# 1 springmvc概述

页面

服务

视图展示

SpringMVC

拦截请求

数据校验

类型转换

数据处理

数据交给服务

数据封装

异常处理

返回数据

抛出异常

ViewResovler

ViewAndModel

请求（数据）

## 1.1是类似与struts2的mvc框架，是springframeWork的后续产品。

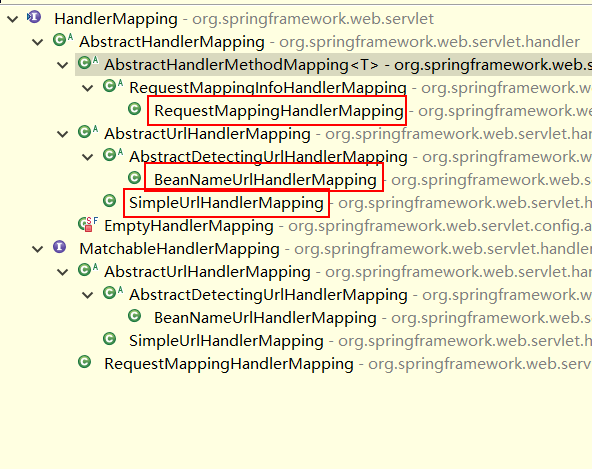
## 1.2 与struts2的区别分析：

|  |  |  |  |
| --- | --- | --- | --- |
| 区分 | Springmvc | Struts2 | 优势 |
| 1 使用率 | 有大量用户，国内新项目都会选用springmvc | 有部分老用户，老项目使用 | 国内springmvc使用率高于struts2 |
| 2 框架入口 | 基于servlet  DispatcherServlet | 基于filter  StrutsPrepareAndExecuteFilter | 本质上没有太大优势之分，知识配置方式不同 |
| 3 框架设计思想 | 控制器基于方法级别的拦截  处理器设计为单例 | 控制器基于类级别的拦截  处理器设计为多例 | 由于设计本身问题，struts2只能设计为多例模式，相比于springmvc的单例模式，struts2会消耗更多的服务器内存 |
| 4 参数传递 | 参数通过方法的入参传递 | 参数通过类的成员变量传递 | Struts2通过成员变量传递参数，导致了参数线程不安全，可能引发并发问题 |
| 5与spring整合 | 与spring同一家公司，  可以与spring无缝整合 | 需要整合包 | Springmvc整合spring更轻松 |

# 2 springmvc组件

2.1前端处理器 DispatcherServlet

2.2处理器映射器 HandlerMapping



2.3处理器适配器 HandlerAdapter

2.4视图解析器 ViewResolver

# 3 参数绑定

## 3.1默认参数

HttpServletRequest

HttpServletResponse

HttpSession

## 3.2 绑定简单参数

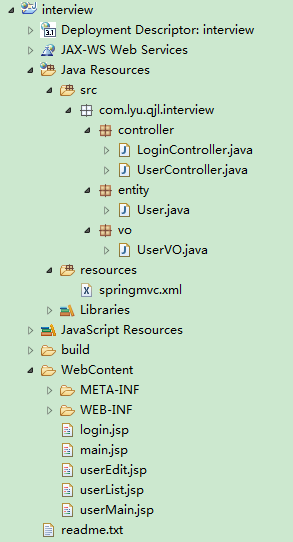
基本数据类型：int、string等

## 3.3 Model/ModelMap(返回数据模型)

### 3.3.1主要有三种：返回字符串，返回 ModelAndView，返回 void

SpringMVC 通过注解实现的 Controller 常用的返回值类型主要有三种：

返回字符串，返回 ModelAndView，返回 void。

先看一下项目的目录结构：   


再看一下 springmvc.xml 配置文件的内容：

<?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:tx="http://www.springframework.org/schema/tx"

xmlns:aop="http://www.springframework.org/schema/aop"

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/tx

http://www.springframework.org/schema/tx/spring-tx.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd

http://www.springframework.org/schema/mvc

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

<!-- 开启spring的组件扫描，自动加载bean -->

<context:component-scan base-package="com.lyu.qjl.interview.controller" />

<!-- 可以用mvc的注解驱动 -->

<mvc:annotation-driven />

<!-- 配置视图解析器 -->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<!-- 配置视图名的默认前缀 -->

<property name="prefix" value="/"></property>

<!-- 配置视图名的默认后缀 -->

<property name="suffix" value=".jsp"></property>

</bean>

</beans>

#### 3.3.1.1 返回字符串

##### 应用场景1：直接返回视图名称，例如跳转到某个功能主页。

下面的 Controller 是用来处理用户页面的请求，gotoUserEdit 方法是用来处理跳转到用户编辑页面的请求。

package com.lyu.qjl.interview.controller;

import java.util.ArrayList;

import java.util.List;

import org.springframework.stereotype.Controller;

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

import org.springframework.web.servlet.ModelAndView;

import com.lyu.qjl.interview.entity.User;

import com.lyu.qjl.interview.vo.UserVO;

/\*\*

\* 类名称：处理用户请求的handler

\* 全限定性类名: com.lyu.qjl.interview.controller.UserController

\* @author 曲健磊

\* @date 2018年5月21日下午7:24:07

\* @version V1.0

\*/

@Controller

public class UserController{

/\*\*

\* 跳转到编辑用户页面

\* @return

\*/

@RequestMapping("/gotoUserEdit")

public String gotoUserEdit() {

return "userEdit";

}

}

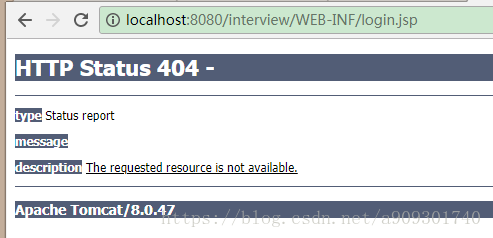
这个 gotoUserEdit 方法匹配 /gotoUserEdit 请求。返回值为一个 userEdit 字符串，表示的是一个视图名，但是由于我们在 springmvc.xml 中已经配置了视图名的前缀和后缀，所以前端控制器实际上获得的视图名为 prefix配置的值 + returnStr（方法返回值） + suffix配置的值，即 /userEdit.jsp，所以页面最终会跳转到网站根目录下的 userEdit.jsp。

**Question：**我想访问某个页面我直接去输入这个页面的地址 /userEdit.jsp 不就可以了吗，为什么还要先去请求 SpringMVC，再由它返回？

**Answer：**对于 WebContent 这一层目录下面的页面，用户如果知道某个 jsp 页面可以直接访问。但是，通常情况下为了保证页面的安全，我们一般的做法是在 WebContent 这一层目录下只留一个引导页面（index.jsp）作为跳转，把网站相关的页面放入 WEB-INF 文件夹下保护起来。因为放在 WEB-INF 文件夹下的页面没有办法通过地址栏直接访问，只能通过后台的跳转来间接的访问，所以就需要请求 SpingMVC 来返回相应的页面。

实际应用中的 web 页面的目录结构：   


**注：login.jsp, main.jsp, userEdit.jsp, userList.jsp, userMain.jsp 都是放在 WEB-INF 这层目录下的，只有 index.jsp 是放在 WebContext 这一层下的。**

测试：直接访问 WEB-INF 文件夹下的 login.jsp 页面的结果如下：   


这样我们就没有办法直接从地址栏去访问 WEB-INF 下的 login.jsp 页面了，只能通过 SpringMVC 来进行页面的跳转了。

这里还需要注意一点：由于页面已经放到了 WEB-INF 目录下，所以 springmvc.xml 配置文件中的视图解析器中的前缀也要对应的有所改动：

<?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:tx="http://www.springframework.org/schema/tx"

xmlns:aop="http://www.springframework.org/schema/aop"

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/tx

http://www.springframework.org/schema/tx/spring-tx.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd

http://www.springframework.org/schema/mvc

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

...

<!-- 配置视图解析器 -->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<!-- 配置视图名的默认前缀 -->

<property name="prefix" value="/WEB-INF/"></property>

<!-- 配置视图名的默认后缀 -->

<property name="suffix" value=".jsp"></property>

</bean>

</beans>

##### 3.3.1.2扩展：

大家注意到那个放在 WebContent 那一层目录下的 index.jsp 页面了吗？它在实际应用中有什么作用呢？

它一般配置为项目的欢迎页面，即 web.xml 中的 welcome-file。当我们在地址栏输入 localhost:8080/interview 的时候，首先就会访问该页面，然后由该页面向后台发请求，通过后台的 Controller 跳转到被 WEB-INF 文件夹保护起来的登录页面 login.jsp，最后，展现在我们面前的就是一个登录页面。   
   
index.jsp 中的内容如下：

<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>

<%

// 转发一个 "/login" 请求给后台的 Controller，由 Controller 返回 WEB-INF 目录下的 login.jsp 页面

request.getRequestDispatcher("/login").forward(request, response);

%>

##### 应用场景2：在登录成功的时候重定向到主页面，登录失败的时候转发回登录页面

String 类型的返回值除了返回一个可以被视图解析器解析的视图名以外，还可以返回 含有 redirect 或 forward 标签的字符串，如下：

package com.lyu.qjl.interview.controller;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

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

/\*\*

\* 类名称：用于处理登录请求的处理器

\* 全限定性类名: com.lyu.qjl.interview.controller.LoginController

\* @author 曲健磊

\* @date 2018年5月21日下午7:02:03

\* @version V1.0

\*/

@Controller

public class LoginController{

@RequestMapping("/loginUser")

public String login(String username, String password, Model model) throws Exception {

if (username != null && password != null) {

if ("admin".equals(username) && "123".equals(password)) {

// 登录成功，重定向到主页面

return "redirect:/main";

} else {

// 向 request 域中设置错误提示信息

model.addAttribute("loginFlag", "用户名或密码错误");

// 登录失败，转发到登录页面

return "forward:/login";

}

} else {

// 只是把模型放到request域里面，并没有真正的安排好

model.addAttribute("loginFlag", "用户名或密码错误");

// 登录失败，转发到登录页面

return "forward:/login";

}

}

@RequestMapping("/login")

// 对外部提供接口的时候需要指定某些参数

public String login() {

return "login";

}

@RequestMapping("/main")

public String main() {

return "main";

}

}

redirect:/main 表示这是一个重定向到 /main 的请求， forward:/login 表示这是一个转发到 /login 的请求。

注意：如果在返回的字符串前加上了 redirect 或者 forward 标签，就不会走视图解析器，而是直接转发或重定向到指定的路径，所以对于重定向标签后面就不能加 WEB-INF ，因为重定向是客户端重新发送的请求，WEB-INF 目录下的页面仍然访问不到，但是对于转发标签的话后面可以跟 WEB-INF。

