



SpringMVC 第二天

第1章 ModelAttribute 和 SessionAttribute[应用]

1.1 ModelAttribute

1.1.1 使用说明

作用：

该注解是 SpringMVC4.3 版本以后新加入的。它可以用于修饰方法和参数。

出现在方法上，表示当前方法会在控制器的方法执行之前，先执行。它可以修饰没有返回值的方法，也可以修饰有具体返回值的方法。

出现在参数上，获取指定的数据给参数赋值。

属性：

value：用于获取数据的 key。key 可以是 POJO 的属性名称，也可以是 map 结构的 key。

应用场景：

当表单提交数据不是完整的实体类数据时，保证没有提交数据的字段使用数据库对象原来的数据。

例如：

我们在编辑一个用户时，用户有一个创建信息字段，该字段的值是不允许被修改的。在提交表单数据是肯定没有此字段的内容，一旦更新会把该字段内容置为 null，此时就可以使用此注解解决问题。

1.1.2 使用示例

1.1.2.1 基于 POJO 属性的基本使用：

jps 代码：

```
<!-- ModelAttribute 注解的基本使用 -->
```

```
<a href="springmvc/testModelAttribute?username=test">测试 modelattribute</a>
```

控制器代码：

```
/**
```

```
 * 被 ModelAttribute 修饰的方法
```

```
 * @param user
```

```
 */
```

```
@ModelAttribute
```

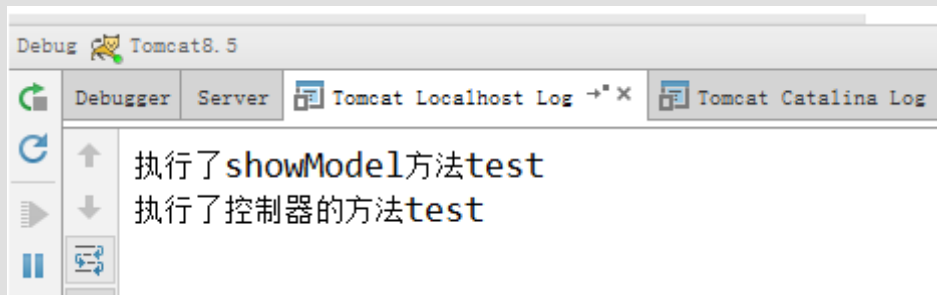
```
public void showModel(User user) {
```



```
        System.out.println("执行了 showModel 方法"+user.getUsername());
    }

    /**
     * 接收请求的方法
     * @param user
     * @return
     */
    @RequestMapping("/testModelAttribute")
    public String testModelAttribute(User user) {
        System.out.println("执行了控制器的方法"+user.getUsername());
        return "success";
    }
}
```

运行结果:



1.1.2.2 基于 Map 的应用场景示例 1: ModelAttribute 修饰方法带返回值

需求:

修改用户信息，要求用户的密码不能修改

jsp 的代码:

<!-- 修改用户信息 -->

```
<form action="springmvc/updateUser" method="post">
    用户名: <input type="text" name="username" ><br/>
    用户年龄: <input type="text" name="age" ><br/>
    <input type="submit" value="保存">
</form>
```

控制的代码:

```
/**
 * 查询数据库中用户信息
 * @param user
 */
@ModelAttribute
public User showModel(String username) {
    //模拟去数据库查询
    User abc = findUserByName(username);
    System.out.println("执行了 showModel 方法"+abc);
}
```



```
        return abc;
    }

    /**
     * 模拟修改用户方法
     * @param user
     * @return
     */
    @RequestMapping("/updateUser")
    public String testModelAttribute(User user) {
        System.out.println("控制器中处理请求的方法：修改用户：" + user);
        return "success";
    }

    /**
     * 模拟去数据库查询
     * @param username
     * @return
     */
    private User findUserByName(String username) {
        User user = new User();
        user.setUsername(username);
        user.setAge(19);
        user.setPassword("123456");
        return user;
    }
}
```

运行结果：



1.1.2.3 基于 Map 的应用场景示例 1：ModelAttribute 修饰方法不带返回值

需求：

修改用户信息，要求用户的密码不能修改

jsp 中的代码：



<!-- 修改用户信息 -->

```
<form action="springmvc/updateUser" method="post">
    用户名称: <input type="text" name="username" ><br/>
    用户年龄: <input type="text" name="age" ><br/>
    <input type="submit" value="保存">
</form>
```

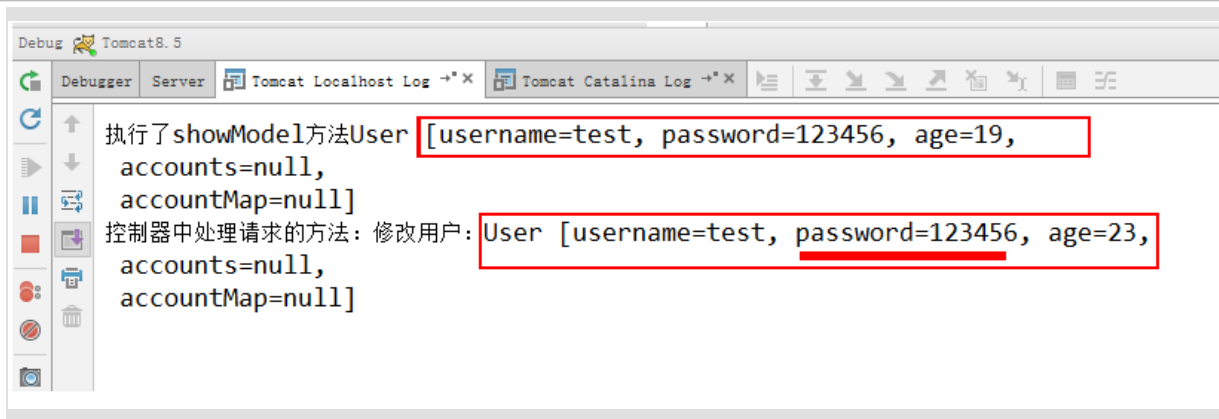
控制器中的代码:

```
/**
 * 查询数据库中用户信息
 * @param user
 */
@RequestMapping("/updateUser")
public void showModel(String username, Map<String, User> map) {
    //模拟去数据库查询
    User user = findUserByName(username);
    System.out.println("执行了 showModel 方法"+user);
    map.put("abc", user);
}

/**
 * 模拟修改用户方法
 * @param user
 * @return
 */
@RequestMapping("/updateUser")
public String testModelAttribute(@ModelAttribute("abc") User user) {
    System.out.println("控制器中处理请求的方法: 修改用户: "+user);
    return "success";
}

/**
 * 模拟去数据库查询
 * @param username
 * @return
 */
private User findUserByName(String username) {
    User user = new User();
    user.setUsername(username);
    user.setAge(19);
    user.setPassword("123456");
    return user;
}
```

