# SpringMVC

## 注解

### @component

@component是spring中的一个注解,它的作用就是**实现bean的注入（把普通pojo实例化到spring容器中）**

### @Autowired

表示一个属性是否需要进行依赖注入，可以使用在属性、普通方法上、构造方法上。注解中的required属性默认是true，如果没有对象可以注入到属性，则会报出异常；

@Autowired加在某个属性上，spring会从ioc容器中找到bean对象注入到属性上，如果找到多个该类型的Bean对象，则再根据属性的名字从多个Bean对象中确认一个；

@Autowired写在set()方法上，在spring会根据方法的参数类型从ioc容器中找到该类型的Bean对象注入到方法的行参中，并且自动反射调用该方法(被@Autowired修饰的方法一定会执行)，所以一般使用在set方法中、普通方法不用；

@Autowired使用在构造方法中：根据构造方法的形参、形参名，从ioc容器中找到该类型的Bean对象，注入到构造方法的形参中，并且执行该方法；

@Autowired注解在进行依赖注入的时候需要指定bean的时候，和@Qualifier注解一起使用使用@qualifier注解指定名称

### @RequestMapping

@RequestMapping注解是用于绑定用户的url请求地址的（为了映射请求路径）

* method:指定请求的method类型，包括GET、POST、PUT、DELETE等

@RequestMapping(value = "/index",method = RequestMethod.POST)

## 2.参数绑定

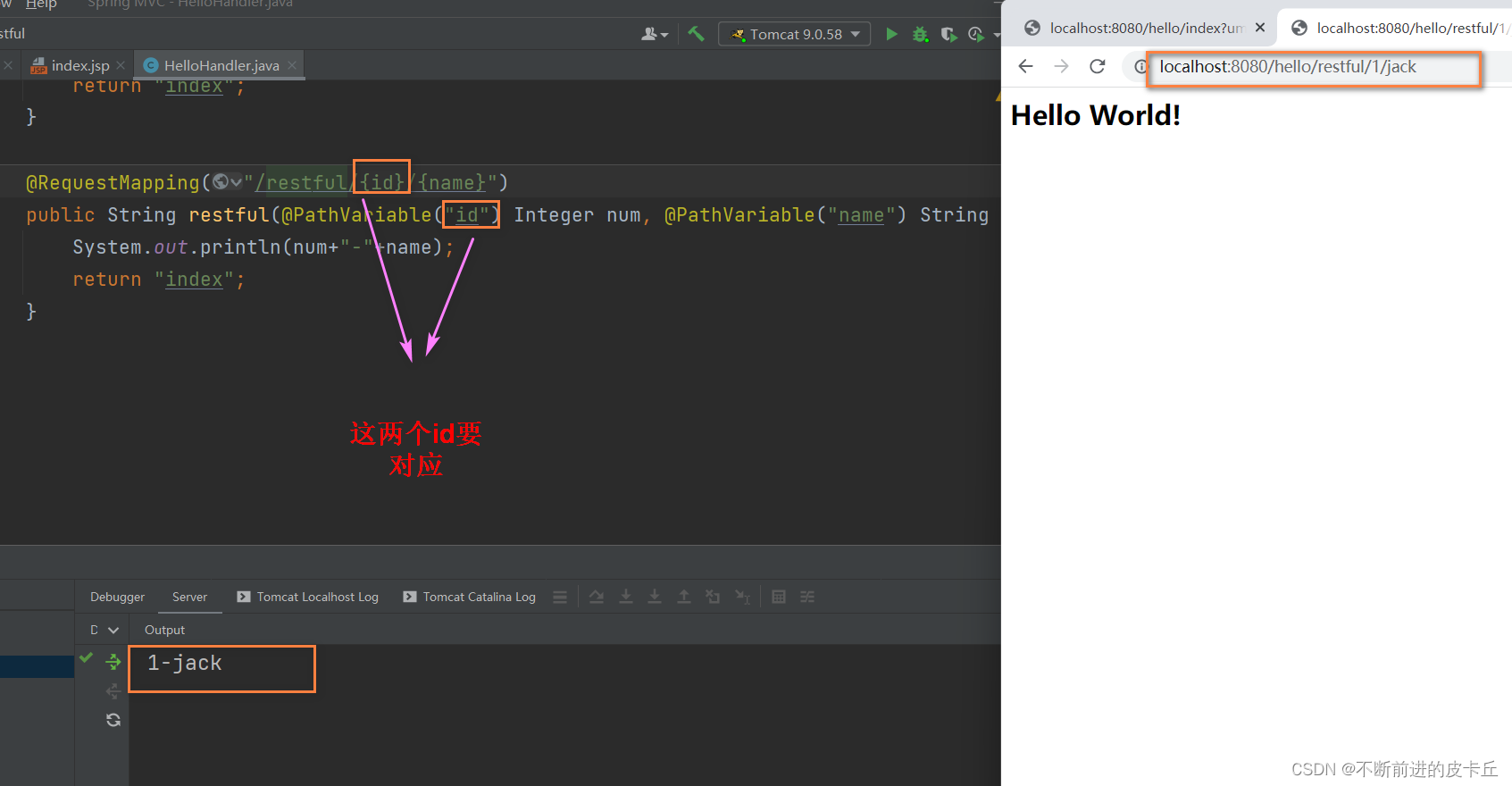
### 2.1 @RequestParam

params:指定request请求中必须包含的参数值，如果不包含的话，就无法调用该方法

1. 给参数列表添加@RequestParam注解进行绑定



### 2.2 [RESTful风格](https://so.csdn.net/so/search?q=RESTful%E9%A3%8E%E6%A0%BC&spm=1001.2101.3001.7020)的URL参数获取



@RequestMapping("/restful/{id}/{name}")

public String restful(@PathVariable("id") Integer num, @PathVariable("name") String name){

System.out.println(num+"-"+name);

return "index";

}

### 2.3映射Cookie

@RequestMapping("/cookie")

public String getCookie(**@CookieValue("JSESSIONID")** String sessionId){

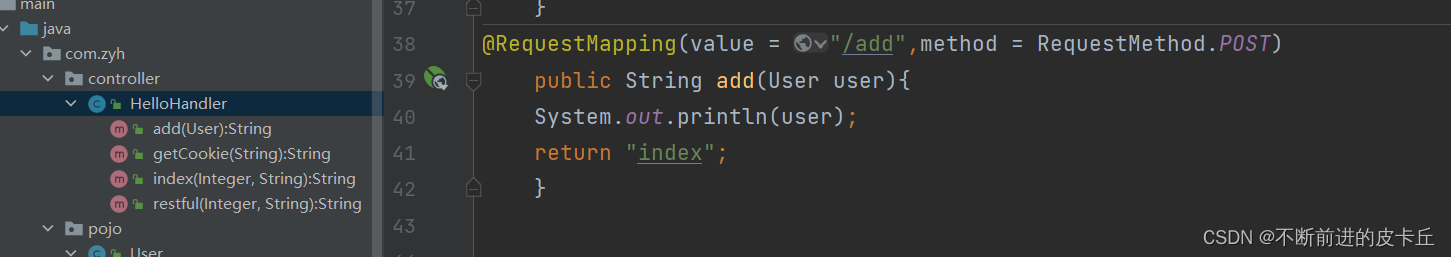
System.out.println(sessionId);

return "index";

}

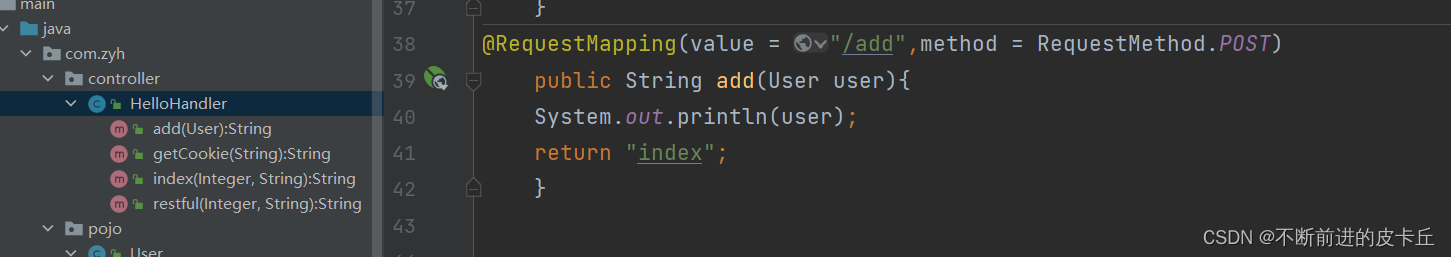
### 2.4使用POJO绑定参数

Spring MVC会根据请求参数名和POJO属性名进行匹配，自动为该对象填充属性值，并且支持属性级联  
首先创建实体类



### 2.5 JSP页面的转发和重定向

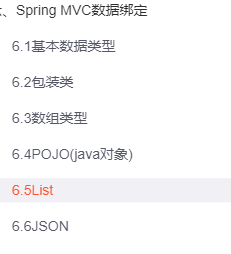
Spring MVC**默认**是通过**转发**的形式响应JSP，可以手动进行修改



比如，我们想把它改成**重定向**的话



## 3.SpringMVC数据绑定



[(190条消息) Spring MVC详解(学习总结)\_springmvc\_不断前进的皮卡丘的博客-CSDN博客](https://blog.csdn.net/qq_52797170/article/details/125591705?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522167868893316800211541002%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=167868893316800211541002&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~blog~top_positive~default-1-125591705-null-null.blog_rank_default&utm_term=springMVC&spm=1018.2226.3001.4450)

