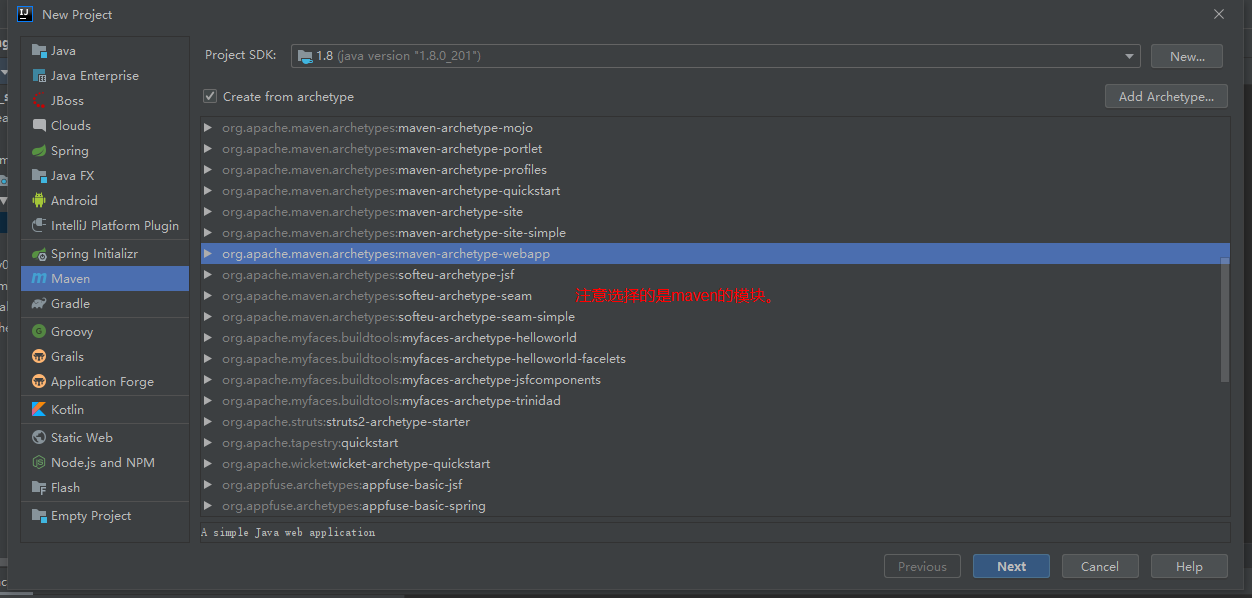
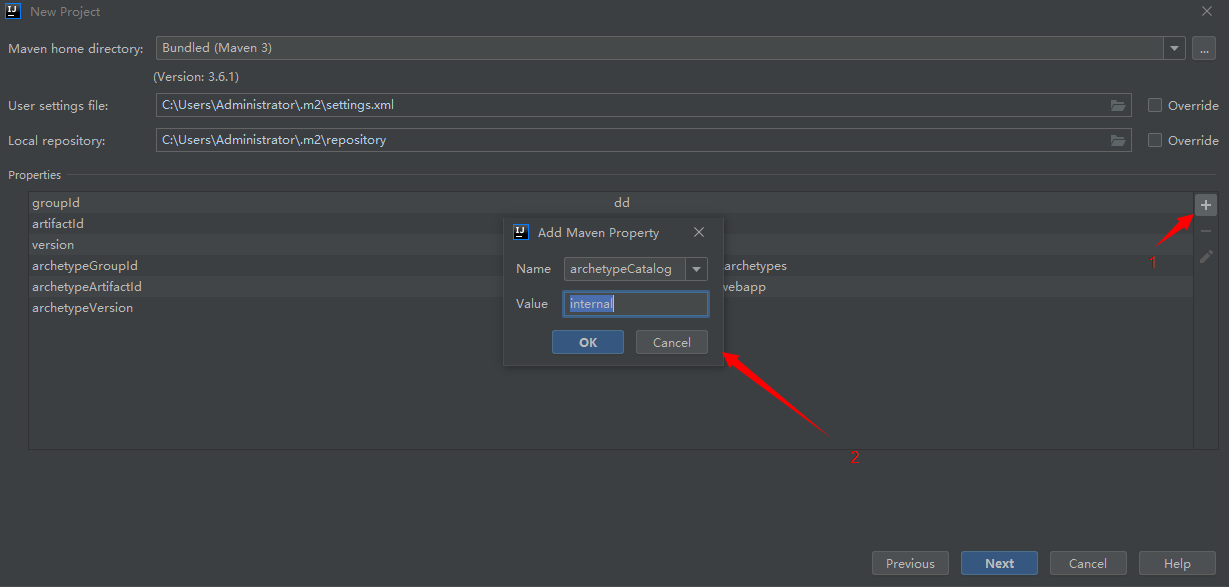
SpringMVC

# 入门案例

**新建**：new->project->





name: archetypeCatalog , value: internal

新建java目录并右键make directory as为source root

新建resources目录比右键make directory as为 resource root

**导入依赖**：

<properties>  
 <!-- Spring 版本 -->  
 <spring.version>5.0.2.RELEASE</spring.version>  
</properties>  
  
