**Springmvc第一天**

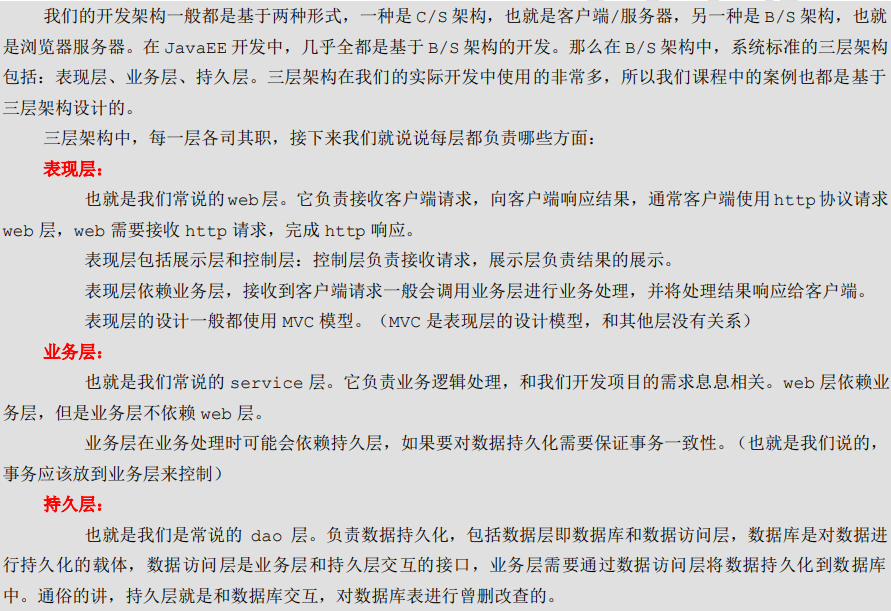
上海千锋 文威

课程目标：

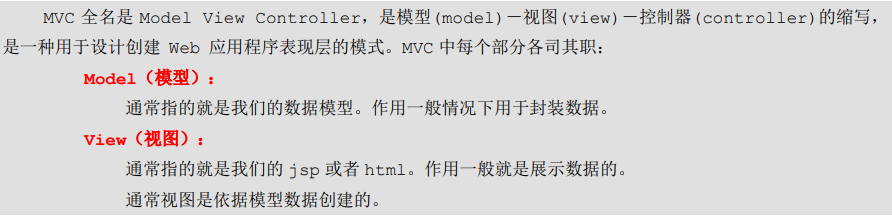
* springmvc的基本概念(理解)
* springmvc案例入门(掌握)
* springmvc执行流程(掌握)
* springmvc参数绑定(掌握)
* springmvc常用注解(掌握)

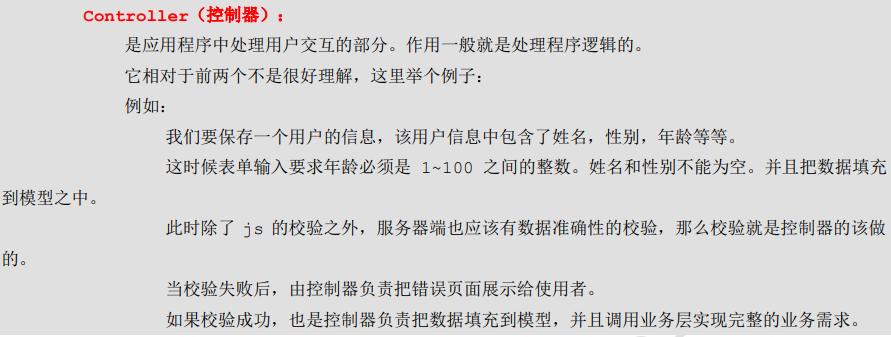
## 1.1 springmvc基本概念

### 1.1.1 三层架构

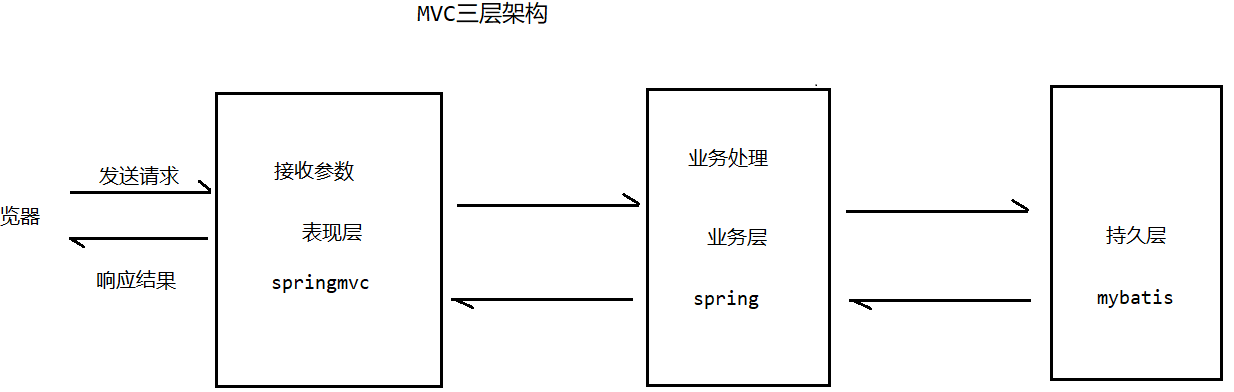


### 1.1.2 MVC架构

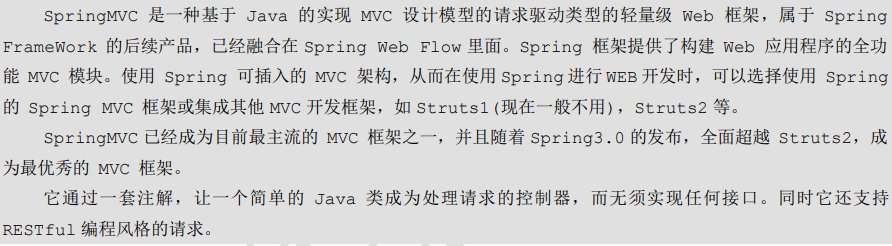




综上所述，我们可以用一幅图来描述MVC三层架构:



### 1.1.3 什么是springmvc

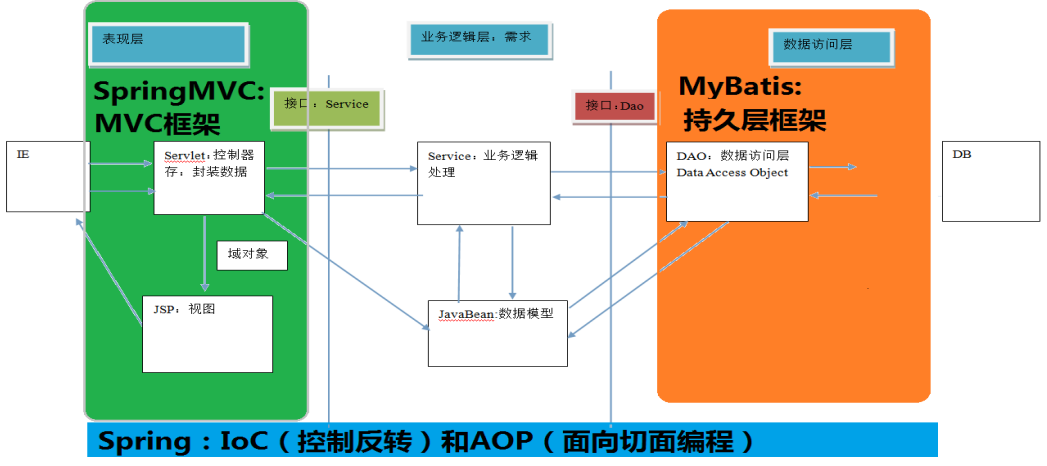


**MVC是模型(Model)、视图(View)、控制器(Controller)的简写，是一种软件设计规范，**MVC是一种架构模式

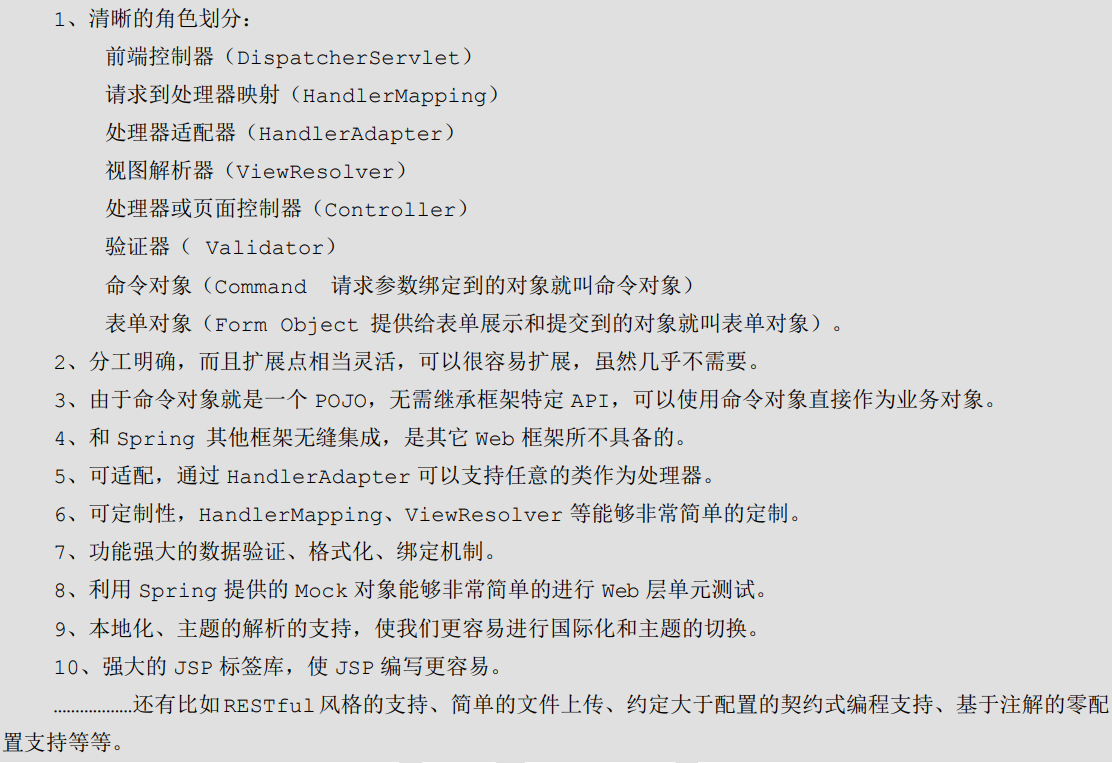
是将业务逻辑、数据、显示分离的方法来组织代码。

MVC主要作用是**降低了视图与业务逻辑间的双向偶合**。

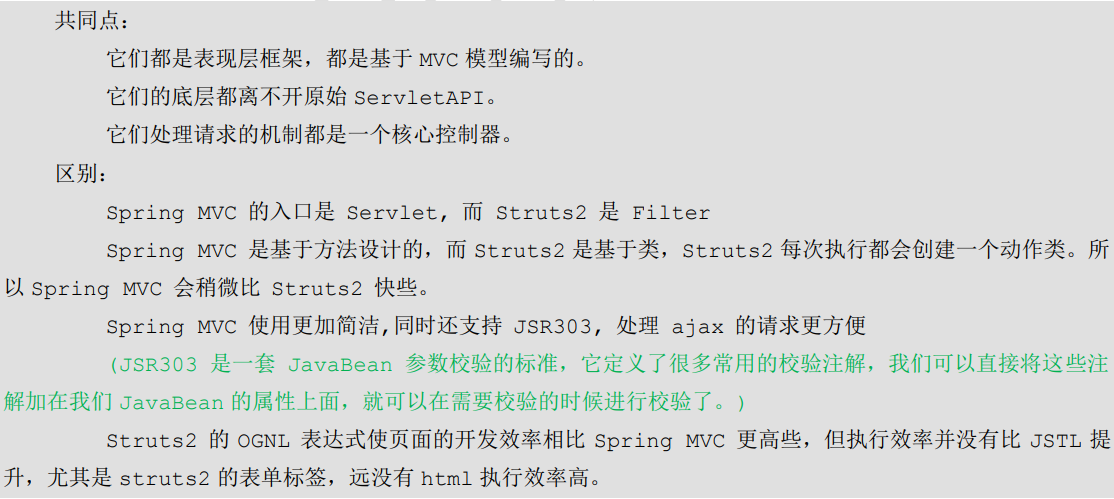
SpringMVC 在三层架构的位置:



### 1.1.4 springmvc的优势



### 1.1.5 SpringMVC和Struts2的优略分析



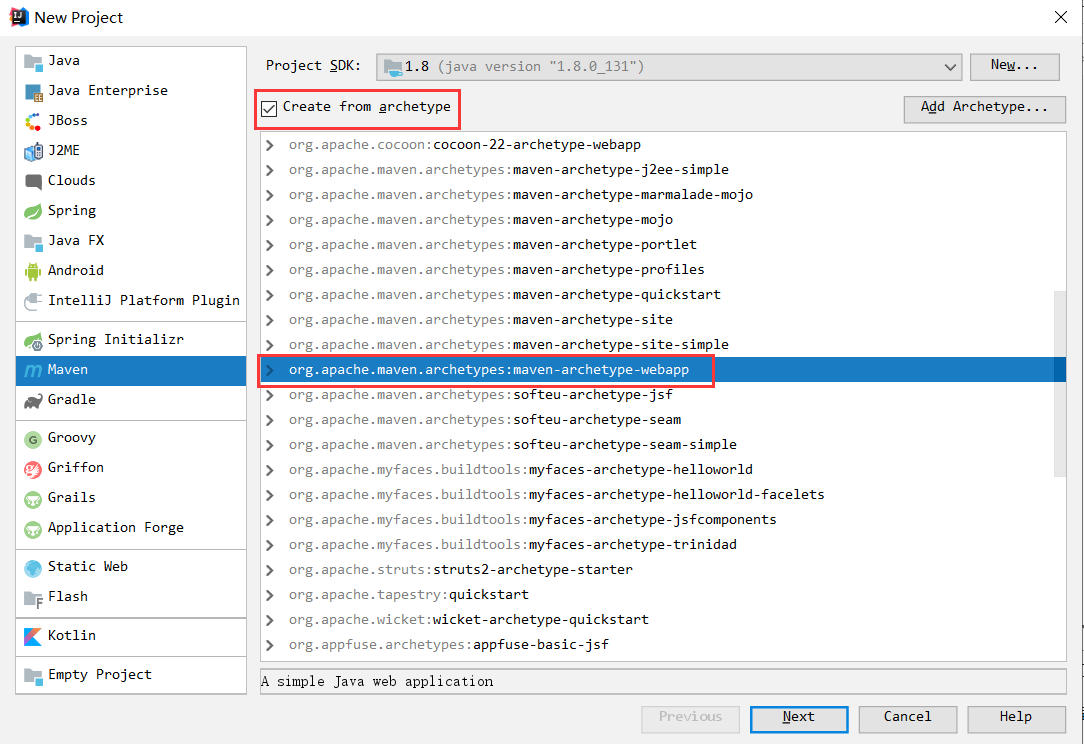
## 1.2 SpringMVC 的入门

### 1.2.1 springmvc的入门案例

需求:用户发送一个请求，跳转到一个jsp页面。

#### 1.2.1.1 搭建项目环境

* 创建一个maven工程(使用骨架创建)



* 导入依赖

|  |
| --- |
| <**properties**>  <**spring.verion**>5.0.2.RELEASE</**spring.verion**> </**properties**>  <**dependencies**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-context</**artifactId**>  <**version**>${spring.verion}</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-web</**artifactId**>  <**version**>${spring.verion}</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-webmvc</**artifactId**>  <**version**>${spring.verion}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>javax.servlet</**groupId**>  <**artifactId**>servlet-api</**artifactId**>  <**version**>2.5</**version**>  <**scope**>provided</**scope**>  </**dependency**>   <**dependency**>  <**groupId**>javax.servlet.jsp</**groupId**>  <**artifactId**>jsp-api</**artifactId**>  <**version**>2.0</**version**>  <**scope**>provided</**scope**>  </**dependency**> </**dependencies**> |

在web.xml里面配置

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns="http://xmlns.jcp.org/xml/ns/javaee"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"  version="3.1"**>  <**display-name**>Archetype Created Web Application</**display-name**>  <**servlet**>  <**servlet-name**>SpringMVCDispatcherServlet</**servlet-name**>  <**servlet-class**>org.springframework.web.servlet.DispatcherServlet</**servlet-class**>  *<!-- 配置初始化参数，用于读取 SpringMVC 的配置文件 -->* <**init-param**>  <**param-name**>contextConfigLocation</**param-name**>  <**param-value**>classpath:springmvc.xml</**param-value**>  </**init-param**>  *<!-- 配置 servlet 的对象的创建时间点：应用加载时创建。  取值只能是非 0 正整数，表示启动顺序 -->* <**load-on-startup**>1</**load-on-startup**>  </**servlet**>  <**servlet-mapping**>  <**servlet-name**>SpringMVCDispatcherServlet</**servlet-name**>  <**url-pattern**>/</**url-pattern**>  </**servlet-mapping**> </**web-app**> |

* 编写springmvc.xml

|  |
| --- |
| <**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"**>  *<!-- 配置创建 spring 容器要扫描的包 -->* <**context:component-scan base-package="com.qf"**></**context:component-scan**>   *<!-- 配置视图解析器 -->* <**bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>  <**property name="prefix" value="/WEB-INF/pages/"**></**property**>  <**property name="suffix" value=".jsp"**></**property**>  </**bean**>  *<!--开启springmvc注解支持-->* <**mvc:annotation-driven**></**mvc:annotation-driven**> </**beans**> |

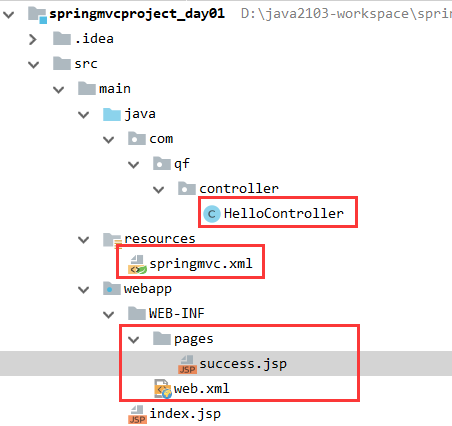
* 编写success.jsp页面

|  |
| --- |
| <**body**>  <**h2**>欢迎你。SpringMVC</**h2**> </**body**> |

* 编写controller

|  |
| --- |
| @Controller **public class** HelloController {  @RequestMapping(**"/hello"**)  **public** String sayHello(){  System.***out***.println(**"hello,springmvc"**);  **return "success"**;  } } |