#### 3.6JSON

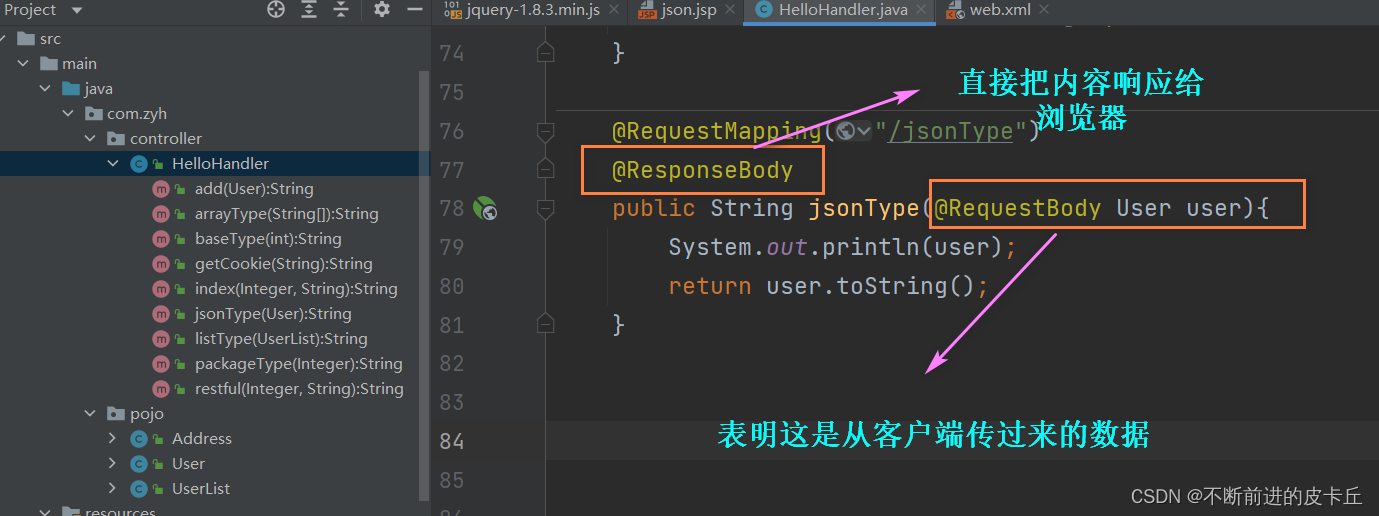
JSON数据必须用JSON.stringfy()方法转换成字符串

contentType:"application/json;charset=UTF-8"不能省略

##### @RequestBody注解

**读取HTTP请求参数，通过Spring MVC提供的HttpMessageConverter接口把读取的参数转换为JSON、XML格式的数据，绑定到业务方法的形参**

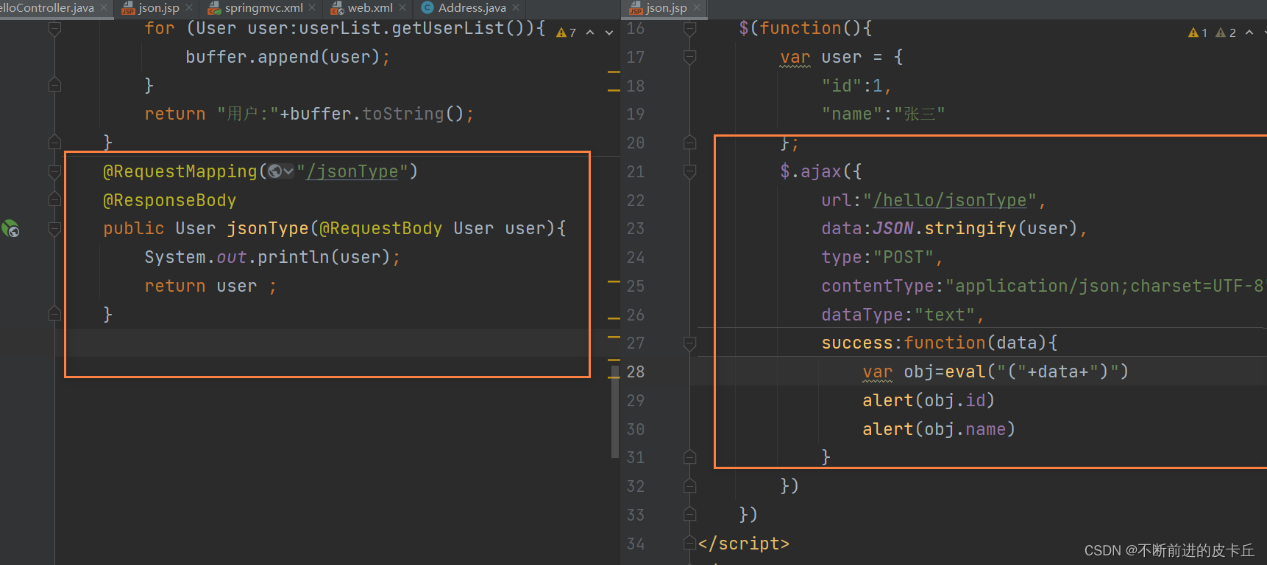
需要使用组件结**合@RequestBody注解把JSON转为JavaBean**，这里使用FastJson，其优势是如果属性为空，就不会将其转为JSON



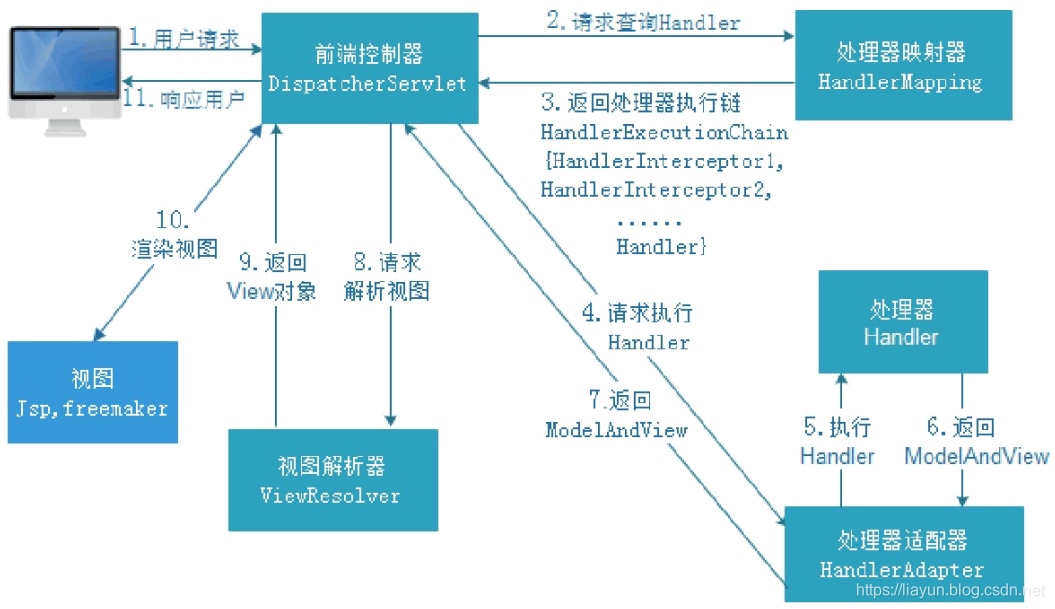
因为我们现在传的参数是json格式的，json格式就需要我们在后端把json格式解析成Java对象，这里我们仅仅加上@RequestBody注解是不够的，我们需要借助第三方工具把json解析成Java对象，这里用到的工具是**fastjson,**所以我们要在pom.xml中，把相关依赖导入进来

导入进来后，还需要在springmvc的配置文件中进行配置





## 4.SpringMVC组件和框架结构



下面我会对SpringMVC的框架结构（也即SpringMVC架构流程）作更加细致入微的讲解。

1. 用户发送请求至前端控制器（即DispatcherServlet）；
2. DispatcherServlet收到请求后调用处理器映射器（即HandlerMapping）；
3. 处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器（如果有则生成）并一起返回给DispatcherServlet；
4. DispatcherServlet通过处理器适配器（HandlerAdapter）调用处理器；
5. 执行处理器（Controller，也叫后端控制器）；
6. Controller执行完成后返回ModelAndView。
7. HandlerAdapter将Controller执行结果（即ModelAndView）返回给DispatcherServlet；
8. DispatcherServlet将ModelAndView传给ViewReslover视图解析器；
9. ViewReslover解析后返回具体View；
10. DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）；
11. 最后，DispatcherServlet响应给用户。

谁来处理业务请求呢？Handler处理器是来真正处理业务请求的，那么问题来了，前端控制器如何找到这个Handler处理器呢？处理器映射器记录的就是请求的url和处理的方法之间的映射关系，这个映射关系是怎么建立起来的呢？就是通过**@RequestMapping**这个注解建立起来的，这个映射关系就相当于一个Map集合（key-value这种形式），key就是请求的url，value就是处理的Handler。现在，前端控制器拿到这个请求之后，要找到对应的Handler



## 5.SpringMVC框架搭建

**（1）在web.xml文件中进行配置，在配置中设置springmvc-context.xml的路径和DispatcherServlet**

<servlet>

<servlet-name>appServlet</servlet-name>

<servlet-class>org.springframework.web.servlet.**DispatcherServlet**</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>**classpath:META-INF/spring/springmvc-context.xml**</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>appServlet</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

**（2）配置springmvc-context.xml文件，这一部分主要是开启注解功能、配置试图解析器，代码如下：**

<?xml version="1.0" encoding="UTF-8"?>

<beans:beans xmlns:beans="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" xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd ">

<mvc:annotation-driven />

**<!-- ①：对web包中的所有类进行扫描，以完成Bean创建和自动依赖注入的功能 -->**

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

**<!-- 这两个类用来启动基于Spring MVC的注解功能，将控制器与方法映射加入到容器中 -->**

<beans:bean

class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping" />

<beans:bean

class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" />

**<!-- 这个类用于Spring MVC视图解析 -->**

<beans:bean id="viewResolver"

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

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

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

</beans:bean>

</beans:beans>

**（3）配置文件完成了，下面开始写代码**两个jsp界面：

create.jsp **：创建用户界面 用户基本信息input 还有个submit提交用户信息**

detail.jsp：提交成功页面

UserController.java

@Controller

public class UserController {

@RequestMapping("")

public String Create(Model model) {

return "create";

}

@RequestMapping("/save")

public String Save(@ModelAttribute("form") User user, Model model) { // user:视图层传给控制层的表单对象；model：控制层返回给视图层的对象

model.addAttribute("user", user);

return "detail";