以上就是 Controller 的返回值为 String 时的另一种常见用途。

**Question：**为什么登录成功需要重定向到主页面？

**Answer：**因为转发客户端只发送了一次请求，如果登录成功转发到主页面的话，浏览器的地址栏中的 url 保留的是提交表单的请求，此时如果按 F5 刷新，就会造成二次提交表单，增大服务端的压力不说，还影响用户的体验，所以登录成功时要重定向到主页面，这样浏览器的地址栏保留的就是当前主页面的 url，无论怎么刷新都没事。   


**Question：**为什么登录失败需要转发回登录页面？

**Answer：**用户登录失败，我们需要给用户提供相应的提示信息，如：“用户名密码不匹配”。如果重定向到登录页面，用户就没有办法获取到服务端设置在 request 域里面的错误提示信息，因为重定向的本质是客户端发送了两次请求，在第二次请求里面获取不到第一次请求中的数据，所以没有办法给用户相应的提示信息。而转发只发送了一次请求，转发回登录页面的时候可以获取到请求中的数据，所以登录失败时需要转发会登录页面。   


#### 3.3.1.3返回ModelAndView

##### 应用场景：其实这种返回值的应用场景比较随意，既可以用来做页面的跳转，也可以在跳转到页面的时候携带一些数据，例如：查询用户列表。

package com.lyu.qjl.interview.controller;

import java.util.ArrayList;

import java.util.List;

import org.springframework.stereotype.Controller;

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

import org.springframework.web.servlet.ModelAndView;

import com.lyu.qjl.interview.entity.User;

import com.lyu.qjl.interview.vo.UserVO;

/\*\*

\* 类名称：处理用户请求的handler

\* 全限定性类名: com.lyu.qjl.interview.controller.UserController

\* @author 曲健磊

\* @date 2018年5月21日下午7:24:07

\* @version V1.0

\*/

@Controller

public class UserController{

@RequestMapping("/getUserList")

public ModelAndView getUserList() {

System.out.println("进入getUseList");

ModelAndView modelAndView = new ModelAndView();

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

User user1 = new User("arry", 18, "男");

user1.setUserId(1L);

User user2 = new User("cc", 28, "女");

user2.setUserId(2L);

User user3 = new User("dd", 38, "男");

user3.setUserId(3L);

userList.add(user1);

userList.add(user2);

userList.add(user3);

// 将数据放入modelAndView对象

modelAndView.addObject("userList", userList);

// 将返回的[逻辑视图](https://www.baidu.com/s?wd=%E9%80%BB%E8%BE%91%E8%A7%86%E5%9B%BE&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)名称放入modelAndView对象

modelAndView.setViewName("userList");

return modelAndView;

}

}

如果想在跳转到某个页面的时候携带一些数据，调用 ModelAndView 对象的 addObject 方法设置一些属性即可，不想带数据也可以不设置，但是必须得设置视图名 setViewName ，要不然它哪知道返回到哪个页面去呢？

#### 3.3.1.4返回void

如果是返回 void 会，就必须要在方法的参数中添加 HttpServletRequest 和 HttpServletResponse 来进行页面的跳转，代码如下：

public void test(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

// 转发到指定页面

request.getRequestDispatcher("xxx").forward(request, response);

// 或者重定向到指定页面

response.sendRedirect("/xxx");

}

其实，从本质上来说前面两种方法（String，ModelAndView）就是间接调用的 request 和 response 这个两个对象来进行页面的跳转，以及数据的存储。

稍微的总结一下，这篇文章主要讲了 SpringMVC 的 Controller 的三种返回值类型：String，ModelAndView，void。在讲 String 的时候扩展了一些 redirect 和 forward 标签，提了一下在实际应用中转发和重定向的用处，以及实际项目中需要把页面放入 WEB-INF 目录里保护起来。ModelAndView 可以携带数据也可以不携带数据，以及前两种方法本质上都是在操作 request 和 response 对象。

### 3.3.2 问题及答案

#### 问答一：我想访问某个页面我直接去输入这个页面的地址 /userEdit.jsp 不就可以了吗，为什么还要先去请求 SpringMVC，再由它返回？

**Question：**我想访问某个页面我直接去输入这个页面的地址 /userEdit.jsp 不就可以了吗，为什么还要先去请求 SpringMVC，再由它返回？

**Answer：**

对于 WebAPP 这一层目录下面的页面，用户如果知道某个 jsp 页面可以直接访问。

但是，通常情况下为了保证页面的安全，我们一般的做法是在 WebAPP这一层目录下只留一个引导页面（index.jsp）作为跳转，把网站相关的页面放入 WEB-INF 文件夹下保护起来。因为放在 WEB-INF 文件夹下的页面没有办法通过地址栏直接访问，只能通过后台的跳转来间接的访问，所以就需要请求 SpingMVC 来返回相应的页面。

#### 问答二：index.jsp的作用

大家注意到那个放在 WebContent 那一层目录下的 index.jsp 页面了吗？它在实际应用中有什么作用呢？

它一般配置为项目的欢迎页面，即 web.xml 中的 welcome-file。当我们在地址栏输入 localhost:8080/interview 的时候，首先就会访问该页面，然后由该页面向后台发请求，通过后台的 Controller 跳转到被 WEB-INF 文件夹保护起来的登录页面 login.jsp，最后，展现在我们面前的就是一个登录页面。   
   
index.jsp 中的内容如下：

<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>

<%

// 转发一个 "/login" 请求给后台的 Controller，由 Controller 返回 WEB-INF 目录下的 login.jsp 页面

request.getRequestDispatcher("/login").forward(request, response);

%>

#### 问答三：请求转发和重定向：

redirect:/main 表示这是一个重定向到 /main 的请求，

forward:/login 表示这是一个转发到 /login 的请求。

两者的不同的适用场景：  
请求转发只需要请求一次服务器，可以提高访问速度。  
重定向可以跳转到任意服务器，可以用在系统间的跳转。

注意：

如果在返回的字符串前加上了 redirect 或者 forward 标签，就不会走视图解析器，而是直接转发或重定向到指定的路径，

所以对于重定向标签后面就不能加 WEB-INF ，因为重定向是客户端重新发送的请求，WEB-INF 目录下的页面仍然访问不到，

但是对于转发标签的话后面可以跟 WEB-INF。

以上就是 Controller 的返回值为 String 时的另一种常见用途。

#### 问答四：为何登录成功要重定向到主页面

**Question：**为什么登录成功需要重定向到主页面？

**Answer：**

因为转发客户端只发送了一次请求，如果登录成功转发到主页面的话，浏览器的地址栏中的 url 保留的是提交表单的请求，此时如果按 F5 刷新，就会造成二次提交表单，增大服务端的压力不说，还影响用户的体验，

所以登录成功时要重定向到主页面，这样浏览器的地址栏保留的就是当前主页面的 url，无论怎么刷新都没事。

#### 问答五： 为什么登录失败需要转发回登录页面？

**Question：**为什么登录失败需要转发回登录页面？

**Answer：**

用户登录失败，我们需要给用户提供相应的提示信息，如：“用户名密码不匹配”。

如果重定向到登录页面，用户就没有办法获取到服务端设置在 request 域里面的错误提示信息，

因为重定向的本质是客户端发送了两次请求，在第二次请求里面获取不到第一次请求中的数据，所以没有办法给用户相应的提示信息。

而转发只发送了一次请求，转发回登录页面的时候可以获取到请求中的数据，所以登录失败时需要转发会登录页面。

#### 问答六：使用ModelAndView时注意事项

如果想在跳转到某个页面的时候携带一些数据，调用 ModelAndView 对象的 addObject 方法设置一些属性即可，不想带数据也可以不设置，

但是必须得设置视图名 setViewName ，要不然它哪知道返回到哪个页面去。

#### 问答七：SpringMVC 中Controller实现页面跳转的三种返回类型

1 返回String

2 返回ModelAndView

3 返回Void，但是必须要有HttpServletRequest 和HttpServletResponse来完成转发或重定向

## 3.4绑定pojo对象

表单提交的name属性必须与pojo的属性名称一致

基本数据类型不是必须一致：get或post表单，但是需要两个注解@RequestParam和@PathVariable 来声明名称一致

@RequestParam(name="name") String namee

## 3.5绑定包装的pojo

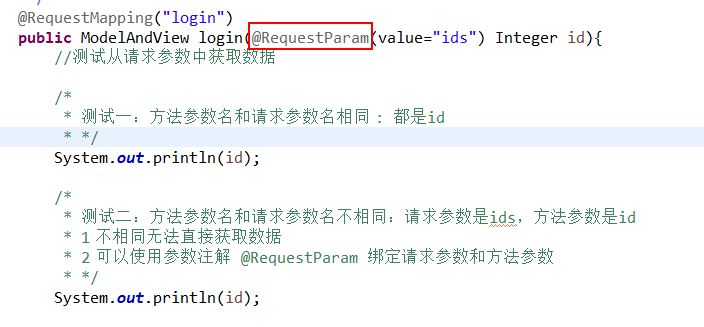
要的：通过点( . ）传递属性

## 3.6 参数中重要注解

### 3.6.1 @RequestParam

测试一：http://localhost:8080/springmvc/login.action?id=1999

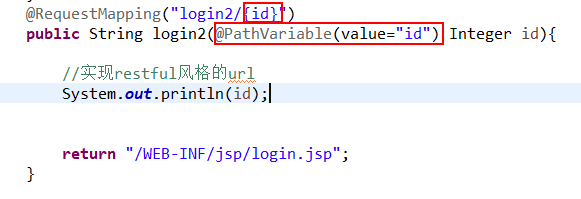
测试二：http://localhost:8080/springmvc/login.action?ids=1999



### 3.6.2 @PathVariable(实现restful风格url)

虽然也是用在方法参数中，但是与@RequestParam 不同，它绑定的不是请求参数。而是请求url地址中的参数.

