SpringMVC_01

学习目标

- [ ] 介绍SpringMVC框架
- [ ] 能够实现SpringMVC的环境搭建
- [ ] 掌握RequestMapping的使用
- [ ] 掌握SpringMVC的参数绑定
- [ ] 掌握SpringMVC的自定义类型转换器的使用
- [ ] 掌握SpringMVC的常用注解
- [ ] 掌握PathVariable注解
- [ ] 掌握Controller的返回值使用
- [ ] 掌握Controller中的转发和重定向使用
- [ ] 掌握SpringMVC与json交互(重点)

第一章-SpringMVC介绍

知识点-SpringMVC概述

1.目标

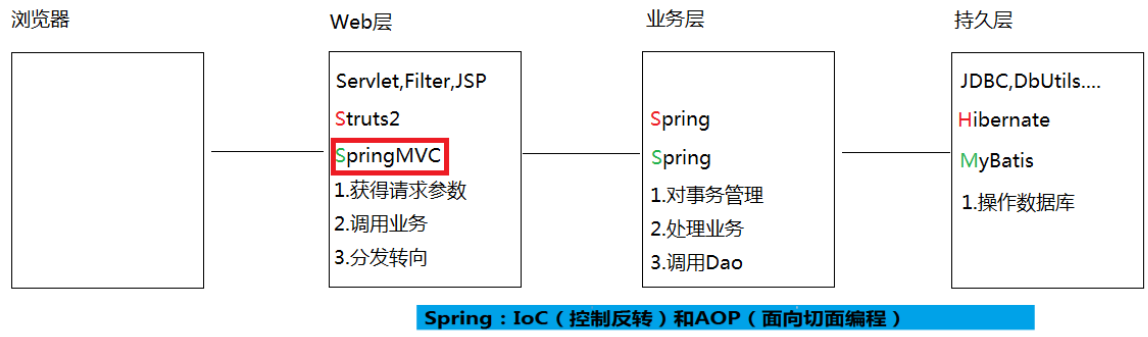
- [ ] 介绍SpringMVC框架

2.路径

1. 三层架构(SpringMVC在三层架构的位置)
2. 什么是SpringMVC
3. SpringMVC 的优点

3.讲解

3.1三层架构



咱们开发服务器端程序，一般都基于两种形式，一种C/S架构程序，一种B/S架构程序. 使用Java语言基本上都是开发B/S架构的程序，B/S架构又分成了三层架构.

- 三层架构

  表现层：WEB层，用来和客户端进行数据交互的。表现层一般会采用MVC的设计模型

  业务层：处理公司具体的业务逻辑的

  持久层：用来操作数据库的

- MVC全名是Model View Controller 模型视图控制器，每个部分各司其职。

  Model：数据模型，JavaBean的类，用来进行数据封装。

  View：指JSP、HTML用来展示数据给用户

  Controller：用来接收用户的请求，整个流程的控制器。用来进行数据校验等

3.2什么是SpringMVC

spring MVC 编辑

Spring MVC属于SpringFrameWork的后续产品，已经融合在Spring Web Flow里面。Spring 框架提供了构建 Web 应用程序的全功能 MVC 模块。使用 Spring 可插入的 MVC 架构，从而在使用Spring进行WEB开发时，可以选择使用Spring的SpringMVC框架或集成其他MVC开发框架，如Struts1(现在一般不用)，Struts2(一般老项目使用)等。

- 用我们自己的话来说:SpringMVC 是一种基于Java实现的MVC设计模型的请求驱动类型的轻量级WEB层框架。
- 作用: 用于替代我们之前Web里面的Servlet的相关代码

  1. 参数绑定(获得请求参数)
  2. 调用业务
  3. 响应数据
  4. 分发转向等等

3.3 SpringMVC 的优点

1.清晰的角色划分：
前端控制器（DispatcherServlet）
请求到处理器映射（HandlerMapping）
处理器适配器（HandlerAdapter）
视图解析器（ViewResolver）
处理器或页面控制器（Controller）
验证器（ Validator）
命令对象（Command 请求参数绑定到的对象就叫命令对象）
表单对象（Form Object 提供给表单展示和提交到的对象就叫表单对象）。
2、分工明确，而且扩展点相当灵活，可以很容易扩展，虽然几乎不需要。
3、由于命令对象就是一个 POJO，无需继承框架特定 API，可以使用命令对象直接作为业务对象。
4、和 Spring 其他框架无缝集成，是其它 Web 框架所不具备的。
5、可适配，通过 HandlerAdapter 可以支持任意的类作为处理器。
6、可定制性， HandlerMapping、 ViewResolver 等能够非常简单的定制。
7、功能强大的数据验证、格式化、绑定机制。
8、利用 Spring 提供的 Mock 对象能够非常简单的进行 Web 层单元测试。
9、本地化、主题的解析的支持，使我们更容易进行国际化和主题的切换。
10、强大的 JSP 标签库，使 JSP 编写更容易。
.....还有比如RESTful风格的支持、简单的文件上传、约定大于配置的契约式编程支持、基于注解的零配
置支持等等。

4.小结

1. SpringMVC 位于web层
2. SpringMVC: Spring家族web层的一个框架,作用
    - 参数绑定(获得请求参数)
    - 调用业务
    - 响应

第二章-SpringMVC入门(重点)

