

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

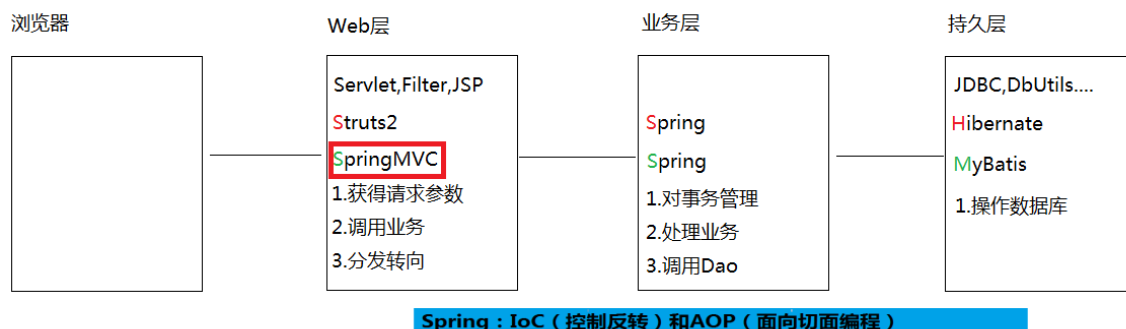
- ☐ 介绍SpringMVC框架

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

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
 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