运行结果:



1.2 SessionAttribute

1.2.1 使用说明

作用：

用于多次执行控制器方法间的参数共享。

属性：

value：用于指定存入的属性名称

type：用于指定存入的数据类型。

1.2.2 使用示例

jsp 中的代码：

```
<!-- SessionAttribute 注解的使用 -->
<a href="springmvc/testPut">存入 SessionAttribute</a>
<hr/>
<a href="springmvc/testGet">取出 SessionAttribute</a>
<hr/>
<a href="springmvc/testClean">清除 SessionAttribute</a>
```

控制器中的代码：

```
/**
 * SessionAttribute 注解的使用
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
@Controller("sessionAttributeController")
@RequestMapping("/springmvc")
@SessionAttributes(value = {"username", "password"}, types = {Integer.class})
public class SessionAttributeController {
```



```
/**
 * 把数据存入 SessionAttribute
 * @param model
 * @return
 * Model 是 spring 提供的一个接口，该接口有一个实现类 ExtendedModelMap
 * 该类继承了 ModelMap，而 ModelMap 就是 LinkedHashMap 子类
 */
@RequestMapping("/testPut")
public String testPut(Model model){
    model.addAttribute("username", "泰斯特");
    model.addAttribute("password", "123456");
    model.addAttribute("age", 31);
    //跳转之前将数据保存到 username、password 和 age 中，因为注解@SessionAttribute 中有
    这几个参数

    return "success";
}

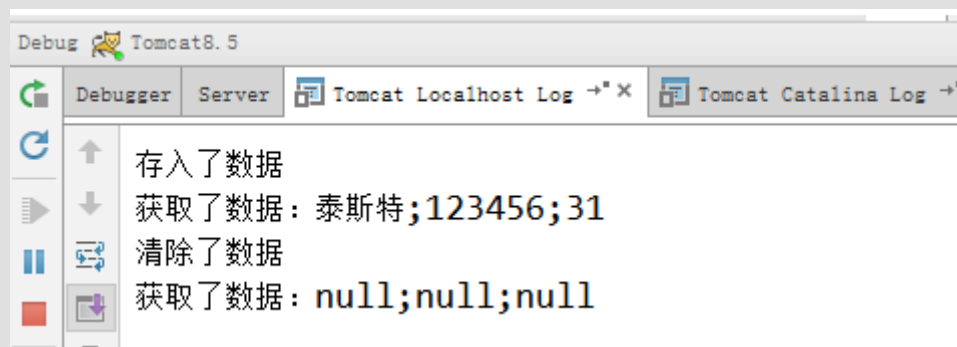
@RequestMapping("/testGet")
public String testGet(ModelMap model){

    System.out.println(model.get("username")+";"+model.get("password")+";"+model.get("a
    ge"));

    return "success";
}

@RequestMapping("/testClean")
public String complete(SessionStatus sessionStatus){
    sessionStatus.setComplete();
    return "success";
}
}
```

运行结果:





第2章 Restful 风格的 URL[应用]

2.1 概述

2.1.1 什么是 rest:

REST (英文: Representational State Transfer, 简称 REST) 描述了一个架构样式的网络系统, 比如 web 应用程序。它首次出现在 2000 年 Roy Fielding 的博士论文中, 他是 HTTP 规范的主要编写者之一。在目前主流的三种 Web 服务交互方案中, REST 相比于 SOAP (Simple Object Access protocol, 简单对象访问协议) 以及 XML-RPC 更加简单明了, 无论是对 URL 的处理还是对 Payload 的编码, REST 都倾向于用更加简单轻量的方法设计和实现。值得注意的是 REST 并没有一个明确的标准, 而更像是一种设计的风格。

它本身并没有什么实用性, 其核心价值在于如何设计出符合 REST 风格的网络接口。

2.1.2 restful 的优点

它结构清晰、符合标准、易于理解、扩展方便, 所以正得到越来越多网站的采用。

2.1.3 restful 的特性

资源 (Resources): 网络上的一个实体, 或者说是网络上的一个具体信息。

它可以是一段文本、一张图片、一首歌曲、一种服务, 总之就是一个具体的存在。可以用一个 URI (统一资源定位符) 指向它, 每种资源对应一个特定的 URI。要

获取这个资源, 访问它的 URI 就可以, 因此 **URI** 即为每一个资源的独一无二的识别符。

表现层 (Representation): 把资源具体呈现出来的形式, 叫做它的表现层 (Representation)。

比如, 文本可以用 txt 格式表现, 也可以用 HTML 格式、XML 格式、JSON 格式表现, 甚至可以采用二进制格式。

状态转化 (State Transfer): 每发出一个请求, 就代表了客户端和服务器的一个交互过程。

HTTP 协议, 是一个无状态协议, 即所有的状态都保存在服务器端。因此, 如果客户端想要操作服务器, 必须通过某种手段, 让服务器端发生“状态转化” (State Transfer)。而这种转化是建立在表现层之上的, 所以就是“表现层状态转化”。具体说, 就是 HTTP 协议里面, 四个表示操作方式的动词: **GET**、**POST**、**PUT**、**DELETE**。它们分别对应四种基本操作: **GET** 用来获取资源, **POST** 用来新建资源, **PUT** 用来更新资源, **DELETE** 用来删除资源。

restful 的示例:

/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



2.2 PathVariable 注解在 rest 风格 url 中的应用

2.2.1 使用说明

作用：

用于绑定 url 中的占位符。例如：请求 url 中 /delete/{id}，这个{id}就是 url 占位符。
url 支持占位符是 spring3.0 之后加入的。是 springmvc 支持 rest 风格 URL 的一个重要标志。

属性：

value：用于指定 url 中占位符名称。
required：是否必须提供占位符。

2.2.2 使用示例

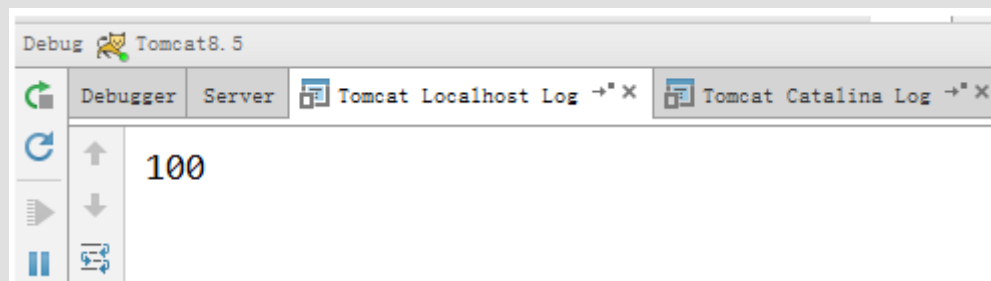
jsp 代码：

```
<!-- PathVariable 注解 -->
<a href="springmvc/usePathVariable/100">pathVariable 注解</a>
```