}

}

User.java:user实体类



## 6. Spring MVC和RESTful的集成

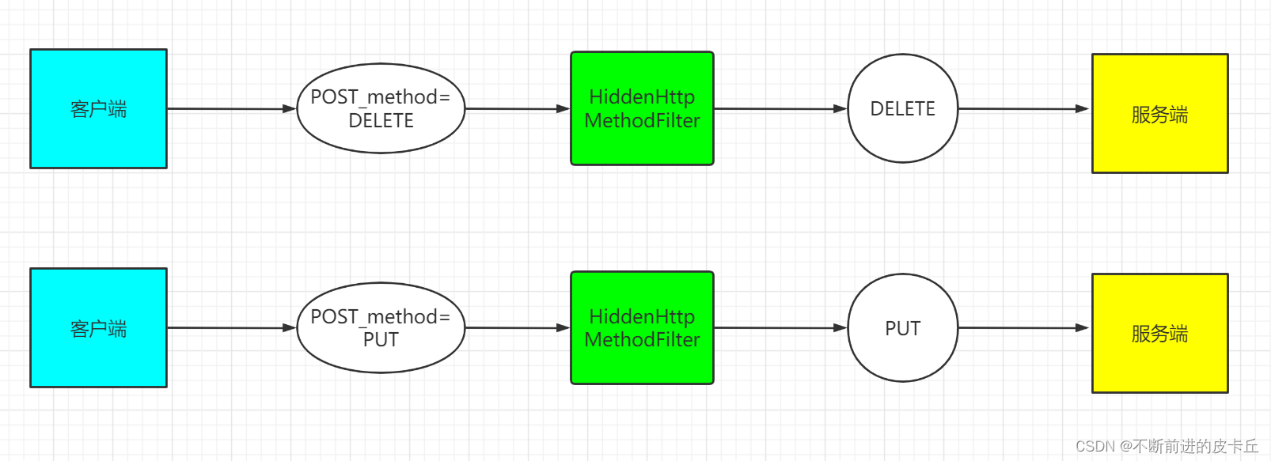
传统的URL：localhost:8080/hello/index?id=1&name=tom

RESTful <URL:localhost:8080/hello/index/1/tom>

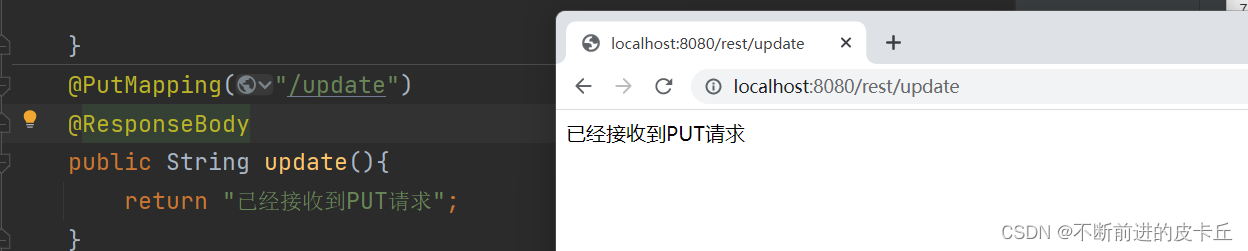
* GET用来获取资源 @GetMapping("/findAll")
* POST用来创建资源 @PostMapping("/save")
* PUT用来修改资源 @PutMapping("/update")
* DELETE用来删除资源 @DeleteMapping("/deleteById/{id}")

两个终端要完成数据交互，基于RESTful的方式，增删改查操作分别需要使用不同的HTTP请求类型来访问。

**1.传统的web开发中form表单只支持GET和POST请求，如何解决呢？我们可以通过添加HiddenHttpMethodFilter过滤器，可以把POST请求转为PUT或者DELETE**



**2.在web.xml中配置HiddenHttpMethodFilter**



<filter>

<filter-name>HiddenHttpMethodFilter</filter-name>

<filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>HiddenHttpMethodFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

Jsp中：

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

<html>

<head>

<title>Title</title>

</head>

<body>

<form action="**/rest/update**" method="post">

<input type="hidden" name="\_method" **value="PUT"**>

<input type="submit" value="提交">

</form>

</body>

</html>

## 7 Spring MVC视图层解析

调用Web资源给域对象传值

page

request

session

application

业务数据的绑定是指把业务数据绑定给JSP域对象，业务数据的绑定是由ViewResolver来完成的，开发时，我们先添加业务数据，再交给ViewResolver来绑定，我们重点是学习如何添加业务数据，**Spring MVC提供了下面几种方式来添加业务数据**：