案例-SpringMVC的快速入门

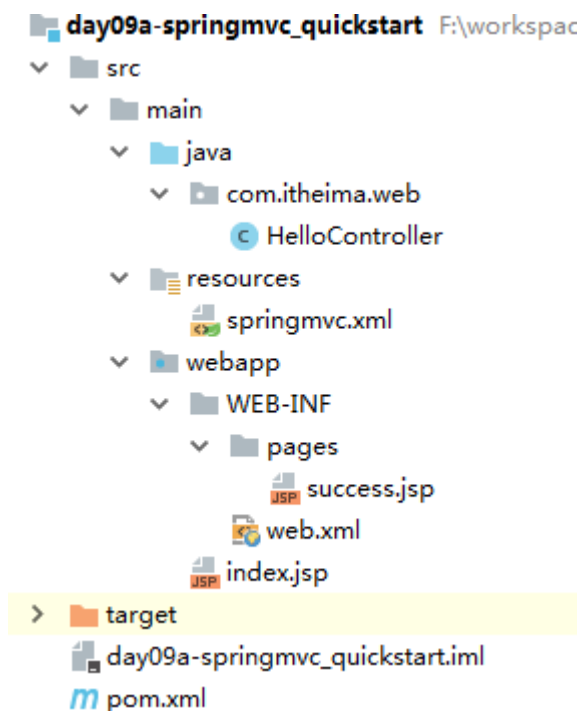
1.需求

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

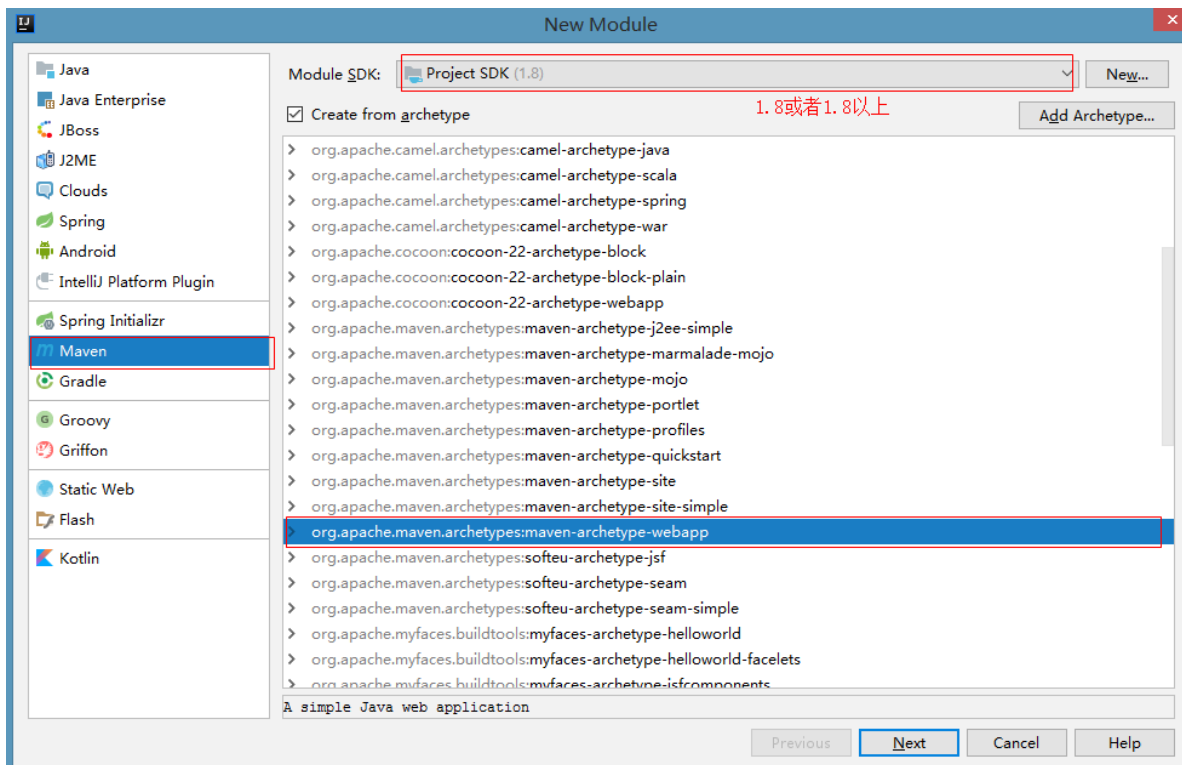
2.分析

1. 创建Maven工程(war),导入坐标
2. 创建Controller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>
</dependencies>
```

3.2编写Controller

```

@Controller
public class HelloController {

    @RequestMapping(value="/hello")
    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springmvc.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">
    <!-- 配置spring创建容器时要扫描的包 -->
    <context:component-scan base-package="com.itheima"></context:component-scan>
    <!-- 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>SpringMVC</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>SpringMVC</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

3.5测试

```
<a href="${pageContext.request.contextPath }/hello">SpringMVC入门</a>
```

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

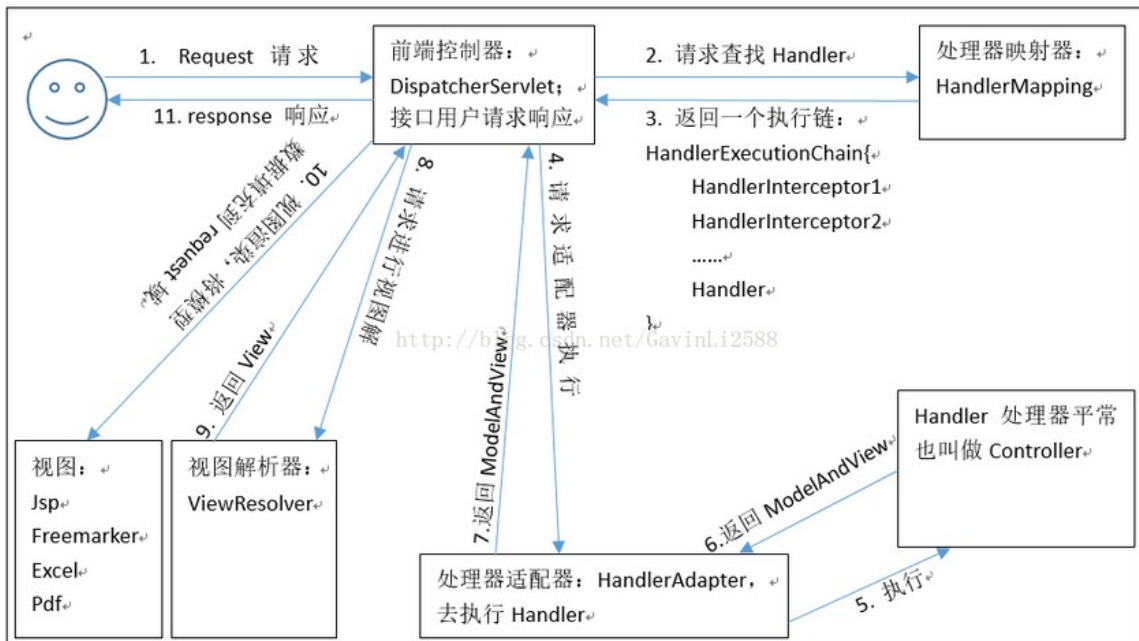
1. 入门案例加载流程
2. SpringMVC 的请求响应流程
3. 入门案例中涉及的组件

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 DispatcherServlet 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。

- HandlerAdapter: 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(默认情况下不配置也是可以使用的)。

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";
    }
}
```

<h3>绝对路径的写法</h3>

```
<a href="${pageContext.request.contextPath }/account/add">添加账户</a><br/>
<a href="${pageContext.request.contextPath }/account/delete">删除账户</a><br/>
<a href="${pageContext.request.contextPath }/account/update">更新账户</a><br/>
<hr/>
```

<h3>相对路径的写法</h3>

```
<a href="account/add">添加账户</a><br/>
<a href="account/delete">删除账户</a><br/>
<a href="account/update">更新账户</a><br/>
```

3.3RequestMapping的属性

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

- 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/getUserId?id=1
```

- UserController

```
@RequestMapping(value = "getUserId")
public String getUserId(int id){
    //在这个方法中接收请求参数id
    //打印ID
    System.out.println("get request..." + id);
    //跳转到success.jsp页面
    //使用请求转发跳转到/WEB-INF/pages/success.jsp
    return "success";
}
```