* 项目最后目录结构

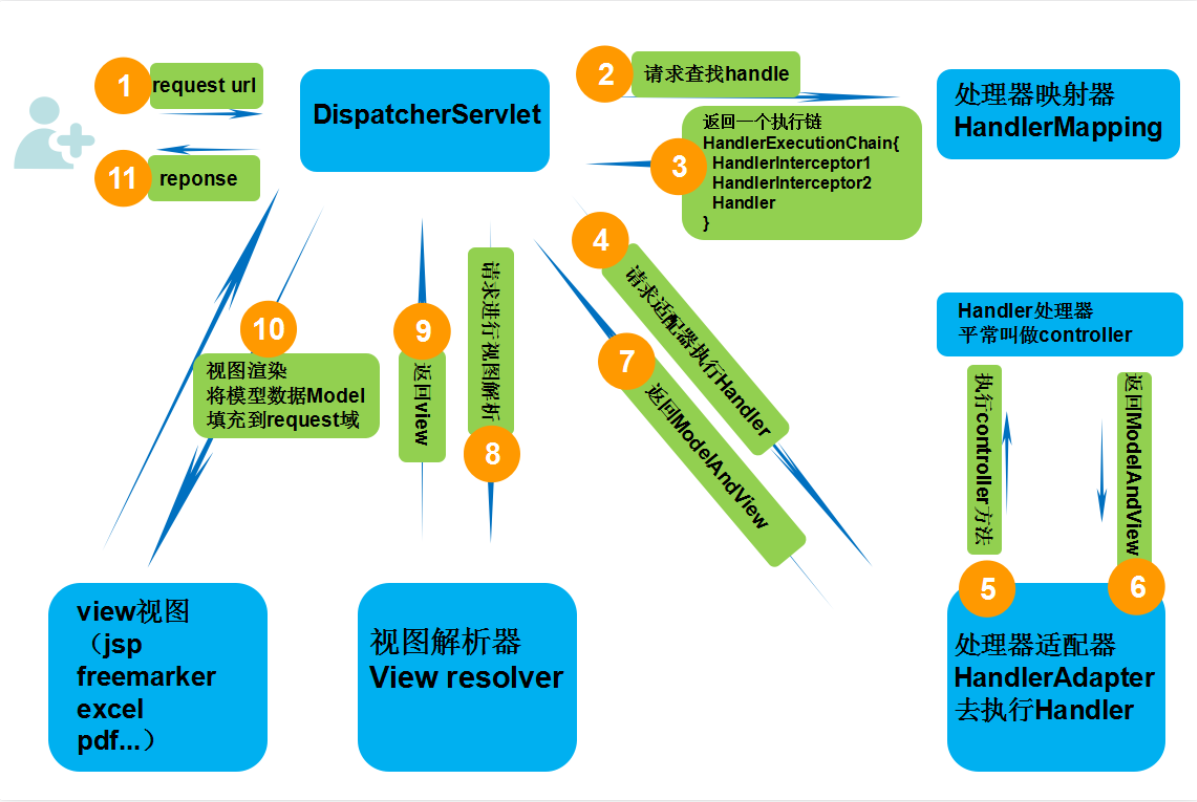


* 配置tomcat(省略)
* 启动项目，测试



#### 1.2.1.2 入门程序分析

* Springmvc的核心执行流程



* 具体步骤
* 第一步：发起请求到前端控制器(DispatcherServlet)
* 第二步：前端控制器请求HandlerMapping查找 Handler （可以根据xml配置、注解进行查找）
* 第三步：处理器映射器HandlerMapping向前端控制器返回Handler，HandlerMapping会把请求映射为HandlerExecutionChain对象（包含一个Handler处理器（页面控制器）对象，多个HandlerInterceptor拦截器对象），通过这种策略模式，很容易添加新的映射策略
* 第四步：前端控制器调用处理器适配器去执行Handler
* 第五步：处理器适配器HandlerAdapter将会根据适配的结果去执行Handler
* 第六步：Handler执行完成给适配器返回ModelAndView
* 第七步：处理器适配器向前端控制器返回ModelAndView （ModelAndView是springmvc框架的一个底层对象，包括 Model和view）
* 第八步：前端控制器请求视图解析器去进行视图解析 （根据逻辑视图名解析成真正的视图(jsp)），通过这种策略很容易更换其他视图技术，只需要更改视图解析器即可
* 第九步：视图解析器向前端控制器返回View
* 第十步：前端控制器进行视图渲染 （视图渲染将模型数据(在ModelAndView对象中)填充到request域）
* 第十一步：前端控制器向用户响应结果

### 1.2.2 springmvc组件

#### 1.2.2.1 DispatcherServlet 前端控制器

用户请求到达前端控制器，它就相当于mvc模式中的c，dispatcherServlet 是整个流程控制的中心，由它调用其它组件处理用户的请求，dispatcherServlet 的存在降低了组件之间的耦合性。

#### 1.2.2.2 HandlerMapping 处理器映射器

HandlerMapping负责根据用户请求找到 Handler 即处理器，SpringMVC 提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

#### 1.2.2.3 Handler 处理器

它就是我们开发中要编写的具体业务控制器。由DispatcherServlet 把用户请求转发到 Handler。由Handler对具体的用户请求进行处理。

#### 1.2.2.4 HandlAdapter 处理器适配器

通过 HandlerAdapter 对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。



#### 1.2.2.5 View Resolver 视图解析器

View Resolver负责将处理结果生成View 视图，View Resolver首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成 View视图对象，最后对View进行渲染将处理结果通过页面展示给用户。

#### 1.2.2.6 View：视图

SpringMVC框架提供了很多View视图类型的支持，包括：jstlView、freemarkerView、pdfView等。我们最常用的视图就是jsp。

一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由程序员根据业务需求开发具体的页面。

#### 1.2.2.7 <mvc:annotation-driven>说明

在SpringMVC 的各个组件中，处理器映射器、处理器适配器、视图解析器称为 SpringMVC 的三大组件。

使用自动加载RequestMappingHandlerMapping(处理映射器)和 RequestMappingHandlerAdapter(处 理 适 配 器) ， 可用在SpringMVC.xml配 置 文件中使用替代注解处理器和适配器的配置。

**注意: 一般开发中，我们都需要写上此标签（虽然从入门案例中看，我们不写也行，随着课程的深入，该标签还有具体的使用场景）。**

## 1.3 RequestMapping注解

### 1.3.1 使用说明

**作用：**用于建立请求URL和处理请求方法之间的对应关系。

**出现位置：**

* 类上：

请求 URL的第一级访问目录。此处不写的话，就相当于应用的根目录。写的话需要以/开头。它出现的目的是为了使我们的 URL 可以按照模块化管理:

例如：

账户模块：

/account/add

/account/update

/account/delete ...

订单模块：

/order/add

/order/update

/order/delete

红色的部分就是把RequsetMappding写在类上，使我们的URL更加精细。

* 方法上：

请求URL的第二级访问目录。

* 属性：

value：用于指定请求的URL。它和path属性的作用是一样的。

method：用于指定请求的方式。

params：用于指定限制请求参数的条件。它支持简单的表达式。要求请求参数的key和value必须和配置的一模一样。

例如：

params = {"accountName"}，表示请求参数必须有accountName

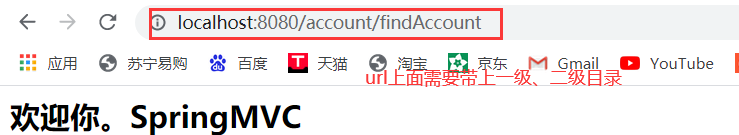
params = {"moeny!100"}，表示请求参数中money不能是 100。

注意：以上四个属性只要出现2个或以上时，他们的关系是与的关系。

### 1.3.2 使用示例

我们使用二级目录访问

|  |
| --- |
| @Controller **@RequestMapping("/account")** **public class** AccountController {  @RequestMapping(**"findAccount"**)  **public** String findAccount(){  **return "success"**;  } } |

此时我们的访问路径应该是:  


如果我们使用超链接访问的话:

在index.jsp里面定义超链接

|  |
| --- |
| <%@ **page language**="**java**" **contentType**="**text/html; charset=UTF-8**"  **pageEncoding**="**UTF-8**"%> <!DOCTYPE **html** PUBLIC **"-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd"**> <**html**> <**head**>  <**meta http-equiv="Content-Type" content="text/html; charset=UTF-8"**>  <**title**>requestmapping 的使用</**title**> </**head**> <**body**>  <**h2**>Hello World!</**h2**>  <**a href="/account/findAccount"**>[超链接1]</**a**>  <**a href="${**pageContext.request.contextPath**}/account/findAccount"**>[超链接2]</**a**> </**body**> </**html**> |

**注意：当我们使用此种方式配置时，在jsp中第二种写法时，不要在访问URL前面加/，否则无法找到资源。**

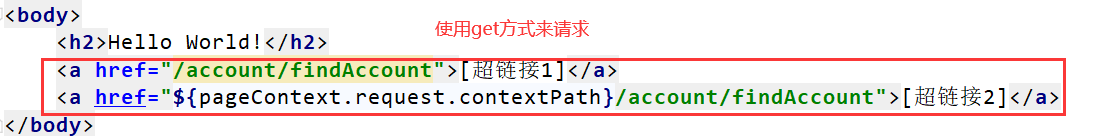
### 1.3.3 method属性示例

描述需要使用指定的请求方式来请求该方法。

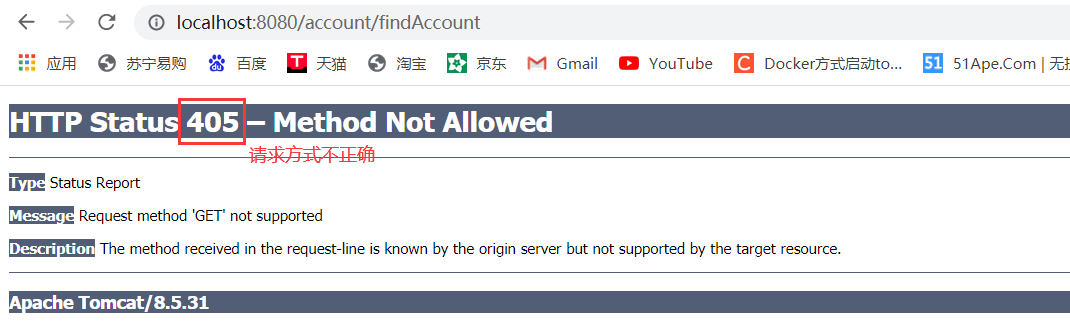
|  |
| --- |
| @Controller @RequestMapping(**"/account"**) **public class** AccountController {  @RequestMapping(value = **"findAccount"**,**method = RequestMethod.*POST***)  **public** String findAccount(){  **return "success"**;  } } |

method = RequestMethod.***POST 使用POST方法来请求该方法***

测试:

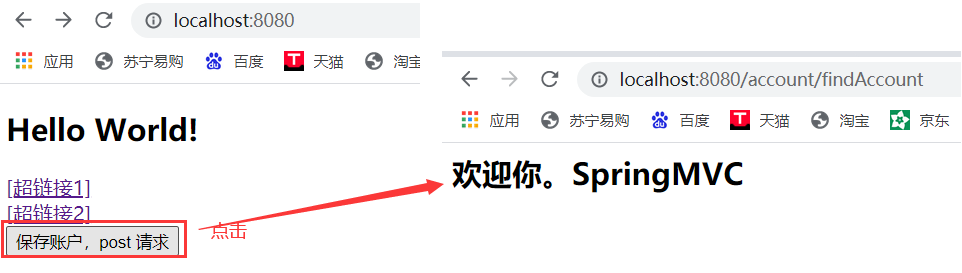


结果:



也就是我们不能通过get方法来请求post方法。我们再换一种请求方式。

|  |
| --- |
| <**form action="account/findAccount" method="post"**>  <**input type="submit" value="保存账户，post 请求"**> </**form**> |