控制器代码：

```
/**
 * PathVariable 注解
 * @param user
 * @return
 */
@RequestMapping("/usePathVariable/{id}")
public String usePathVariable(@PathVariable("id") Integer id){
    System.out.println(id);
    return "success";
}
```

运行结果：



2.3 基于 HiddenHttpMethodFilter 的示例

作用：

由于浏览器 form 表单只支持 GET 与 POST 请求，而 DELETE、PUT 等 method 并不支持，Spring3.0 添加了一个过滤器，可以将浏览器请求改为指定的请求方式，发送给我们的控制器方法，使得支持 GET、POST、PUT



与 DELETE 请求。

使用方法:

第一步: 在 web.xml 中配置该过滤器。

第二步: 请求方式必须使用 post 请求。

第三步: 按照要求提供_method 请求参数, 该参数的取值就是我们需要的请求方式。

源码分析:

```
/** Default method parameter: {@code _method} */
public static final String DEFAULT_METHOD_PARAM = "_method";

private String methodParam = DEFAULT_METHOD_PARAM;

/**
 * Set the parameter name to look for HTTP methods.
 * @see #DEFAULT_METHOD_PARAM
 */
public void setMethodParam(String methodParam) {
    Assert.hasText(methodParam, "'methodParam' must not be empty");
    this.methodParam = methodParam;
}

@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
    throws ServletException, IOException {

    HttpServletRequest requestToUse = request;

    if ("POST".equals(request.getMethod()) && request.getAttribute(WebUtils.ERROR_EXCEPTION_ATTRIBUTE) == null) {
        String paramValue = request.getParameter(this.methodParam);
        if (StringUtils.hasLength(paramValue)) {
            requestToUse = new HttpMethodRequestWrapper(request, paramValue);
        }
    }

    filterChain.doFilter(requestToUse, response);
}
```

获取请求参数

把request转成固定请求方式的request对象

jsp 中示例代码:

```
<!-- 保存 -->
<form action="springmvc/testRestPOST" method="post">
    用户名称: <input type="text" name="username"><br/>
    <!-- <input type="hidden" name="_method" value="POST"> -->
    <input type="submit" value="保存">
</form>
<hr/>

<!-- 更新 -->
<form action="springmvc/testRestPUT/1" method="post">
    用户名称: <input type="text" name="username"><br/>
    <input type="hidden" name="_method" value="PUT">
    <input type="submit" value="更新">
</form>
<hr/>

<!-- 删除 -->
<form action="springmvc/testRestDELETE/1" method="post">
    <input type="hidden" name="_method" value="DELETE">
    <input type="submit" value="删除">
</form>
<hr/>
```



<!-- 查询一个 -->

```
<form action="springmvc/testRestGET/1" method="post">
    <input type="hidden" name="_method" value="GET">
    <input type="submit" value="查询">
</form>
```

控制器中示例代码:

```
/**
 * post 请求: 保存
 * @param username
 * @return
 */
@RequestMapping(value="/testRestPOST",method=RequestMethod.POST)
public String testRestfulURLPOST(User user){
    System.out.println("rest post"+user);
    return "success";
}

/**
 * put 请求: 更新
 * @param username
 * @return
 */
@RequestMapping(value="/testRestPUT/{id}",method=RequestMethod.PUT)
public String testRestfulURLPUT(@PathVariable("id") Integer id,User user){
    System.out.println("rest put "+id+", "+user);
    return "success";
}

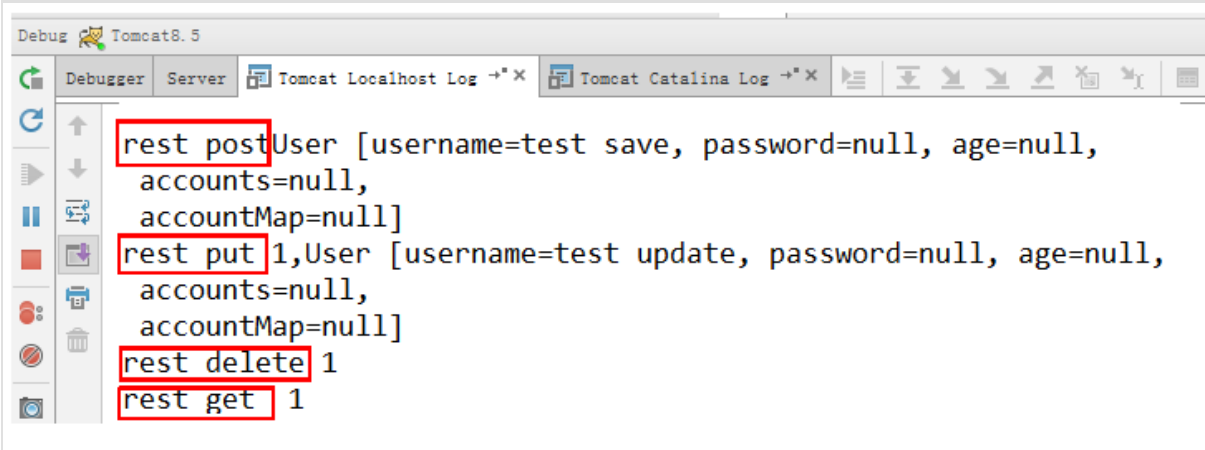
/**
 * post 请求: 删除
 * @param username
 * @return
 */
@RequestMapping(value="/testRestDELETE/{id}",method=RequestMethod.DELETE)
public String testRestfulURLDELETE(@PathVariable("id") Integer id){
    System.out.println("rest delete "+id);
    return "success";
}

/**
 * post 请求: 查询
 * @param username
```



```
* @return
*/
@RequestMapping(value="/testRestGET/{id}",method=RequestMethod.GET)
public String testRestfulURLGET(@PathVariable("id") Integer id){
    System.out.println("rest get "+id);
    return "success";
}
```

运行结果:



第3章 控制器方法的返回值[掌握]