Map

Model

ModelAndView

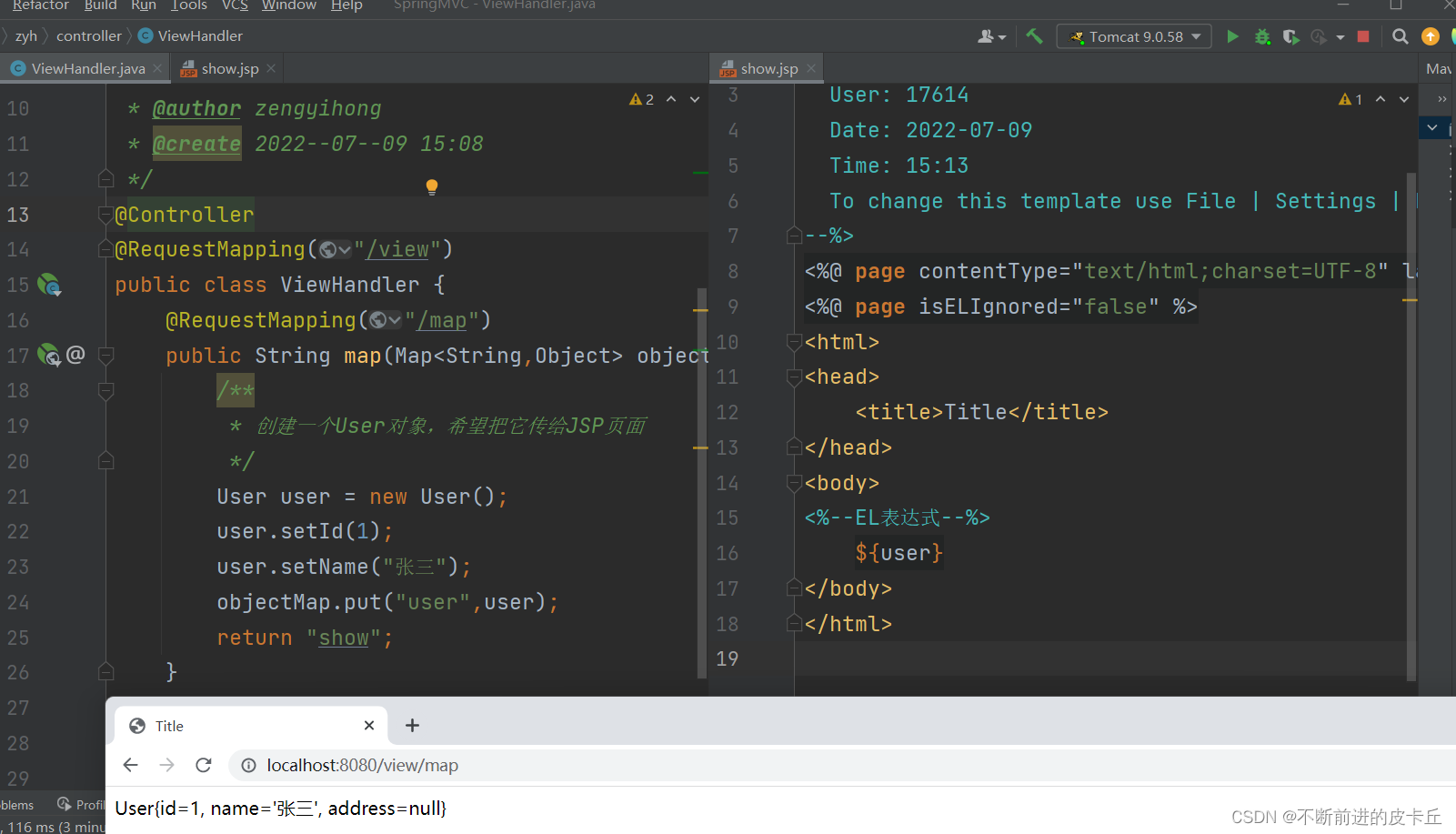
@SessionAttribue

@ModelAttribute

Servlet的API

### 7.1业务数据绑定到request域对象

#### 7.1.1Map



#### 7.1.2 Model

Model和Map类似，业务方法通过入参来完成业务数据的绑定



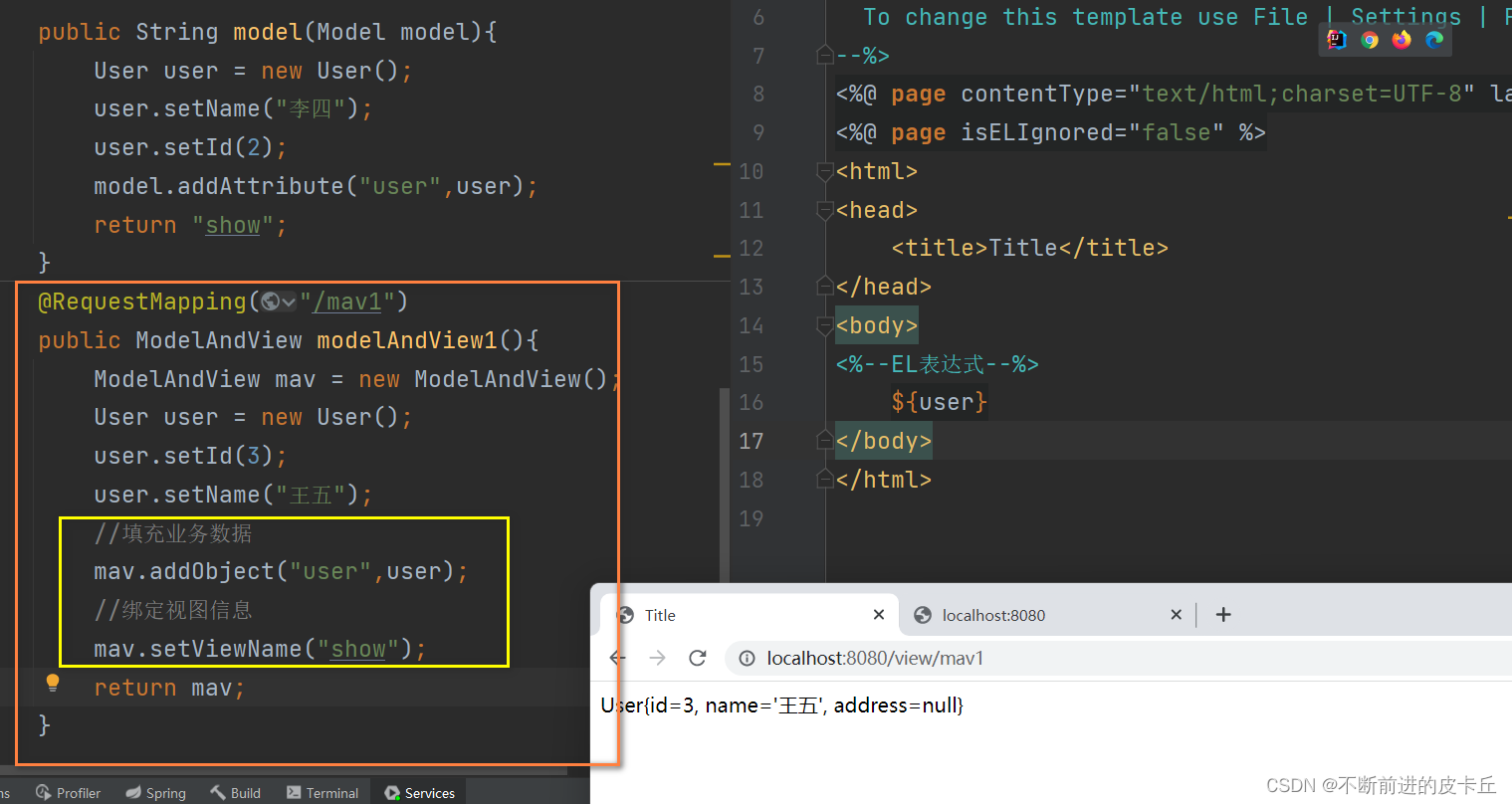
#### 7.1.3ModelAndView

和Map,Model不同的是，ModelAndView不仅包含业务数据，同时也封装了视图信息，如果使用ModelAndView来处理业务数据，业务方法的返回值必须是ModelAndView对象