3.2POJO 类型作为参数

- pojo

```
public class User implements Serializable {
    private String username;
    private String password;
    private String nickname;
```

```

private Date birthday;
@Override
public String toString() {
    return "User{" +
        "username='" + username + '\'' +
        ", password='" + password + '\'' +
        ", nickname='" + nickname + '\'' +
        ", birthday=" + birthday +
        '}';
}

public String getUsername() {
    return username;
}

public void setUsername(String username) {
    this.username = username;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

public String getNickname() {
    return nickname;
}

public void setNickname(String nickname) {
    this.nickname = nickname;
}

public Date getBirthday() {
    return birthday;
}

public void setBirthday(Date birthday) {
    this.birthday = birthday;
}
}

```

- 前端页面

```

<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>
        <input type="submit" value="注册">
    </form>

```

```
</form>
</body>
</html>
```

- UserController.java

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

3.3 POJO 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

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

    @Override
    public String toString() {
        return "User{" +
            "username='" + username + '\'' +
            ", password='" + password + '\'' +
            ", nickname='" + nickname + '\'' +
            ", birthday=" + birthday +
            ", hobbies=" + hobbies +
            '}';
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getNickname() {
        return nickname;
    }
}
```

```

    public void setNickname(String nickname) {
        this.nickname = nickname;
    }

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public List<String> getHobbies() {
        return hobbies;
    }

    public void setHobbies(List<String> hobbies) {
        this.hobbies = hobbies;
    }
}

```

- 前端页面

```

<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";
}

```

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){
    //接收多个同名参数，我们可以使用数组进行接收，前提是方法的参数名和请求参数名要一致
    System.out.println("delete +" + ids);
    return "success";
}

//或者使用数组接收(用的较少)
@RequestMapping("deleteMore")
public String deleteMore(int[] ids){
    //接收多个同名参数，我们可以使用数组进行接收，前提是方法的参数名和请求参数名要一致
    System.out.println("delete +" + ids.length);
    return "success";
}
```

前端页面

```
<form action="/user/deleteMore.do" 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)

一定要记得在接收参数的Map前添加@RequestParam注解

UserController的代码

```
@RequestMapping("getAccountInfo")
public String getAccountInfo(@RequestParam Map parameterMap){
    System.out.println(parameterMap);
    return "success";
}
```

前端代码

```

<html>
<head>
  <title>账户页面</title>
</head>
<body>
  <form action="/user/getAccountInfo.do" method="post">
    账户名<input type="text" name="accountName"><br>
    账户金额<input type="text" name="money"><br>
    <input type="submit" value="提交">
  </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. 请求参数类型是简单(基本,String)类型
 - 方法的形参和请求参数的name一致就可以
2. 请求参数类型是pojo对象类型
 - 形参就写pojo对象
 - pojo的属性必须和请求参数的name一致就可以
3. 获取参数直接封装到List或者Map中的时候，要加上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spring提供的字符集过滤器 -->
<filter>
  <filter-name>CharacterEncodingFilter</filter-name>

```



```

<filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
<!-- 配置初始化参数, 指定字符集 -->
<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, Spring MVC 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
    </form>

```

```
        <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

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

    @Override
    public String toString() {
        return "User{" +
            "username='" + username + '\'' +
            ", password='" + password + '\'' +
            ", nickname='" + nickname + '\'' +
            ", birthday=" + birthday +
            ", hobbies=" + hobbies +
            '}';
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getNickname() {
        return nickname;
    }

    public void setNickname(String nickname) {
        this.nickname = nickname;
    }
}
```

```

    }

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public List<String> getHobbies() {
        return hobbies;
    }

    public void setHobbies(List<String> hobbies) {
        this.hobbies = hobbies;
    }
}

```

- 报错了:

HTTP Status 400 – Bad Request

Type Status Report

Description The server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).

Apache Tomcat/8.5.27

- 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DateTimeFormat注解进行局部类型转换

在要进行转换的变量上添加DateTimeFormat注解，指定转换的格式

```

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

    @Override
    public String toString() {
        return "User{" +
            "username='" + username + '\'' +
            ", password='" + password + '\'' +
            ", nickname='" + nickname + '\'' +
            ", birthday=" + birthday +
            ", hobbies=" + hobbies +
            '}';
    }

    public String getUsername() {
        return username;
    }
}

```

```

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getNickname() {
        return nickname;
    }

    public void setNickname(String nickname) {
        this.nickname = nickname;
    }

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public List<String> getHobbies() {
        return hobbies;
    }

    public void setHobbies(List<String> hobbies) {
        this.hobbies = 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)

```

public class StringToDateConverter implements Converter<String,Date> {
    @Nullable
    @Override
    public Date convert(String s) {