3.1 返回值分类

3.1.1 字符串

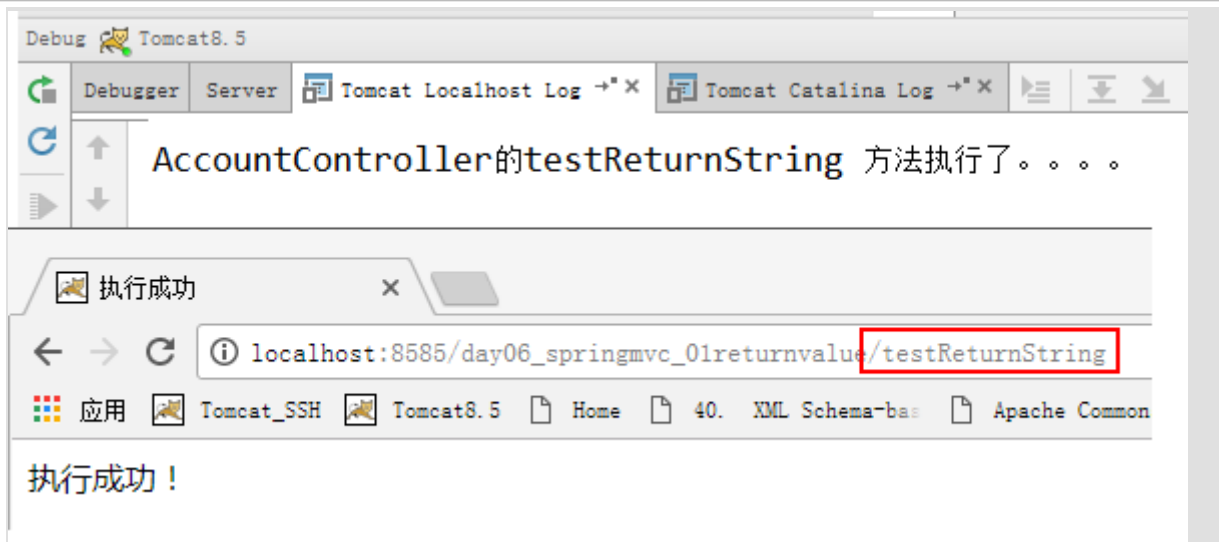
controller 方法返回字符串可以指定逻辑视图名，通过视图解析器解析为物理视图地址。

//指定逻辑视图名，经过视图解析器解析为 jsp 物理路径: /WEB-INF/pages/success.jsp

```
@RequestMapping("/testReturnString")
```

```
public String testReturnString() {
    System.out.println("AccountController 的 testReturnString 方法执行了。。。");
    return "success";
}
```

运行结果:



3.1.2 void

在昨天的学习中，我们知道 Servlet 原始 API 可以作为控制器中方法的参数：

```
@RequestMapping("/testReturnVoid")  
  
public void testReturnVoid(HttpServletRequest request, HttpServletResponse response)  
throws Exception {  
}
```

在 controller 方法形参上可以定义 request 和 response，使用 request 或 response 指定响应结果：

1、使用 request 转向页面，如下：

```
request.getRequestDispatcher("/WEB-INF/pages/success.jsp").forward(request,  
response);
```

2、也可以通过 response 页面重定向：

```
response.sendRedirect("testReturnString")
```

3、也可以通过 response 指定响应结果，例如响应 json 数据：

```
response.setCharacterEncoding("utf-8");  
response.setContentType("application/json;charset=utf-8");  
response.getWriter().write("json 串");
```

3.1.3 ModelAndView

ModelAndView 是 SpringMVC 为我们提供的一个对象，该对象也可以用作控制器方法的返回值。
该对象中有两个方法：



```
/**
 * Add an attribute to the model.
 * @param attributeName name of the object to add to the model
 * @param attributeValue object to add to the model (never {@code null})
 * @see ModelMap#addAttribute(String, Object)
 * @see #getModelMap() 添加模型到该对象中，通过源码分析可以看出，和昨天我们讲的请求参数封装中用到的对象是同一个
 */
public ModelAndView addObject(String attributeName, Object attributeValue) {
    getModelMap().addAttribute(attributeName, attributeValue);
    return this; 我们在页面上可以直接用el表达式获取。 获取方式: ${attributeName}
}
```

```
/**
 * Set a view name for this ModelAndView, to be resolved by the
 * DispatcherServlet via a ViewResolver. Will override any
 * pre-existing view name or View.
 */ 用于设置逻辑视图名称，视图解析器会根据名称前往指定的视图。
public void setViewName(@Nullable String viewName) {
    this.view = viewName;
}
```

示例代码:

```
/**
 * 返回 ModelAndView
 * @return
 */
@RequestMapping("/testReturnModelAndView")
public ModelAndView testReturnModelAndView() {
    ModelAndView mv = new ModelAndView();
    mv.addObject("username", "张三");
    mv.setViewName("success");
    return mv;
}
```

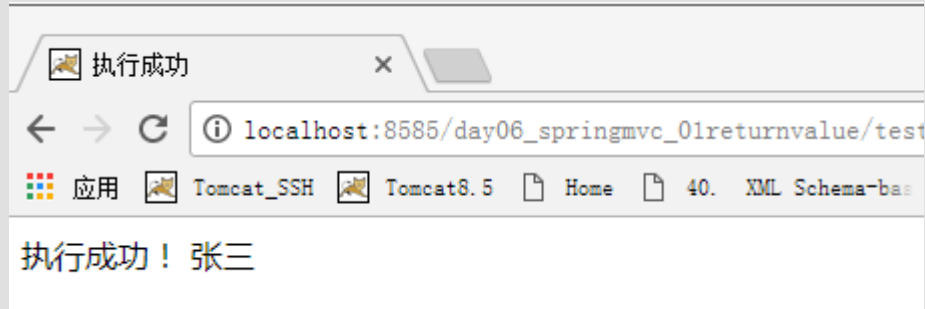
响应的 jsp 代码:

```
<%@ 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>执行成功</title>
</head>
<body>
执行成功!
```



```
${requestScope.username}  
</body>  
</html>
```

输出结果:



注意:

我们在页面上获取使用的是 `requestScope.username` 取的，所以返回 `ModelAndView` 类型时，浏览器跳转只能是请求转发。

3.2 转发和重定向

3.2.1 forward 转发

`controller` 方法在提供了 `String` 类型的返回值之后，默认就是请求转发。我们也可以写成：

```
/**  
 * 转发  
 * @return  
 */  
  
@RequestMapping("/testForward")  
public String testForward() {  
    System.out.println("AccountController 的 testForward 方法执行了。。。");  
    return "forward:/WEB-INF/pages/success.jsp";  
}
```

需要注意的是，如果用了 **forward**，则路径必须写成实际视图 url，不能写逻辑视图。

它相当于 “`request.getRequestDispatcher("url").forward(request, response)`”。使用请求转发，既可以转发到 `jsp`，也可以转发到其他的控制器方法。

3.2.2 Redirect 重定向

`controller` 方法提供了一个 `String` 类型返回值之后，它需要在返回值里使用 **redirect**：

```
/**  
 * 重定向  
 * @return  
 */  
  
@RequestMapping("/testRedirect")
```



```
public String testRedirect() {  
    System.out.println("AccountController 的 testRedirect 方法执行了。。。");  
    return "redirect:testReturnModelAndView";  
}
```

它相当于“`response.sendRedirect(url)`”。需要注意的是，如果是重定向到 jsp 页面，则 jsp 页面不能写在 WEB-INF 目录中，否则无法找到。