测试一：http://localhost:8080/springmvc/login/ 1999



输入结果是：1999

真正实现restful风格，要把拦截路径改为/，如果是\*.action则要在最后加上.action，不符合



# 4 异步加载

## 4.1 json交互

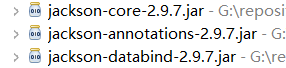
### 4.1.1 配置文件设置

springmvc.xml中开启

<!-- 打开该注解相当于使用最新的HandlerMapping和HandlerAdapter，而且还对json交互提供了支持 -->

<mvc:annotation-driven/>

### 4.1.2 需要jar包



<!-- jackson相关包 -->

<dependency>

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

<artifactId>jackson-core</artifactId>

<version>2.9.7</version>

</dependency>

<dependency>

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

<artifactId>jackson-annotations</artifactId>

<version>2.9.7</version>

</dependency>

<dependency>

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

<artifactId>jackson-databind</artifactId>

<version>2.9.7</version>

</dependency>

### 4.1.3 关键注解 @ResponseBody 和 @RequestBody

@ResponseBody :

负责管理向页面异步发送：基本数据类型/json对象/json对象数组（List、Map）

@RequestBody

负责接收页面发来的：json对象/json对象数组

### 4.1.4 异步请求测试

#### 4.1.4.1返回字符串（基本类型）

#### 4.1.4.2返回json对象

#### 4.1.4.3返回json对象数组（集合）

#### 4.1.4.4给服务器发送json对象

#### 3.1.4.5给服务器发送json对象列表(数组)



服务器代码：

**package** cn.htb.controller;

**import** java.util.Arrays;

**import** java.util.Date;

**import** java.util.List;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.ui.Model;

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

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

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

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

**import** cn.htb.pojo.User;

@Controller

**public** **class** UserController {

/\*\*

\* springmvcr入门测试：返回ModelAndView对象

\*

\* **@return**

\*/

@RequestMapping("login")

**public** ModelAndView login(){

ModelAndView mav = **new** ModelAndView();

mav.addObject("msg", "springmvc hello");

mav.setViewName("/WEB-INF/jsp/login.jsp");

**return** mav;

}

/\*\*

\* 异步加载测试一：返回字符串

\* **@return**

\*/

@RequestMapping("ajax")

@ResponseBody

**public** String ajax(){

**return** "ajax==============";

}

/\*\*

\* 异步加载测试二：返回json对象

\* **@return**

\*/

@RequestMapping("ajax2")

@ResponseBody

**public** User ajax2(){

User user = **new** User();

user.setId(1);

user.setName("张三");

user.setAge(18);

**return** user;

}

/\*\*

\* 异步加载测试三：接收页面传来的json对象

\* **@return**

\*/

@RequestMapping("ajax3")

@ResponseBody

**public** User ajax3(@RequestBody User user){

System.***out***.println(user);

user.setId(2);

user.setName("李四");

user.setAge(18);

**return** user;

}

/\*\*

\* 异步加载测试四：接收页面传来的json对象，并发送json对象列表（数组）

\* **@return**

\*/

@RequestMapping("ajax4")

@ResponseBody

**public** List<User> ajax4(@RequestBody User user){

System.***out***.println(user);

List<User> userList = Arrays.*asList*(**new** User(1,"1号",16),

**new** User(2,"2号",17),

**new** User(3,"3号",18),

**new** User(4,"4号",19));

**return** userList;

}

/\*\*

\* 异步加载测试五：接收页面传来的json对象，并发送json对象列表（数组）

\* **@return**

\*/

@RequestMapping("ajax5")

@ResponseBody

**public** List<User> ajax5(@RequestBody List<User> userList){

**if**(userList != **null** && userList.size()>0){

**for**(User temp:userList){

System.***out***.println(temp.getName());

}

}

**return** userList;

}

}

<%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"* isELIgnored=*"false"*%>

<%

String path = request.getContextPath();

String basePath = request.getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+path+"/";

%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

<head>

<base href=*"*<%=basePath%>*"*>

<title>My JSP 'logini.jsp' starting page</title>

<meta http-equiv=*"pragma"* content=*"no-cache"*>

<meta http-equiv=*"cache-control"* content=*"no-cache"*>

<meta http-equiv=*"expires"* content=*"0"*>

<meta http-equiv=*"keywords"* content=*"keyword1,keyword2,keyword3"*>

<meta http-equiv=*"description"* content=*"This is my page"*>

<!--

<link rel="stylesheet" type="text/css" href="styles.css">

-->

<script type=*"text/javascript"*>

/\*

异步五步走：

1、获取XMLHttpRequest

2、打开与服务器的连接open(method,url,true);

3、设置请求头 Content-Type

4、发送send(null) 参数为相应体参数，如果是get方法，则要填写null，不然部分浏览器可能不能发送

5、给onreadaystatechange注册监听

\*/

/\* =============================1 ======================================== \*/

window.onload = **function**(){

**var** button = document.getElementById("button");

button.onclick = **function**(){

**var** xmlHttp = **new** XMLHttpRequest();

xmlHttp.open("POST","ajax.action",**true**);

xmlHttp.setRequestHeader("Content-Type", "application/x-www-form-urlencoded")

xmlHttp.send(**null**);

xmlHttp.onreadystatechange = **function**(){

**var** div = document.getElementById("div");

**if**(xmlHttp.readyState==4 && xmlHttp.status==200){//说明发送相应成功

**var** text = xmlHttp.responseText;//接收string数据的方法

alert(text);

div.innerHTML=text;

}

}

}

/\* =======================2 ============================================== \*/

**var** button2 = document.getElementById("button2");

button2.onclick = **function**(){

**var** xmlHttp = **new** XMLHttpRequest();

xmlHttp.open("POST","ajax2.action",**true**);

xmlHttp.setRequestHeader("Content-Type", "application/x-www-form-urlencoded")

xmlHttp.send(**null**);

xmlHttp.onreadystatechange = **function**(){

**var** div2 = document.getElementById("div2");

**if**(xmlHttp.readyState==4 && xmlHttp.status==200){//说明发送相应成功

**var** text = xmlHttp.responseText;

alert(text);

div2.innerHTML=text;

}

}

}

/\* =====================3 ================================================ \*/

**var** button3 = document.getElementById("button3");

button3.onclick = **function**(){

**var** xmlHttp = **new** XMLHttpRequest();

xmlHttp.open("POST","ajax3.action",**true**);

xmlHttp.setRequestHeader("Content-Type", "application/json")

/\* === 发送json对象格式要正确，不然会请求参数格式错误，用引号括着大括号 === \*/

xmlHttp.send('{"id":1,"name":"张三","age":18}');

xmlHttp.onreadystatechange = **function**(){

**var** div3 = document.getElementById("div3");

**if**(xmlHttp.readyState==4 && xmlHttp.status==200){//说明发送相应成功

**var** data = xmlHttp.responseText;

**var** user = eval("("+data+")");

div3.innerHTML=data+user.name;

}

}

}

/\* =====================4 ================================================ \*/

**var** button4 = document.getElementById("button4");

button4.onclick = **function**(){

**var** xmlHttp = **new** XMLHttpRequest();

xmlHttp.open("POST","ajax4.action",**true**);

xmlHttp.setRequestHeader("Content-Type", "application/json")

xmlHttp.send('{"id":1,"name":"张三","age":18}');

xmlHttp.onreadystatechange = **function**(){

**var** div4 = document.getElementById("div4");

**if**(xmlHttp.readyState==4 && xmlHttp.status==200){//说明发送相应成功

**var** data = xmlHttp.responseText;

**var** userList = eval("("+data+")");

**for**(**var** i = 0;i<userList.length;i++){

alert(userList[i].name);

}

div4.innerHTML=data;

}

}

}

/\* =====================5 ================================================ \*/

**var** button5 = document.getElementById("button5");

button5.onclick = **function**(){

alert(123);

**var** xmlHttp = **new** XMLHttpRequest();

xmlHttp.open("POST","ajax5.action",**true**);

xmlHttp.setRequestHeader("Content-Type", "application/json")

xmlHttp.send('[{"id":1,"name":"张三","age":18},{"id":1,"name":"李四","age":18},{"id":1,"name":"王五","age":18}]');

xmlHttp.onreadystatechange = **function**(){

**var** div5 = document.getElementById("div5");

**if**(xmlHttp.readyState==4 && xmlHttp.status==200){//说明发送相应成功

**var** data = xmlHttp.responseText;

**var** userList = eval("("+data+")");

**for**(**var** i = 0;i<userList.length;i++){

alert(userList[i].name);

}

div5.innerHTML=data;

}

}

}

}

</script>

</head>

<body>

${msg}

<div>异步加载测试一：返回字符串</div>

<input id=*"button"* type=*"button"* value=*"点击"* onclick="ajax()"/>

<div id=*"div"* style="width: *500px*;height: *100px*;background-color: *red*"></div>

<div>异步加载测试二：返回json对象</div>

<input id=*"button2"* type=*"button"* value=*"点击"* onclick="ajax()"/>

<div id=*"div2"* style="width: *500px*;height: *100px*;background-color: *red*"></div>

<div>异步加载测试三：发送json对象给服务器</div>

<input id=*"button3"* type=*"button"* value=*"点击"* onclick="ajax()"/>

<div id=*"div3"* style="width: *500px*;height: *100px*;background-color: *red*"></div>

<div>异步加载测试四：发送json对象给服务器，并接收json对象列表</div>

<input id=*"button4"* type=*"button"* value=*"点击"* onclick="ajax()"/>

<div id=*"div4"* style="width: *500px*;height: *100px*;background-color: *red*"></div>

<div>异步加载测试五：发送json对象数组给服务器，并接收json对象列表</div>

<input id=*"button5"* type=*"button"* value=*"点击"* onclick="ajax()"/>

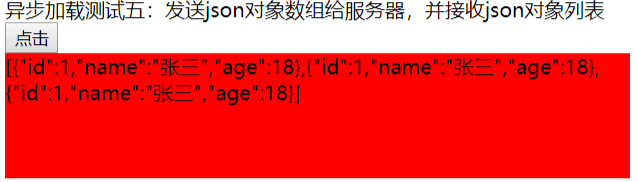
<div id=*"div5"* style="width: *500px*;height: *100px*;background-color: *red*"></div>