```

```

        try {
            if(StringUtils.isEmpty(s)){
                throw new RuntimeException("字符串不能为null");
            }
            SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
            return dateFormat.parse(s);
        } catch (ParseException e) {
            e.printStackTrace();
            return null;
        }
    }
}

```

- 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.web.StringToDateConverter"></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

`使用 ServletAPI 对象作为方法参数`

- AccountController.java

```
@RequestMapping("/testServletAPI")
public String testServletAPI(HttpServletRequest request, HttpServletResponse
response, HttpSession session){
    System.out.println("request="+request);
    System.out.println("response="+response);
    System.out.println("session="+session);
    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request, 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uid，而controller中接收请求参数的变量叫id时候

```

public String getUserId(@RequestParam("uid") int id){
    //在这个方法中接收请求参数id
    //打印ID
    System.out.println("get request..." + id);
    //跳转到success.jsp页面
    //使用请求转发跳转到/WEB-INF/pages/success.jsp
    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("userRequestBody")
public String userRequestBody(@RequestBody User user){
    //RequestBody注解的作用1: 获取请求体的数据
    System.out.println(user);
    //真正的目标: 将请求体中的数据拿到, 封装到POJO
    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。

- 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

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

- 页面

```
<a href="user/info/1">测试PathVaribale</a><br/>
```

- UserController.java

```
@RequestMapping(value = "info/{id}",method = RequestMethod.GET)
public String getInfo(@PathVariable("id") int id,){
    System.out.println("查询信息:" + id);
    return "success";
}
@RequestMapping(value = "info/{id}",method = RequestMethod.POST)
public String addInfo(@PathVariable("id") int id,){
    System.out.println("添加信息:" + id);
    return "success";
}
@RequestMapping(value = "info/{id}",method = RequestMethod.DELETE)
public String deleteInfo(@PathVariable("id") int id,){
    System.out.println("删除信息:" + id);
    return "success";
}
@RequestMapping(value = "info/{id}",method = RequestMethod.PUT)
public String updateInfo(@PathVariable("id") int id,){
    System.out.println("更新信息:" + id);
    return "success";
}
```

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

- 页面

```
<a href="user/testRequestHeader">测试RequestHeader</a><br/>
```

- UserController.java

```

@RequestMapping("testRequestHeader")
public String testRequestHeader(@RequestHeader(value = "User-Agent") String
requestHeader){
    System.out.println("requestHeader="+requestHeader);
    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

- 页面

```
<a href="user/testCookieValue">测试CookieValue</a><br/>
```

- UserController.java

```

@RequestMapping("testCookieValue")
public String testCookieValue(@CookieValue(value="JSESSIONID") String sessionId)
{
    System.out.println("sessionId="+sessionId);
    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。
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。

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)

```
@RequestMapping("/testModelAttribute")
public String testModelAttribute(User user){
    System.out.println("testModelAttribute ...收到了请求..." + user);
    return "success";
}

@ModelAttribute
public User getModel(String username, String password){
    System.out.println("getModel ...收到了请求...");
    //模拟查询数据库，把性别也查询出来
    User user = new User();
    user.setUsername(username);
    user.setUsername(password);
    user.setSex("男");
    return user;
}
```

- UserController.java(用在参数上面)

```
@RequestMapping("/testModelAttribute")
public String testModelAttribute(@ModelAttribute("u") User user){
    System.out.println("testModelAttribute ...收到了请求..." + user);
    return "success";
}

@ModelAttribute
//没有返回值
public void getModel(String username, String password, Map<String, User>
map){
    System.out.println("getModel ...收到了请求...");
    //模拟查询数据库，把性别也查询出来
    User user = new User();
    user.setUsername(username);
    user.setUsername(password);
    user.setSex("男");
    map.put("u", user);
}
```

3.7.SessionAttributes (了解)

3.7.1使用说明

- 作用:

用于多次执行(多次请求)控制器方法间的参数共享。(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

- 页面

```
<a href="sessionController/setAttribute?name=张三&age=18">测试  
SessionAttributes(存)</a><br/>  
<a href="sessionController/getAttribute">测试SessionAttribute(取)</a><br/>  
<a href="sessionController/removeAttribute">测试SessionAttribute(清空)</a><br/>
```

- SessionController.java

```
@Controller  
@RequestMapping("/sessionController")  
@SessionAttributes(value = {"name", "age"})  
public class SessionController {  
  
    @RequestMapping("/setAttribute")  
    public String setAttribute(String name, int age, Model model){  
        model.addAttribute("name", name);  
        model.addAttribute("age", age);  
        return "success";  
    }  
  
    @RequestMapping("/getAttribute")  
    public String getAttribute(ModelMap modelMap){  
        System.out.println("name="+modelMap.get("name"));  
        System.out.println("age="+modelMap.get("age"));  
        return "success";  
    }  
  
    @RequestMapping("/removeAttribute")  
    public String removeAttribute(SessionStatus sessionStatus){  
        sessionStatus.setComplete();  
        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");
    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-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...");
    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里面的资源

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:

- 在springmvc.xml里面设置过滤资源

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`

- 页面