案例-SpringMVC的快速入门

1.需求

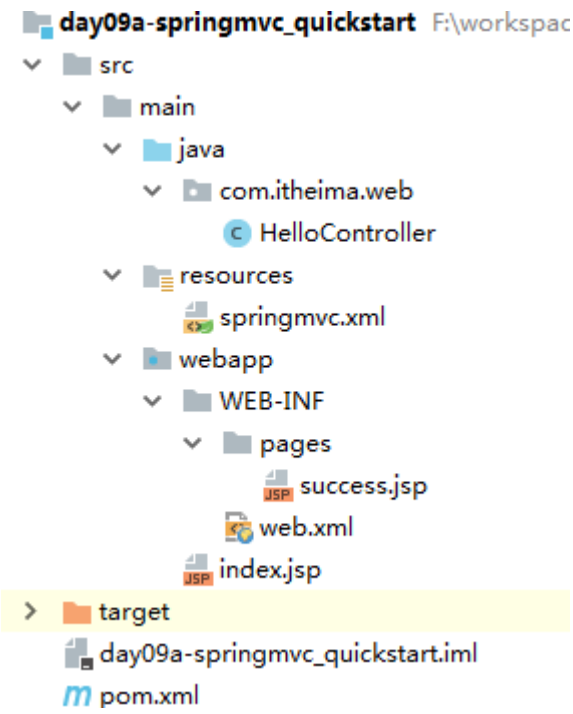
浏览器请求服务器(SpringMVC)里面HelloController，会打印一句话, 然后响应跳转到成功页面

1. HelloController中的某一个方法，可以接收处理请求
2. 使用SpringMVC的方式进行跳转

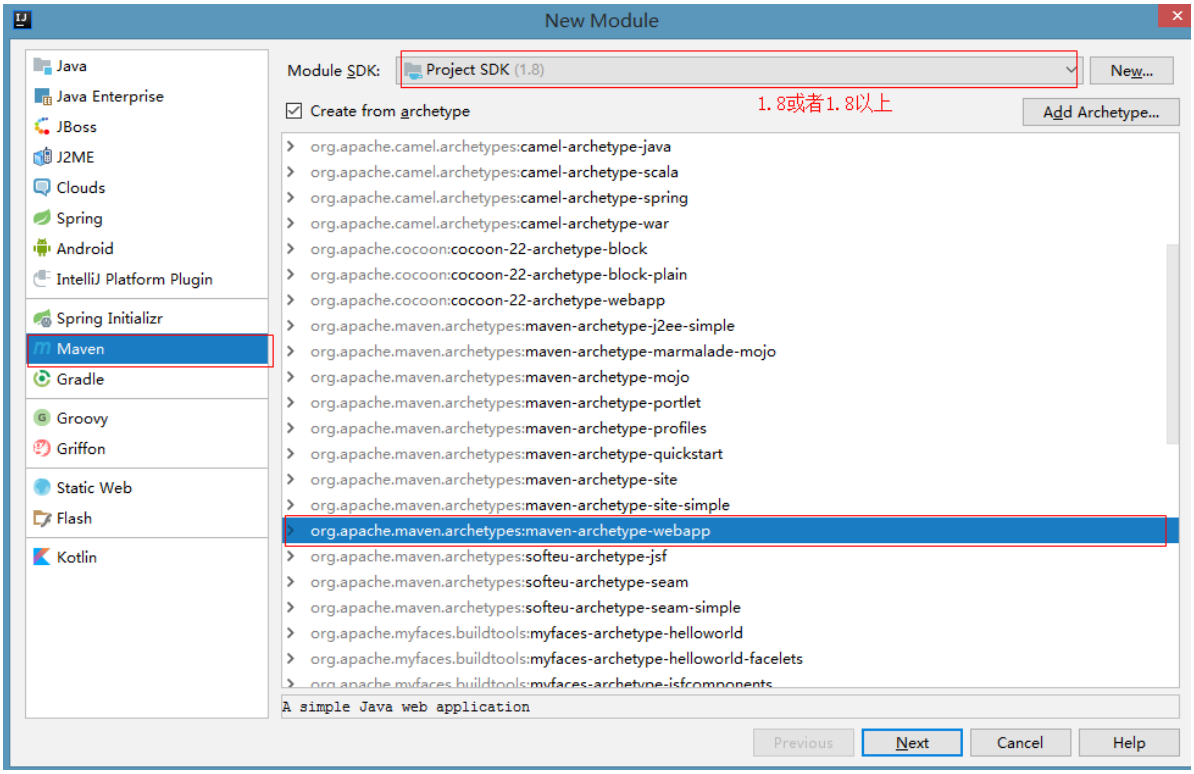
2.分析

1. 创建Maven工程(war),导入坐标
2. 创建HelloController类, 添加注解, 定义方法
3. 创建springmvc.xml(开启包扫描,视图解析器)
4. 配置web.xml

3.实现



3.1 创建web项目,引入坐标




```
<!-- 版本锁定 -->
<properties>
    <spring.version>5.0.2.RELEASE</spring.version>
</properties>

<dependencies>
    <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>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>jsp-api</artifactId>
        <version>2.0</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.18.12</version>
    </dependency>
</dependencies>
```

3.2编写HelloController

```
@Controller
public class HelloController {

    @RequestMapping(value="/hello/sayHello")
    public String sayHello(){
        System.out.println("HelloController 的 sayHello 方法执行了。。。。 ");
        return "success";
    }
}
```

3.3编写SpringMVC的配置文件

- 在classpath目录下创建applicationContext.xml

```
<?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">
    <!--1. 配置spring创建容器时要扫描的包 -->
    <context:component-scan base-package="com.itheima"></context:component-scan>

    <!--2. 加载mvc注解驱动-->
    <mvc:annotation-driven />
    <!--3. 配置视图解析器 -->
    <bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/pages/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>

</beans>
```

3.4在web.xml里面配置核心控制器

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
```

```
  </welcome-file-list>
  <servlet>
    <servlet-name>dispatcherServlet</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:applicationContext.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>dispatcherServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