第4章 交互 json 数据[应用]

4.1RequestBody

4.1.1 使用说明

作用：

用于获取请求体内容。直接使用得到是 `key=value&key=value...` 结构的数据。
get 请求方式不适用。

属性：

`required`：是否必须有请求体。默认值是 `true`。当取值为 `true` 时，get 请求方式会报错。如果取值为 `false`，get 请求得到是 `null`。

4.1.2 使用示例

post 请求 jsp 代码：

```
<!-- request body 注解 -->  
<form action="springmvc/useRequestBody" method="post">  
    用户名: <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>
```

get 请求 jsp 代码：

```
<a href="springmvc/useRequestBody?body=test">requestBody 注解 get 请求</a>
```

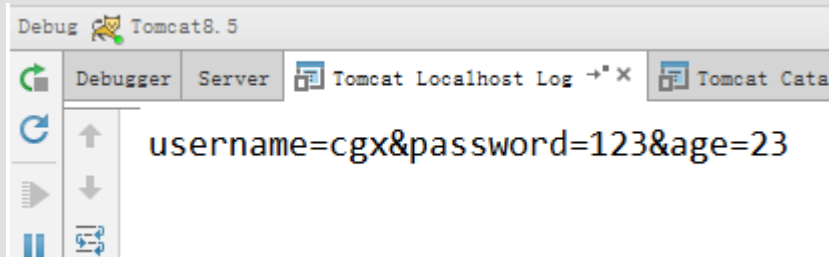
控制器代码：

```
/**  
 * RequestBody 注解  
 * @param user  
 * @return  
 */  
@RequestMapping("/useRequestBody")
```

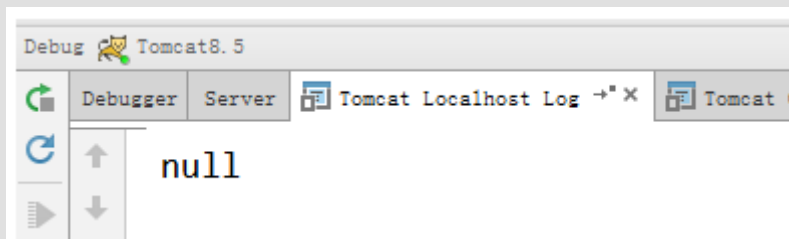


```
public String useRequestBody(@RequestBody(required=false) String body){  
    System.out.println(body);  
    return "success";  
}
```

post 请求运行结果:



get 请求运行结果:



4.2ResponseBody

4.2.1 使用说明

作用:

该注解用于将 Controller 的方法返回的对象，通过 `HttpMessageConverter` 接口转换为指定格式的数据如: json,xml 等，通过 Response 响应给客户端

4.2.2 使用示例

需求:

使用 `@ResponseBody` 注解实现将 controller 方法返回对象转换为 json 响应给客户端。

前置知识点:

Springmvc 默认用 `MappingJacksonHttpMessageConverter` 对 json 数据进行转换，需要加入 jackson 的包。mavne 工程直接导入坐标即可。

```
jackson-annotations-2.9.0.jar  
jackson-databind-2.9.0.jar  
jackson-core-2.9.0.jar
```

```
<dependency>
```

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




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

注意：2.7.0 以下的版本用不了

jsp 中的代码：

```
<script type="text/javascript"
src="${pageContext.request.contextPath}/js/jquery.min.js"></script>
<script type="text/javascript">
    $(function() {
        $("#testJson").click(function() {
            $.ajax({
                type:"post",
                url:"${pageContext.request.contextPath}/testResponseJson",
                contentType:"application/json;charset=utf-8",
                data:'{"id":1,"name":"test","money":999.0}',
                dataType:"json",
                success:function(data) {
                    alert(data);
                }
            });
        });
    })
</script>
<!-- 测试异步请求 -->
<input type="button" value="测试 ajax 请求 json 和响应 json" id="testJson"/>
```

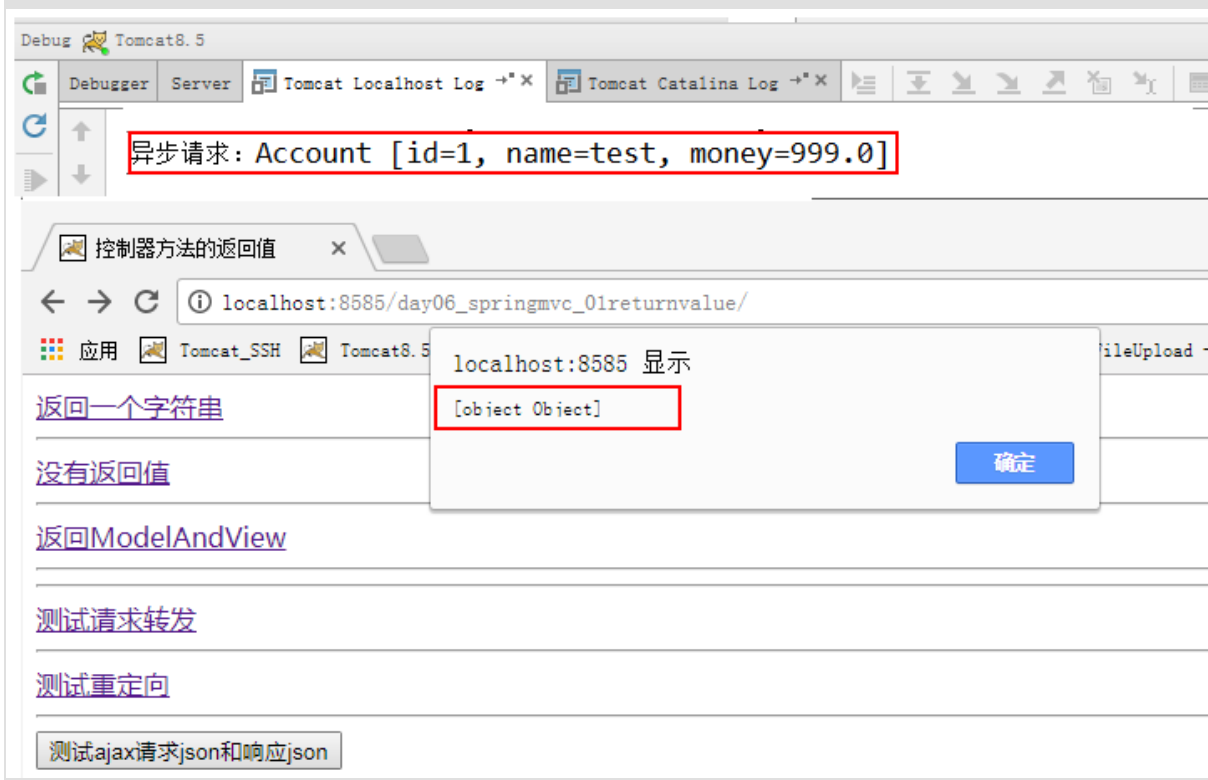
控制器中的代码：

```
/**
 * 响应 json 数据的控制器
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
@Controller("jsonController")
```



```
public class JsonController {  
  
    /**  
     * 测试响应 json 数据  
     */  
    @RequestMapping("/testResponseJson")  
    public @ResponseBody Account testResponseJson(@RequestBody Account account) {  
        System.out.println("异步请求: "+account);  
        return account;  
    }  
}
```