<dependencies>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-web</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-webmvc</artifactId>  
 <version>${spring.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>javax.servlet</groupId>  
 <artifactId>servlet-api</artifactId>  
 <version>2.5</version>  
 <!-- provided：provided依赖范围只有在当JDK或者一个容器已提供该依赖之后才使用，provided依赖在编译和测试时需要，在运行时不需要。比如servlet api被tomcat容器提供 -->  
 <scope>provided</scope>  
 </dependency>  
 <dependency>  
 <groupId>javax.servlet.jsp</groupId>  
 <artifactId>jsp-api</artifactId>  
 <version>2.0</version>  
 <scope>provided</scope>  
 </dependency>  
</dependencies>

在resources目录下new>xml...file>spring config新建springmvc.xml配置文件

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:mvc="http://www.springframework.org/schema/mvc"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/mvc  
 http://www.springframework.org/schema/mvc/spring-mvc.xsd  
 http://www.springframework.org/schema/context  
 http://www.springframework.org/schema/context/spring-context.xsd">  
  
 <!-- 注解扫描的包 -->  
 <context:component-scan base-package="com.chq"/>  
  
 <!-- 配置视图解析器 -->  
 <bean id="internalResourceViewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
 <property name="prefix" value="/WEB-INF/pages/"/>  
 <property name="suffix" value=".jsp"/>  
 </bean>  
  
 <!-- 配置 Spring 开启注解 MVC 的支持 -->  
 <mvc:annotation-driven/>  
</beans>

在**web.xml下编写**

<!DOCTYPE web-app PUBLIC  
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"  
 "http://java.sun.com/dtd/web-app\_2\_3.dtd" >  
  
<web-app>  
 <display-name>Archetype Created Web Application</display-name>  
  
 <servlet>  
 <servlet-name>dispatcherServlet</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <init-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>classpath:springmvc.xml</param-value>  
 </init-param>  
 <!-- 服务器启动就创建该控制器，并且加载 spring 配置文件，创建容器 -->  
 <load-on-startup>1</load-on-startup>  
 </servlet>  
 <servlet-mapping>  
 <servlet-name>dispatcherServlet</servlet-name>  
 <url-pattern>/</url-pattern>  
 </servlet-mapping>  
</web-app>

在 webapp 目录下创建 index.jsp

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

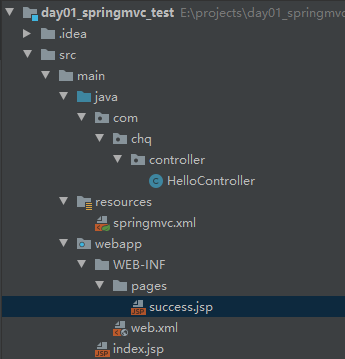
编写控制器 com.chq.controller.HelloController，并且使用注解配置

package com.chq.controller;  
  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
  
@Controller  
public class HelloController {  
 @RequestMapping(path = "/hello")  
 public String hello() {  
 System.*out*.println("Hello SpringMVC");  
 return "success"; // 被视图解析器解析后为 /WEB-INF/pages/success.jsp  
 }  
}

在 webapp/WEB-INF/pages 目录下创建 success.jsp，用于展示成功跳转页面

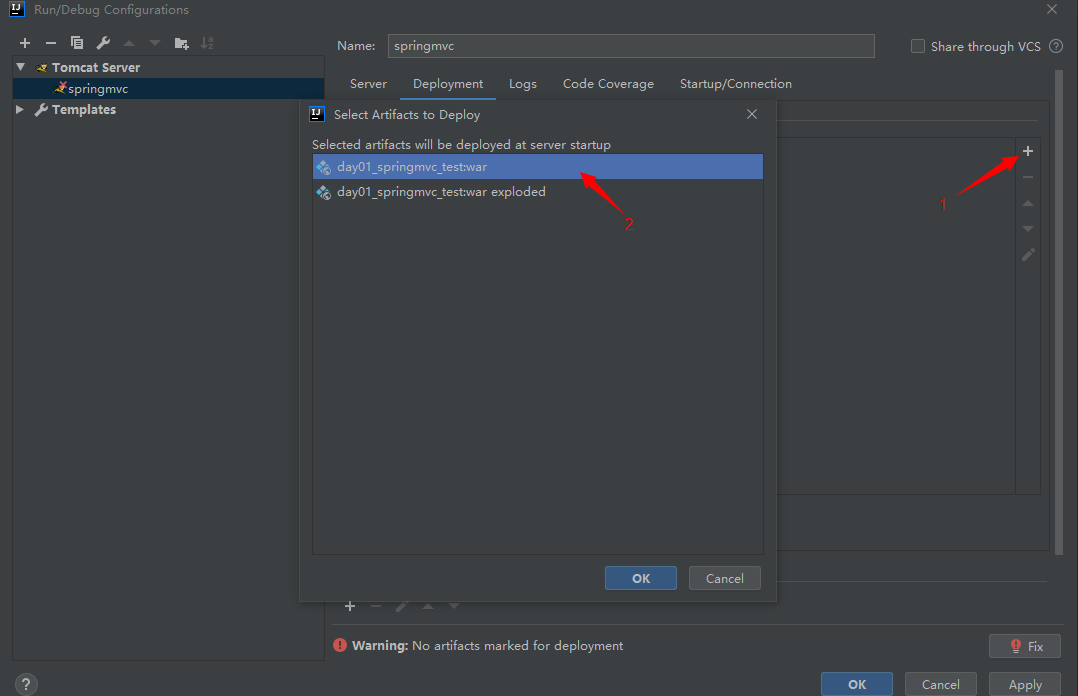
<%@ **page** contentType="text/html;charset=UTF-8" language="java" %>  
<html>  
<head>  
 <title>Title</title>  
</head>  
<body>  
<h3>访问成功</h3>  
</body>  
</html>