</body>

</html>

测试效果：





# 5 [四大器](自定义类型转换器和数据验证.oxps)（异常处理器/拦截器、校验器/转换器）

## 5.1 异常处理器

异常处理方法：

1 自定义全局异常处理器，实现HandlerExceptionResolver接口

2 springmvc提供的SimpleMappingExceptionResolver

3 使用@ExceptionHandler注解 / 还可结合使用@ControllerAdvice

### 5.1.1 自定义全局异常处理器（重点）

#### 5.1.1.1 自定义异常处理器



#### 5.1.1.2 springmvc配置

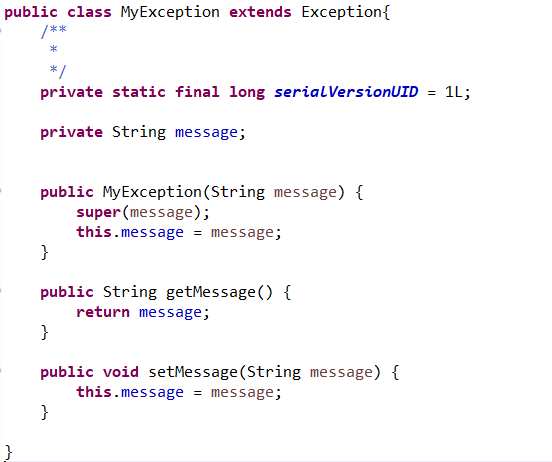


### 5.1.2 使用SimpleMappingExceptionResolver

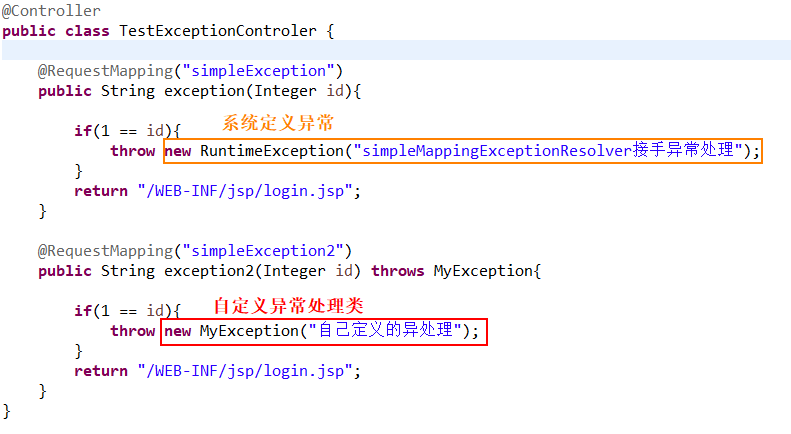
#### 5.1.2.1springmvc配置文件：

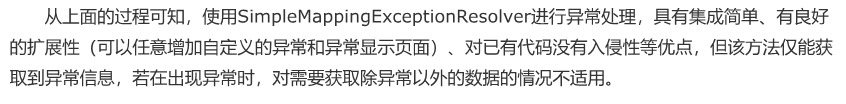


#### 5.1.2.2自定义的异常处理类：



#### 5.1.2.3测试代码：





SimpleMappingExceptionResolver进行异常处理，具有配置简单，扩展性强，对已有代码没有入侵等优点， 但是只能获取到异常信息，如果想要获得更多的数据则不适用。

### 5.1.3 使用@ExceptionHandler 注解

使用方法：使用@ExceptionHandler注解异常处理方法。

缺陷：必须是 处理异常的方法和发生异常的方法在同一个Controller中。

解决：

方法一：创建一个BaseController ，在里面创建异常处理方法，别的Controller都继承它。

方法二：结合@ControllerAdvice注解。使用它注解包含处理异常处理方法的controller，达到增强目的，使之也能够处理别的controller中的异常。

仍有缺陷：代码入侵比较明现，不常用。

#### 5.1.3.1 常规使用@ExceptionHandler

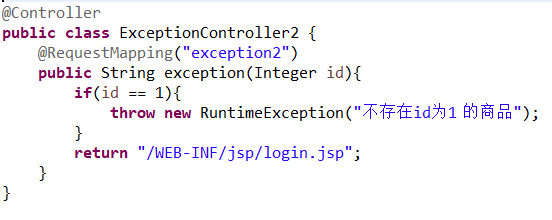


#### 5.1.3.2结合@ControllerAdvice

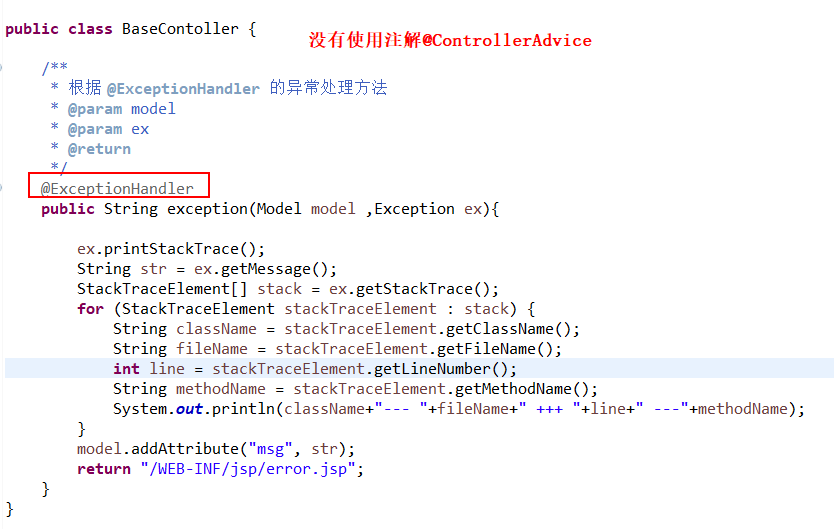
包含异常处理方法的Controller



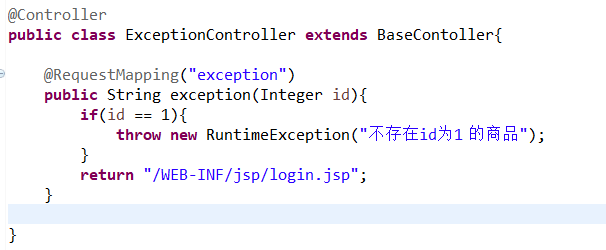
发生异常的controller



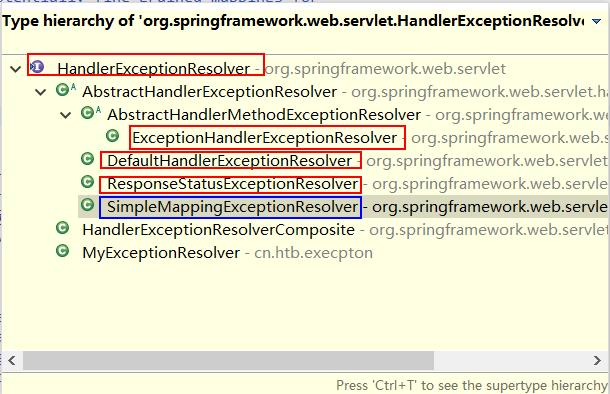
#### 5.1.3.3使用BaseController



实现给类的：



### 5.1.4 异常处理扩展



#### 5.1.4.1. AbstractHandlerMethodExceptionResolver

和ExceptionHandlerExceptionResolver一起使用，完成使用@ExceptionHandler注释的方法进行对异常的解析。

#### 5.1.4.2. ResponseStatusExceptionResolver:

解析有@ResponseStatus注解的异常。

在我们自定义的异常上使用ResponseStatus注解。当我们的Controller抛出异常，并且没有被处理的时候，他将返回HTTP STATUS 为指定值的 HTTP RESPONSE，比如：

@ResponseStatus(value=HttpStatus.NOT\_FOUND, reason="No such Order") // 404

public class OrderNotFoundException extends RuntimeException {

// ...

}

我们的Controller为：

@RequestMapping(value="/orders/{id}", method=GET)

public String showOrder(@PathVariable("id") long id, Model model) {

Order order = orderRepository.findOrderById(id);

if (order == null) throw new OrderNotFoundException(id);

model.addAttribute(order);

return "orderDetail";

}

这时候会返回404，转到404页面而不是错误页面。

在一个Controller中，通过增加使用注解@ExceptionHandler的方法来处理@RequestMapping方法抛出的异常，注意这种只在单个Controller中有效。

这么做可以：

1发生异常后，改变Response status，一般而言，发生异常返回HTTP STATUS 500.我们可以变为其他。

2发生错误后转到错误页面

3可以为不同异常定义不同处理（如不同的错误页面，不同的Response status）

@Controller

public class ExceptionHandlingController {

// 我们标注了@RequestMapping的方法

...

//处理异常的方法。

// 把我们定义的异常转换为特定的Http status code

@ResponseStatus(value=HttpStatus.CONFLICT, reason="Data integrity violation") // 409

@ExceptionHandler(DataIntegrityViolationException.class)

public void conflict() {

// Nothing to do

}

// 捕获到SQLException，DataAccessException异常之后，转到特定的页面。

@ExceptionHandler({SQLException.class,DataAccessException.class})

public String databaseError() {

//仅仅转到错误页面，我们在页面上得不到这个Exception的值，要得到值，我们可以通过下面的方法得到

return "databaseError";

}

// 通过ModelAndView返回页面，以及往页面传相应的值

@ExceptionHandler(Exception.class)

c public ModelAndView handleError(HttpServletRequest req, Exception exception) {

logger.error("Request: " + req.getRequestURL() + " raised " + exception);

ModelAndView mav = new ModelAndView();

mav.addObject("ex eption", exception);

mav.addObject("url", req.getRequestURL());

mav.setViewName("error");

return mav;

}

}

#### 5.1.4.3. DefaultHandlerExceptionResolver：

按照不同的类型分别对异常进行解析。

#### 5.1.4.4. SimpleMappingExceptionResolver:

通过配置的异常类和view的对应关系来解析异常。

## 6.1 拦截器

### 6.1.1 DispatherServlet中重要方法doDispach()，中拦截器信息分析

protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {

try {

try {

//.......省略代码

//返回HandlerExecutionChain 其中包含了拦截器队列

mappedHandler = getHandler(processedRequest);

// 1 调用拦截器PreHandle方法，若返回false不执行Controller逻辑，并不调用下面的PostHandle方法

if (!mappedHandler.applyPreHandle(processedRequest, response)) {

return;

}

// 处理Controller层

mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

applyDefaultViewName(processedRequest, mv);

// 2 调用拦截器的PostHandle方法

mappedHandler.applyPostHandle(processedRequest, response, mv);

}

catch (Exception ex) {

dispatchException = ex;

}

processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);

}

catch (Exception ex) {

// 3 抛出异常后都会调用拦截器AfterCompletion方法

triggerAfterCompletion(processedRequest, response, mappedHandler, ex);

}

finally {

if (asyncManager.isConcurrentHandlingStarted()) {

// Instead of postHandle and afterCompletion

if (mappedHandler != null) {

//若Controller方法为异步调用，则执行拦截器afterConcurrentHandlingStarted（只有AsyncHandlerInterceptor拦截器才有）

mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);