3.5测试

```
浏览器地址栏访问:http://localhost:8080/hello/sayHello
```

4.小结

1. 创建Maven工程(war), 导入坐标
2. 创建Controller类, 创建一个方法, 添加注解
3. 创建springmvc.xml(开启包扫描, 注册视图解析器)
4. 配置web.xml(前端控制器)

知识点-入门案例的执行过程及原理分析

1.目标

- [ ] 掌握入门案例的执行过程

2.路径

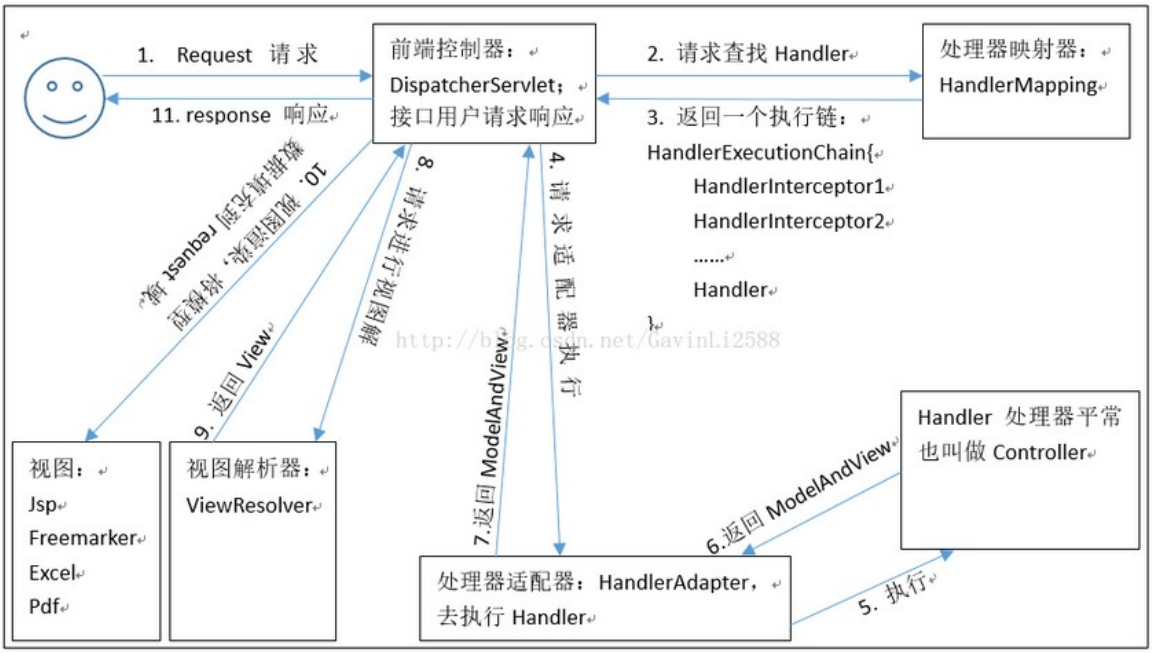
1. 入门案例加载流程
2. SpringMVC 的请求响应流程
3. 入门案例中涉及的组件

3.讲解

3.1入门案例加载流程

1、服务器启动，应用被加载。 读取到 web.xml 中的配置创建 spring 容器并且初始化容器中的对象。
2、浏览器发送请求，被 DispatherServlet 捕获，该 Servlet 并不处理请求，而是把请求转发出去。转发的路径是根据请求 URL，匹配@RequestMapping 中的内容。
3、匹配到了后，执行对应方法。该方法有一个返回值。
4、根据方法的返回值，借助 InternalResourceViewResolver 找到对应的结果视图。
5、渲染结果视图，响应浏览器

3.2 SpringMVC 的请求响应流程 【面试】



什么东西配置在web.xml里面:之前web里面学习到的Servlet、Filter、Listener

其它的东西都配置在springmvc的配置文件里面

3.3 入门案例中涉及的组件(了解)

- DispatcherServlet：前端控制器**(最重要最关键)**

  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(符合面向对象设计的"迪米特法则")。

- 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。

- Handler：处理器 (自己写的Controller类)

  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。

- 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。

- View Resolver：视图解析器:将Controller返回的字符串，解析成具体的视图对象

  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。

- View：视图

  SpringMVC 框架提供了很多的 View 视图类型的支持，包括：jsp，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。

`<mvc:annotation-driven>` 注解说明：

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 `<mvc:annotation-driven>` 自动加载 RequestMappingHandlerMapping （处理映射器）RequestMappingHandlerAdapter （处理适配器），可用在 SpringMVC.xml 配置文件中使用
`<mvc:annotation-driven/>` 替代注解处理器和适配器的配置。

4.小结

1. 服务器启动的时候 就会初始化Spring容器(web.xml配置了启动项), 初始化所有的单例bean
2. 来了请求, 经过DispatcherServlet. 调度其它的组件进行处理, 处理的结果都需要汇报给DispatcherServlet

知识点-RequestMapping注解详解

1.目标

- [ ] 掌握RequestMapping的使用

2.路径

1. 介绍RequestMapping作用
2. RequestMapping的使用的位置
3. RequestMapping的属性

3.讲解

3.1RequestMapping作用

RequestMapping注解的作用是建立请求URL和处理方法之间的对应关系

RequestMapping注解可以作用在方法和类上

3.2RequestMapping的使用的位置

- 使用在类上:

  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

- 使用在方法上:

  请求 URL 的第二级访问目录

```
@Controller
@RequestMapping("/account")
public class AccountController {
    @RequestMapping("/add")
    public String add(){
        System.out.println("添加账户");
        return "success";
    }
    @RequestMapping("/deletet")
    public String deletet(){
        System.out.println("删除账户");
        return "success";
    }
    @RequestMapping("/update")
    public String update(){
        System.out.println("更新账户");
        return "success";
    }
}
```