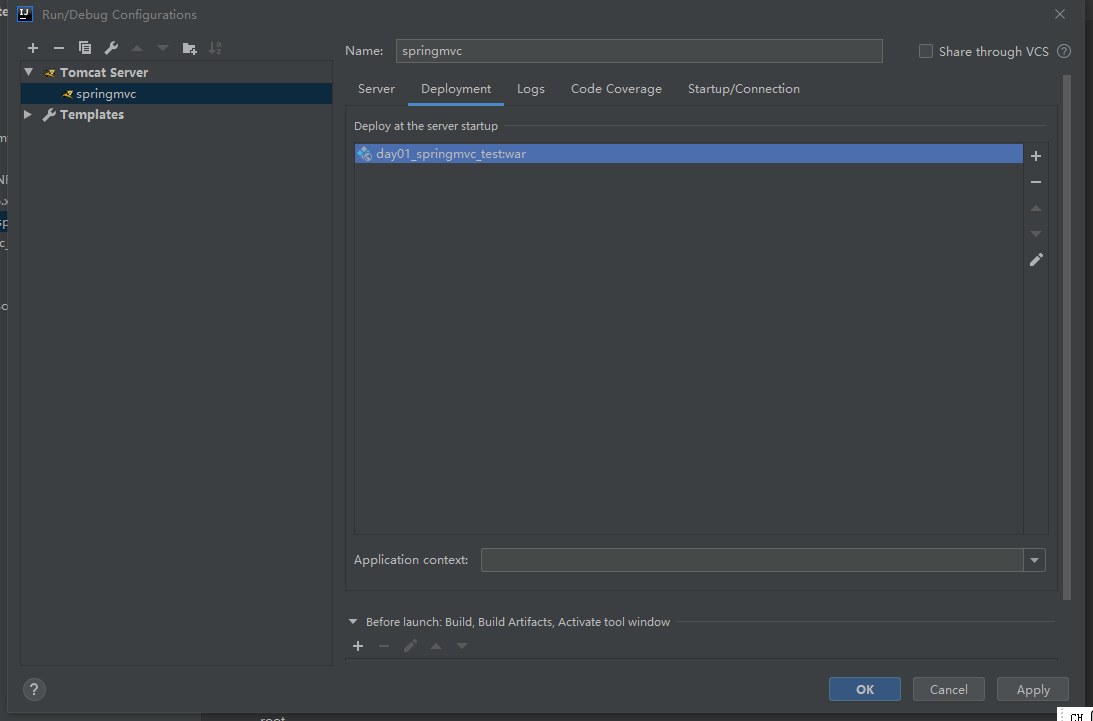
目录如下：



**配置Tomcat**

idea新添 add configuration>+>tomcat server>local





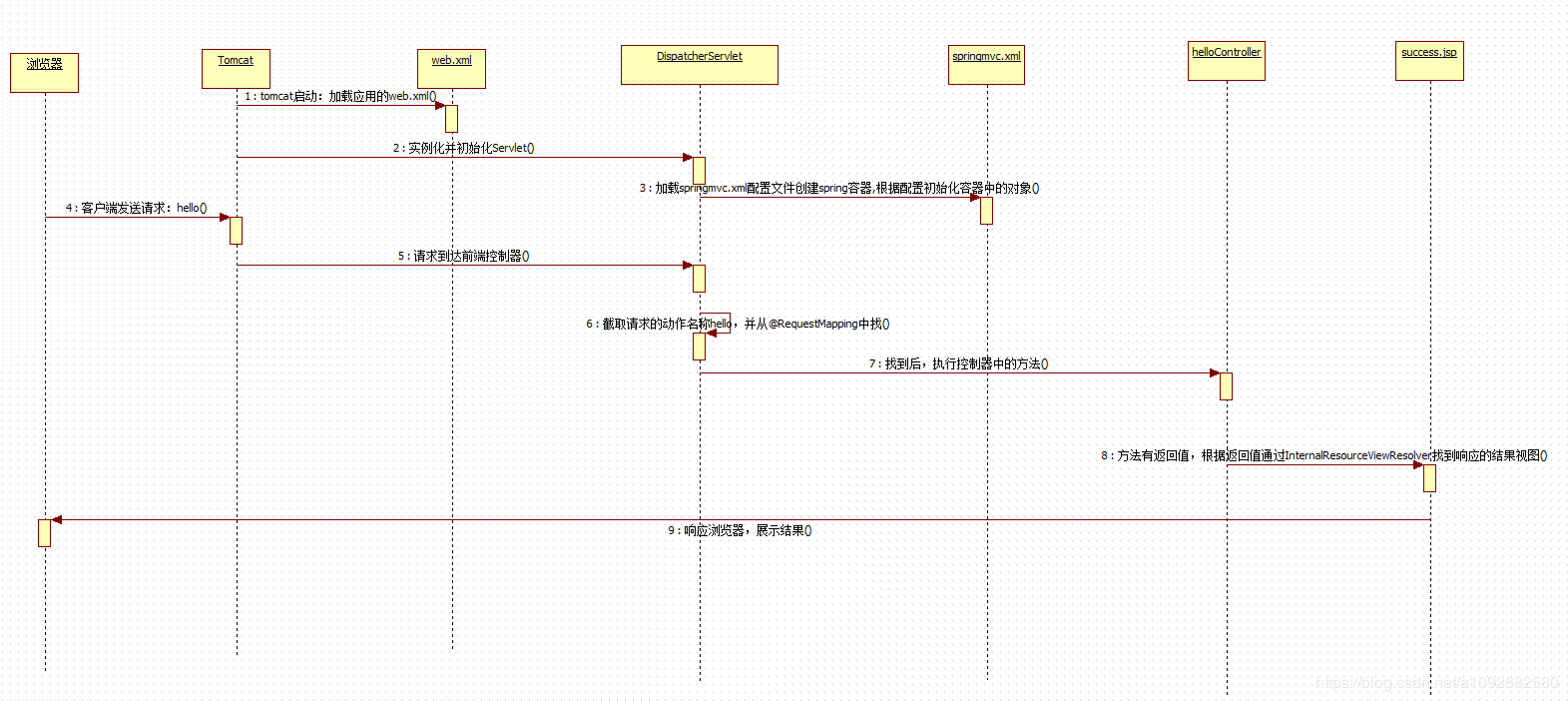
案例分析：

1.当 Tomcat 服务器启动的时候，应用被加载到 Tomcat 容器中。首先读取 web.xml，因为 DispatcherServlet 配置了 <load-on-startup> 标签，所以会在服务器一启动就被创建。同时因为 DispatcherServlet 指定了 Spring MVC的配置文件 springmvc.xml 的位置，所以会读取springmvc.xml，创建视图解析器 InternalResourceViewResolver 和控制器 HelloController 对象并加入到 IoC 容器中。（实际上并不止创建了这两个对象）

2.当 index.jsp 向服务器发送请求时，由于 DispatcherServlet 所映射的路径为 <url-pattern>/</url-patttern>，所以这个请求会先被 DispatcherServlet 所拦截到。但是该 Servlet 并不处理请求，而是根据我们请求的 URI 路径 /hello ，去匹配 @RequestMapping 绑定的路径。

3.根据 @RequestMapping 找到匹配的方法 hello() 后，执行并取得返回值 success 。借助视图解析器 InternalResouceViewResolver 得到最终的结果视图为 /WEB-INF/pages/success.jsp

4.Tomcat 服务器渲染页面，对客户端浏览器做出响应。



**@RequestMapping** 注解

@RequestMapping 用于建立请求 URL 路径和处理请求方法之间的对应关系。

出现的位置可以在类上或者方法上。