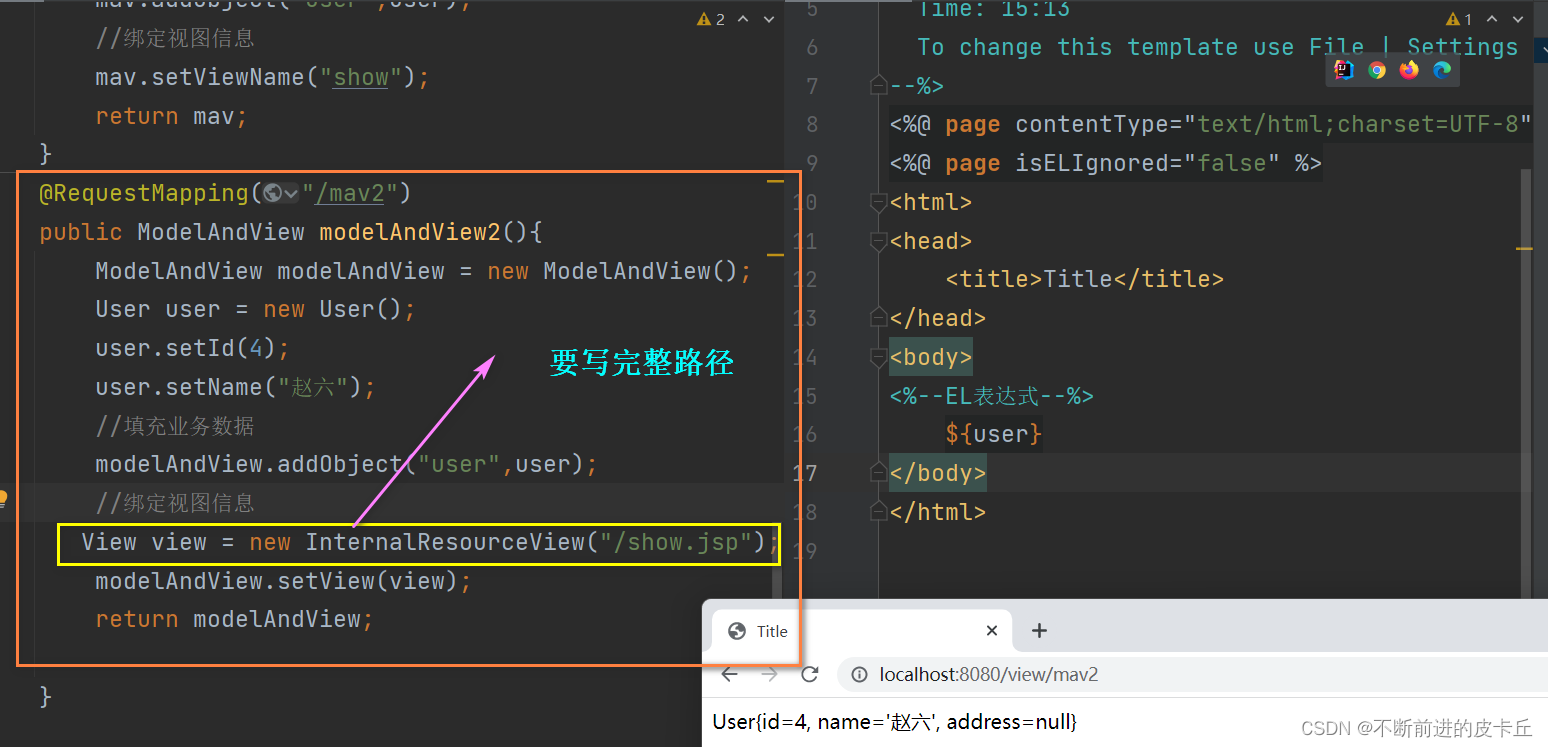
业务方法中对ModelAndView进行两个操作：

* 填充业务数据
* 绑定视图信息

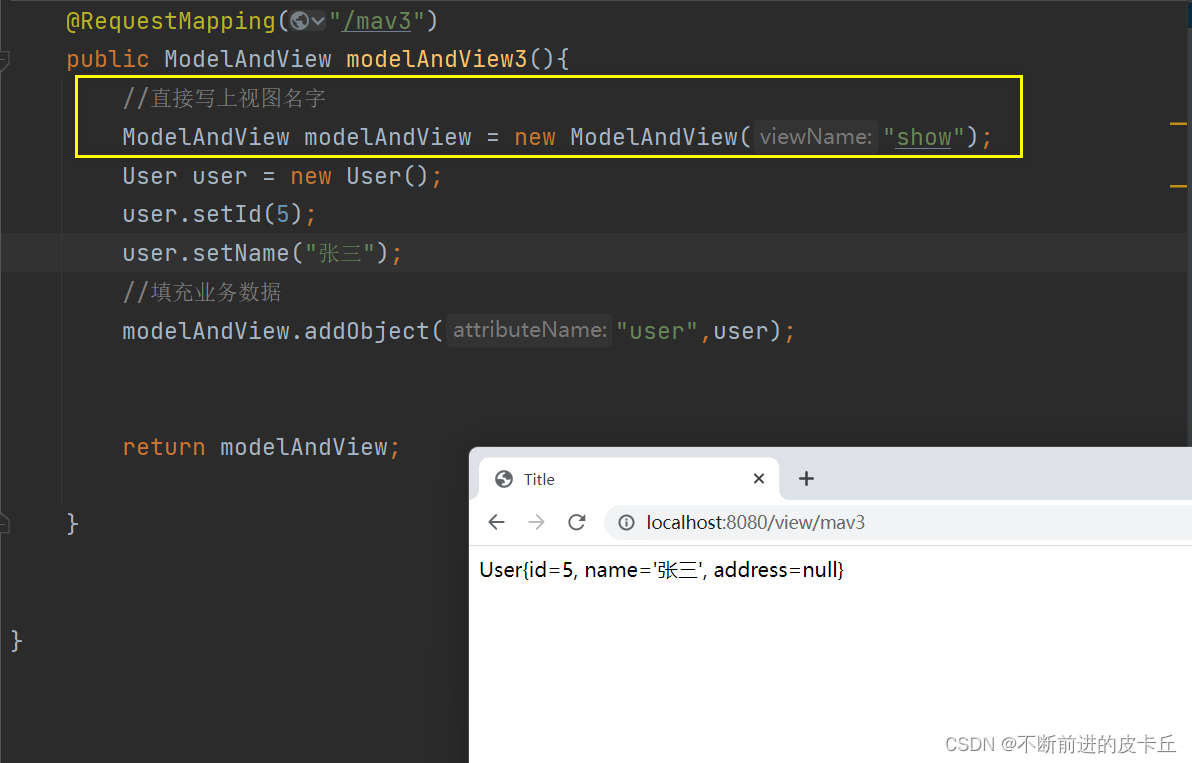
第一种方式



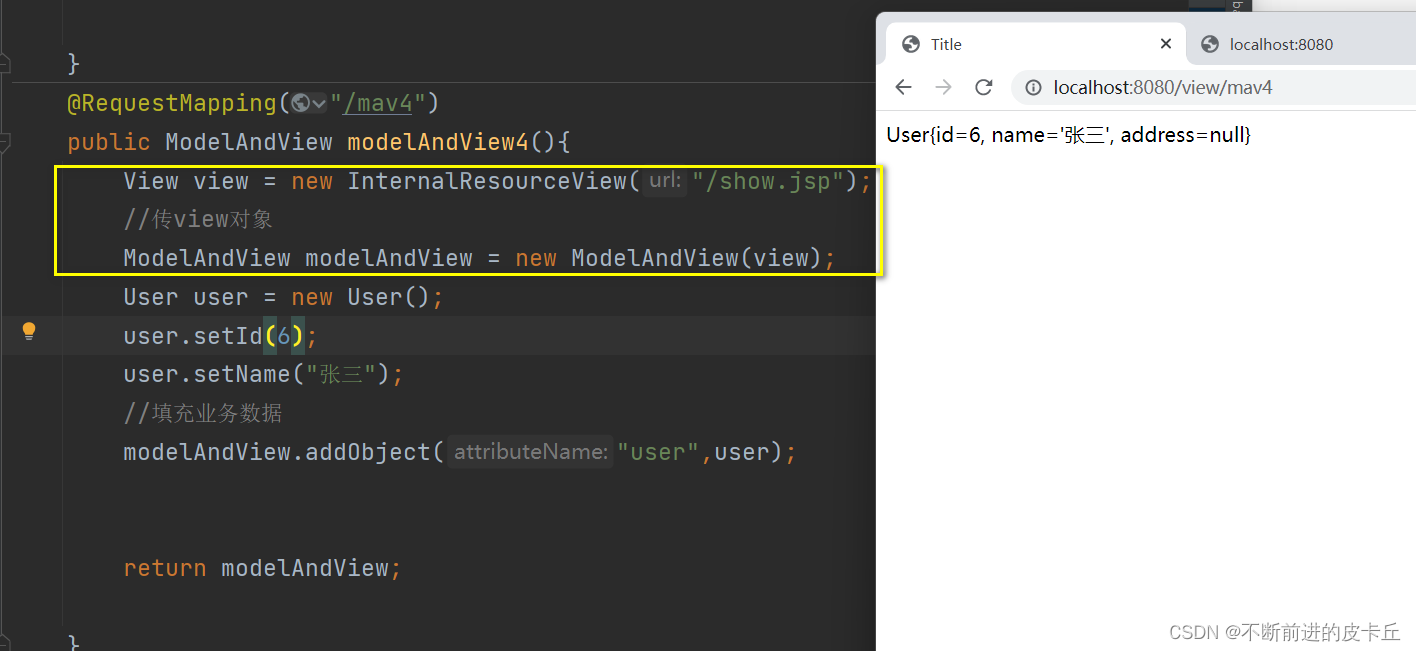
第二种方式



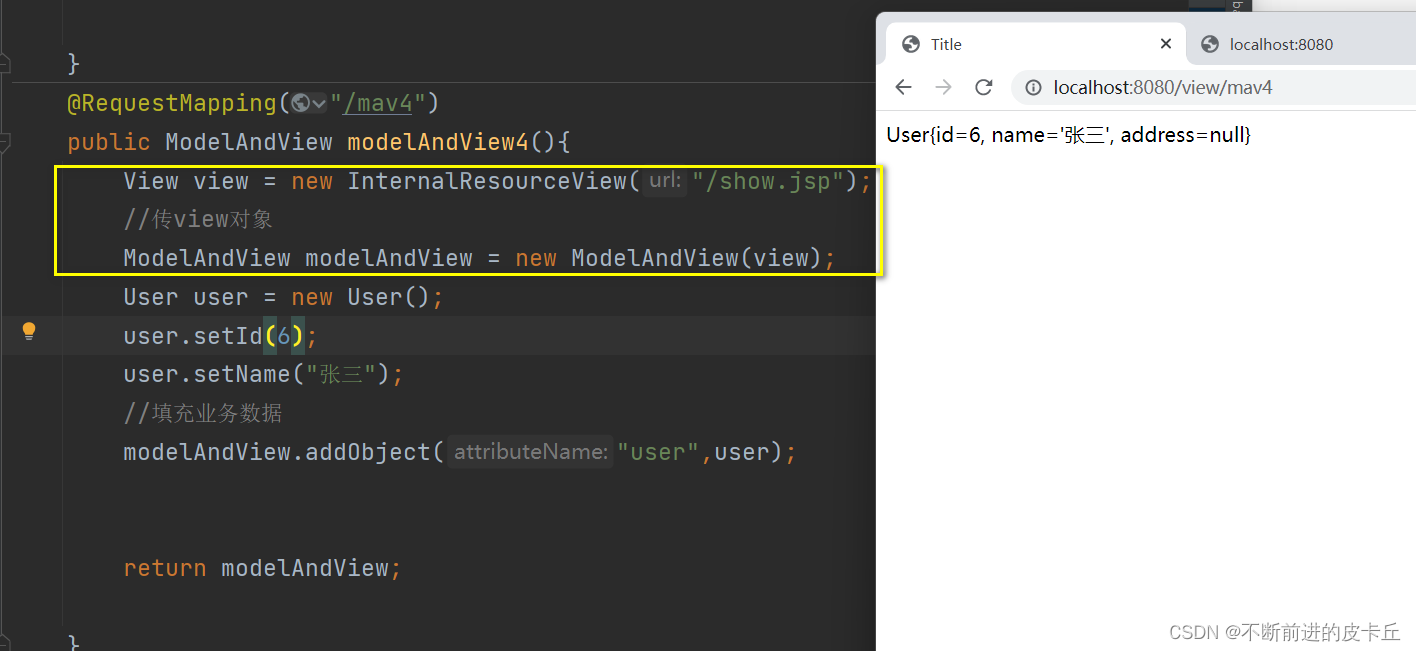
第三种方式



第四种方式



第五种方式



#### 7.1.4 Servlet的API

Spring MVC可以在业务方法种直接获取Servlet原生Web资源，只需要在方法定义时添加HttpServletRequest输入参数就可以，在方法体种直接使用request对象  
先在pom.xml导入相关依赖

<!--导入servlet API -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

</dependency>



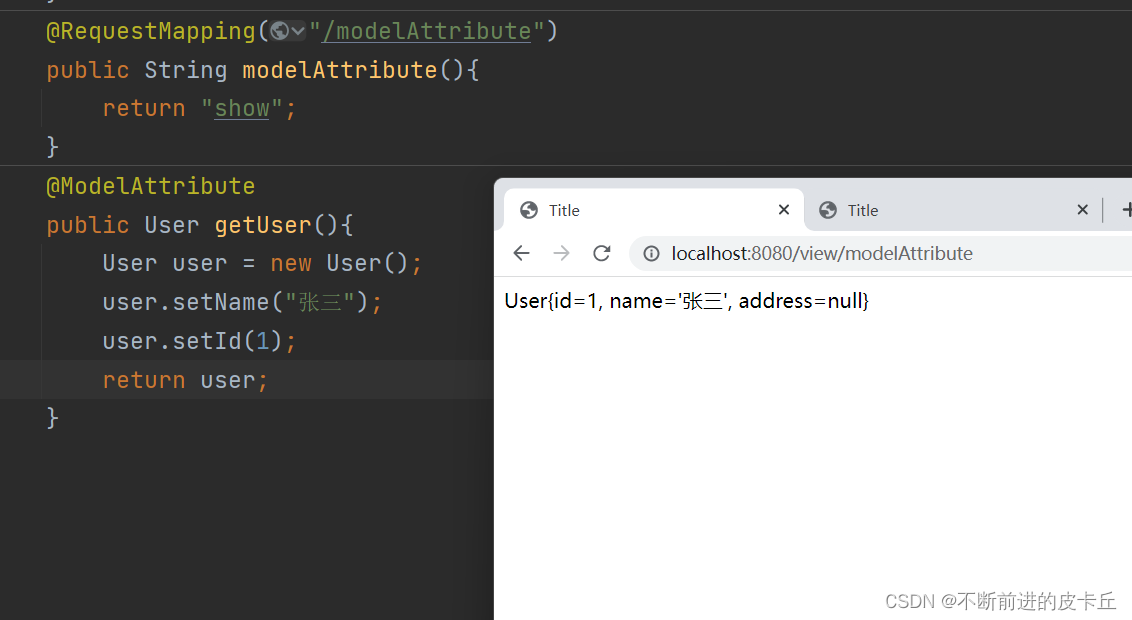
#### 7.1.5@ModelAttribute

Spring MVC还可以通过@ModelAttribute注解的方式添加业务数据，具体使用步骤如下：

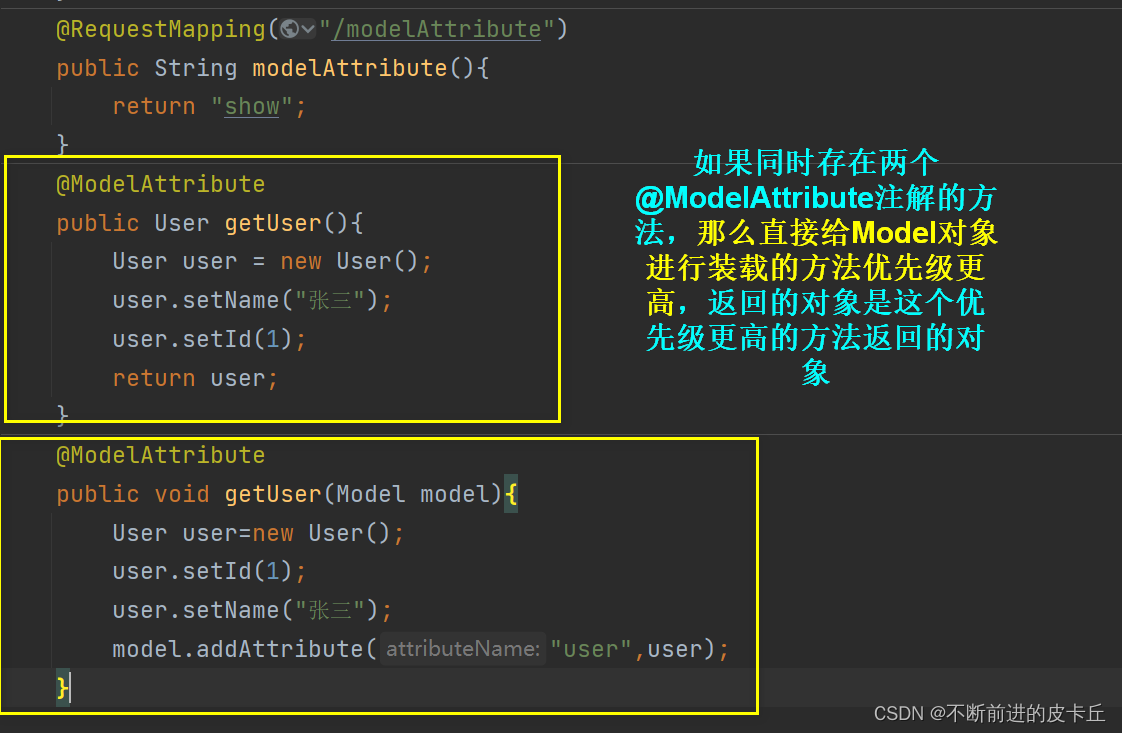
定义一个方法，这个方法用来填充到业务数据中的对象

给该方法添加@ModelAttribute注解，不是响应请求的业务方法

@ModelAttribute注解的作用，将**请求参数绑定到Model对象**。被@ModelAttribute注释的方法会在Controller每个方法执行前被执行（如果在一个Controller映射到多个URL时，要谨慎使用）。



@ModelAttribute的作用是当Handler接收到一个客户端请求以后，不管调用哪一个业务方法，都会优先调用被



### 7.2业务数据绑定到Session域对象

#### 7.2.1使用原生的Servlet API

****

#### 7.2.2@SessionAttribute

@SessionAttribute这个注解不是给方法添加的，**而是给类添加的**



**@SessionAttributes除了可以通过key值绑定，也可以通过业务数据的数据类型进行绑定**

@Controller

@SessionAttributes(type=User.class)

public class ViewHandler{

...