3.3 RequestMapping的属性(了解)

3.3.1掌握的属性

- path: 指定请求路径的url

```
@RequestMapping(path = "/delete")
public String deletet(){
    System.out.println("删除账户");
    return "success";
}
```

- value: value属性和path属性是一样的(重点)

```
@RequestMapping(value = "/delete")
public String deletet(){
    System.out.println("删除账户");
    return "success";
}
```

- method： 指定该方法的请求方式

```
//只能接受post方式请求
@RequestMapping(value = "/add",method ={RequestMethod.POST} )
 public String add(){
    System.out.println("添加账户");
    return "success";
}
```

可以使用GetMapping、PostMapping注解指定只能使用get、post方式访问

3.3.2 了解的属性（不用练习）

- params: 指定限制请求参数的条件

```
//请求参数必须是money=18,如果不是,则会报错(HTTP Status 400 - Bad Request)
@RequestMapping(value = "/add",params = {"money=18"})
 public String add(){
     System.out.println("添加账户");
     return "success";
}

<a href="${pageContext.request.contextPath }/account/add?money=19">添加账户</a>
<br/> //会报错
```

- headers: 发送的请求中必须包含的请求头

```
 //请求头必须有content-type=text/*,否则就会报错
 @RequestMapping(value = "/add",headers ="content-type=text/*" )
 public String add(){
      System.out.println("添加账户");
      return "success";
}
```

4.小结

1. RequestMapping: URL和方法进行绑定
2. RequestMapping定义位置
    - 类上面
    - 方法上面

    > 如果类上面使用了, 方法上面也使用了. 访问: 类上面的RequestMapping/方法上面的RequestMapping

3. 属性
    - value/path: 访问的路径(可以配置多个)
    - method: 配置访问的请求方式(可以配置多个, 默认就是任何请求方式都可以)

第三章-SpringMVC进阶

知识点-请求参数(formdata类型)的绑定【重点】

1.目标

- [ ] 使用SpringMVC在Controller中获取客户端的请求参数，在这里讲解的绑定请求参数只是"key=value"类型的请求参数，而没有讲到json格式的请求参数

2.分析

- 绑定机制

  表单提交的数据都是key=value格式的(username=zs&password=123),SpringMVC的参数绑定过程是把表单提交的请求参数，作为控制器中方法的参数进行绑定的(要求：提交表单的name和参数的名称是相同的)

- 支持的数据类型

  基本数据类型和字符串类型，统称为简单数据类型

  实体类型（POJO类型）

  集合数据类型（List、map集合等）

- 使用要求
  - 如果是基本类型或者 String 类型： 要求我们的参数名称必须和控制器中方法的形参名称保持一致。 (严格区分大小写) .
  - 如果是 POJO 类型，或者它的关联对象： 要求表单中参数名称和 POJO 类的属性名称保持一致。并且控制器方法的参数类型是 POJO 类型 .
  - 如果是集合类型,有两种方式： 第一种：要求集合类型的请求参数必须在 POJO 中。在表单中请求参数名称要和 POJO 中集合属性名称相同。给 List 集合中的元素赋值， 使用下标。给 Map 集合中的元素赋值， 使用键值对。第二种：接收的参数直接封装到List或者Map中，需要加入RequestParam注解

3.讲解

3.1基本类型和 String 类型作为参数

- 前端页面访问路径

```
http://localhost:8080/user/findByName?username=jay&age=29
```

- UserController

```
@RequestMapping("/findByName")
public String findByName(String username, int age){
    System.out.println("根据name为"+username+",查询用户，年龄为:"+age);
    return "success";
}
```

3.2POJO 类型作为参数

- pojo

```
@Data
public class User implements Serializable {
    private String username;
    private String password;
    private String nickname;
    private Date birthday;
}
```

- 前端页面

```
<html>
<head>
    <title>注册页面</title>
</head>
<body>
    <form method="post" action="/user/register">
        用户名<input type="text" name="username"><br>
        密码<input type="text" name="password"><br>
        昵称<input type="text" name="nickname"><br>
        生日<input type="text" name="birthday"><br>
        <input type="submit" value="注册">
    </form>
</body>
</html>
```

- UserController.java

```
@PostMapping("register")
public String register(User user){
    System.out.println(user);
    return "success";
}
```

3.3POJO 类中包含集合类型参数(使用场景比较少)

3.3.1 POJO 类中包含List

- User

```
@Data
public class User implements Serializable {
    private String username;
    private String password;
    private String nickname;
    private Date birthday;
    private List<String> hobbies;
}
```

- 前端页面

```
<html>
<head>
    <title>注册页面</title>
```

```
</head>
<body>
    <form method="post" action="/user/register">
        用户名<input type="text" name="username"><br>
        密码<input type="text" name="password"><br>
        昵称<input type="text" name="nickname"><br>
        生日<input type="text" name="birthday"><br>
        兴趣爱好
        <input type="checkbox" name="hobbies" value="basketball">篮球
        <input type="checkbox" name="hobbies" value="football">足球
        <input type="checkbox" name="hobbies" value="yumaoball">羽毛球
        <input type="checkbox" name="hobbies" value="pingpangball">乒乓球<br>
        <input type="submit" value="注册">
    </form>
</body>
</html>
```

- UserController.java

```
@PostMapping("register")
public String register(User user){
    System.out.println(user);
    return "success";
}
```

3.4 获取请求参数直接封装到List集合中，比如批量删除(项目中会用到)