当 @RequestMapping 在类上时，相当于是 URL 路径的 一级目录，可以让我们的 URL 按照模块化管理。比如在入门案例的 HelloController 类上使用 @RequestMapping("/user") ，那么此时我们的请求路径就应该为 user/hello。

当 @RequestMapping 在方法上时，如果搭配类上的 @RequestMapping，那么就是 URL 路径的 二级目录；如果单独在方法上时，那么就是一级目录。

属性如下：

value/path ：这两个属性互为别名，都是用来指定请求的 URL 路径，

methods ：用于指定请求的方式，值为 RequestMethod.POST、RequestMethod.POST 等，出现多个值时为或的关系。

params ： 用于指定限制请求参数的条件，支持简单的表达式，要求请求参数的 key 和 value 必须和条件一模一样。譬如：

params={"username=ykf","arg"} 表示请求参数中必须要有参数 arg 和值为 ykf 的 username 参数（此时请求参数中这两个参数必须都要存在）。

params={"money!=100"} 表示请求参数中不能有值为 100 的 money 参数（此时请求参数中没有 money 也是可以的）。

headers：用于指定限制请求消息头的条件。

以上的限制条件属性出现多个时，为与的关系。

# 参数绑定

## 基本类型参数

当我们从表单提交数据到后台的时候，请求参数都是基于 key=value 的形式。

<a href="user/deleteUser?uid=1">删除用户</a>

@RequestMapping("/deleteUser")  
public String deleteUser(Integer uid){  
 System.*out*.println("删除了 id 为 " + uid+ " 的用户...");  
 return "success";  
}