}

**@SessionAttributes可以同时绑定多个业务数据**

**@Controller**

**@SessionAttributes(value={"user","address"})**

**public class ViewHandler{**

**...**

**}**

**或**

**@Controller**

**@SessionAttributes(value={"user","address"})**

**public class ViewHandler{**

**...**

**}**

## 8.Spring MVC自定义数据转换器

**Spring MVC默认情况下可以对基本类型进行类型转换**，例如可以将String转换为Integer,Double,Float等。但是**Spring MVC并不能转换日期类型**（java.util.Date），如果希望把字符串参数转换为日期类型，必须**自定义类型转换器**

1.**创建DateConverter类**，并且实现org.springframework.core.convert.converter.Converter接口，这样它就成为了一个自定义数据类型转换器，需要**指定泛型<String,Date>**,表示把String类型转为Date类型

public class DateConverter implements **Converter<String, Date>** {

private String pattern;

public DateConverter(String pattern) {

this.pattern = pattern;

}

@Override

public Date convert(String s) {

SimpleDateFormat simpleDateFormat = new SimpleDateFormat(this.pattern);

try {

return simpleDateFormat.parse(s);

} catch (ParseException e | java.text.ParseException e) {

e.printStackTrace();

}

return null;

}

}

1. 在springmvc.xml中配置**conversionService bean**,这个bean是org.springframework.context.support.ConversionServiceFactoryBean的实例化对象，**同时bean中必须包含一个converters属性，在其中注册所有需要使用的自定义转换器**

<bean id="conversionService" class="org.springframework.context.support.ConversionServiceFactoryBean">

<property name="converters">

<list>

<bean class="com.zyh.converter.DateConverter">

<constructor-arg type="java.lang.String" value="yyyy-MM-dd"></constructor-arg>

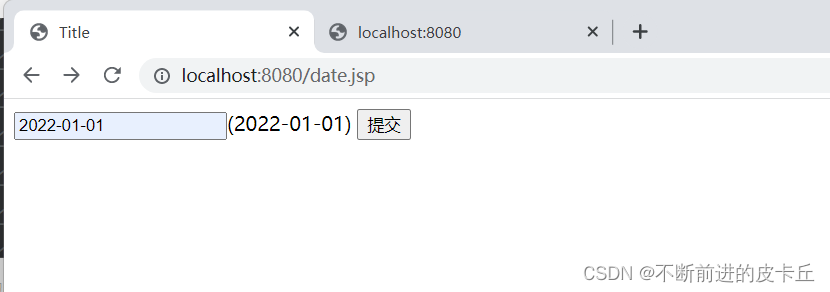
</bean>

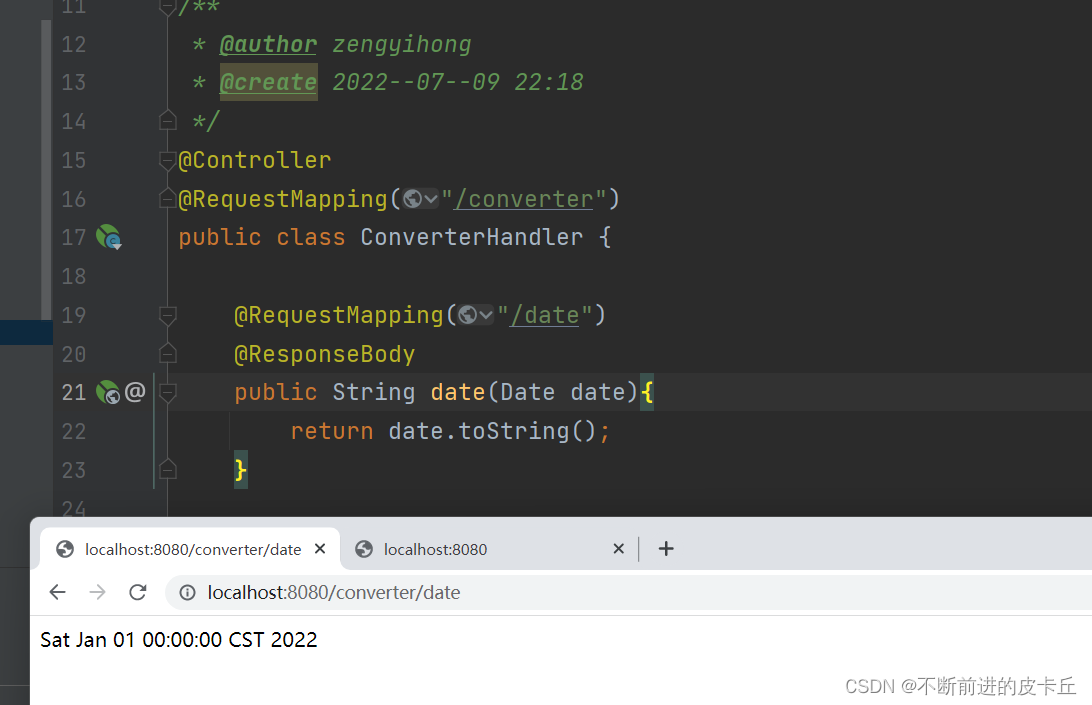
</list>

</property>

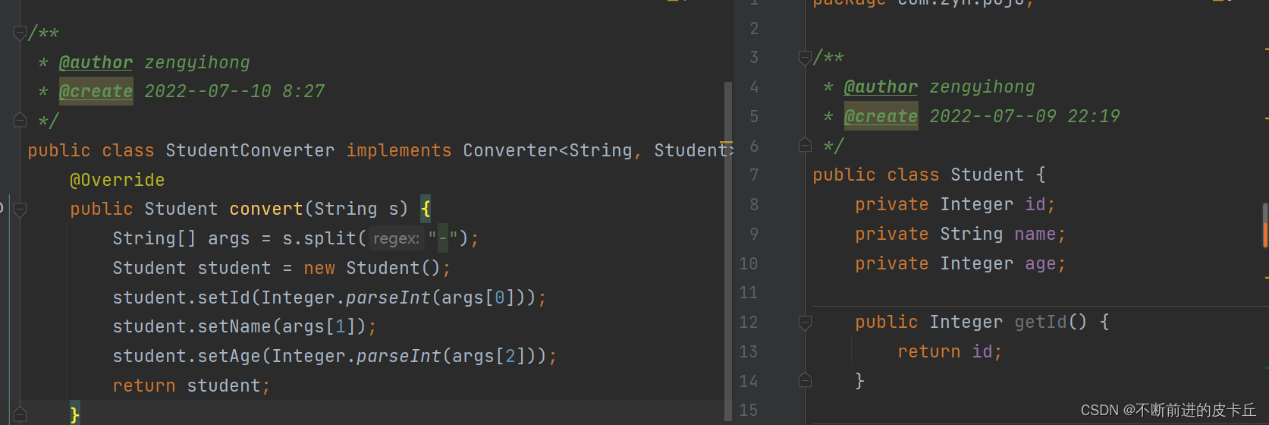
</bean>

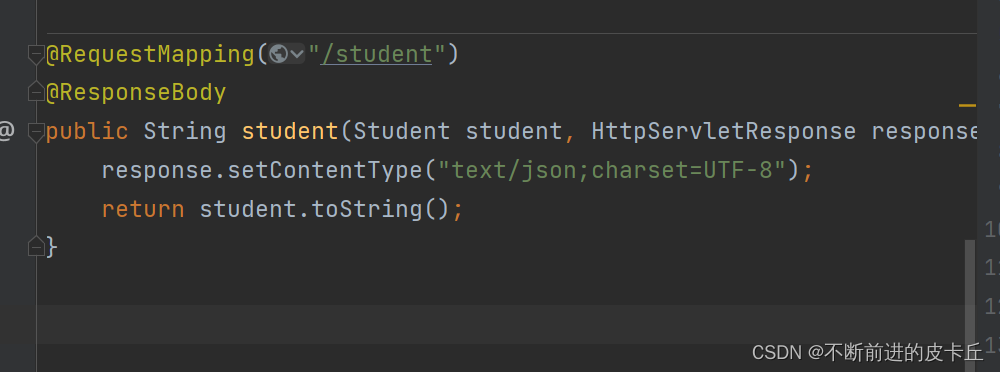
<mvc:annotation-driven conversion-service="conversionService"></mvc:annotation-driven>