什么是批量删除:客户端会想服务器端传入一个id的数组，包含要删除的说个id，那么服务器端得获取到客户端提交的数组中的所有id，然后遍历删除

UserController的代码

```
@RequestMapping("/deleteMore")
public String deleteMore(@RequestParam List<Integer> ids){
    System.out.println("要删除的数据:" + ids);
    return "success";
}
```

前端页面

```
<form action="/user/deleteMore" method="post">
    <input type="checkbox" name="ids" value="1">1<br>
    <input type="checkbox" name="ids" value="2">2<br>
    <input type="checkbox" name="ids" value="3">3<br>
    <input type="checkbox" name="ids" value="4">4<br>
    <input type="checkbox" name="ids" value="5">5<br>
    <input type="checkbox" name="ids" value="6">6<br>
    <input type="submit" value="删除">
</form>
```

注意点: 请求参数名，要和方法的参数名一致；并且要在List参数前添加@RequestParam注解

3.5 获取多个请求参数直接封装到Map中(项目中也会使用到)

- map的key就是参数名，map的value就是参数值只能是String类型
- 一定要记得在接收参数的Map前添加@RequestParam注解

UserController的代码

```
@RequestMapping("/register")
public String register(@RequestParam Map map){
    System.out.println("获取到的请求参数：" + map);
    return "success";
}
```

前端代码

```
<html>
    <head>
        <title>注册页面</title>
    </head>
    <body>
        <form method="post" action="/user/register">
            用户名<input type="text" name="username"><br>
            密码<input type="text" name="password"><br>
            昵称<input type="text" name="nickname"><br>
            生日<input type="text" name="birthday"><br>
            兴趣爱好
            <input type="checkbox" name="hobbies" value="basketball">篮球
            <input type="checkbox" name="hobbies" value="football">足球
            <input type="checkbox" name="hobbies" value="yumaoball">羽毛球
            <input type="checkbox" name="hobbies" value="pingpangball">乒乓球<br>
            <input type="submit" value="注册">
        </form>
    </body>
</html>
```

规律:

1. 如果客户端是单个请求参数，肯定以简单数据类型接收

```
2. 如果客户端的请求参数是多个同名参数，以List接收
            3. 如果客户端的请求参数是多个不同名的请求参数就以POJO对象接收
            4. 但是有的时候，可能没有对应的pojo对象,此时就使用map接收，接受参数的map中的key，就和请求参数名是一一对应的，map中的value就和请求参数值是一一对应
```

4.小结

1. "参数名=参数值&参数名=参数值&参数名=参数值"---->参数类型form-data: get请求
2. "参数名=参数值&参数名=参数值&参数名=参数值"----->x-www-form-urlencoded: 表单post方式提交
    - 1.1 以单个参数接收: 在处理请求的方法中添加参数，保证方法的参数名和请求参数名一致就行 springmvc对一些常用的类型，内置有转换器
    - 1.2 以对象接收
        - 1.2.1 POJO 对象，在处理请求的方法中添加一个POJO类型的参数，保证POJO中的参数名和要接收的参数名对应

- 1.2.2 Map 对象(map的key就是参数名，map的value就是参数值只能是String类型) 必须在参数前添加上RequestParam注解
- 1.2.3 接收多个同名的请求参数，直接封装到List中,此时要在List类型的参数前添加RequestParam注解

知识点-请求参数细节和特殊情况

1.目标

- [ ] 掌握乱码处理和自定义类型转换器

2.路径

1. 请求参数乱码处理
2. 自定义类型转换器
3. 使用 ServletAPI 对象作为方法参数

3.讲解

3.1请求参数乱码处理

- 在web.xml里面配置编码过滤器

```
<!--
  配置解决全局乱码的过滤器
 -->
<filter>
    <filter-name>characterEncodingFilter</filter-name>
    <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>characterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

3.2 自定义类型转换器(了解)

默认情况下,SpringMVC已经实现一些数据类型自动转换。 内置转换器全都在：`org.springframework.core.convert.support` 包下 ,如遇特殊类型转换要求，需要我们自己编写自定义类型转换器。

```
java.lang.Boolean -> java.lang.String : ObjectToStringConverter
java.lang.Character -> java.lang.Number : CharacterToNumberFactory
java.lang.Character -> java.lang.String : ObjectToStringConverter
java.lang.Enum -> java.lang.String : EnumToStringConverter
java.lang.Number -> java.lang.Character : NumberToCharacterConverter
java.lang.Number -> java.lang.Number : NumberToNumberConverterFactory
java.lang.Number -> java.lang.String : ObjectToStringConverter
java.lang.String -> java.lang.Boolean : StringToBooleanConverter
```

```
java.lang.String -> java.lang.Character : StringToCharacterConverter
java.lang.String -> java.lang.Enum : StringToEnumConverterFactory
java.lang.String -> java.lang.Number : StringToNumberConverterFactory
java.lang.String -> java.util.Locale : StringToLocaleConverter
java.lang.String -> java.util.Properties : StringToPropertiesConverter
java.lang.String -> java.util.UUID : StringToUUIDConverter
java.util.Locale -> java.lang.String : ObjectToStringConverter
java.util.Properties -> java.lang.String : PropertiesToStringConverter
java.util.UUID -> java.lang.String : ObjectToStringConverter
 ....
```

3.2.1场景

- 页面，前端输入生日的时候，使用 1999-03-04