前端的请求参数名称必须与控制器方法的形参名称保持一致（严格区分大小写

## POJO 类型作为参数

<form action="param/testParam2" method="post">  
 用户名: <input type="text" name="username"/><br/>  
 密码：<input type="text" name="password"/><br/>  
 金额：<input type="text" name="money"/><br/>  
<%-- 外部属性名 user 内部属性名 name，因此请求参数为 user.name --%>  
 真实姓名：<input type="text" name="user.name"/><br/>  
 年龄：<input type="text" name="user.age"/><br/>  
<input type="submit" value="提交">  
</form>

@RequestMapping("/testParam2")  
public String testParam3(Account account) {  
 System.*out*.println(account);  
 return "success";  
}

public class Account implements Serializable {  
 private String username;  
 private String password;  
 private Double money;  
 // 其他 JavaBean 对象引用  
 private User user;  
  
 // getters and setters  
 // ...  
}  
  
// 内部类  
public class User implements Serializable {  
 private String name;  
 private Integer age;  
  
 // getters and setters  
 // ...  
}

## 集合类型作为参数

<form action="param/testParam4" method="post">  
 用户名: <input type="text" name="username"/><br/>  
 密码：<input type="text" name="password"/><br/>  
 金额：<input type="text" name="money"/><br/>  
<%-- 绑定到 List 集合 --%>  
 持卡人1名称：<input type="text" name="userList[0].name"/><br/>  
 持卡人1年龄：<input type="text" name="userList[0].age"/><br/>  
 持卡人2名称：<input type="text" name="userList[1].name"/><br/>  
 持卡人2年龄：<input type="text" name="userList[1].age"/><br/>  
<%-- 绑定到 Map 集合 --%>  
 持卡人3名称：<input type="text" name="userMap['one'].name"/><br/>  
 持卡人3年龄：<input type="text" name="userMap['one'].age"/><br/>  
 持卡人4名称：<input type="text" name="userMap['two'].name"/><br/>  
 持卡人4年龄：<input type="text" name="userMap['two'].age"/><br/>  
<input type="submit" value="提交">  
</form>

@RequestMapping("/testParam4")  
public String testParam4(Account account) {  
 System.*out*.println(account);  
 return "success";  
}

public class Account implements Serializable {  
 private String username;  
 private String password;  
 private Double money;  
 // List 集合  
 private List<User> userList;  
 // Map 集合  
 private Map<String, User> userMap;  
  
 // getters and setters  
 // ...  
}

## 使用 Servlet 原生 API 作为参数

|  |
| --- |
| 类名 |
| javax.servlet.http.HttpServletRequest |
| javax.servlet.http.HttpServletResponse |
| javax.servlet.http.HttpSession |
| java.security.Principal |
| java.util.Locale |
| java.io.InputStream |
| java.io.OutputStream |
| java.io.Reader |
| java.io.Writer |

@RequestMapping("/testServlet")  
public String testServlet(HttpServletRequest request, HttpSession session, HttpServletResponse response) {  
 // ...  
}

# 常用注解

## @RequestParam

该注解的作用是把请求中指定名称的参数绑定到控制器方法的某个参数上。常用属性如下：

value / name ： 这两个属性互为别名，都是指定请求参数中的名称

required ： 请求参数中是否必须提供此参数，默认值为 true，表示必须提供，否则报错

<a href="${pageContext.request.contextPath}/anno/testRequestParam?name=鱼开饭">RequestParam</a>

@RequestMapping("/testRequestParam")  
public String testRequestParam(@RequestParam("name") String username) {  
 System.*out*.println(username);  
  
 return "success";  
}

此时使用 @RequestParam("name") ，那么如果请求参数中没有 name 参数，程序就会报错。同时，由于已经使用 @RequestParam 进行参数的绑定，所以控制器方法的参数名称 username 已经无法绑定了，也就是说，即使请求参数中有参数 username，那么也不会绑定到控制器方法的 username 参数上了

## @RequestBody

该注解的作用是获取请求体内容，得到的内容为 key1=value1&key2=value2...，只适用 POST 方式提交的请求，因为 GET 方式无请求体。属性如下：

required ： 是否必须有请求体，默认值为 true，表示必须有请求体，GET 方式请求会报错；当为 false 时，GET 方式请求得到的为 null

@RequestMapping("/testRequestBody")  
public String testRequestBody(@RequestBody String body) throws UnsupportedEncodingException {  
 System.*out*.println(body);  
 // 中文乱码  
 System.*out*.println(URLDecoder.*decode*(body, "UTF-8"));  
 return "success";  
}

## @PathVariable

该注解的作用是将 URL 中的占位符绑定到控制器方法的参数上，例如 localhost:8080/springmvc/user/{uid}，这里的 {uid} 就是一个占位符。属性如下：

value ： 指定 URL 中的占位符名称

required ：是否必须提供占位符

<a href="${pageContext.request.contextPath}/anno/testPathVariable/10001">PathVariable</a>

@RequestMapping("/testPathVariable/{uid}")  
public String testPathVariable(@PathVariable("uid") String id) {  
 System.*out*.println(id);  
 return "success";  
}