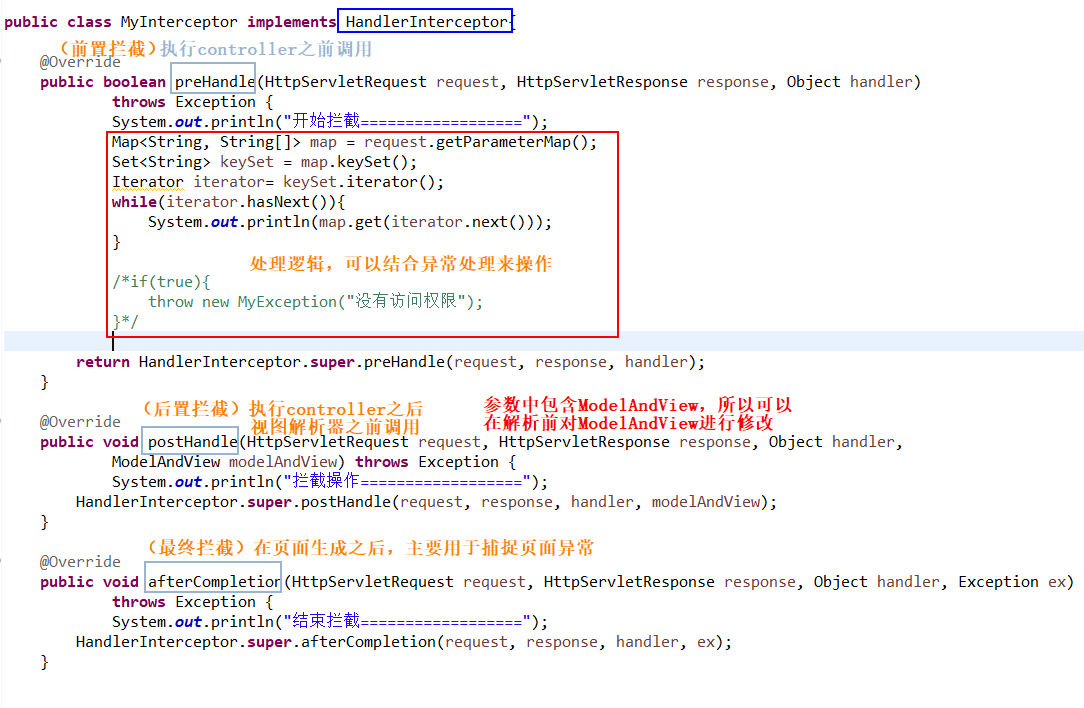
}

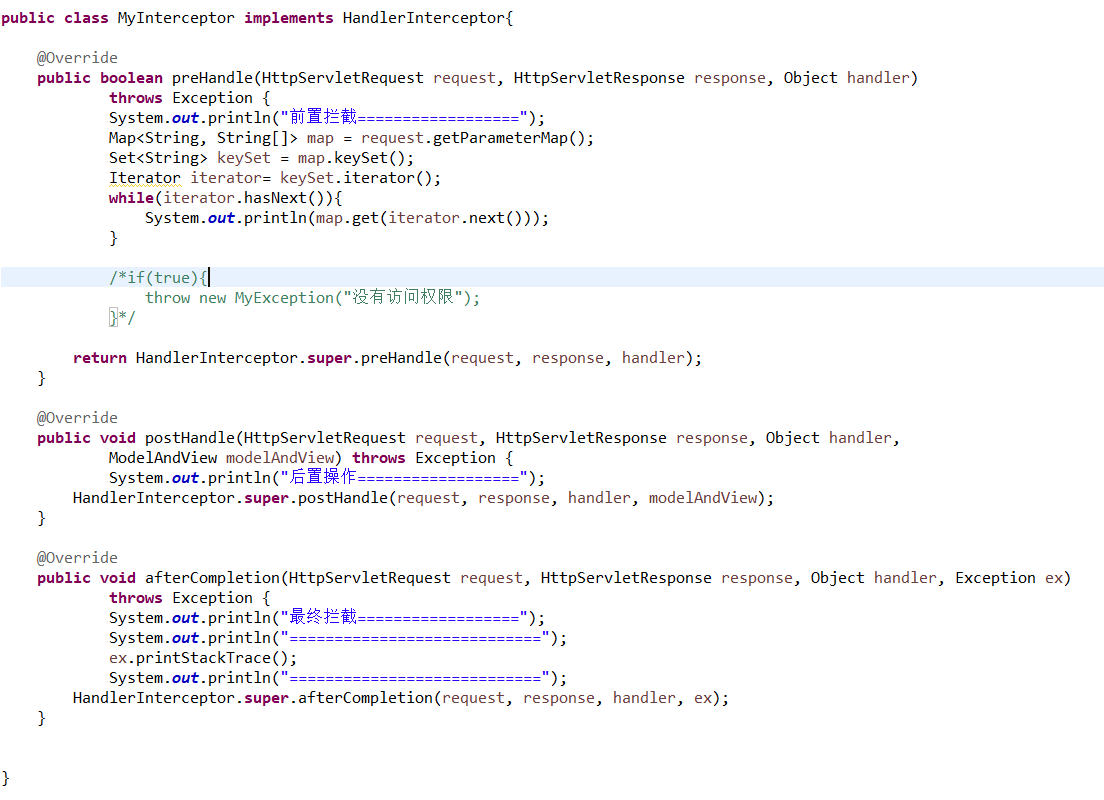
}

}

}

### 6.1.2 拦截器分析





#### 6.1.2.1 preHandle( )

该方法在controller之前调用，可用于权限校验，数据验证等

#### 6.1.2.2 postHandle()

该方法在Controller之后，ViewResolver之前调用，所以可以对ModelAndView进行修改操作。比如：增加额外的信息。

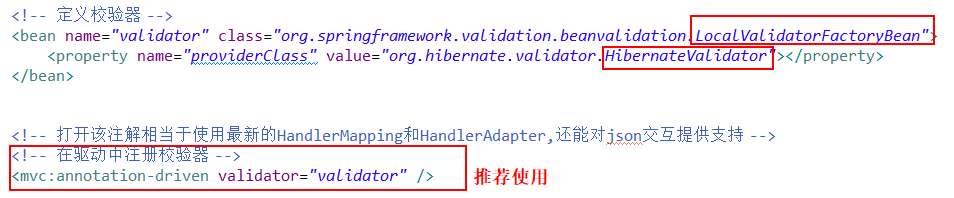
#### 6.1.2.3 afterComplet()