### 1.3.4 params属性

要求请求的时候，必须携带指定的参数。用法:param={}

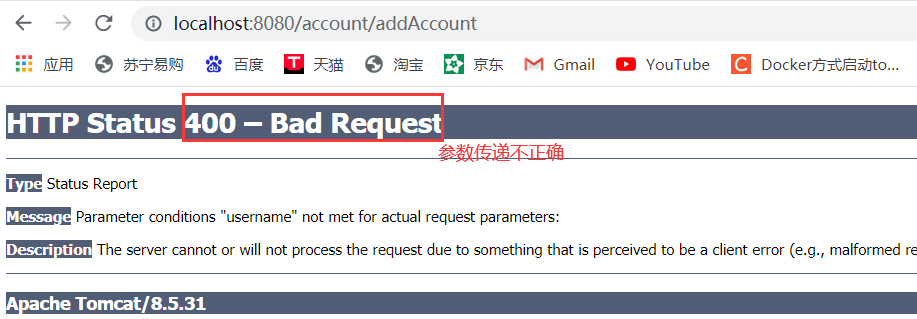
实例:

|  |
| --- |
| @RequestMapping(value = **"addAccount"**,params = {**"username"**}) **public** String addAccount(){  **return "success"**; } |

上面的代码要求请求必须携带username参数。

此时我们不携带参数测试一下:

|  |
| --- |
| <**a href="${**pageContext.request.contextPath**}/account/addAccount"**>[超链接3]</**a**><**br**> |



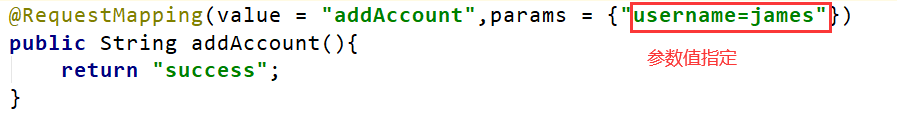
此时报错。

然后我们携带参数(使用?号拼接参数)，参数的值是任意的。

|  |
| --- |
| <**a href="${**pageContext.request.contextPath**}/account/addAccount?username=eric"**>[超链接3]</**a**><**br**> |

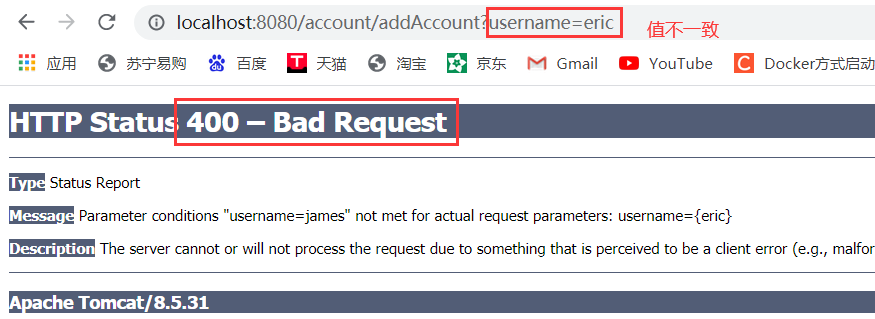
测试没有问题。

当然我们也可以指定参数的值，要求请求携带的参数值必须和我们指定参数的值一致。



|  |
| --- |
| <**a href="${**pageContext.request.contextPath**}/account/addAccount?username=eric"**>[超链接3]</**a**><**br**>  **携带的参数值和后台指定的不一样** |

值不一致: 此时请求也会报错



值一致:成功跳转

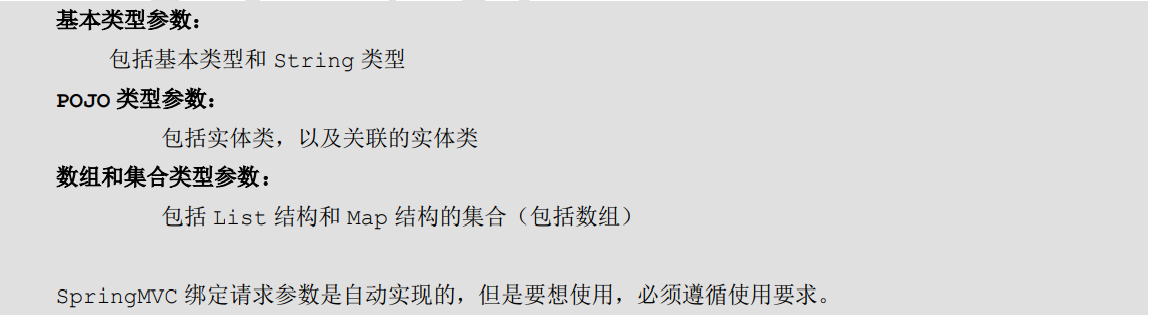


## 1.4 请求参数绑定

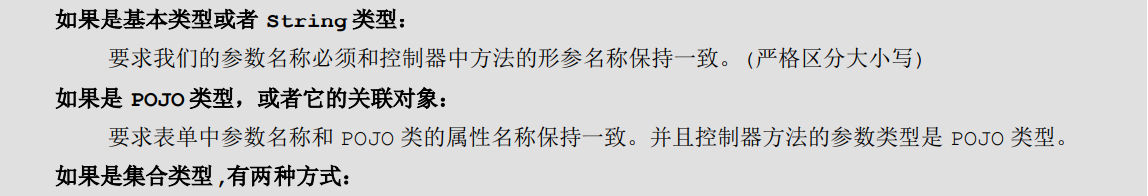
**绑定机制:**

我们都知道，用户请求参数都是基于key=value的。SpringMVC绑定请求参数的过程是通过把请求提交的参数，作为控制器中的方法参数进行绑定的。

Springmvc支持的参数绑定类型:



简单的:



### 1.4.1 基本数据类型参数绑定

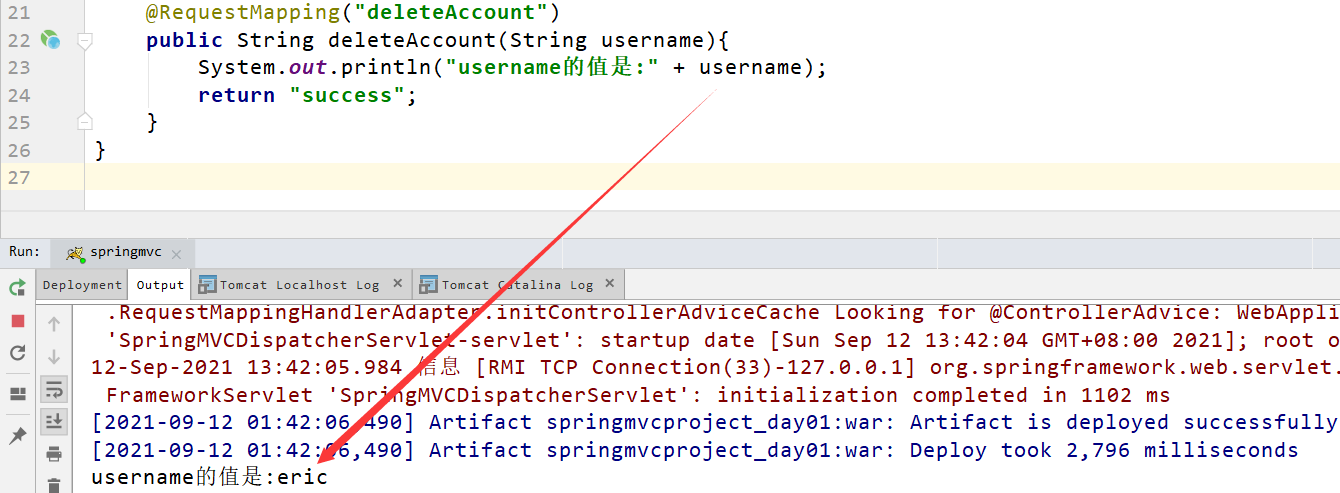
* 编写controller

|  |
| --- |
| @RequestMapping(**"deleteAccount"**) **public** String deleteAccount(String username){  System.***out***.println(**"username的值是:"** + username);  **return "success"**; } |

* 编写jsp param.jsp

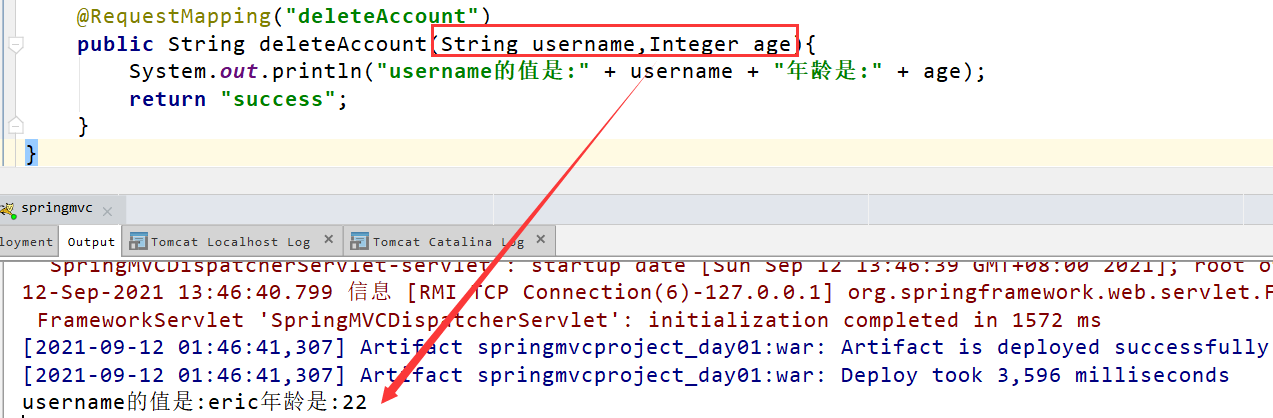
|  |
| --- |
| <**body**>  <**a href="${**pageContext.request.contextPath**}/account/deleteAccount?username=eric"**>[提交请求参数]</**a**> </**body**> |

* 测试



如果携带多个参数的话，也是一样的。

|  |
| --- |
| <**body**>  <**a href="${**pageContext.request.contextPath**}/account/deleteAccount?username=eric&age=22"**>[提交请求参数]</**a**> </**body**> |



### 1.4.2 POJO类型参数绑定

* 编写pojo类

|  |
| --- |
| **public class** Account **implements** Serializable {  **private** Integer **id**;  **private** String **name**;  **private** Float **money**;  **private** Address **address**;  //省略get set方法  } |

|  |
| --- |
| **public class** Address **implements** Serializable {  **private** String **provinceName**;  **private** String **cityName**;  //省略get set方法  } |