该注解可以用于绑定 REST 风格 URL 的占位符，restful 有以下特点：

资源（Resources）：每一个 URI 对应着一个资源

表现层（Representation）：把资源具体呈现出来的形式

状态转化（State Transfer）：客户端使用 GET、POST、PUT、DELETE 4个表示操作方式的动词对服务端资源进行操作：GET 用来获取资源，POST 用来新建资源（也可以用于更新资源），PUT 用来更新资源，DELETE 用来删除资源，譬如：

/account/1 HTTP GET ： 得到 id = 1 的 account

/account/1 HTTP DELETE： 删除 id = 1 的 account

/account/1 HTTP PUT： 更新 id = 1 的 account

/account HTTP POST： 新增 account

## @RequestHeader 和 @CookieValue

@RequestHeader 用于获取请求消息头。属性如下：

value ： 要获取的消息头名称

required ： 是否必须有此消息头

@CookieValue 用于获取 Cookie 的值。属性如下：

value ： 要获取的 Cookie名称

required ： 是否必须有此 Cookie

@RequestMapping("/testRequestHeader")  
public String testRequestHeader(@RequestHeader("Accept") String header){  
 System.*out*.println(header);  
 return "success";  
}  
  
@RequestMapping("/testCookieValue")  
public String testCookieValue(@CookieValue("JSESSIONID") String cookieValue){  
 System.*out*.println(cookieValue);  
 return "success";  
}

## @ModelAttribute

该注解用于修饰方法或者参数，如果修饰方法，那么该方法（有无返回值均可）会在控制器方法之前执行；如果修饰参数，可以获取指定的数据并赋值给参数。当表单提交的数据不是完整的实体类数据时，我们就可以借助该注解，对空属性进行预处理（譬如赋值为数据库记录中该对象原本的值）。属性如下：

value ： 用于获取数据的 key ，key 可以是 POJO 类的属性名称，也可以是 Map 集合的 key

## @SessionAttributes

该注解可以将数据存入 Session 域，使得数据在多次执行控制器方法的时候共享，只能作用在类上。属性如下：

value ： 用于指定存入的属性名称

type ： 用于指定存入的数据类型

<%-- SessionAttributes --%>  
<a href="${pageContext.request.contextPath}/anno/testPut">存入数据到 SessionAttribute</a>  
<a href="${pageContext.request.contextPath}/anno/testGet">从 SessionAttribute 取出数据</a>  
<a href="${pageContext.request.contextPath}/anno/testRemove">从 SessionAttribute 移除数据</a>

@Controller  
@RequestMapping("/anno")  
// 指定要存入 session 中的数据  
@SessionAttributes({"msg"})  
public class AnnotationController {  
 */\*\*  
 \* 存数据到 session  
 \*  
 \** ***@param*** *model  
 \** ***@return*** *\*/* @RequestMapping("/testPut")  
 public String testPut(Model model) {  
 model.addAttribute("msg", "存入的数据");  
 return "success";  
 }  
  
 */\*\*  
 \* 从 session 中取数据  
 \*  
 \** ***@param*** *model  
 \** ***@return*** *\*/* @RequestMapping("/testGet")  
 public String testGet(ModelMap model) {  
 System.*out*.println(model.get("msg"));  
 return "success";  
 }  
  
 */\*\*  
 \* 从 session 中移除数据  
 \*  
 \** ***@param*** *status  
 \** ***@return*** *\*/* @RequestMapping("/testRemove")  
 public String testRemove(SessionStatus status) {  
 status.setComplete();  
 return "success";  
 }  
}

# 响应数据和结果视图

## 使用 @RequestBody 和 @ResponseBody 进行 json 交互

在开发中，很多时候都是客户端提交 json 数据，服务器接收 json 数据并解析，然后生成 json 响应给客户端。这个时候我们就可以使用 @RequestBody 和 @ResponseBody 来完成 json 数据的交互。见代码

<%@ page contentType="text/html;charset=UTF-8" language="java" %>  
<html>  
<head>  
<title>Json 数据交互</title>  
<script src="js/jquery-1.8.2.min.js"></script>  
<script type="text/javascript">  
 // 页面加载  
 $(function () {  
 // 绑定单击事件  
 $("#btn").click(function () {  
 // 发送ajax请求  
 $.ajax({  
 url: "${pageContext.request.contextPath}/user/testAjax",  
 // 发送数据的 MIME 类型  
 contentType: "application/json;charset=utf-8",  
 // 发送数据为一个 json 串  
 data: '{"username":"鱼开饭","password":"123456","age":22}',  
 // 返回的数据类型  
 dataType: "json",  
 // 请求方式  
 type: "post",  
 // 成功回调函数  
 success: function (data) {  
 console.log(data);  
 console.log(data.username);  
 console.log(data.password);  
 console.log(data.age);  
 }  
 });  
 });  
 });  
</script>  
</head>  
<body>  
<button id="btn">测试 ajax 请求 json 和响应 json</button>  
</body>  
</html>