该方法在页面生成之后调用，可用于捕捉页面异常信息，比如在页面中：<%= 1/0%>,会捕获该异常。

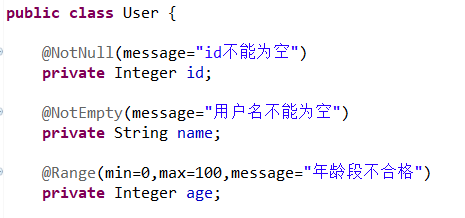
## 7.1数据校验器

### 7.1.1 使用hibernate注解校验

#### 7.1.1.1 文件配置一：驱动注册校验器



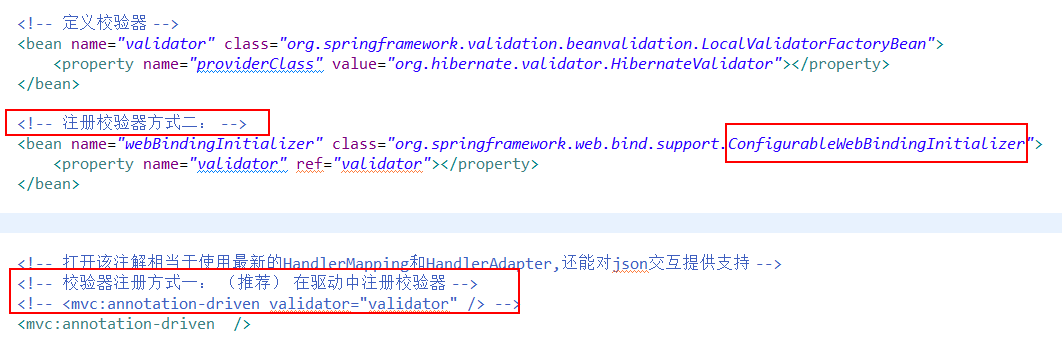
javaBean例子：



控制器例子：Errors /BindingResult 要紧跟被校验的参数，每个校验参数跟一个。可以有多个

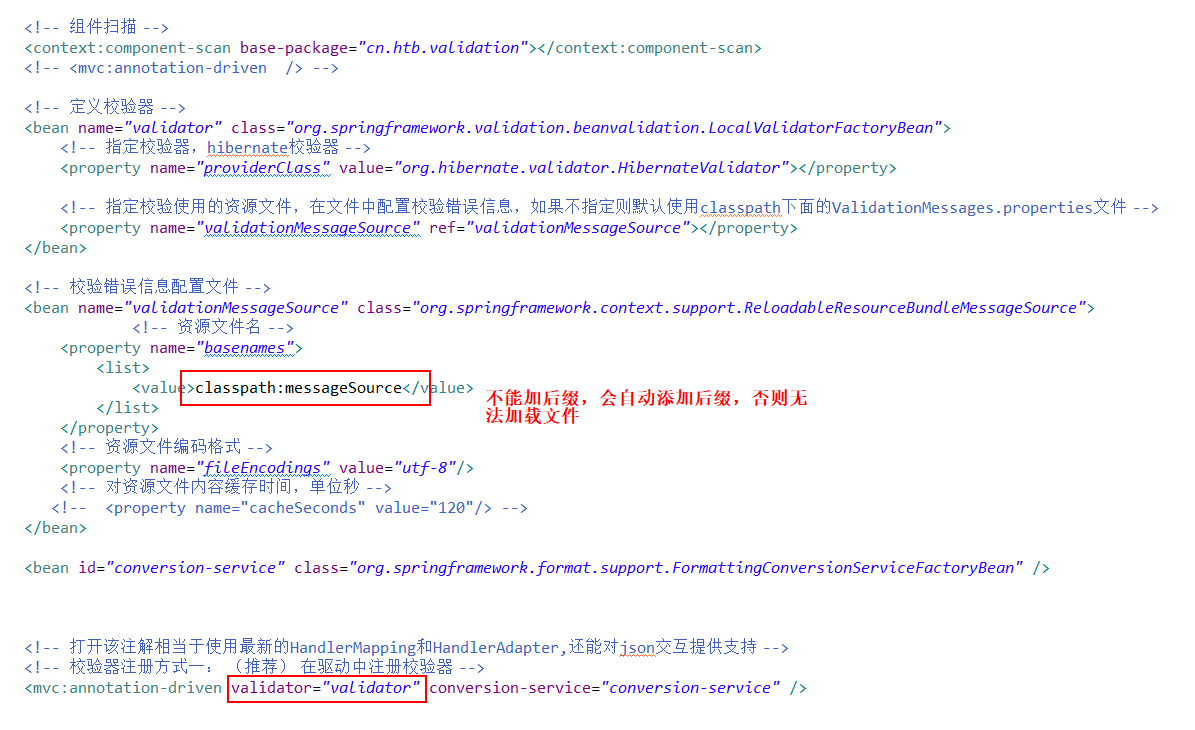


#### 7.1.1.2 文件配置方式二：

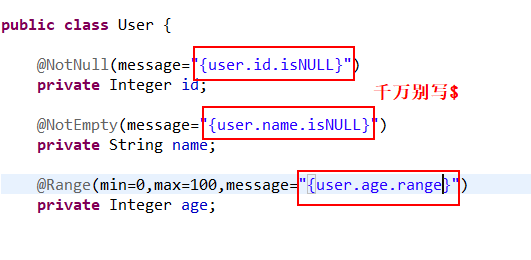


#### 7.1.1.3 使用外部信息文件

springmvc文件配置：



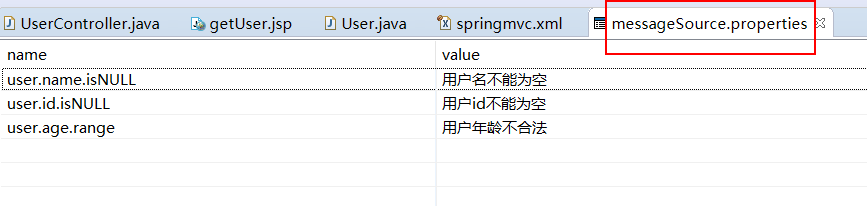
javaBean代码：



controller代码：



属性文件信息：

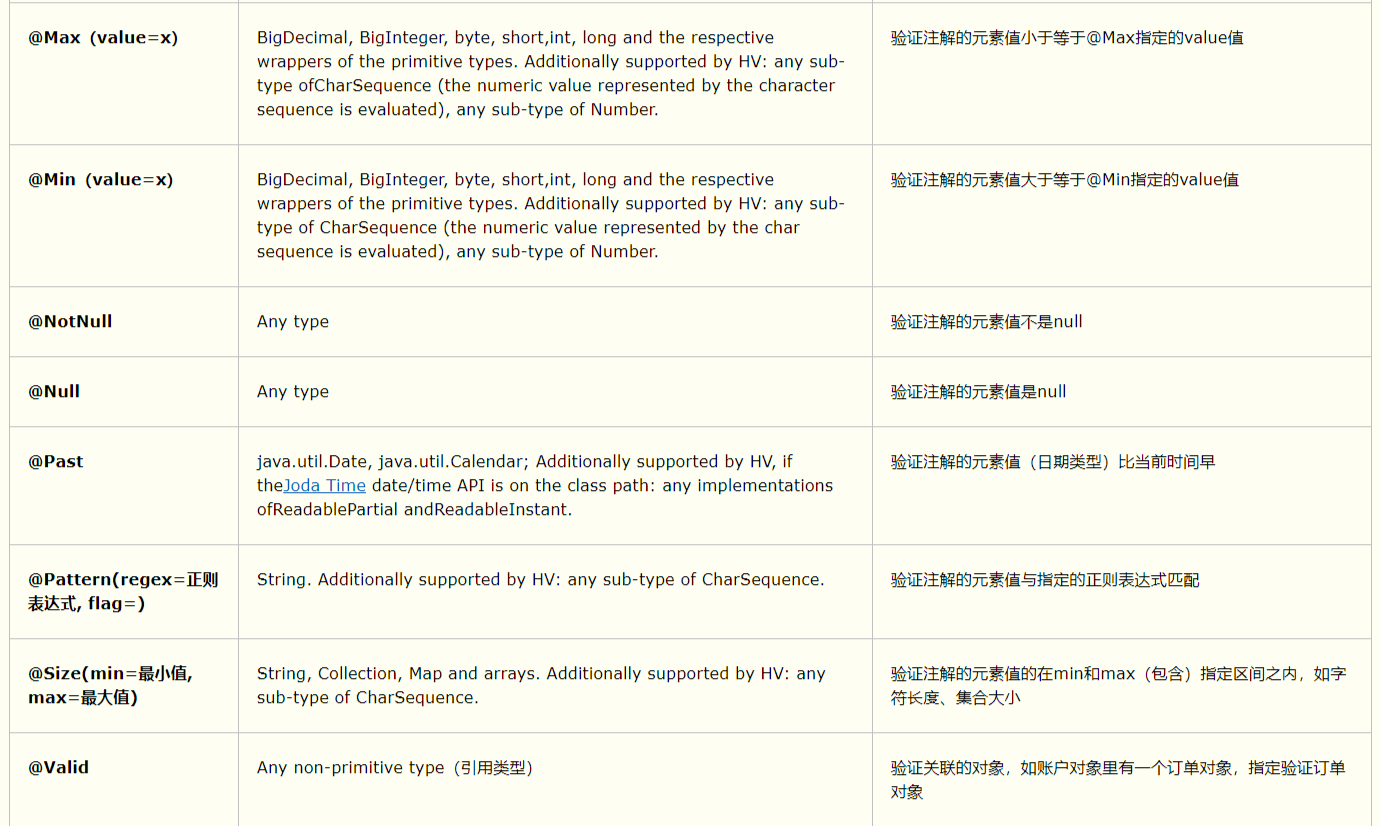


jsp页面配置：



#### 7.1.1.4 常用校验注解







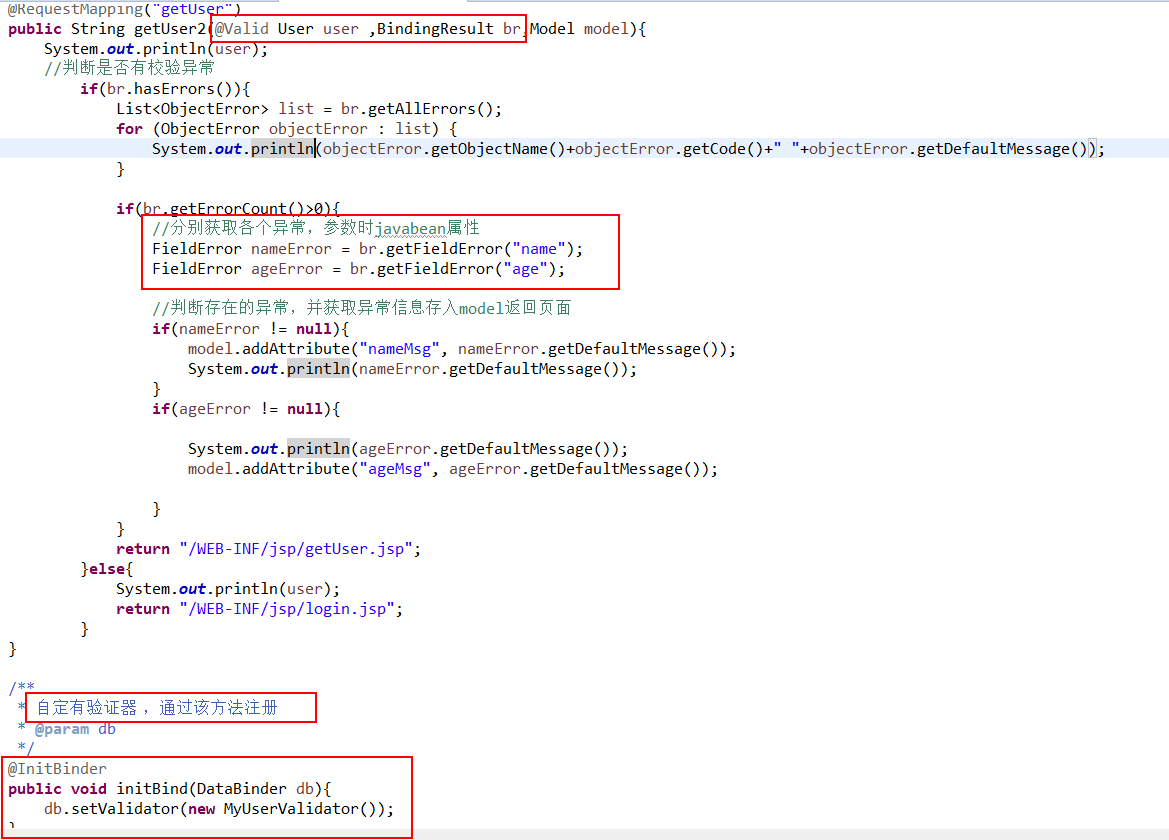
### 7.1.2 自定义数据验证器

#### 7.1.5.1 自定义校验器



#### 7.1.5.2controller

自定义校验器，不通过配置文件配置。主要通过以下方法配置：



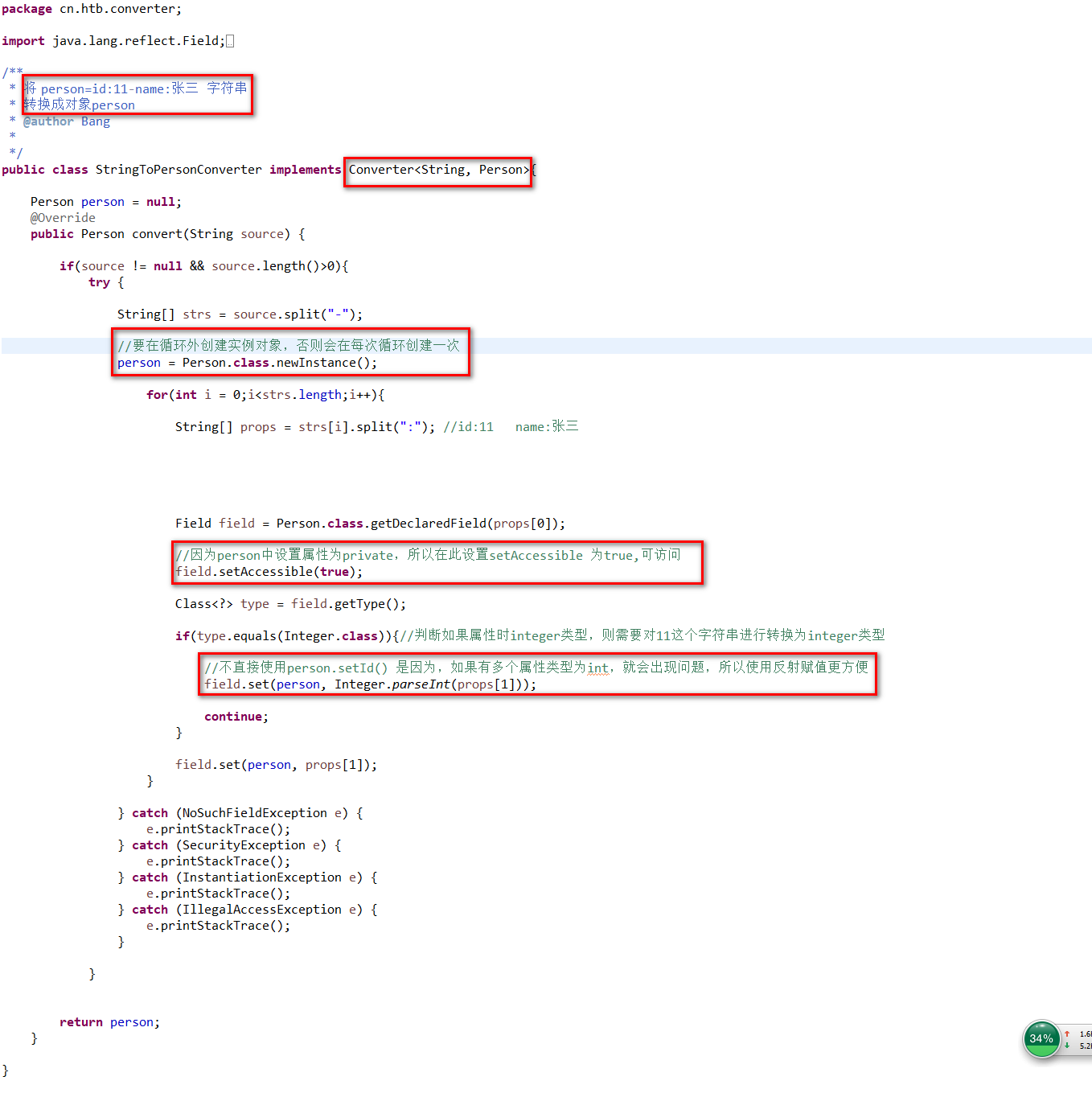
## 8.1转换器（格式化）

### 8.1.1 自定义类型转换器

#### 8.1.1.1目标：将字符转换为对象

将 person=id:11-name:张三 字符串转换成对象person

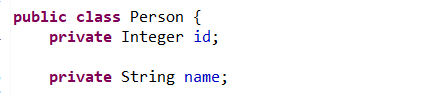
#### 8.1.1.2 转换器代码



#### 8.1.1.3 springmvc配置信息



#### 8.1.1.4 javaBean代码



### 8.1.2 注解转换器（格式化）



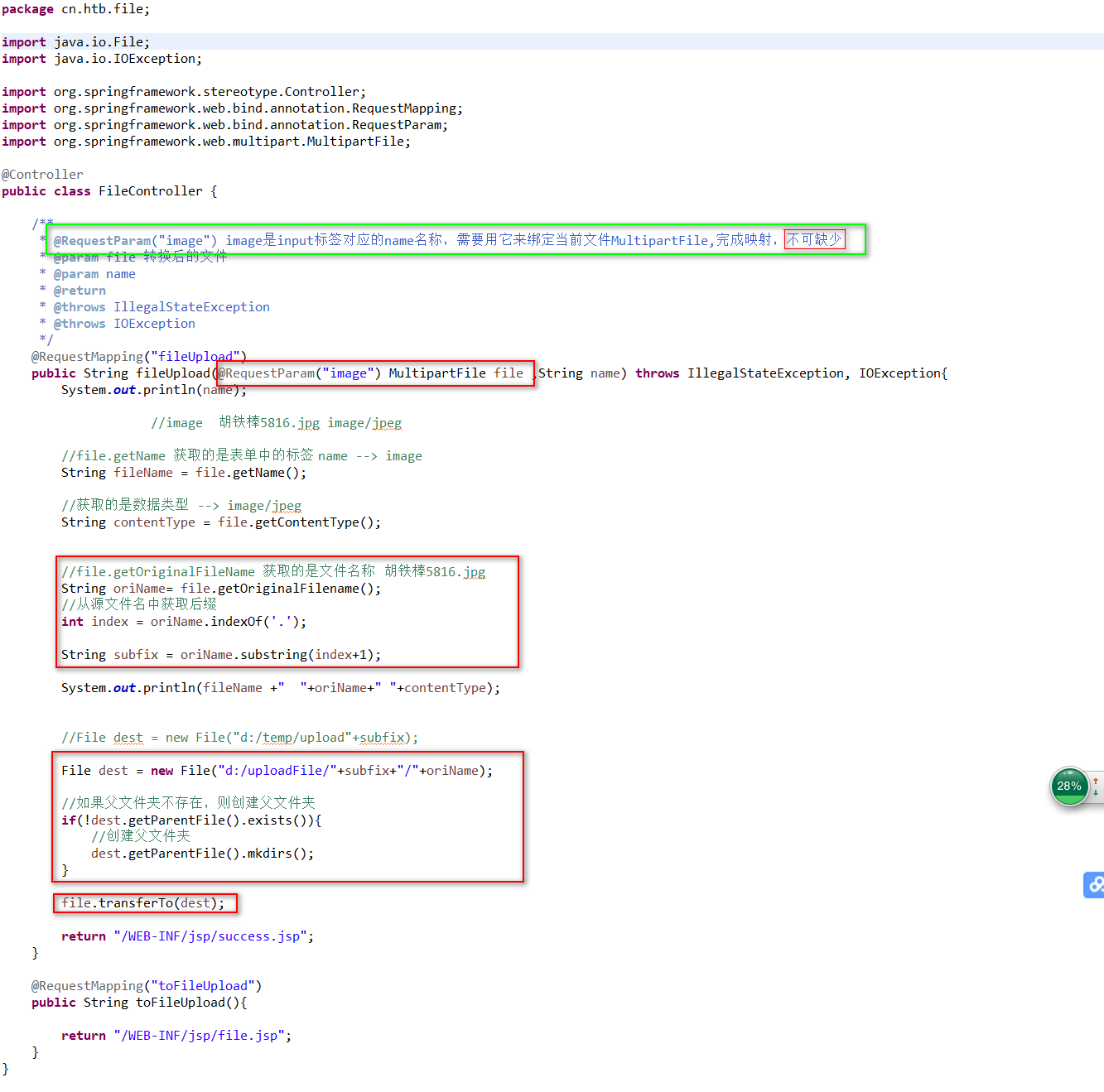
# 8文件上传下载

## 8.1 文件上传

### 8.1.1 springmvc配置

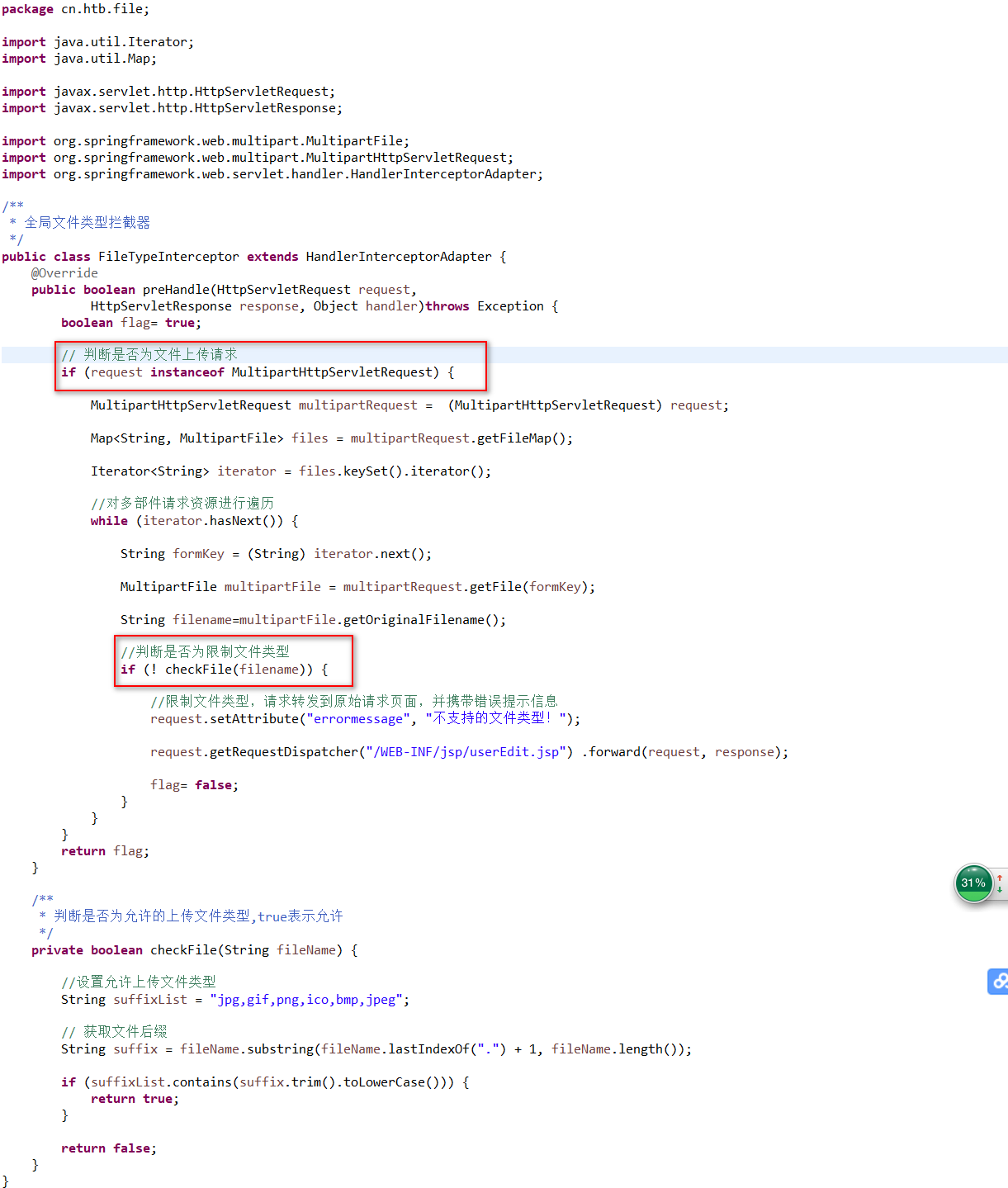


### 8.1.2 controller代码



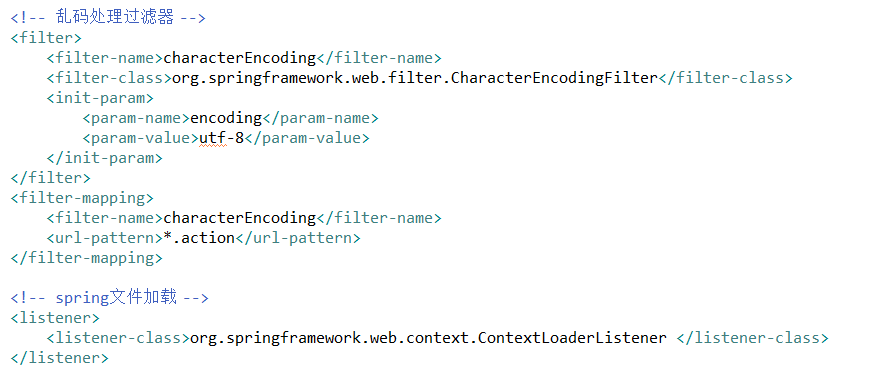
### 8.1.3 文件上传类型限制

springmvc中没有类型的限制，可以通过拦截器进行限制。

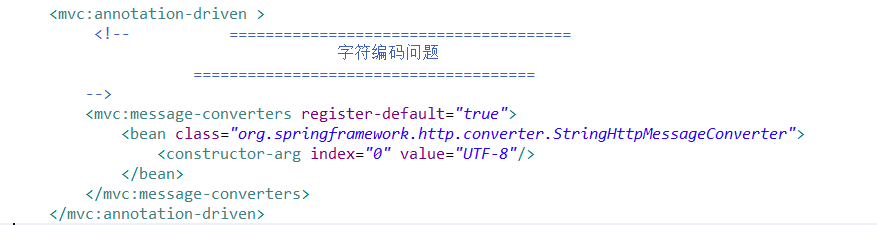


# 9 乱码处理

方法一：web.xml

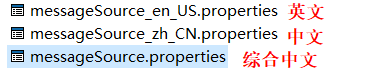


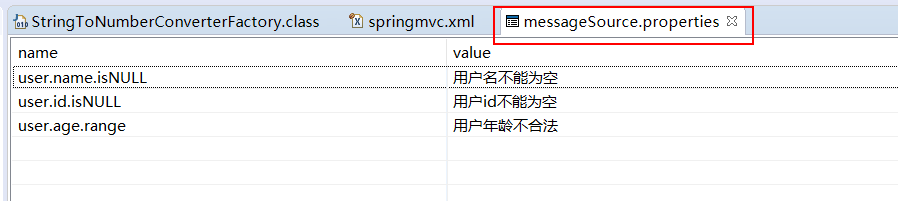
方法二：springmvc.xml

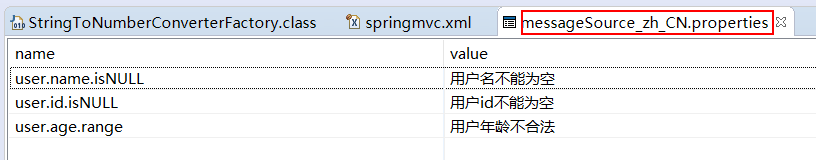


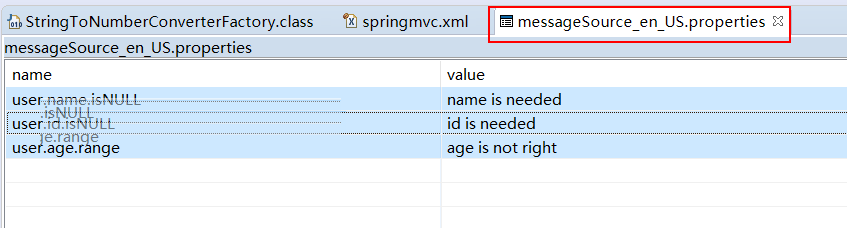
# 10 国际化（结合数据验证示例）

## 10.1国际化文件准备









## 10.2springmvc配置



页面

服务

视图展示

SpringMVC

拦截请求

数据校验

类型转换

数据处理

数据交给服务

数据封装

异常处理

返回数据

抛出异常

ViewResovler

ViewAndModel

请求（数据）

# 11转发和重定向

## 11.1 不同系统：重定向到其他系统的页面的两种方式

//测试重定向到另外的一个系统  
@RequestMapping("/tttt")

**//@ResponseBody**  
public void testRed(HttpServletResponse response) throws Exception{  
System.out.println("tttt");  
response.sendRedirect("https://www.baidu.com");  
}  
  
@RequestMapping("/ssss")  
public String testRedSSS(HttpServletResponse response) throws Exception{  