* 编写acount.jsp

|  |
| --- |
| <**body**>  <**form action="account/saveAccount" method="post"**>  账户名称：<**input type="text" name="name"**><**br**/>  账户金额：<**input type="text" name="money"**><**br**/>  账户省份：<**input type="text" name="address.provinceName"**><**br**/>  账户城市：<**input type="text" name="address.cityName"**><**br**/>  <**input type="submit" value="保存"**>  </**form**> </**body**> |

* 测试

控制台打印:

|  |
| --- |
| Account{id=null, name='aaa', money=12.0, address=Address{provinceName='aa', cityName='aa'}} |

### 1.4.3 集合类型参数绑定

* 编写pojo类

|  |
| --- |
| **public class** User {   **private** String **username**;   **private** String **password**;   **private** Integer **age**;   **private** List<Account> **accountList**;   **private** Map<String,Account> **accountMap**;  //省略get set方法  } |

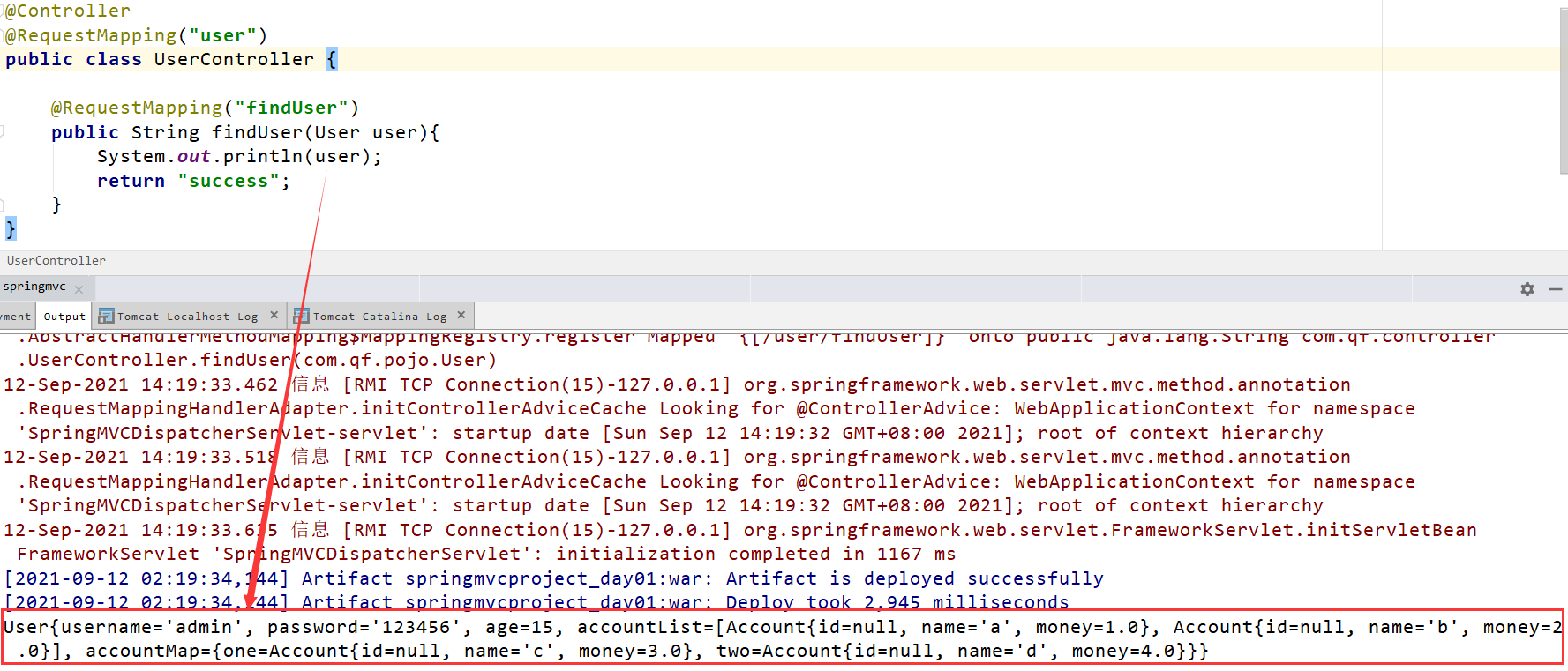
* 编写controller

|  |
| --- |
| @Controller @RequestMapping(**"user"**) **public class** UserController {   @RequestMapping(**"findUser"**)  **public** String findUser(User user){  System.***out***.println(user);  **return "success"**;  } } |

* 编写user.jsp

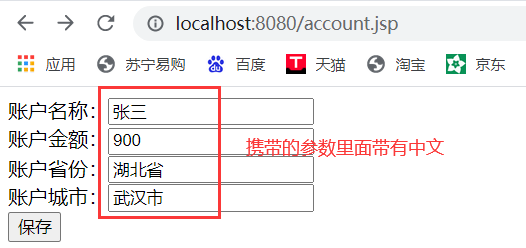
|  |
| --- |
| <**form action="user/findUser" method="post"**>  用户名称：<**input type="text" name="username"** ><**br**/>  用户密码：<**input type="password" name="password"** ><**br**/>  用户年龄：<**input type="text" name="age"** ><**br**/>  账户 1 名称：<**input type="text" name="accountList[0].name"** ><**br**/>  账户 1 金额：<**input type="text" name="accountList[0].money"** ><**br**/>  账户 2 名称：<**input type="text" name="accountList[1].name"** ><**br**/>  账户 2 金额：<**input type="text" name="accountList[1].money"** ><**br**/>  账户 3 名称：<**input type="text" name="accountMap['one'].name"** ><**br**/>  账户 3 金额：<**input type="text" name="accountMap['one'].money"** ><**br**/>  账户 4 名称：<**input type="text" name="accountMap['two'].name"** ><**br**/>  账户 4 金额：<**input type="text" name="accountMap['two'].money"** ><**br**/>  <**input type="submit" value="保存"**> </**form**> |

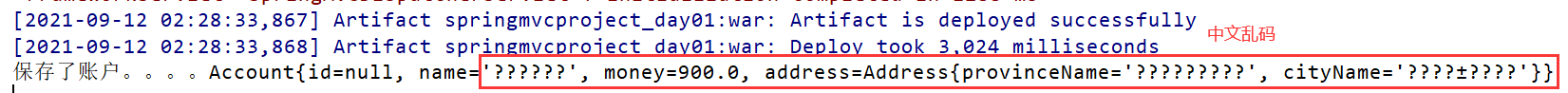
* 测试



### 1.4.4 请求参数乱码问题

如果我们请求的参数里面带有中文，数据还能够正常显示吗?  
我们可以测试一下:





我们发现中文乱码。

如何解决中文乱码呢?

* POST请求方式解决乱码问题

在web.xml里面设置编码过滤器

|  |
| --- |
| <**filter**>  <**filter-name**>CharacterEncodingFilter</**filter-name**>  <**filter-class**>  org.springframework.web.filter.CharacterEncodingFilter  </**filter-class**>  *<!-- 设置过滤器中的属性值 -->* <**init-param**>  <**param-name**>encoding</**param-name**>  <**param-value**>UTF-8</**param-value**>  </**init-param**>  *<!-- 启动过滤器 -->* <**init-param**>  <**param-name**>forceEncoding</**param-name**>  <**param-value**>true</**param-value**>  </**init-param**> </**filter**> *<!-- 过滤所有请求 -->* <**filter-mapping**>  <**filter-name**>CharacterEncodingFilter</**filter-name**>  <**url-pattern**>/\*</**url-pattern**> </**filter-mapping**> |

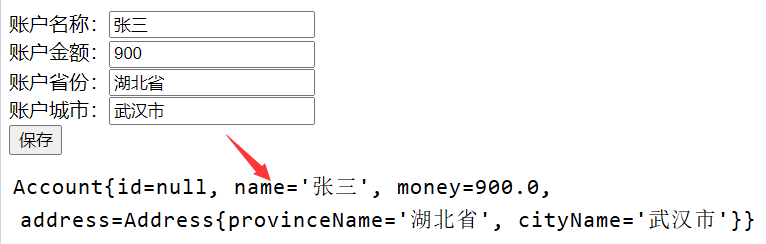
但是需要注意的是:

我们web应用里面有静态资源，比如js css image等。我们不能够把静态资源也过滤了，否则会造成静态资源不能正常加载的问题。所以在配置编码过滤器的时候。我们需要设置静态资源不过滤。

在springmvc.xml里面设置

|  |
| --- |
| *<!-- location 表示路径，mapping 表示文件，\*\*表示该目录下的文件以及子目录的文件 -->* <**mvc:resources location="/css/" mapping="/css/\*\*"**/> <**mvc:resources location="/images/" mapping="/images/\*\*"**/> <**mvc:resources location="/scripts/" mapping="/javascript/\*\*"**/> |

然后我们再次测试:



* GET请求方式解决乱码问题

tomcat对GET和POST请求处理方式是不同的，GET请求的编码问题，要改tomcat的 server.xml 配置文件，如下：

|  |
| --- |
| <Connector port="8080" protocol="HTTP/1.1"  connectionTimeout="20000"  redirectPort="8443" useBodyEncodingForURI="true"/> |

如果遇到 ajax 请求仍然乱码，请把：

**useBodyEncodingForURI="true"改为 URIEncoding="UTF-8"**即可。

### 1.4.5 自定义类型转换器

#### 1.4.5.1 使用场景

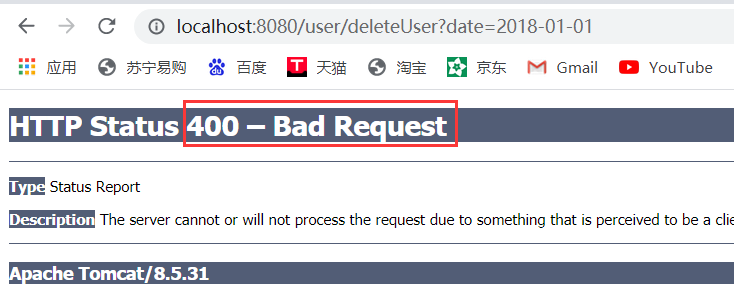
* 定义jsp代码

|  |
| --- |
| *<!-- 特殊情况之：类型转换问题 -->* <**a href="user/deleteUser?date=2018-01-01"**>根据日期删除账户</**a**> |