```
<html>
<head>
    <title>注册页面</title>
</head>
<body>
    <form method="post" action="/user/register.do">
        用户名<input type="text" name="username"><br>
        密码<input type="text" name="password"><br>
        昵称<input type="text" name="nickname"><br>
        生日<input type="text" name="birthday"><br>
        兴趣爱好
        <input type="checkbox" name="hobbies" value="basketball">篮球
        <input type="checkbox" name="hobbies" value="football">足球
        <input type="checkbox" name="hobbies" value="yumaoball">羽毛球
        <input type="checkbox" name="hobbies" value="pingpangball">乒乓球<br>
        <input type="submit" value="注册">
    </form>
</body>
</html>
```

- UserController.java

```
@PostMapping("register")
public String register(User user){
    System.out.println(user);
    return "success";
}
```

- POJO 类

```
@Data
public class User implements Serializable {
    private String username;
    private String password;
    private String nickname;
    private Date birthday;
    private List<String> hobbies;
}
```

- 报错了:

- 原因是因为SpringMVC内置的是将 1999/03/04这种格式的字符串转换成Date，而客户端传入的是1999-03-04这种格式的，所以无法进行转换，才报错了

3.2.2 使用DateTimeFormate注解进行局部类型转换

在要进行转换的变量上添加DateTimeFormate注解，指定转换的格式

```
@Data
public class User implements Serializable {
    private String username;
    private String password;
    private String nickname;
    @DateTimeFormate("yyyy-MM-dd")
    private Date birthday;
    private List<String> hobbies;
}
```

这种方式有局限性，只能是添加了DateTimeFormate注解的变量能够进行转换，没有添加的变量还是会使用SpringMVC默认的转换规则

3.2.3 自定义类型转换器 （了解）

其实SpringMVC中内置有类型转换器，它可以进行一些基本类型的转换，比如说String转成int等等操作

步骤:

1. 创建一个类实现Converter 接口
2. 配置类型转换器

实现:

- 定义一个类，实现 Converter 接口

  该接口有两个泛型,,S:表示接受的类型， T：表示目标类型(需要转的类型)

```
package com.itheima.convert;

import org.springframework.core.convert.converter.Converter;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * 包名:com.itheima.convert
 * @author Leevi
 * 日期2020-08-13  11:12
 */
public class DateConverter implements Converter<String, Date>{
    @Override
    public Date convert(String source) {
        //source就是要进行类型转换的那个字符串,比如"1999-03-03"
```

```
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd");
        //返回值就是转换成的Date
        try {
            return simpleDateFormat.parse(source);
        } catch (ParseException e) {
            e.printStackTrace();
            throw new RuntimeException("Date转换异常!!!");
        }
    }
}
```

- 在springmvc.xml里面配置转换器

  spring 配置类型转换器的机制是，将自定义的转换器注册到类型转换服务中去

```
<!-- 配置类型转换器 -->
<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.itheima.convert.DateConverter"></bean>
        </array>
    </property>
</bean>
```

- 在 annotation-driven 标签中引用配置的类型转换服务

```
<!--配置Spring开启mvc注解-->
<mvc:annotation-driven conversion-service="converterService"></mvc:annotation-
driven>
```

3.3 使用 ServletAPI 对象作为方法参数(了解)

SpringMVC 还支持使用原始 ServletAPI 对象作为控制器方法的参数。我们可以把它们直接写在控制的方法参数中使用。 支持原始 ServletAPI 对象有：

HttpServletRequest

HttpServletResponse

HttpSession

java.security.Principal

Locale

InputStream

OutputStream

Reader

Writer

- 页面

```
<a href="account/testServletAPI?name=zs">使用 ServletAPI 对象作为方法参数</a>
```

- AccountController.java

```
@RequestMapping("/useServletApi")
public String userServletApi(HttpSession session, HttpServletRequest request){
    String header = request.getHeader("User-Agent");
    System.out.println(header);
    //向使用session对象存值
    session.setAttribute("username","奥巴马");
    return "success";
}
```

4.小结

1. 处理post乱码 直接在web.xml 配置编码过滤器
2. 类型转换器
    - 创建一个类实现Converter
    - 在springmvc.xml进行配置
3. 直接方法的形参里面绑定requet, response. session...
    - 前提是pom导入相关的坐标

知识点-常用注解

1.目标

- [ ] 掌握常用注解

2.路径

1. @RequestParam 【重点】
2. @RequestBody 【重点】
3. @PathVariable 【重点】
4. @RequestHeader【了解】
5. @CookieValue【了解】

3.讲解

3.1 RequestParam【重点】

3.1.1使用说明

- 作用:
    1. 把请求中指定名称的参数给控制器中的形参赋值。
    2. 获取List、Map类型的请求参数必须添加

3.1.2使用实例

1. 当客户端请求参数名为username，而controller中接收请求参数的变量叫name时候

```
@RequestMapping("/findByName")
public String findByName(@RequestParam("username") String name, int age){
    System.out.println("根据name为"+name+",查询用户，年龄为:"+age);
    return "success";
}
```

2. 当使用RequestParam获取请求参数封装到List或者Map中的时候(前面已经讲过了)

3.2. RequestBody【最重点之一】

请求体: post方式的请求参数,get方式没有请求体

3.2.1使用说明

客户端传入给服务器端的请求参数一般分为两种:

1. formdata类型的请求参数，例如:"username=aobama&pwd=123&nickname=圣枪游侠"，获取这类请求参数已经在前面详细讲解过了
2. json类型的请求参数，例如使用axios发送异步的post请求携带的请求参数，而RequestBody注解的作用就是将json类型的请求参数封装到POJO对象或者Map中

3.2.2 具体使用

1. 添加jackson的依赖，因为Springmvc 默认用 MappingJacksonHttpMessageConverter 对 json 数据进行转换，需要添加jackson依赖。

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

2. 用于获取请求参数的POJO前加上RequestBody注解