我们也可以自定义类







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

<html>

<head>

<title>Title</title>

</head>

<body>

<form action="/converter/student" method="post">

<input type="text" name="student"/>(1-张三-22)<br/>

<input type="submit" value="提交"/>

</form>

</body>

</html>

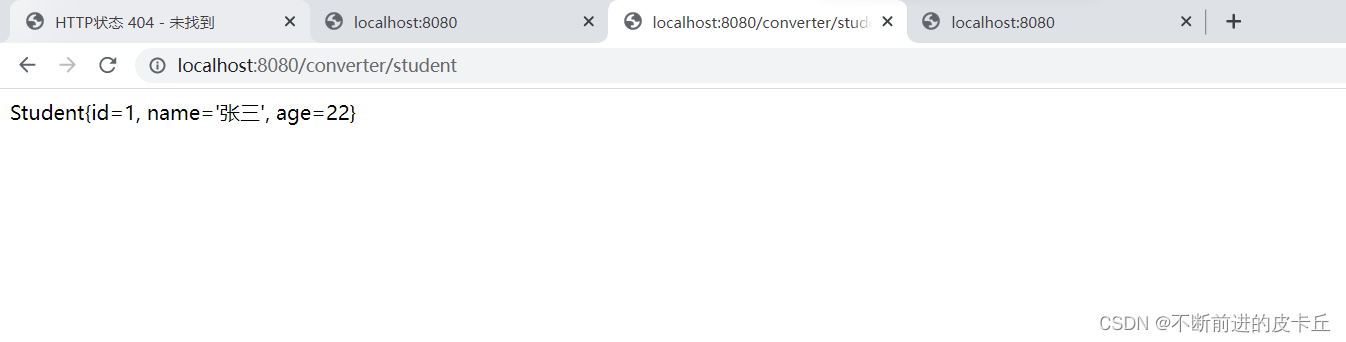
**怎么解决中文乱码问题**

<!-- 解决中文乱码问题 -->

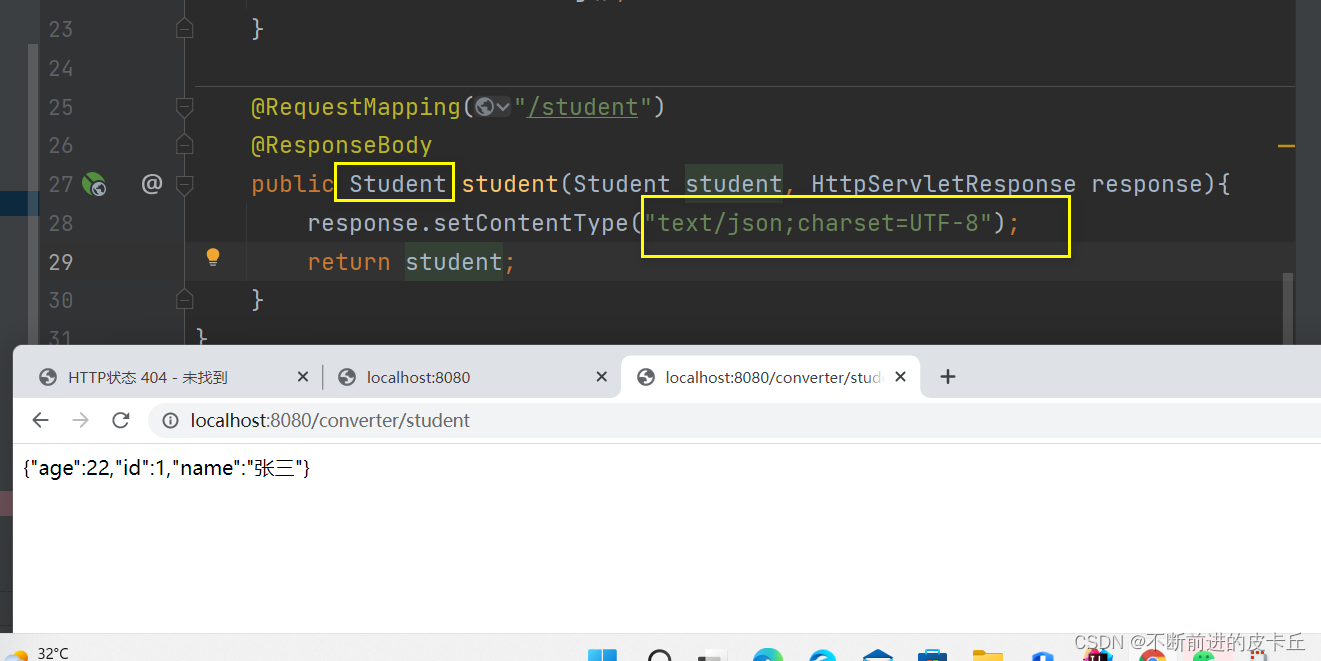
<bean class="org.springframework.http.converter.StringHttpMessageConverter">

<property name="supportedMediaTypes" value="text/html;charset=UTF-8"></property>

</bean>



如果我们想要在**浏览器显示的是JSON格式**，中文乱码需要在业务方法中通过设置response的编码方式来解决，springmvc.xml的bean不起作用，如果不需要把业务数据转换成json格式，springmvc.xml的配置可以起到中文乱码的作用



## 十、文件的上传下载

[(190条消息) Spring MVC详解(学习总结)\_springmvc\_不断前进的皮卡丘的博客-CSDN博客](https://blog.csdn.net/qq_52797170/article/details/125591705?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522167868893316800211541002%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=167868893316800211541002&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~blog~top_positive~default-1-125591705-null-null.blog_rank_default&utm_term=springMVC&spm=1018.2226.3001.4450)

## 十一、拦截器

### 11.1过滤器、监听器、拦截器的对比

* **Servlet**:处理Reequest请求和Response响应
* **过滤器(Filter)**:对Request请求起到**过滤**作用，作用在Servlet之前，**如果配置为/\*可以为所有的资源(servlet、js/css静态资源等)进行过滤处理**
* **监听器(Listener)**:实现了javax.servlet.ServletContextListener接口的服务器端组件，它随Web应用的启动而启动，**只初始化一次，然后一直监视，随Web应用的停止而销毁**

作用一：做初始化工作，web应用中spring容器启动ContextLoaderListener

作用二：监听web中的特定事件，比如HttpSession,ServletRequest的创建和销毁；变量的创建、销毁和修改等可以在某些动作 前后增加处理，实现监控，比如说统计在线人数，利用HttpSessionListener等

* **拦截器(Interceptor)**:是Spring MVC、Struts等表现层框架自己的，不会拦截jsp/html/css/image等的访问，**只会拦截访问的控制器方法(Handler)**

servlet、filter、listener是配置在web.xml中，interceptor是配置在表现层框架自己的配置文件中

在Handler业务逻辑执行之前拦截一次

在Handler逻辑执行完但是还没有跳转页面之前拦截一次

在跳转页面后拦截一次

### 11.2拦截器基本概念

**Spring MVC中的拦截器（Interceptor）类似于Servlet中的过滤器（Filter），它主要用于拦截用户请求并作相应的处理。例如通过拦截器可以进行权限验证、记录请求信息的日志、判断用户是否登录等。**

要使用Spring MVC中的拦截器，就需要对拦截器类进行定义和配置。通常拦截器类可以通过两种方式来定义。

**通过实现HandlerInterceptor接口**

public class MyInterceptor implements HandlerInterceptor {

/\*\*

\* 在目标Handler(方法)执行前执行

\* 返回true:执行Handler方法

\* 返回false:阻止目标Handler方法执行

\* @param request

\* @param response

\* @param handler

\* @return

\* @throws Exception

\*/

@Override

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

System.out.println("目标Handler执行前执行MyInterceptor---->preHandle方法...");

return true;

}

/\*\*

\* 在目标Handler(方法)执行后，视图生成前执行

\* @param request

\* @param response

\* @param handler

\* @param modelAndView

\* @throws Exception

\*/

@Override

public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {

System.out.println("目标Handler执行后，视图执行前执行MyInterceptor---->postHandle方法...");

}

/\*\*

\* 在目标方法执行后，视图生成后执行

\* @param request

\* @param response

\* @param handler

\* @param ex

\* @throws Exception

\*/

@Override

public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {

System.out.println("目标Handler执行后，视图执行后执行MyInterceptor---->afterCompletion方法...");