* 定义controller代码

|  |
| --- |
| @RequestMapping(**"deleteUser"**) **public** String findUser(**Date date**){  System.***out***.println(date);  **return "success"**; } |

* 运行结果



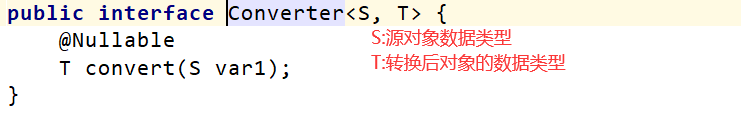


原因:我们前台传递的是字符串类型的参数，但是后台使用的是Date类型接收的。我们期望springmvc可以帮我们做数据类型的自动转换，显然没有做。

所以我们需要自己自定义类型转换器。

#### 1.4.5.2 自定义类型转换器

第一步：定义一个类，实现Converter接口，该接口有两个泛型。



|  |
| --- |
| **public class** StringToDateConverter **implements** Converter<String, Date> {  @Override  **public** Date convert(String source) {  DateFormat format = **null**;  **try** {  **if**(StringUtils.*isEmpty*(source)) {  **throw new** NullPointerException(**"请输入要转换的日期"**);  }  format = **new** SimpleDateFormat(**"yyyy-MM-dd"**);  Date date = format.parse(source);  **return** date;  } **catch** (Exception e) {  **throw new** RuntimeException(**"输入日期有误"**);  }  } } |

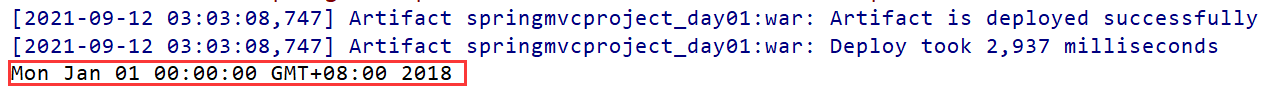
第二步：在 springmvc.xml配置文件中配置类型转换器。

|  |
| --- |
| *<!-- 配置类型转换器工厂 -->* <**bean id="converterService"  class="org.springframework.context.support.ConversionServiceFactoryBean"**>  *<!-- 给工厂注入一个新的类型转换器 -->* <**property name="converters"**>  <**array**>  *<!-- 配置自定义类型转换器 -->* <**bean class="com.qf.convert.StringToDateConverter"**></**bean**>  </**array**>  </**property**> </**bean**> *<!--开启springmvc注解支持-->* <**mvc:annotation-driven conversion-service="converterService"**></**mvc:annotation-driven**> |

第三步：在annotation-driven标签中引用配置的类型转换服务。

|  |
| --- |
| *<!--开启springmvc注解支持-->* <**mvc:annotation-driven conversion-service="converterService"**></**mvc:annotation-driven**> |

第四步:测试



没有问题。

### 1.4.6 使用@DateTimeFormat注解进行数据格式转换

* 使用get请求，传递日期类型的数据。

|  |
| --- |
| <**a href="${**pageContext.request.contextPath**}/user/getDate?date=2021-12-7"**>[链接到]</**a**> |

|  |
| --- |
| @RequestMapping(**"getDate"**) **public** String getDate( **@DateTimeFormat(pattern="yyyy-MM-dd")** Date date){  System.***out***.println(date);  **return "success"**; } |

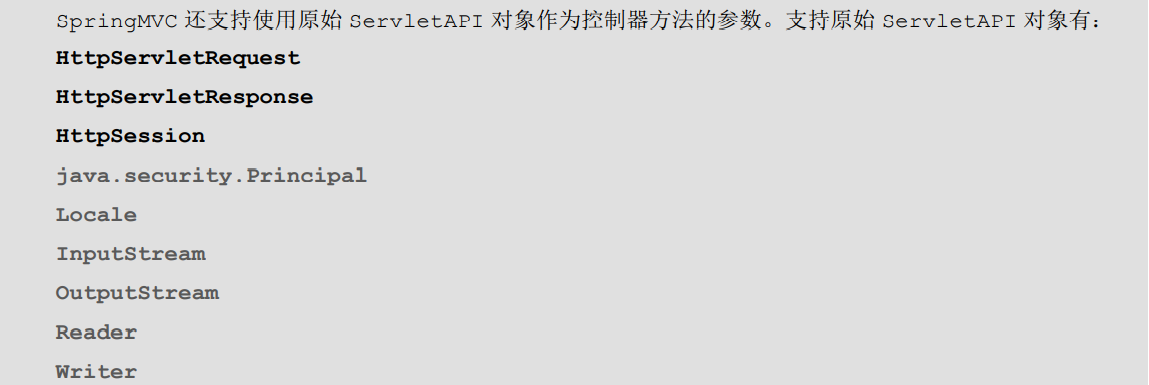
* 使用post请求，传递日期类型的数据。

|  |
| --- |
| <**body**>  <**form action="/user/getStudentInfo" method="post"**>  学生姓名: <**input type="text" name="name"**> <**br**>  学生生日: <**input type="text" name="birthday"**> <**br**>  学生年龄: <**input type="text" name="age"**> <**br**>  <**input type="submit" value="提交"**>  </**form**> </**body**> |

|  |
| --- |
| **public class** Student {  **private** String **name**;   **@DateTimeFormat(pattern="yyyy-MM-dd")**  **private** Date **birthday**;  **private** Integer **age**;   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public** Date getBirthday() {  **return birthday**;  }   **public void** setBirthday(Date birthday) {  **this**.**birthday** = birthday;  }   **public** Integer getAge() {  **return age**;  }   **public void** setAge(Integer age) {  **this**.**age** = age;  }   @Override  **public** String toString() {  **return "Student{"** +  **"name='"** + **name** + **'\''** +  **", birthday="** + **birthday** +  **", age="** + **age** +  **'}'**;  } } |

|  |
| --- |
| @RequestMapping(**"getStudentInfo"**) **public** String getStudentInfo(Student student){  System.***out***.println(student);  **return "success"**; } |

### 1.4.7 使用ServletAPI传递参数



**我们可以把上述对象，直接写在控制的方法参数中使用。**

jsp代码：

|  |
| --- |
| <**a href="account/testServletAPI?name=eric&age=19"**>测试访问 ServletAPI</**a**> |

控制器中的代码：

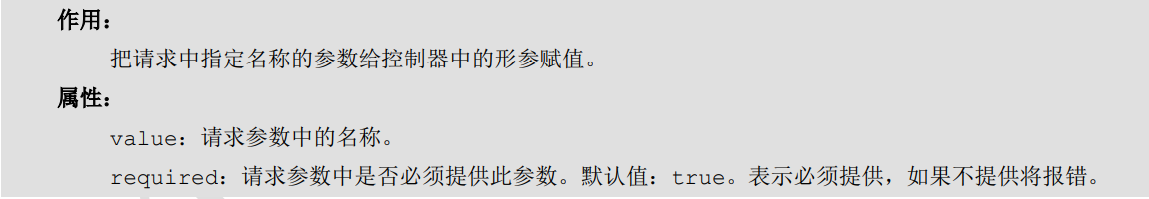
|  |
| --- |
| @RequestMapping(**"testServletAPI"**) **public void** testServletAPI(HttpServletRequest request, HttpServletResponse response){  String name = request.getParameter(**"name"**);  String age = request.getParameter(**"age"**);  System.***out***.println(**"姓名:"** + name + **"年龄:"** + age); } |

测试略。

## 1.5 springmvc常用注解

### 1.5.1 RequestParam注解

#### 1.5.1.1 使用说明



#### 1.5.1.2 使用示例

前面我们已经理解了其基本用法，下面主要演示几个特性。

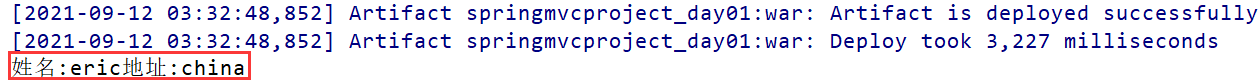
* jsp代码

|  |
| --- |
| <**a href="user/addUser?username=eric&address=china"**>[超链接]</**a**> |

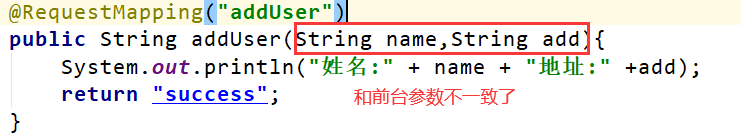
* 控制器代码

|  |
| --- |
| @RequestMapping(**"addUser"**) **public** String addUser(String username,String address){  System.***out***.println(**"姓名:"** + username + **"地址:"** +address);  **return "success"**; } |

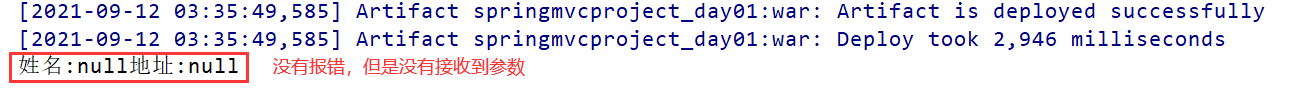
测试是没有问题的。后台也能成功接收参数。



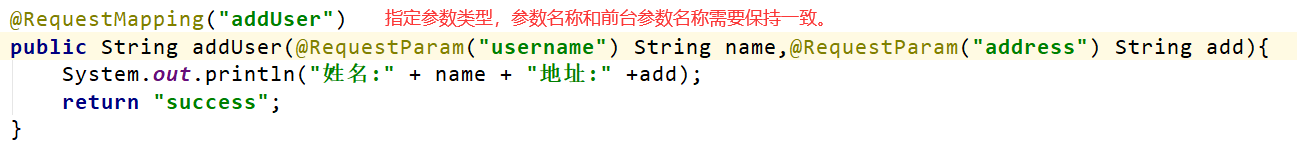
如果后台接收的参数名称和前台参数名称不一样,台还能接收到参数吗?