```
<h1>三, 响应json</h1>
<input id="btnId" type="button" value="发送Ajax请求"/><br/>

<script>
    $(function () {
        $("#btnId").click(function () {
            $.post("response/testJson", {name: "zs", age: 18}, function (result)
            {
                alert("result="+result);
            }, "json");
        });
    });
</script>
```


- 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

```
@Controller
@RequestMapping("/response")
public class ResponseController {
    @RequestMapping("/testJson")
    public @ResponseBody User testJson(User user){
        System.out.println("收到了请求..." + user);
        user.setName("李四");
        return user;
    }
}
```

4.小结

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

4.2注意事项

1. Dispatcher的路径是/, 除了jsp以为所有的资源都匹配, 要使用JQ 忽略静态资源

```
<!--忽略静态资源-->
<mvc:default-servlet-handler/>
<!--<mvc:resources mapping="/js/**" location="/js/" />
<mvc:resources mapping="/css/**" location="/css/" />
<mvc:resources mapping="/img/**" location="/img/" />-->
```

综合案例

####步骤

1. 引入相关依赖(直接copy老师的代码)
2. 编写包结构、接口、类
3. 编写web.xml配置文件, 进行DispatcherServlet、CharacterEncodingFilter的配置
4. 编写mybatis的核心配置文件、映射配置文件
5. 编写springmvc的配置文件
 1. 包扫描
 2. 加载mvc的注解驱动
 3. 整合mybatis
6. 给类添加IOC、DI的注解
7. 前端代码

springmvc第二天

学习目标

- ☐ 掌握图片上传(重点)
- ☐ 了解SpringMVC的统一异常处理
- ☐ 掌握SpringMVC拦截器(重点)
- ☐ 将面面项目改成ssm架构(重点)

第一章-SpringMVC 实现文件上传

知识点-文件上传介绍

1.目标

- ☐ 掌握文件上传的要求

2.路径

1. 文件上传概述
2. 文件上传要求
3. 常见的文件上传jar包和框架

3.讲解

3.1文件上传概述以及使用场景

就是把客户端(浏览器)的文件保存一份到服务器 说白了就是文件的拷贝

常见的使用场景:上传头像、上传各种照片、上传word、Excel等等文件

3.2文件上传要求

3.2.1浏览器端要求(通用浏览器的要求)

- 表单提交方式 post
- 提供文件上传框(组件) input type="file"
- 表单的enctype属性必须为 `multipart/form-data` (没有这个属性值的话, 文件的内容是提交不过去的)

3.2.2服务器端要求(如果有框架就不是这样的)

要使用`request.getInputStream()`来获取数据.

注意:

- 若表单使用了 `multipart/form-data` ,使用原生`request.getParameter()`去获取参数的时候都为null

我们做文件上传一般会借助第三方组件(jar, 框架 SpringMVC)实现文件上传.

3.3常见的文件上传jar包和框架

serlvet3.0(原生的文件上传的API)

commons-fileupload : apache出品的一款专门处理文件上传的工具包 (我们肯定不会直接使用)

struts2(底层封装了:commons-fileupload)

SpringMVC(底层封装了:commons-fileupload)

4.小结

案例-springmvc 传统方式文件上传

1.需求

- ☐ 使用springmvc 完成传统方式文件上传

2.步骤

1. 把commons-fileupload坐标导入进来
2. 在控制器的方法的形参里面定义和文件相关的变量 `MultipartFile`
3. 把文件存到服务器
4. 配置文件解析器

3.实现

- 创建Maven工程,添加依赖

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

- 创建前端页面

```
<h1>一, springmvc传统方式文件上传</h1>
<form action="springmvc/fileUpload02" method="post" enctype="multipart/form-data">
    图片: <input type="file" name="upload"/><br/>
    图片描述:<input type="text" name="pdesc"/>
    <input type="submit" value="上传"/>
</form>
```

- 控制器

```
@RequestMapping("fileUpload02")
public String fileUpload02(HttpServletRequest request, String pdesc,
MultipartFile upload) throws IOException {
    System.out.println("pdesc="+pdesc);
    //1. 获得文件名
    String filename = upload.getOriginalFilename();
    //2. 获得随机文件名
    String uuidName = UploadUtils.getUUIDName(filename);
    //3. 获得绝对路径
    String basePath =
request.getSession().getServletContext().getRealPath("/upload");
    //4. 获得两层目录
    String dir = UploadUtils.getDir();
    //5. 创建两层目录
    File fileDir = new File(basePath, dir);
    if(!fileDir.exists()){
        fileDir.mkdirs();
    } //准备一个文件夹，用于上传文件

    //6.使用 MultipartFile 接口中方法，把上传的文件写到指定位置
    upload.transferTo(new File(fileDir, uuidName));

    return "success";
}
```

- 在springmvc.xml配置文件解析器

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)