因为我们配置了前端控制器 DispatcherServlet ，因此对于所有的请求都会进行拦截，包括静态资源。为了使引入的 jquery.js 不被拦截，我们需要在配置文件 springmvc.xml 中配置不拦截静态资源

<!-- 设置静态资源不过滤 -->

<mvc:resources location="/css/" mapping="/css/\*\*"/> <!-- 样式 -->

<mvc:resources location="/images/" mapping="/images/\*\*"/> <!-- 图片 -->

<mvc:resources location="/js/" mapping="/js/\*\*"/> <!-- javascript -->

location 属性 ： 指定 webapp 目录下的包

mapping 属性 ： 表示以 /static 开头的所有请求路径

@RequestMapping("/testAjax")

@ResponseBody  
public User testAjax(@RequestBody User user) {  
 System.*out*.println("testAjax方法执行了...");  
 System.*out*.println(user);  
  
 // 模拟查询出新的用户  
 user.setPassword("654321");  
 user.setAge(10);  
  
 return user;  
}

客户端发送了一个 json 串，服务器使用 @RequestBody 获取请求体内容，并使用 jackson 将 json 串封装到 User 对象中；服务器使用 @ResponseBody 生成响应，通过 jackson 将 User 对象转成 json 串。

SpringMVC 默认使用 MappingJacksonHttpMessageConverter 进行转换，因此需要导入 jackson 的依赖

<dependency>  
<groupId>com.fasterxml.jackson.core</groupId>  
<artifactId>jackson-databind</artifactId>  
<version>2.9.0</version>  
</dependency>

使用 jackson 进行 json 和 JavaBean 对象的自动转换时，需要保证 json 串的键与 JavaBean 的属性名称一一对应

# SpringMVC 方式文件上传

我们只需要配置一个文件解析器，由文件解析器来替我们解析，然后我们在控制器使用 MultipartFile 对象接收即可（该对象表示要上传的文件）。

导入依赖

<dependency>  
 <groupId>commons-fileupload</groupId>  
 <artifactId>commons-fileupload</artifactId>  
 <version>1.3.1</version>  
</dependency>

配置文件解析器

<!-- 配置文件解析器 -->  
<bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">  
 <!-- 配置上传文件最大大小，字节（Byte）为单位。这里设置 10MB=10\*1024KB=10\*1024\*1024B -->  
 <property name="maxUploadSize" value="10485760"/>  
 <!-- 设置编码，防止中文乱码 -->  
 <property name="defaultEncoding" value="utf-8"/>  
</bean>

编写 jsp 代码

<form action="/fileupload/test1" method="post" enctype="multipart/form-data">  
 选择文件：<input type="file" name="upload"/><br/>  
 <input type="submit" value="上传文件">  
</form>

编写控制器代码

package com.chq.controller;  
  
import org.springframework.http.HttpRequest;  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.multipart.MultipartFile;  
  
import javax.servlet.http.HttpServletRequest;  
import java.io.File;  
import java.io.IOException;  
import java.text.SimpleDateFormat;  
import java.util.Date;  
  
@Controller  
@RequestMapping(path = "fileupload")  
public class FileuploadController {  
  
 @RequestMapping(path = "/test1")  
 public String hello(HttpServletRequest request, MultipartFile upload) throws IOException {  
 System.*out*.println("SpringMVC方式文件上传...");  
 // 获取要保存的真实目录路径  
 String path = request.getSession().getServletContext().getRealPath("/uploads");  
 // 判断文件夹是否存在，不存在则创建  
 File file = new File(path);  
 if (!file.exists()) {  
 file.mkdirs();  
 }  
 // 获取文件原始名  
 String filename = upload.getOriginalFilename();  
 // 加时间戳  
 filename = new SimpleDateFormat("yyyyMMddHHmmssSSS").format(new Date()) + "\_" + filename;  
 // 上传文件  
 upload.transferTo(new File(path, filename));  
 System.*out*.println(path);  
 return "success";  
 }  
  
}

使用该方式上传文件时，要确保 MultipartFile 对象名称与表单中 file 组件的名称保持一致。如果不一致，则需要使用 @RequestParam 进行参数的绑定。譬如表单为：<input type="file" name="file"/>，则控制器方法应该为 test2(HttpServletRequest request, @RequestParam("file") MultipartFile upload)

获取上传文件的文件名时，应该使用 getOriginalFilename() ，不能使用 getName() ，getName() 是获取表单中 file 组件的名称。

如果要想实现多文件上传，那么需要在 file 组件加上属性 multiple="multiple"（HTML5 的新属性），然后控制器用 List<MultipartFile> 接收即可。

# 拦截器

## 说明

SpringMVC 中的 Interceptor（拦截器） 类似于 Servlet 中的 Filter（过滤器），拦截器是用于对 Controller（处理器）进行预处理和后处理的。和过滤器类似，过滤器可以定义过滤链，拦截器也可以定义拦截器链（Interceptor Chain），也就是将拦截器按一定的顺序联结成一条链，在访问被拦截的方法之前，拦截器链中的拦截器就会按之前定义的顺序进行拦截。