```
@RequestMapping("/useJson")
public String useJson(@RequestBody User user){
    //目标: 使用body参数来获取请求体的内容----->从请求体中获取参数
    System.out.println("请求体的内容是:" + user);
    return "success";
}
```

3. 前端可以使用postman发送post请求测试

3.3.PathVariable【重点】在以后的项目中使用很多

该注解的作用是，获取RestFul风格的url上的参数

3.3.1RESTFul 风格 URL

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。

- 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。

- 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次交互过程。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 .

- 实例

```
保存
    传统：http://localhost:8080/user/save
    REST：http://localhost:8080/user                          POST方式   执行保存

更新
    传统：http://localhost:8080/user/update?id=1
    REST：http://localhost:8080/user/1                        PUT方式    执行更新    1
代表id

删除
    传统：http://localhost:8080/user/delete?id=1
    REST：http://localhost:8080/user/1                       DELETE方式 执行删除 1代
表id

查询
    传统：http://localhost:8080/user/findAll
    REST：http://localhost:8080/user                        GET方式 查所有

    传统：http://localhost:8080/user/findById?id=1
    REST：http://localhost:8080/user/1                      GET方式 根据id查1个
```

3.3.2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。

3.3.3使用实例

- UserController.java

```
/**
     *  .../findCheckItem?mealId=1&groupId=3&itemId=10
     * 目标: /套餐id/检查组的id/检查项的id ----> .../1/3/10
     * @return
     */
@RequestMapping(method = RequestMethod.GET,path =
"/{mealId}/{groupId}/{itemId}")
public String findCheckItem(@PathVariable("mealId") int
mealId,@PathVariable("groupId") int groupId,@PathVariable("itemId") int itemId){
    System.out.println("根据id查询:mealId=" + mealId + ",groupId=" + groupId +
",itemId=" + itemId);
    return "success";
}

@RequestMapping(method = RequestMethod.DELETE,path =
"/{mealId}/{groupId}/{itemId}")
public String deleteCheckItem(@PathVariable("mealId") int
mealId,@PathVariable("groupId") int groupId,@PathVariable("itemId") int itemId){
    System.out.println("根据id删除:mealId=" + mealId + ",groupId=" + groupId +
",itemId=" + itemId);

    return "success";
}
```

- postman访问
  - http://localhost:8080/user/1/11/13 使用delete请求测试
  - http://localhost:8080/user/1/11/13 使用get请求测试

3.4. RequestHeader【了解】，也有使用场景

3.4.1使用说明

- 作用：
  用于获取请求消息头。
- 属性：
  value：提供消息头名称
  required：是否必须有此消息头

3.4.2使用实例

- UserController.java

```
/**
 * 获取userAgent请求头的信息
 * @return
 */
@RequestMapping("/getUserAgent")
public String getUserAgent(@RequestHeader(value="user-Agent") String userAgent){
    System.out.println("请求头userAgent的信息:" + userAgent);
    return "success";
}
```

3.5. CookieValue【了解】

3.5.1使用说明

- 作用:

  用于把指定 cookie 名称的值传入控制器方法参数。

- 属性:

  value：指定 cookie 的名称。
  required：是否必须有此 cookie。

3.5.2使用实例

- UserController.java

```
/**
 * 获取名字叫做JSESSIONID的cookie的值
 * @param cookieValue
 * @return
 */
@RequestMapping("/getCookieValue")
public String getCookieValue(@CookieValue("JSESSIONID") String cookieValue){
    System.out.println("获取的cookie的值为:" + cookieValue);
    return "success";
}
```

3.6. ModelAttribute (了解)

3.6.1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。

- 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。

3.6.2使用实例

- 页面

```
<form action="user/testModelAttribute" method="post">
    用户名:<input type="text" name="username"/><br/>
    密码:<input type="text" name="password"/><br/>
    <input type="submit" value="testModelAttribute"/>
</form>
```

- UserController.java(用在方法上面)

```
@ModelAttribute
public Account getAccountById(int id){
    System.out.println("根据id查询账户信息:" + id);
    Account account = new Account();
    account.setName("张三");
    account.setMoney(1000.0);
    return account;
}

@RequestMapping("/updateAccount")
public String updateAccount(Account account){
    System.out.println("要修改的用账户的数据是:" + account);
    return "success";
}
```

3.7. SessionAttributes (了解)

3.7.1使用说明

- 作用：

  用于多次执行(多次请求)控制器方法间的数据共享。(该注解定义在类上)

- 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。

3.7.2使用实例

- SessionController.java

```
@Controller
@RequestMapping("/user")
@SessionAttributes(value = {"msg"})
public class UserController {
/**
 * 调用该方法:往域对象中存放一个msg的值
```

```
 * @return
 */
@RequestMapping("/msg")
public String msg(Model model){
    //SpringMVC中model表示数据模型,其实也就是往域对象中存放数据
    model.addAttribute("msg","我是域对象中msg信息");
    return "success";
  }
}
```

第四章-响应数据和结果视图(了解)

知识点-返回值分类(了解)

1.目标

- [ ] 掌握Controller的返回值使用

2.路径

1. 字符串
2. ModelAndView

3.讲解

3.1 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。

- 页面

```
<a href="response/testReturnString">返回String</a><br>
```

- Controller

```
@Controller
@RequestMapping("/response")
public class ResponseController {

    //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("testReturnString");
        return "success";
    }
}
```

3.2 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。

- 页面

```
<a href="response/testReturnModelAndView">ModelAndView类型返回值</a><br>
```

- Controller

```
@RequestMapping("testReturnModelAndView")
public ModelAndView testReturnModelAndView() {
    ModelAndView mv = new ModelAndView();//数据模型和视图
    //ModelAndView对象既能够绑定视图名字，又能够绑定数据(将数据存储到request域对象)
    mv.addObject("name","张三");
    mv.addObject("age",18);
    mv.setViewName("success");