```
<!--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"></property>
</bean>
```

4.小结

案例-springmvc 跨服务器方式的文件上传(了解)

1.需求

- 了解使用springmvc 跨服务器方式的文件上传

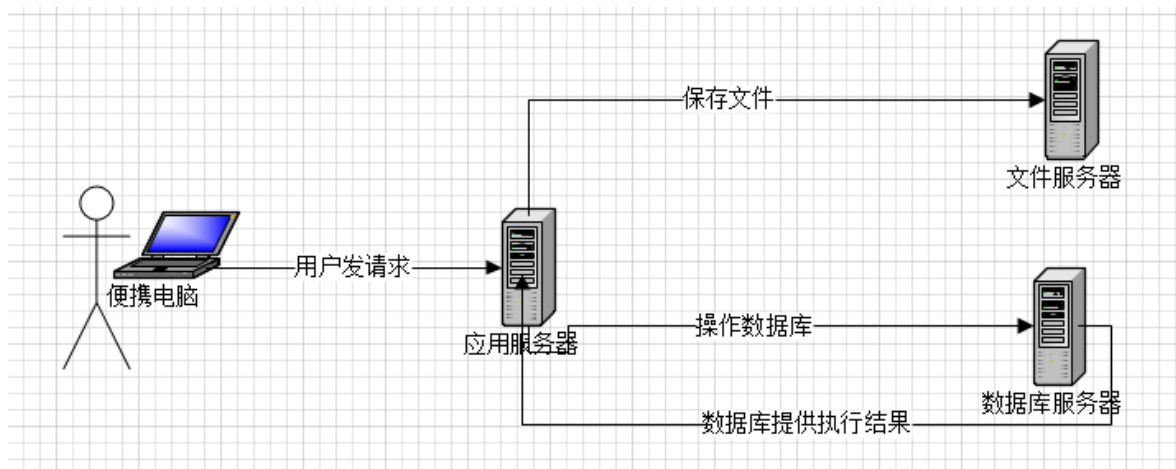
2.分析

2.1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(注意：此处说的不是服务器集群)。例如：

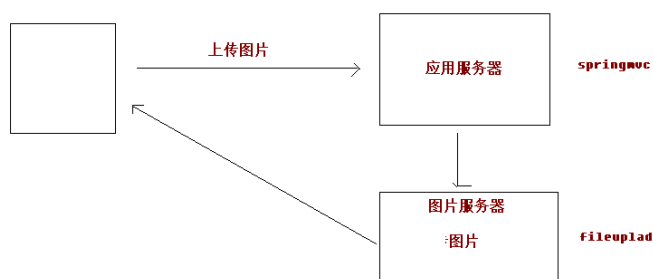
- 应用服务器：负责部署我们的应用
- 数据库服务器：运行我们的数据库
- 缓存和消息服务器：负责处理大并发访问的缓存和消息
- 文件服务器：负责存储用户上传文件的服务器。

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。



2.2跨服务器方式的文件上传图解

跨服务器的文件上传



- 准备两个服务器(默认情况下，tomcat是不允许其他服务器往它里面写入数据的)，修改tomcat的的conf目录下的web.xml，添加readonly参数为false

```

web.xml
103     <servlet>
104         <servlet-name>default</servlet-name>
105         <servlet-class>org.apache.catalina.servlet
106     <init-param>
107         <param-name>debug</param-name>
108         <param-value>0</param-value>
109     </init-param>
110     <init-param>
111         <param-name>readonly</param-name>
112         <param-value>>false</param-value>
113     </init-param>
114     <init-param>
115         <param-name>listings</param-name>
116         <param-value>>false</param-value>
117     </init-param>
118     <load-on-startup>1</load-on-startup>

```

3.实现

- 添jersey依赖 (跨服务器上传图片的代码)

```

<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>

```

- 前端页面

```

<h1>二,springmvc 跨服务器方式的文件上传</h1>
<form action="springmvc/fileUpload03" method="post" enctype="multipart/form-
data">
    图片: <input type="file" name="upload"/><br/>
    图片描述:<input type="text" name="pdesc"/>
    <input type="submit" value="上传"/>
</form>

```

- 控制器

```

private static final String FILESERVERURL="http://localhost:9090/upload/";

@RequestMapping("fileupload03")
public String fileUpload03(String pdesc, MultipartFile upload) throws
IOException {
    System.out.println("pdesc="+pdesc);
    //1. 获得文件名
    String filename = upload.getOriginalFilename();
    //2. 获得随机文件名
    String uuidName = UploadUtils.getUUIDName(filename);

```