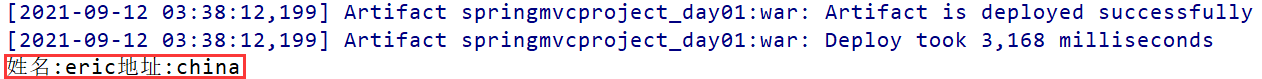
我们发送请求，之后再测试:



那么如何在前后端参数不一致的情况下，也能保证后台数据能够正常接收呢



我们再次测试:



同时我们在后台指定参数是否是必须的。

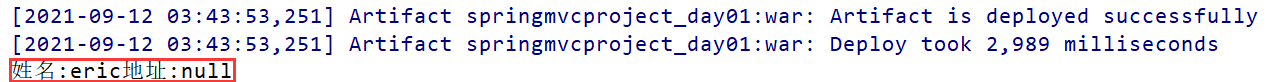
* jsp代码

|  |
| --- |
| <**a href="user/addUser?username=eric"**>[超链接]</**a**> |

|  |
| --- |
| @RequestMapping(**"addUser"**) **public** String addUser(@RequestParam(**"username"**) String name,@RequestParam(value = **"address"**,**required = false**) String add){  System.***out***.println(**"姓名:"** + name + **"地址:"** +add);  **return "success"**; } |

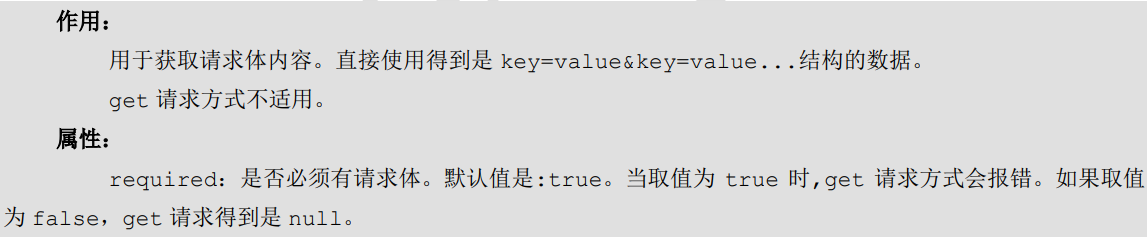
测试前台只有一个参数，后台两个参数接收，如果不指定,required = **false**就会报错。

指定之后测试:



### 1.5.2 RequestBody参数

#### 1.5.2.1 使用说明



#### 1.5.2.2 使用示例

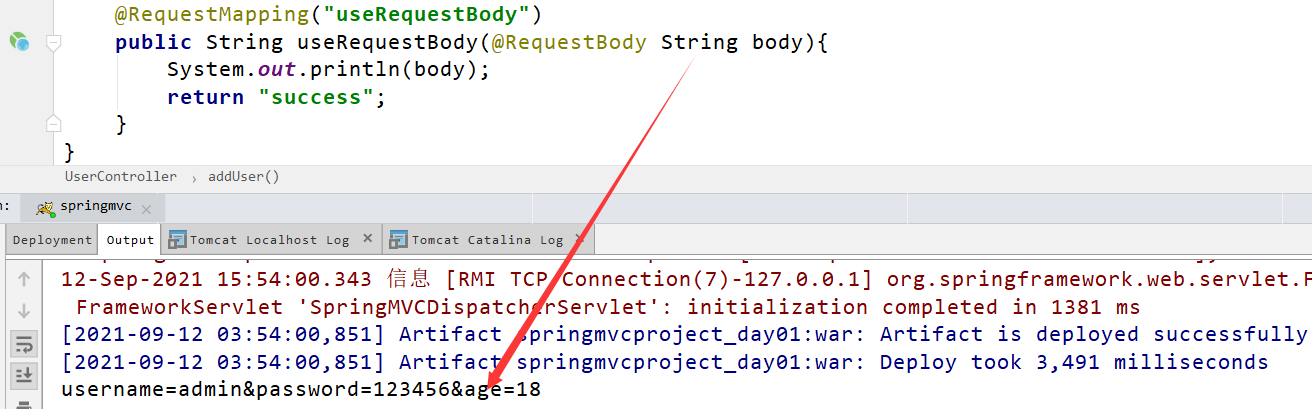
* jsp代码

|  |
| --- |
| <**form action="user/useRequestBody" method="post"**>  用户名称：<**input type="text" name="username"** ><**br**/>  用户密码：<**input type="password" name="password"** ><**br**/>  用户年龄：<**input type="text" name="age"** ><**br**/>  <**input type="submit" value="保存"**> </**form**> |

* 控制器代码

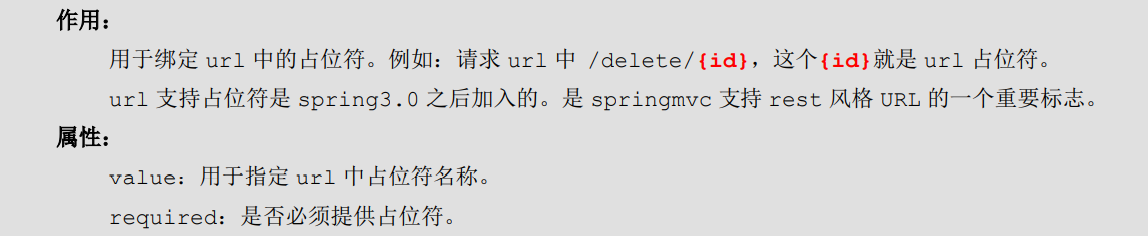
|  |
| --- |
| @RequestMapping(**"useRequestBody"**) **public** String useRequestBody(@RequestBody String body){  System.***out***.println(body);  **return "success"**; } |

* 测试



### 1.5.3 PathVaribale

#### 1.5.3.1 使用说明



#### 1.5.3.2 使用示例

* jsp代码

|  |
| --- |
| <**a href="user/usePathVariable/100"**>pathVariable注解</**a**> |

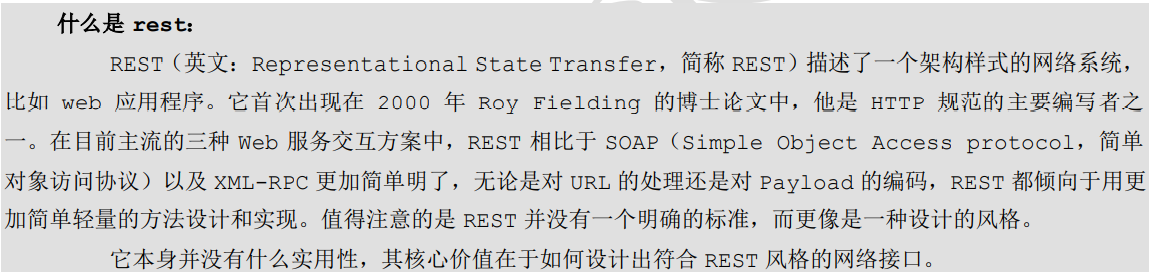
* 控制器代码

|  |
| --- |
| @RequestMapping(**"usePathVariable/{id}"**) **public** String usePathVariable(@PathVariable(**"id"**) Integer id){  System.***out***.println(**"值是："** + id);  **return "success"**; } |

* 测试



#### 1.5.3.3 REST风格URL



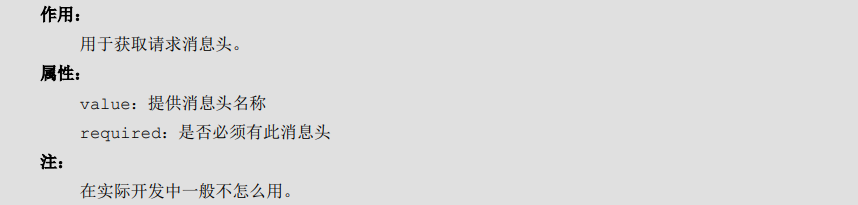


总结:

Restful风格就是请求url统一，根据不同的请求方式，请求不同的后台方法。如果需要携带参数，在url上使用/{}占位符。

### 1.5.4 RequestHeader注解

#### 1.5.4.1 使用说明



#### 1.5.4.2 使用示例

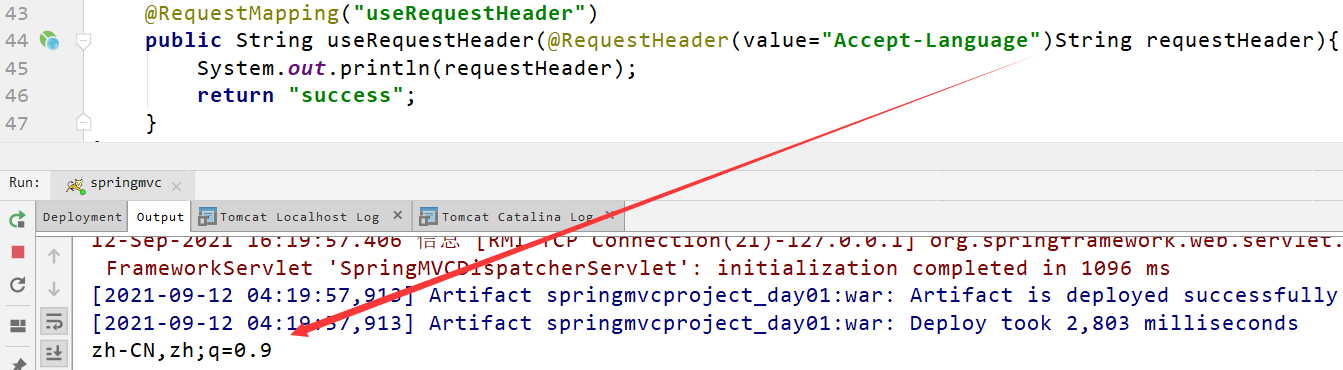
* jsp代码

|  |
| --- |
| <**a href="user/useRequestHeader"**>获取请求消息头</**a**> |

* 控制层代码

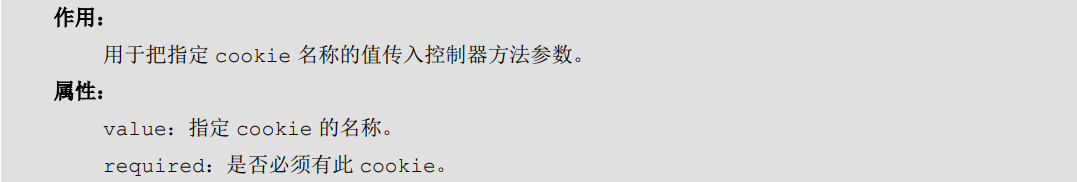
|  |
| --- |
| @RequestMapping(**"useRequestHeader"**) **public** String useRequestHeader(@RequestHeader(value=**"Accept-Language"**)String requestHeader){  System.***out***.println(requestHeader);  **return "success"**; } |