    //去找视图解析器，解析viewName，使用请求转发跳转到对应的视图
    return  mv;
}
```

4.小结

1. 返回String. 返回值是逻辑视图, 通过视图解析器解析成物理视图
2. 返回ModelAndView
    - 设置数据 向request存
    - 设置视图 逻辑视图

知识点-请求转发和重定向(了解)

1.目标

- [ ] 掌握Controller中的转发和重定向使用

2.路径

1. forward 转发
2. Redirect 重定向

3.讲解

3.1forward 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。我们也可以加上 `forward:` 可以转发到页面,也可以转发到其它的controller方法

- 转发到页面

  需要注意的是，如果用了 formward： 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)”

```
    //转发到页面
    @RequestMapping("forwardToPage")
    public String forwardToPage(){
        System.out.println("forwardToPage...");

        //请求转发，不需要视图解析器
        return "forward:/WEB-INF/pages/success.jsp";
    }
```

- 转发到其它的controller方法

  语法: `forward:/类上的RequestMapping/方法上的RequestMapping`

```
    //转发到其它controller
    @RequestMapping("forwardToOtherController")
    public String forwardToOtherController(){
        System.out.println("forwardToOtherController...");
        return "forward:/response/testReturnModelAndView";
    }
```

3.2Redirect 重定向

contrller 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:` 同样可以重定向到页面,也可以重定向到其它controller

- 重定向到页面

  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。

```
    //重定向到页面
    @RequestMapping("redirectToPage")
    public String redirectToPage(){
        System.out.println("redirectToPage...");
        return "redirect:/redirect.jsp";
    }
```

- 重定向到其它的controller方法

  语法: `redirect:/类上的RequestMapping/方法上的RequestMapping`

```
    //重定向到其它Controller
    @RequestMapping("redirectToOtherController")
    public String redirectToOtherController(){
        System.out.println("redirectToOtherController...");
        return "redirect:/response/testReturnModelAndView";
    }
```

4.小结

4.1转发和重定向区别

1. 转发是一次请求, 重定向是两次请求
2. 转发路径不会变化, 重定向的路径会改变
3. 转发只能转发到内部的资源,重定向可以重定向到内部的(当前项目里面的)也可以是外部的(项目以外的)
4. 转发可以转发到web-inf里面的资源, 重定向不可以重定向到web-inf里面的资

4.2 转发和重定向(返回String)

1. 转发到页面

```
forward:/页面的路径
```

2. 转发到Controller

```
forward:/类上面的RequestMapping/方法上面的RequestMapping
```

3. 重定向到页面

```
redirect:/页面的路径
```

4. 重定向到Controller

```
redirect:/类上面的RequestMapping/方法上面的RequestMapping
```

知识点-ResponseBody响应 json数据【重点】

1.目标

- [ ] 掌握SpringMVC与json交互

2.路径

1. 使用说明
2. 使用示例

3.讲解

3.1使用说明

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

3.2使用示例

需求: 发送Ajax请求，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

步骤:

1. 导入jackson坐标
2. 把什么对象转json, 方法返回值就定义什么类型
3. 添加@ResponseBody注解

实现:

- Springmvc 默认用 MappingJacksonHttpMessageConverter 对 json 数据进行转换，需要添加 jackson依赖。

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

- JsonController.java

```
@RequestMapping("/responseJson")
@ResponseBody
public User responseJson(){
    //模拟从数据库查询数据
    User user = new User();
    user.setUsername("周杰伦");
    user.setPassword("123456");
    user.setNickname("双节棍");
    user.setBirthday(new Date());
    List<String> hobbiesList = new ArrayList<>();
    hobbiesList.add("basketball");
    hobbiesList.add("football");
    user.setHobbies(hobbiesList);
    //将user转换成json响应给客户端
    return user;
}
```

4.小结

1. 可以在方法上添加ResponseBody注解
2. 可以在类上添加ResponseBody注解
3. 可以将Controller注解改成RestController注解

4.1实现步骤

1. 添加jackson坐标
2. 把什么对象转成json, 方法的返回值就是什么类型
3. 在方法上面或者方法的返回值前面添加@ResponseBody

解决DispatcherServlet拦截静态资源的问题(重要)

方式一(不常用): DispatcherServlet如果配置映射路径为"/"会拦截到所有的资源(除了JSP)，导致一个问题就是静态资源（img、css、js）也会被拦截到，从而不能被使用。解决问题就是需要配置静态资源不进行拦截.

语法: `<mvc:resources location="/css/" mapping="/css/**"/>`, location:webapp目录下的包,mapping:匹配请求路径的格式

在springmvc.xml配置文件添加如下配置

```
<!-- 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 -->
```

方式二:让DefaultServlet去处理静态资源，DispatcherServlet就不处理静态资源了
<mvc:default-servlet-handler />

方式三:改变DispatcherServlet的映射路径,改成"*.do"

综合案例

####步骤

1. 引入相关依赖(直接copy老师的代码)
    1. spring相关的依赖
        1. spring-webmvc
        2. spring-jdbc
    2. mysql驱动
    3. mybatis
    4. mybatis-spring
    5. lombok
    6. 日志相关的依赖
    7. jackson
2. 编写包结构、接口、类
3. 编写mybatis的映射配置文件
4. 编写springmvc的配置文件
5. 包扫描
6. 加载mvc的注解驱动
7. 整合mybatis
8. 编写web.xml配置文件，进行DispatcherServlet、CharacterEncodingFilter的配置
9. 给类添加IOC、DI的注解
10. 前端代码