System.out.println("ssss");  
return "redirect:http://www.qq.com";  
}

## 11.2同一系统：从一个controller跳到另一个controller

### 第一种情况，不带参数跳转：

方法一：使用ModelAndView

　　　　return new ModelAndView("redirect:/toList");

 　　　  这样可以重定向到toList这个方法

方法二：在return后直接，redirect 加上要跳转的地址，即可以从第一个controller跳到第二个controller，如下图代码中方法一

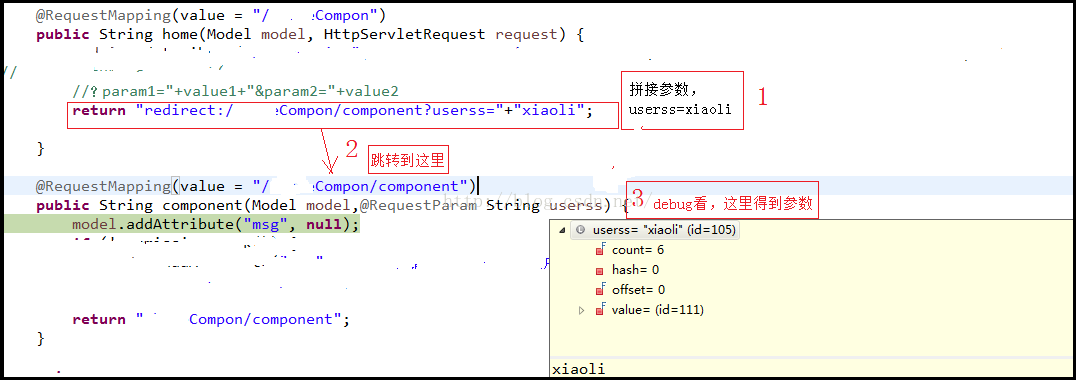
方法三：见蓝色框，只要在return后直接加想要跳到的controller的方法名即可，注意，这个方法名不是RequestMapping里影射的路径，是controller里具体的方法，

如图片中的3和4，走完3后，他会找到4而不是2（2是RequestMapping里映射的路径），这个像不像[Java](http://lib.csdn.net/base/17)方法的重载，如下图代码中方法二



### 第二种情况，带参数跳转

方法一：直接在后面用？拼接如图。



方法二：用RedirectAttributes，这个是发现的一个比较好用的一个类这里用它的addAttribute方法，这个实际上重定向过去以后你看url，是它自动给你拼了你的url。

使用方法：attr.addAttribute("param", value);

return "redirect:/namespace/toController";

这样在toController这个方法中就可以通过获得参数的方式获得这个参数，再传递到页面。过去的url还是和方式一一样的。

方法三：带参数不拼接url页面也能拿到值（重点是这个）

@RequestMapping("/save")

public String save(@ModelAttribute("form") Bean form,RedirectAttributes attr)

throws Exception {

String code = service.save(form);

if(code.equals("000")){

attr.addFlashAttribute("name", form.getName());

attr.addFlashAttribute("success", "添加成功!");

return "redirect:/index";

}else{

attr.addAttribute("projectName", form.getProjectName());

attr.addAttribute("enviroment", form.getEnviroment());

attr.addFlashAttribute("msg", "添加出错!错误码为："+rsp.getCode().getCode()+",错误为："+rsp.getCode().getName());

return "redirect:/maintenance/toAddConfigCenter";

}

}

　　@RequestMapping("/index")

　　public String save(@ModelAttribute("form") Bean form,RedirectAttributes attr) throws Exception {

　　　　　　return "redirect:/main/list";

　　}

//页面取值，直接用el表达式就能获得到，这里的原理是放到session中，session在跳到页面后马上移除对象。所以你刷新一下后这个值就会丢掉。

//3. 最底层还是两种跳转，只是spring又进行了封装而已，所以说跳转的方式其实有很多很多种，你自己也可以封一个，也可以用最原始的response来，也没有问题。

## 11.3 扩展阅读

### 转发和重定向区别

servlet中转发request.getRequestDispatcher().forward()和重定向response.sendRedirect()的区别：

　　①、转发是一次请求，一次响应，而重定向是两次请求，两次响应

　　②、转发：servlet和jsp共享一个request，重定向：两次请求request独立，所以前面request里面setAttribute()的任何东西，在后面的request里面都获取不到