* 测试



### 1.5.5 CookieValue注解

#### 1.5.5.1 使用说明



#### 1.5.5.2 使用示例

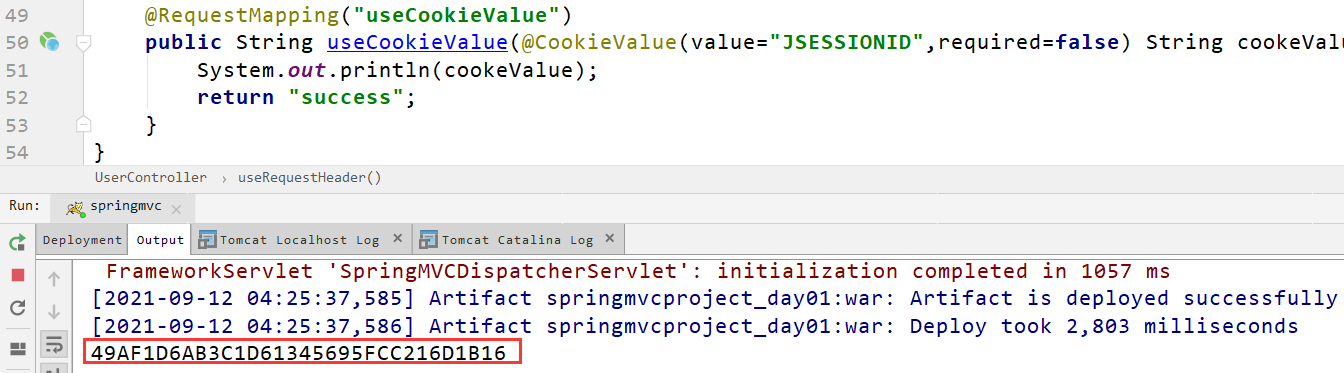
* jsp代码

|  |
| --- |
| <**a href="user/useCookieValue"**>绑定 cookie 的值</**a**> |

* 控制器代码

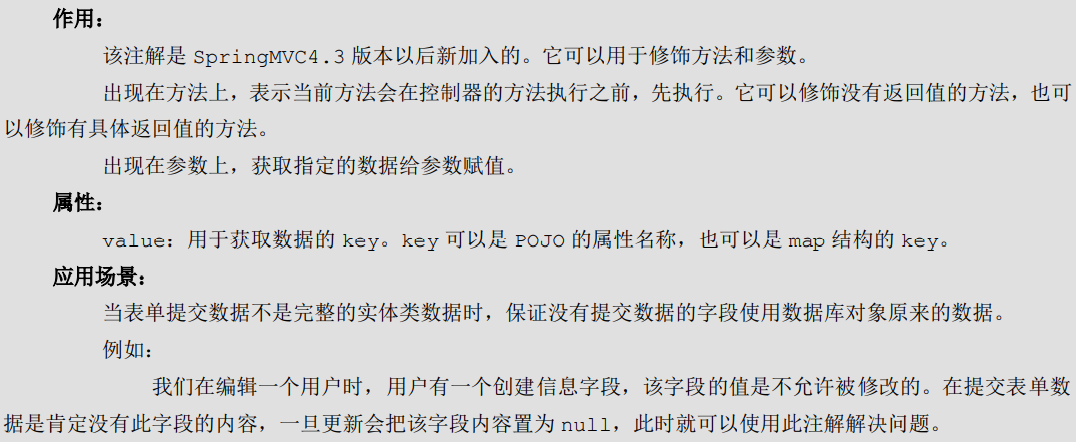
|  |
| --- |
| @RequestMapping(**"useCookieValue"**) **public** String useCookieValue(@CookieValue(value=**"JSESSIONID"**,required=**false**) String cookeValue){  System.***out***.println(cookeValue);  **return "success"**; } |

* 测试



### 1.5.6 ModelAttribute注解

#### 1.5.6.1 使用说明



#### 1.5.6.2 使用示例

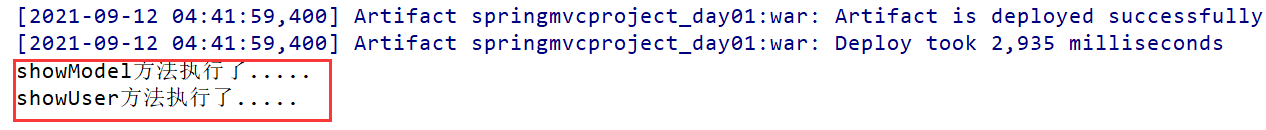
* 放在方法上，在控制器方法前面先执行
* 准备jsp代码

|  |
| --- |
| <**form action="user/showUser" method="post"**>  姓名: <**input type="text" name="username"**><**br**>  年龄: <**input type="text" name="age"**><**br**>  <**input type="submit" value="保存"**> </**form**> |

* 准备控制层代码

|  |
| --- |
| @ModelAttribute **public void** showModel(){  System.***out***.println(**"showModel方法执行了....."**); }  @RequestMapping(**"showUser"**) **public** String showUser(){  System.***out***.println(**"showUser方法执行了....."**);  **return "success"**; } |

* 测试



* 表单数据提交不完整，保证没有提交数据的字段使用数据表原来的数据
* 定义pojo User

|  |
| --- |
| **public class** User {  **private** String **username**;  **private** String **password**;  **private** Integer **age**;  //省略get set方法  } |

* 定义表单

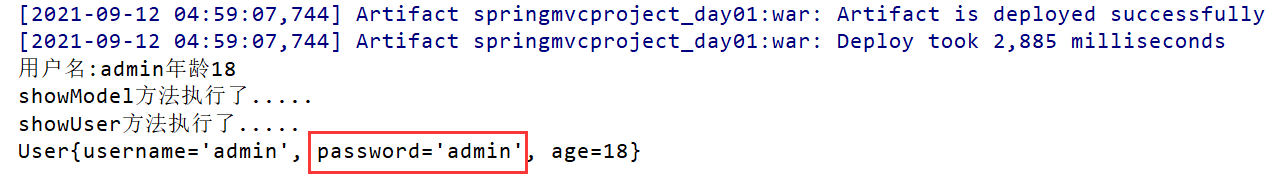
|  |
| --- |
| <**form action="user/showUser" method="post"**>  姓名: <**input type="text" name="username"**><**br**>  年龄: <**input type="text" name="age"**><**br**>  <**input type="submit" value="保存"**> </**form**> |

注意:这个表单只提交了两个字段，还有password字段没有提交。

* 定义控制器方法(ModelAndView注解修饰的方法带返回值)

|  |
| --- |
| @ModelAttribute **public** User showModel(String username,Integer age){  System.***out***.println(**"用户名:"** + username + **"年龄"** + age);  User user = **new** User();  user.setUsername(username);  user.setAge(18);  *//模拟根据用户名查询从数据表查询到了一个用户信息* user.setPassword(**"admin"**);  System.***out***.println(**"showModel方法执行了....."**);  **return** user; }  @RequestMapping(**"showUser"**) **public** String showUser(User user){  System.***out***.println(**"showUser方法执行了....."**);  System.***out***.println(user);  **return "success"**; } |

* 测试

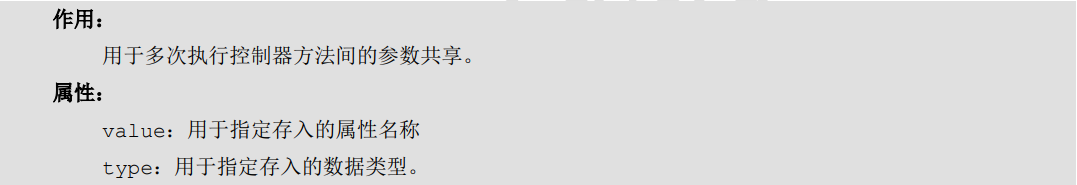


* 定义控制器方法(ModelAndView注解修饰的方法也可以不带返回值)

|  |
| --- |
| @ModelAttribute **public void** showModel(String username, Integer age, **Map<String,Object> map**){  System.***out***.println(**"用户名:"** + username + **"年龄"** + age);  User user = **new** User();  user.setUsername(username);  user.setAge(18);  *//模拟根据用户名查询从数据表查询到了一个用户信息* user.setPassword(**"admin"**);  System.***out***.println(**"showModel方法执行了....."**);   **map.put("user",user);** }  @RequestMapping(**"showUser"**) **public** String showUser(**@ModelAttribute("user")User user**){  System.***out***.println(**"showUser方法执行了....."**);  System.***out***.println(user);  **return "success"**; } |

### 1.5.7 SessionAttributes注解

#### 1.5.7.1 使用说明



#### 1.5.7.2 使用示例

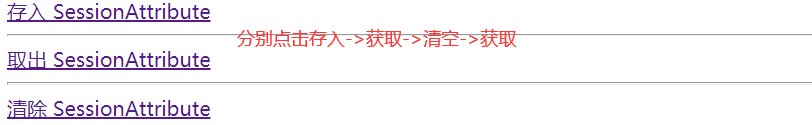
* 编写jsp页面

|  |
| --- |
| <**a href="springmvc/putMethod"**>存入 SessionAttribute</**a**> <**hr**/> <**a href="springmvc/getMethod"**>取出 SessionAttribute</**a**> <**hr**/> <**a href="springmvc/cleanMethod"**>清除 SessionAttribute</**a**> |

* 编写控制器代码

|  |
| --- |
| @Controller @RequestMapping(**"/springmvc"**) @SessionAttributes(value ={**"username"**,**"password"**,**"age"**}) **public class** SessionAttributeController {   @RequestMapping(**"putMethod"**)  **public** String putMethod(Model model){  model.addAttribute(**"username"**, **"泰斯特"**);  model.addAttribute(**"password"**,**"123456"**);  model.addAttribute(**"age"**,**"admin"**);  **return "success"**;  }   @RequestMapping(**"getMethod"**)  **public** String getMethod(ModelMap model){  System.***out***.println(model.get(**"username"**)+**";"**+model.get(**"password"**)+**";"**+model.get(**"age"**));  **return "success"**;  }   @RequestMapping(**"cleanMethod"**)  **public** String cleanMethod(SessionStatus sessionStatus){  sessionStatus.setComplete();  **return "success"**;  } } |

* 测试



* 观察控制台信息