```
//3.创建jersey包中提供的client对象
Client client = Client.create();//它就可以和文件服务器建立联系
//4.使用client和文件服务器建立联系
WebResource resource = client.resource(FILESERVERURL+uuidName);
//5.把web资源对象写到文件服务器
resource.put(upload.getBytes());
return "success";
}
```

4.小结

第二章-SpringMVC 中的异常处理(了解)

知识点-SpringMVC 中的异常处理

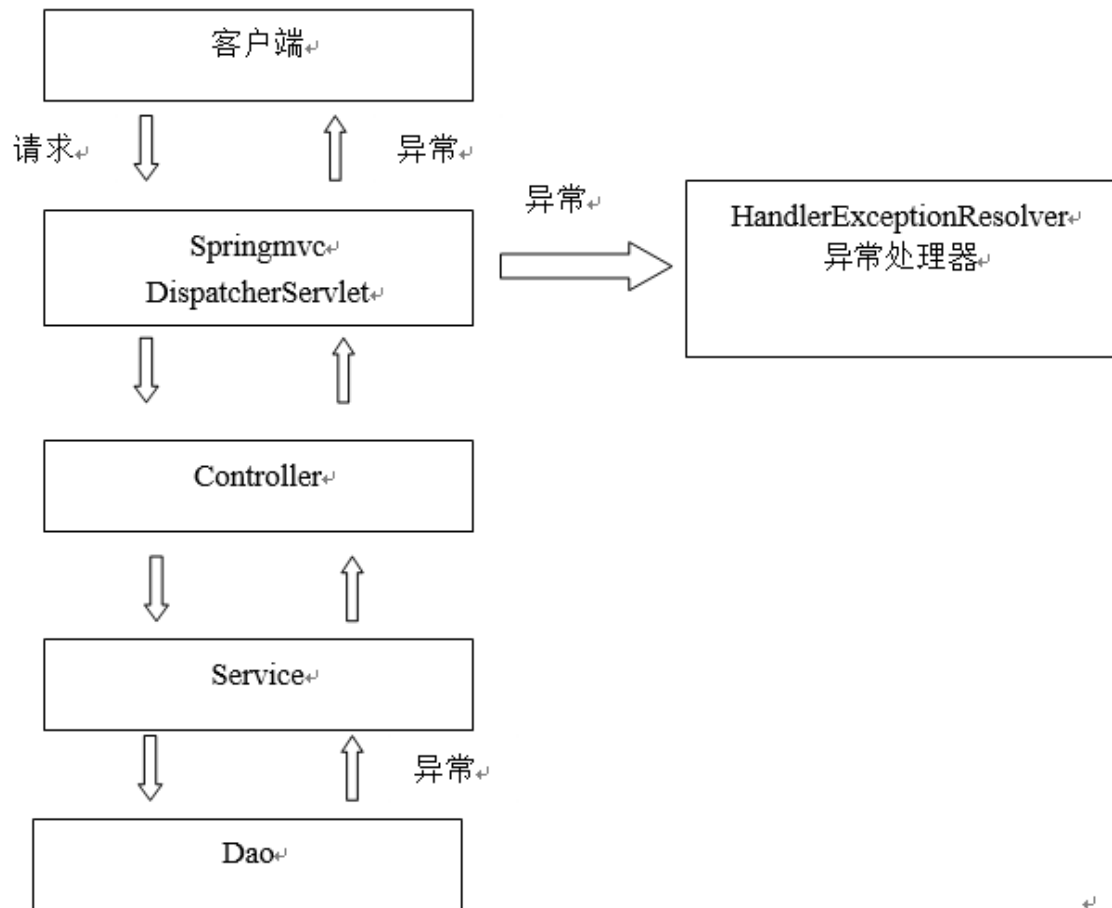
1.目标

- ☐ 掌握SpringMVC的统一异常处理

2.分析

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：



springmvc在处理请求过程中出现异常信息交由异常处理器进行处理，自定义异常处理器可以实现一个系统的异常处理逻辑。

3.代码实现

3.1自定义异常类

目的: 统一的管理异常, 方便统一管理错误提示语

```
/**
 * 自定义异常类
 */
public class SysException extends Exception{
    private String msg;
    public SysException(String msg) {
        this.msg = msg;
    }
    public void setMsg(String msg) {
        this.msg = msg;
    }
    public String getMsg() {
        return msg;
    }
}
```

3.2自定义异常处理器


```

/**
 * 自定义异常处理器
 */
public class SysExceptionHandler implements HandlerExceptionResolver {

    @Nullable
    @Override
    public ModelAndView resolveException(HttpServletRequest request,
        HttpServletResponse response, @Nullable Object handler, Exception ex) {
        SysExceptionHandler sysExceptionHandler = null;
        if(ex instanceof SysExceptionHandler){
            sysExceptionHandler = (SysExceptionHandler) ex;
        }else{
            sysExceptionHandler = new SysExceptionHandler("服务器异常,请联系管理员");
        }
        ModelAndView mv = new ModelAndView();
        mv.setViewName("msg");
        mv.addObject("msg", sysExceptionHandler.getMessage());
        return mv;
    }
}

```

3.3配置异常处理器

- 在springmvc.xml配置