③、转发：地址栏不会改变，重定向：地址栏发生变化。

### 两中重定向方法

|  |
| --- |
| import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpSession;  import org.springframework.stereotype.Controller;  import org.springframework.ui.Model;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestParam;  import org.springframework.web.servlet.mvc.support.RedirectAttributes;  @Controller  public class HelloController {    /\*\*  \* 转发形式  \* @param name  \* @param model  \* @return  \*/  @RequestMapping("/helloForward")  public String helloForward(@RequestParam(value="name", required=false, defaultValue="World2017") String name, Model model) {  model.addAttribute("name", name);  return "hello";  }        @RequestMapping("/hello")  public String hello() {  return "hello";  }    /\*\*  \* 使用RedirectAttributes类  \* @param name  \* @param redirectAttributes  \* @return  \*/  @RequestMapping("/helloRedirect")  public String helloRedirect(@RequestParam(value="name", required=false ) String name, RedirectAttributes redirectAttributes) {    redirectAttributes.addFlashAttribute("name", name);    return "redirect:/hello";  }      @RequestMapping("/hello2")  public String hello2(Model model,HttpServletRequest request) {    HttpSession session = request.getSession();  model.addAttribute("name",session.getAttribute("name"));  return "hello";  }    /\*\*  \* 常规做法，重定向之前把参数放进Session中，在重定向之后的controller中把参数从Session中取出并放进ModelAndView  \* @param name  \* @param request  \* @return  \*/  @RequestMapping("/helloRedirect2")  public String helloRedirect2(@RequestParam(value="name", required=false ) String name, HttpServletRequest request) {    request.getSession().setAttribute("name", name);    return "redirect:/hello2";  }      } |

在使用redirect进行重定向时请求的URL链接发生了改变，并且在controller中如果像reward一样 model.addAttribute("name", name)放置的参数，重定向之后是无法获取到的，所以重定向需要另外的方式进行参数的传递，上面的程序介绍了两种重定向传参的方式：

　　①、重定向之前把参数放进Session中，在重定向之后的controller中把参数从Session中取出并放进ModelAndView

　　②、使用RedirectAttributes类，这种实现方式比较简单。

# 12解决请求为\*.html无法解析JSON的问题

在springmvc中请求\*.html不可以返回json数据。

修改web.xml，添加url拦截格式。