运行结果:



第5章 SpringMVC 实现文件上传[应用]

5.1 文件上传的回顾

5.1.1 文件上传的必要前提

A form 表单的 enctype 取值必须是: `multipart/form-data`
(默认值是: `application/x-www-form-urlencoded`)
enctype: 是表单请求正文的类型

B method 属性取值必须是 Post

C 提供一个文件选择域<input type="file" />

5.1.2 文件上传的原理分析

当 form 表单的 enctype 取值不是默认值后，request.getParameter() 将失效。

enctype="application/x-www-form-urlencoded" 时，form 表单的正文内容是：

key=value&key=value&key=value

当 form 表单的 enctype 取值为 Multipart/form-data 时，请求正文内容就变成：

每一部分都是 MIME 类型描述的正文

-----7de1a433602ac

分界符

Content-Disposition: form-data; name="userName"

协议头

aaa

协议的正文

-----7de1a433602ac

Content-Disposition: form-data; name="file";

filename="C:\Users\zhy\Desktop\fileupload_demo\file\b.txt"

Content-Type: text/plain



协议的类型 (MIME 类型)

bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb

-----7de1a433602ac--

5.1.3 借助第三方组件实现文件上传

使用 Commons-fileupload 组件实现文件上传，需要导入该组件相应的支撑 jar 包：Commons-fileupload 和 commons-io。commons-io 不属于文件上传组件的开发 jar 文件，但 Commons-fileupload 组件从 1.1 版本开始，它工作时需要 commons-io 包的支持。

 commons-fileupload-1.3.1.jar
 commons-io-2.4.jar

5.2 springmvc 传统方式的文件上传

5.2.1 说明

传统方式的文件上传，指的是我们上传的文件和访问的应用存在于同一台服务器上。
并且上传完成之后，浏览器可能跳转。



5.2.2 实现步骤

5.2.2.1 第一步：创建 maven 工程并导入 commons-fileupload 坐标

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

5.2.2.2 第二步：编写 jsp 页面

```
<form action="/fileUpload" method="post" enctype="multipart/form-data">
    名称: <input type="text" name="picname"/><br/>
    图片: <input type="file" name="uploadFile"/><br/>
    <input type="submit" value="上传"/>
</form>
```

5.2.2.3 第三步：编写控制器

```
/**
 * 文件上传的控制器
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
@Controller("fileUploadController")
public class FileUploadController {

    /**
     * 文件上传
     */
    @RequestMapping("/fileUpload")
    public String testResponseJson(String picname, MultipartFile
uploadFile, HttpServletRequest request) throws Exception{
        //定义文件名
        String fileName = "";
        //1.获取原始文件名
        String uploadFileName = uploadFile.getOriginalFilename();
        //2.截取文件扩展名
        String extendName =
```



```
uploadFileName.substring(uploadFileName.lastIndexOf(".") + 1,
uploadFileName.length());

//3.把文件加上随机数，防止文件重复
String uuid = UUID.randomUUID().toString().replace("-", "").toUpperCase();
//4.判断是否输入了文件名
if(!StringUtils.isEmpty(picname)) {
    fileName = uuid + "_" + picname + "." + extendName;
} else {
    fileName = uuid + "_" + uploadFileName;
}
System.out.println(fileName);
//2.获取文件路径
ServletContext context = request.getServletContext();
String basePath = context.getRealPath("/uploads");
//3.解决同一文件夹中文件过多问题
String datePath = new SimpleDateFormat("yyyy-MM-dd").format(new Date());
//4.判断路径是否存在
File file = new File(basePath + "/" + datePath);
if(!file.exists()) {
    file.mkdirs();
}
//5.使用 MultipartFile 接口中方法，把上传的文件写到指定位置
uploadFile.transferTo(new File(file, fileName));
return "success";
}
}
```

5.2.2.4 第四步：配置文件解析器

```
<!-- 配置文件上传解析器 -->
<bean id="multipartResolver" <!-- id 的值是固定的-->
    class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- 设置上传文件的最大尺寸为 5MB -->
    <property name="maxUploadSize">
        <value>5242880</value>
    </property>
</bean>
```