}

}

**拦截器配置1**

<mvc:interceptors>

**<!-- 拦截器配置 -->**

**<!--**

**使用bean定义一个Interceptor**

**直接定义在mvc:interceptors根下面的Interceptor将拦截所有的请求**

**-->**

<bean class="com.zyh.interceptor.MyInterceptor"></bean>

</mvc:interceptors>

**拦截器配置方式2**

<!-- 拦截器配置2 -->

<mvc:interceptors>

**<!--定义在mvc:interceptor下面,可以自定义需要拦截的请求**

**如果有多个拦截器满足拦截处理的要求，则依据配置的先后顺序来执行**

**-->**

<mvc:interceptor>

**<!--通过mvc:mapping配置需要拦截的资源。支持通配符，可以配置多个 -->**

<mvc:mapping path="/\*\*"/>

<!-- /\*\*表示拦截所有的请求-->

**<!--通过mvc:exclude-mapping配置不需要拦截的资源。支持通配符，可以配置多个 -->**

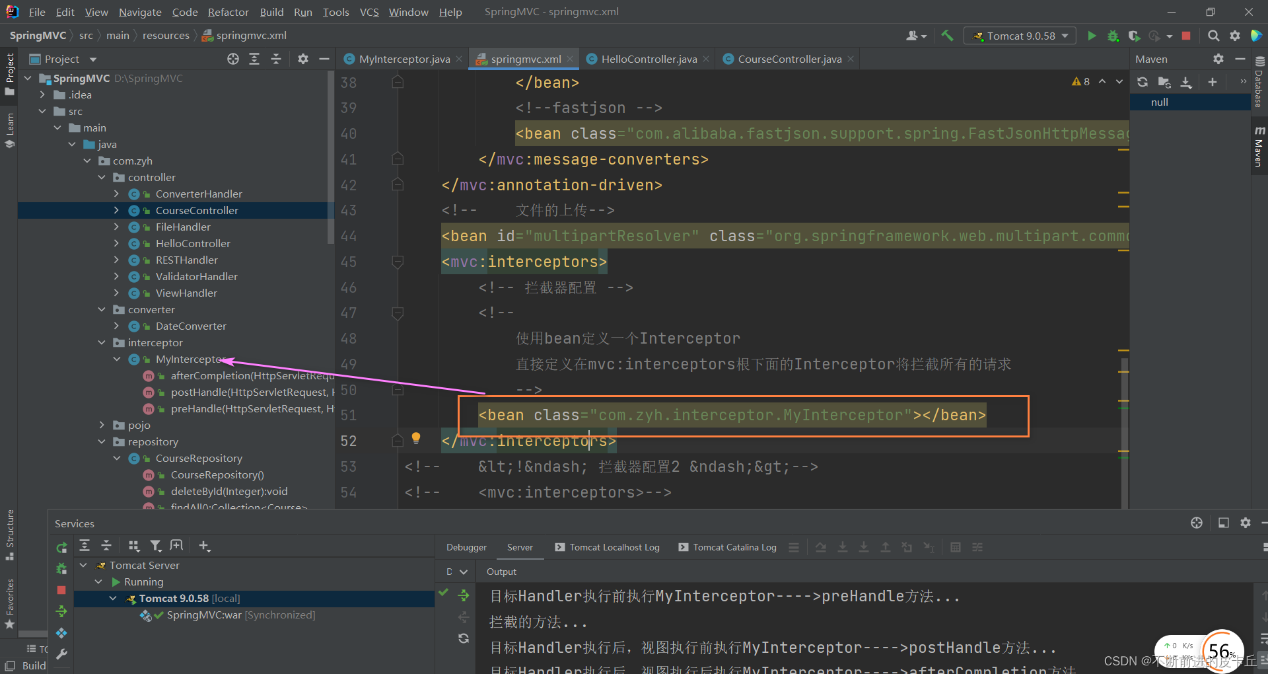
<mvc:exclude-mapping path="/hello/\*"/>

<!-- /hello/\*表示放行hello路径下的请求 -->

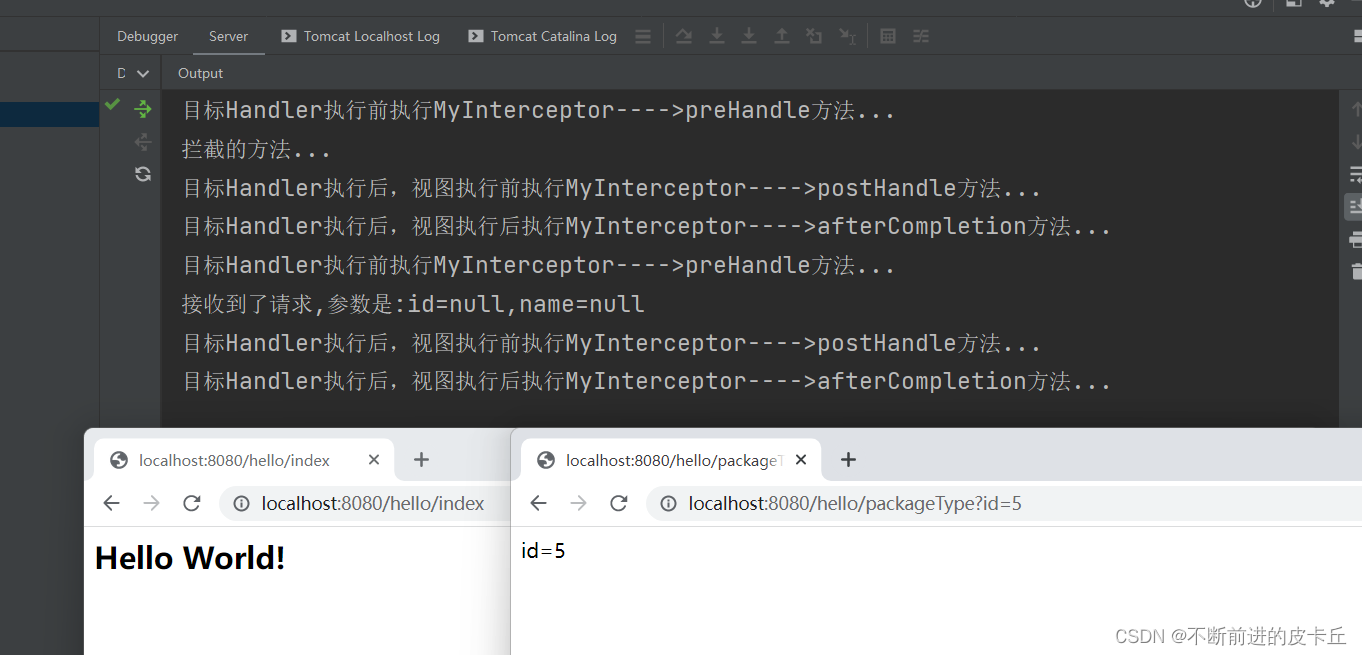
<bean class="com.zyh.interceptor.MyInterceptor"></bean>

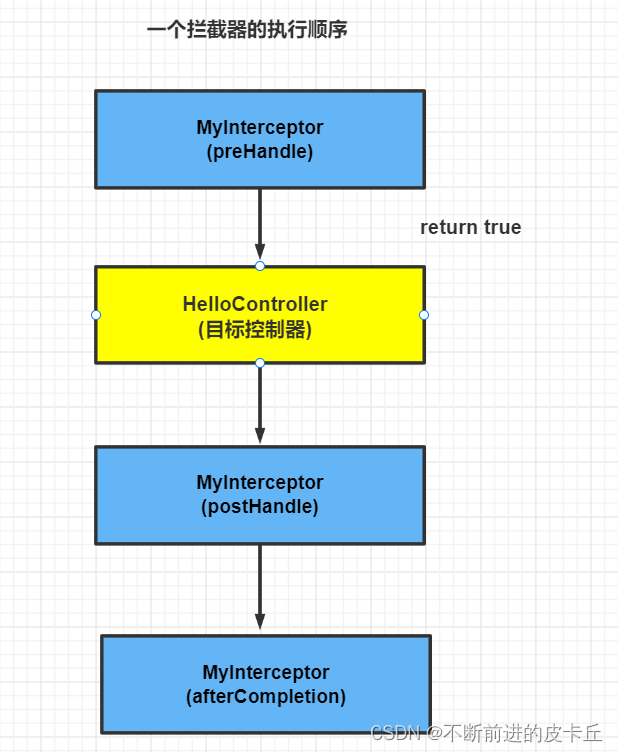
</mvc:interceptor>

</mvc:interceptors>









**继承HandlerInterceptor接口的实现类（如：HandlerInterceptorAdapter）来定义**。

### 11.4多个拦截器的实现

Spring MVC框架支持多个拦截器的配置，从而构成拦截器链，对客户端进行多次拦截操作

**过滤器配置**  
<mvc:interceptors>

<mvc:interceptor>

<mvc:mapping path="/\*\*"/>

<bean class="com.zyh.interceptor.MyInterceptor"></bean>

</mvc:interceptor>

<mvc:interceptor>

<mvc:mapping path="/\*\*"/>

<bean class="com.zyh.interceptor.MyInterceptor2"></bean>

</mvc:interceptor>

</mvc:interceptors>

**自定义第二个过滤器**

public class MyInterceptor2 implements HandlerInterceptor {

/\*\*

\* 在目标Handler(方法)执行前执行

\* 返回true:执行Handler方法

\* 返回false:阻止目标Handler方法执行

\* @param request

\* @param response

\* @param handler

\* @return

\* @throws Exception

\*/

@Override

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

System.out.println("2.目标Handler执行前执行MyInterceptor2---->preHandle方法...");

return true;

}

/\*\*

\* 在目标Handler(方法)执行后，视图生成前执行

\* @param request

\* @param response

\* @param handler

\* @param modelAndView

\* @throws Exception

\*/

@Override

public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {

System.out.println("3.目标Handler执行后，视图执行前执行MyInterceptor2---->postHandle方法...");

}

/\*\*

\* 在目标方法执行后，视图生成后执行

\* @param request

\* @param response

\* @param handler

\* @param ex

\* @throws Exception

\*/

@Override

public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {

System.out.println("5.目标Handler执行后，视图执行后执行MyInterceptor2---->afterCompletion方法...");

}

}

**Handler**

@RequestMapping("/hello")

@Controller

public class HelloContro

@RequestMapping("/packageType")

@ResponseBody

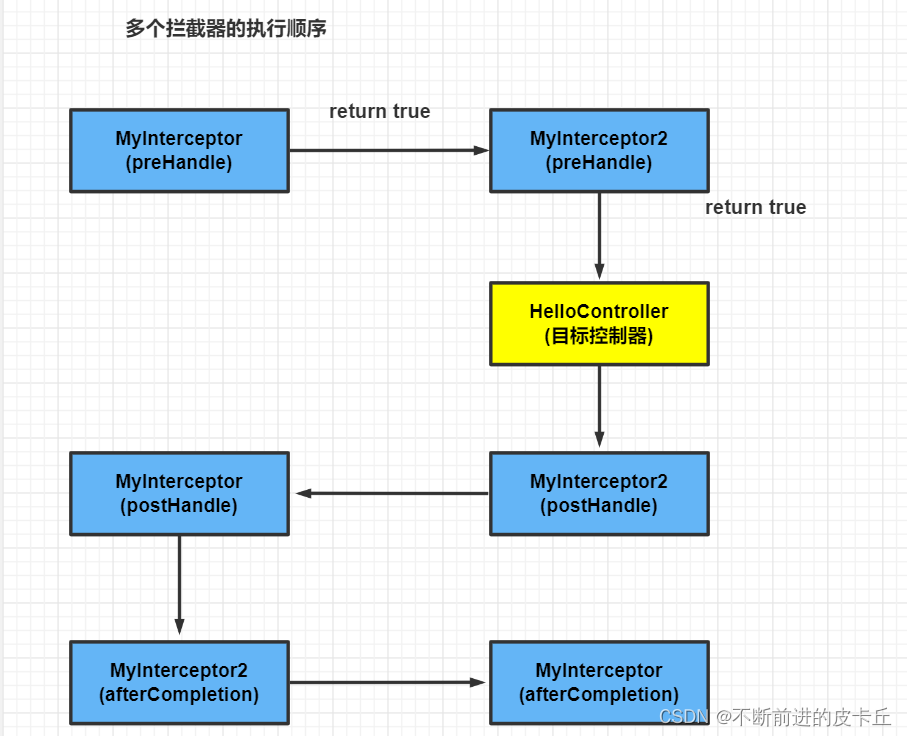
public String packageType(@RequestParam(value = "id", required = true) Integer id) {

System.out.println("拦截的方法...");

return "id=" + id;

}

}

****