过滤器和拦截器虽然比较类似，但是还是有区别的：

过滤器是 Servlet 规范中的，也就是说，只要是 JavaWeb 工程就可以使用；而拦截器是 SpringMVC 框架的，只有使用了 SpringMVC 框架的工程才可以使用

过滤器可以拦截所有的资源，也就是不仅可以拦截 Controller ，其它的所有资源（像 jsp、js 这些静态资源）它都可以拦截；而拦截器是针对 Controller 的，它只能拦截控制器。

拦截器是 AOP 思想的一种实现方式

如果想自定义拦截器，那么就需要实现 HandlerInterceptor 接口，该接口中有 3 个 default 方法：

boolean preHandle(..) ： 该方法在 Controller 方法执行前调用。如果返回 true 就代表放行，执行下一个拦截器，如果没有拦截器，那就执行 Controller 的方法；如果返回 false 就代表不放行，此时可以使用转发或者重定向跳转到指定的页面。该方法是按拦截器定义顺序进行调用的。

void postHandle(..) ： 该方法在 Controller 方法执行后（但是在视图渲染之前）调用。此时我们可以通过 ModelAndView （模型和视图对象）来对模型数据或视图进行处理，也可以通过 HttpServletRequest 和 HttpServletResponse 进行转发或者重定向。该方法是按拦截器定义逆序进行调用的。

void afterCompletion(..) ： 该方法在 整个请求处理完毕后调用（也就是在视图渲染完毕后）。 可以在该方法中进行一些资源清理的操作（不能在该方法中进行页面的跳转了），相当于 try...catch...finally 中的 finally。该方法是按拦截器定义逆序进行调用的，而且只有 preHandle() 返回 true 才调用。

## 操作步骤

按照入门案例先创建。。

创建自定义拦截器，实现 Interceptor 接口并重写其中的方法

package com.chq.springmvc.interceptor;  
  
import org.springframework.web.servlet.HandlerInterceptor;  
import org.springframework.web.servlet.ModelAndView;  
  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
  
public class MyInterceptor1 implements HandlerInterceptor {  
 @Override  
 public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  
 System.*out*.println("第 1 个拦截器中的preHandle方法执行了...");  
 return true;  
 }  
  
 @Override  
 public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {  
 System.*out*.println("第 1 个拦截器中的postHandle方法执行了...");  
 }  
  
 @Override  
 public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {  
 System.*out*.println("第 1 个拦截器中的afterCompletion方法执行了...");  
 }  
}

在 SpringMVC 配置文件中配置拦截器

<!-- 配置拦截器链 -->  
<mvc:interceptors>  
 <!-- 配置第一个拦截器 -->  
 <mvc:interceptor>  
 <!-- 要拦截的路径 -->  
 <mvc:mapping path="/user/\*"/>  
 <!-- 不拦截的路径 -->  
 <!--<mvc:exclude-mapping path=""/>-->  
 <!-- 注册拦截器对象 -->  
 <bean class="com.chq.springmvc.interceptor.MyInterceptor1"/>  
 </mvc:interceptor>  
</mvc:interceptors>

<mvc:mapping> 标签用于指定要拦截的路径，而 <mvc:exclude-mapping> 标签用于指定不拦截的路径，两者选其一即可。

编写控制器代码

package com.chq.springmvc;  
  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
  
@Controller  
@RequestMapping("/user")  
public class TestInterceptor {  
 @RequestMapping("testInterceptor")  
 public String testInterceptor(){  
 System.*out*.println("testInterceptor方法执行了...");  
 return "success";  
 }  
}

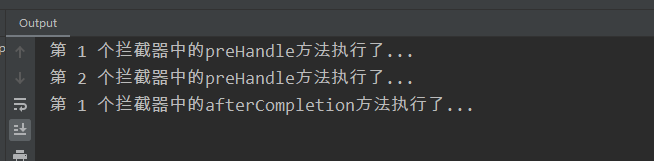
编写jsp代码

<%@ **page** contentType="text/html;charset=UTF-8" language="java" %>  
<html>  
<head>  
 <title>Title</title>  
</head>  
<body>  
<h3>执行成功</h3>  
**<%** System.*out*.println("success.jsp 执行了");  
**%>**</body>  
</html>

## 特殊情况

如果有多个拦截器，此时拦截器 1 的 preHandle 方法返回 true，但是拦截器 2 的 preHandle 方法返回 false，那么拦截器 1 的 afterCompletion 的方法是否执行？

因为我们上面说过，afterCompletion 方法只有当 preHandle 返回 true 的时候执行。所以，拦截器 1 的 afterCompletion 方法会执行，而拦截器 2 的 afterCompletion 方法不会执行。



如果某个拦截器在 postHandle 方法中使用 request 和 response 进行转发，那么最终的结果视图（这里是 success.jsp）还会不会执行？

我们上面说过，postHandle 方法是在控制器执行后，视图渲染前被调用。所以，当我们转发到某个页面时，浏览器会把这个页面展示出来，但是 postHandle 方法还是会继续执行，也就是说 success.jsp 还是会被执行到，只不过不会再被渲染展示了。