注意：

文件上传的解析器 `id` 是固定的，不能起别的名称，否则无法实现请求参数的绑定。（不光是文件，其他字段也将无法绑定）

5.3springmvc 跨服务器方式的文件上传

5.3.1 分服务器的目的

在实际开发中，我们会有很多处理不同功能的服务器。例如：

应用服务器：负责部署我们的应用

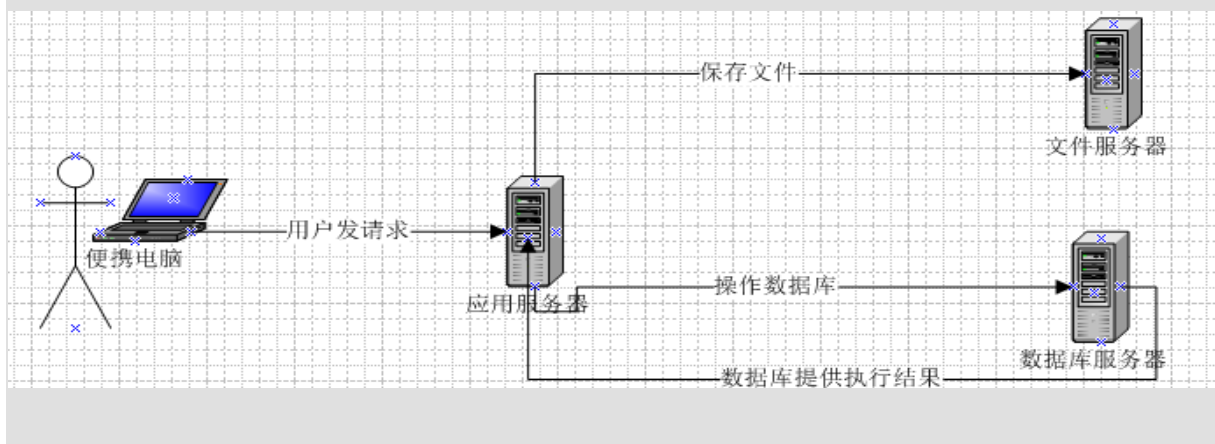
数据库服务器：运行我们的数据库

缓存和消息服务器：负责处理大并发访问的缓存和消息

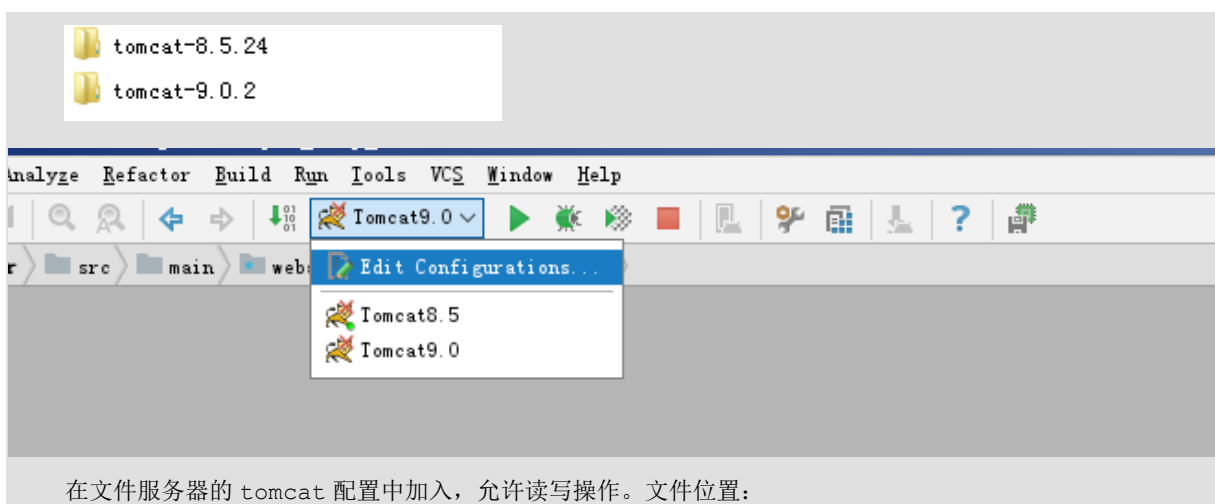
文件服务器：负责存储用户上传文件的服务器。

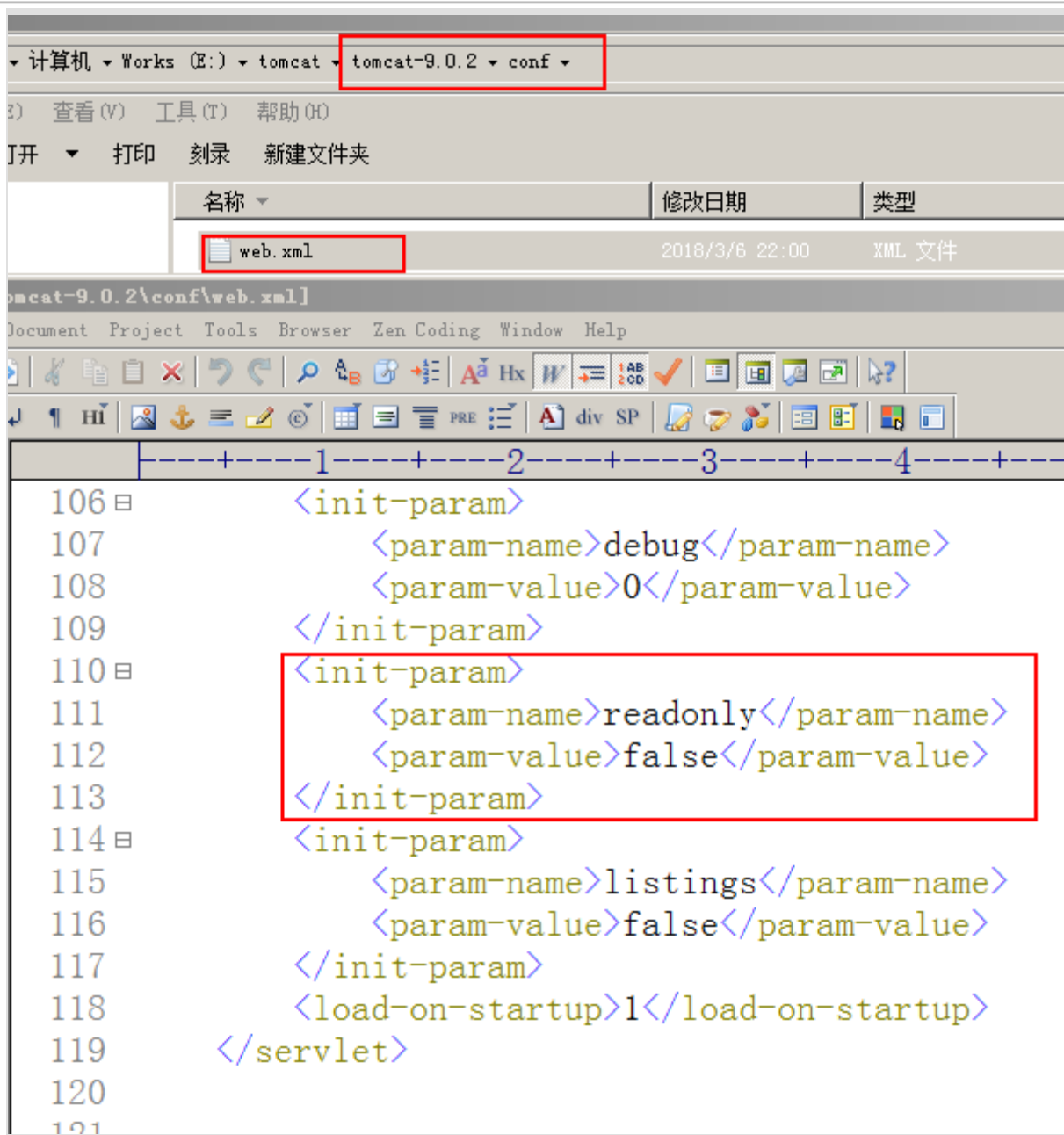
(注意：此处说的不是服务器集群)

分服务器处理的目的是让服务器各司其职，从而提高我们项目的运行效率。



5.3.2 准备两个 tomcat 服务器，并创建一个用于存放图片的 web 工程





5.3.3 导入 jersey 的坐标

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-core</artifactId>
    <version>1.18.1</version>
</dependency>
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-client</artifactId>
    <version>1.18.1</version>
</dependency>
```



5.3.4 编写控制器实现上传图片

```
/**
 * 响应 json 数据的控制器
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
@Controller("fileUploadController2")
public class FileUploadController2 {

    public static final String FILESERVERURL =
"http://localhost:9090/day06_spring_image/uploads/";

    /**
     * 文件上传，保存文件到不同服务器
     */
    @RequestMapping("/fileUpload2")
    public String testResponseJson(String picname, MultipartFile uploadFile) throws
Exception{
        //定义文件名
        String fileName = "";
        //1.获取原始文件名
        String uploadFileName = uploadFile.getOriginalFilename();
        //2.截取文件扩展名
        String extendName =
uploadFileName.substring(uploadFileName.lastIndexOf(".") + 1,
uploadFileName.length());
        //3.把文件加上随机数，防止文件重复
        String uuid = UUID.randomUUID().toString().replace("-", "").toUpperCase();
        //4.判断是否输入了文件名
        if(!StringUtils.isEmpty(picname)) {
            fileName = uuid + "_" + picname + "." + extendName;
        } else {
            fileName = uuid + "_" + uploadFileName;
        }
        System.out.println(fileName);
        //5.创建 sun 公司提供的 jersey 包中的 Client 对象
        Client client = Client.create();
        //6.指定上传文件的地址，该地址是 web 路径
        WebResource resource = client.resource(FILESERVERURL + fileName);
        //7.实现上传
        String result = resource.put(String.class, uploadFile.getBytes());
        System.out.println(result);
        return "success";
    }
}
```




```
}
```

```
}
```

5.3.5 编写 jsp 页面

```
<form action="fileUpload2" method="post" enctype="multipart/form-data">
    名称: <input type="text" name="picname"/><br/>
    图片: <input type="file" name="uploadFile"/><br/>
    <input type="submit" value="上传"/>
</form>
```

5.3.6 配置解析器

```
<!-- 配置文件上传解析器 -->
<bean id="multipartResolver"
    class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- 设置上传文件的最大尺寸为 5MB -->
    <property name="maxUploadSize">
        <value>5242880</value>
    </property>
</bean>
```

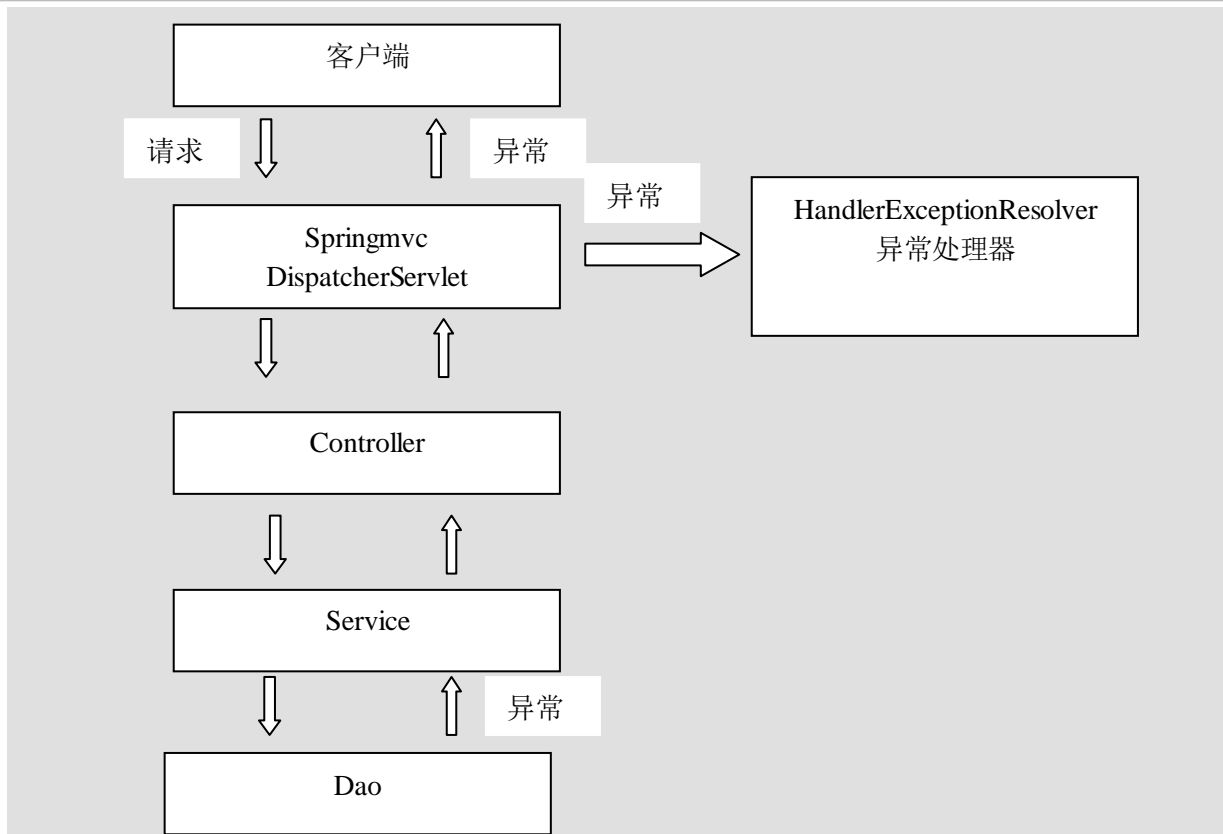
注意：此处 bean 的 id 取值是固定的。

第6章 SpringMVC 中的异常处理[理解]

6.1 异常处理的思路

系统中异常包括两类：预期异常和运行时异常 RuntimeException，前者通过捕获异常从而获取异常信息，后者主要通过规范代码开发、测试通过手段减少运行时异常的发生。

系统的 dao、service、controller 出现都通过 throws Exception 向上抛出，最后由 springmvc 前端控制器交由异常处理器进行异常处理，如下图：



6.2 实现步骤

6.2.1 编写异常类和错误页面

```
/**
 * 自定义异常
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
public class CustomException extends Exception {

    private String message;

    public CustomException(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}
```



```
jsp 页面:
<%@ 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>执行失败</title>
</head>
<body>
执行失败!

${message }
</body>
</html>
```

6.2.2 自定义异常处理器

```
/**
 * 自定义异常处理器
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
public class CustomExceptionHandler implements HandlerExceptionResolver {

    @Override
    public ModelAndView resolveException(HttpServletRequest request,
        HttpServletResponse response, Object handler, Exception ex) {

        ex.printStackTrace();
        CustomException customException = null;
        //如果抛出的是系统自定义异常则直接转换
        if(ex instanceof CustomException){
            customException = (CustomException)ex;
        }else{
            //如果抛出的不是系统自定义异常则重新构造一个系统错误异常。
            customException = new CustomException("系统错误，请与系统管理员联系！");
        }
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("message", customException.getMessage());
        modelAndView.setViewName("error");
        return modelAndView;
    }
}
```



```
}
```

```
}
```

6.2.3 配置异常处理器

```
<!-- 配置自定义异常处理器 -->
```

```
<bean id="handlerExceptionResolver"
```

```
class="com.ithema.exception.CustomExceptionResolver"/>
```

6.2.4 运行结果：