```

<!--注册异常处理器-->
<bean id="sysExceptionHandler" class="com.itheima.web.SysExceptionHandler">
</bean>

```

4.小结

第三章-SpringMVC 中的拦截器(掌握)

知识点-SpringMVC入门

1.目标

- ☐ 掌握SpringMVC基本使用

2.路径

1. 拦截器概述
2. 自定义拦截器入门

3.讲解

3.1.拦截器概述

Spring MVC 的处理器拦截器类似于 Servlet 开发中的过滤器 Filter，用于对处理器(自己编写的 Controller)进行预处理和后处理。用户可以自己定义一些拦截器来实现特定的功能。谈到拦截器，还要向大家提一个词——拦截器链 (Interceptor Chain)。拦截器链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或字段时，拦截器链中的拦截器就会按其之前定义的顺序被调用。说到这里，可能大家脑海中有了一个疑问，这不是我们之前学的过滤器吗？是的它和过滤器是有几分相似，但是也有区别，接下来我们就来说说他们的区别：

类别	使用范围	拦截范围
拦截器	SpringMVC项目	只会拦截访问的控制器方法的请求
过滤器	任何web项目	任何资源(servlet,控制器,jsp,html等)

我们要想自定义拦截器，要求必须实现：HandlerInterceptor 接口。

3.2.自定义拦截器入门

- 编写一个普通类实现 HandlerInterceptor 接口

```
public class Interceptor01 implements HandlerInterceptor {
    @Override
    //在达到目标方法之前执行(拦截的方法)
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        System.out.println("Interceptor01 收到了请求...");
        return true;//返回true放行，返回false拦截
    }
}
```

- 在springmvc.xml配置拦截器

```
<!--配置拦截器-->
<mvc:interceptors>
    <mvc:interceptor>
        <!--用于指定拦截的路径-->
        <mvc:mapping path="/*" />
        <bean id="interceptor01"
class="com.itheima.web.interceptor.Interceptor01"></bean>
    </mvc:interceptor>
</mvc:interceptors>
```

4.小结

知识点-自定义拦截器进阶

1.目标

- ☐ 掌握自定义拦截器的高级使用

2.路径

1. 拦截器的放行

2. 拦截后跳转
3. 拦截器的路径
4. 拦截器的其它方法
5. 多个拦截器执行顺序

3.讲解

3.1拦截器的放行

```
public class Interceptor01 implements HandlerInterceptor {  
    @Override  
    //在达到目标方法之前执行(拦截的方法)  
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response) throws Exception {  
        System.out.println("preHandle 执行了...");  
        return true; //返回true放行, 返回false拦截  
    }  
}
```

3.2拦截后跳转

拦截器的处理结果，莫过于两种：

放行： 如果后面还有拦截器就执行下一个拦截器，如果后面没有了拦截器，就执行Controller方法

拦截： 但是注意，拦截后也需要返回到一个具体的结果(页面,Controller)。

- 在preHandle方法返回false,通过request进行转发,或者通过response对象进行重定向,输出

```
public class Interceptor01 implements HandlerInterceptor {  
    @Override  
    //在达到目标方法之前执行(拦截的方法)  
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  
        System.out.println("preHandle 执行了...");  
        //转发到拦截后的页面  
  
        //request.getRequestDispatcher("/interceptor01.jsp").forward(request, response);  
        //转发到controller  
        request.getRequestDispatcher("/demo01/fun02").forward(request, response);  
        return false; //返回true放行, 返回false拦截  
    }  
}
```

3.3 拦截器的路径

```
<!--配置拦截器-->  
<mvc:interceptors>  
    <mvc:interceptor>  
        <!--用于指定拦截的路径-->  
        <mvc:mapping path="/**"/>  
        <!--用于指定忽略(不拦截的路径)-->  
        <mvc:exclude-mapping path="/demo01/fun02"/>  
        <bean id="interceptor01" class="com.itheima.web.interceptor.Interceptor01"/>  
    </mvc:interceptor>  
</mvc:interceptors>
```

3.4拦截器的其它方法

- afterCompletion 在目标方法完成视图层渲染后执行。
- postHandle 在目标方法执行完毕获得了返回值后执行。
- preHandle 被拦截的目标方法执行之前执行。

```
public class Interceptor01 implements HandlerInterceptor {
    @Override
    //在达到目标方法之前执行(拦截的方法)
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        System.out.println("preHandle 执行了...");
        return true;//返回true放行, 返回false拦截
    }
    @Override
    //在目标方法执行完成之后,完成页面渲染之前执行
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, @Nullable ModelAndView modelAndView) throws Exception
{
        System.out.println("postHandle 执行了...");
    }
    @Override
    //完成页面渲染之后执行
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, @Nullable Exception ex) throws Exception {
        System.out.println("afterCompletion 执行了...");
    }
}
```

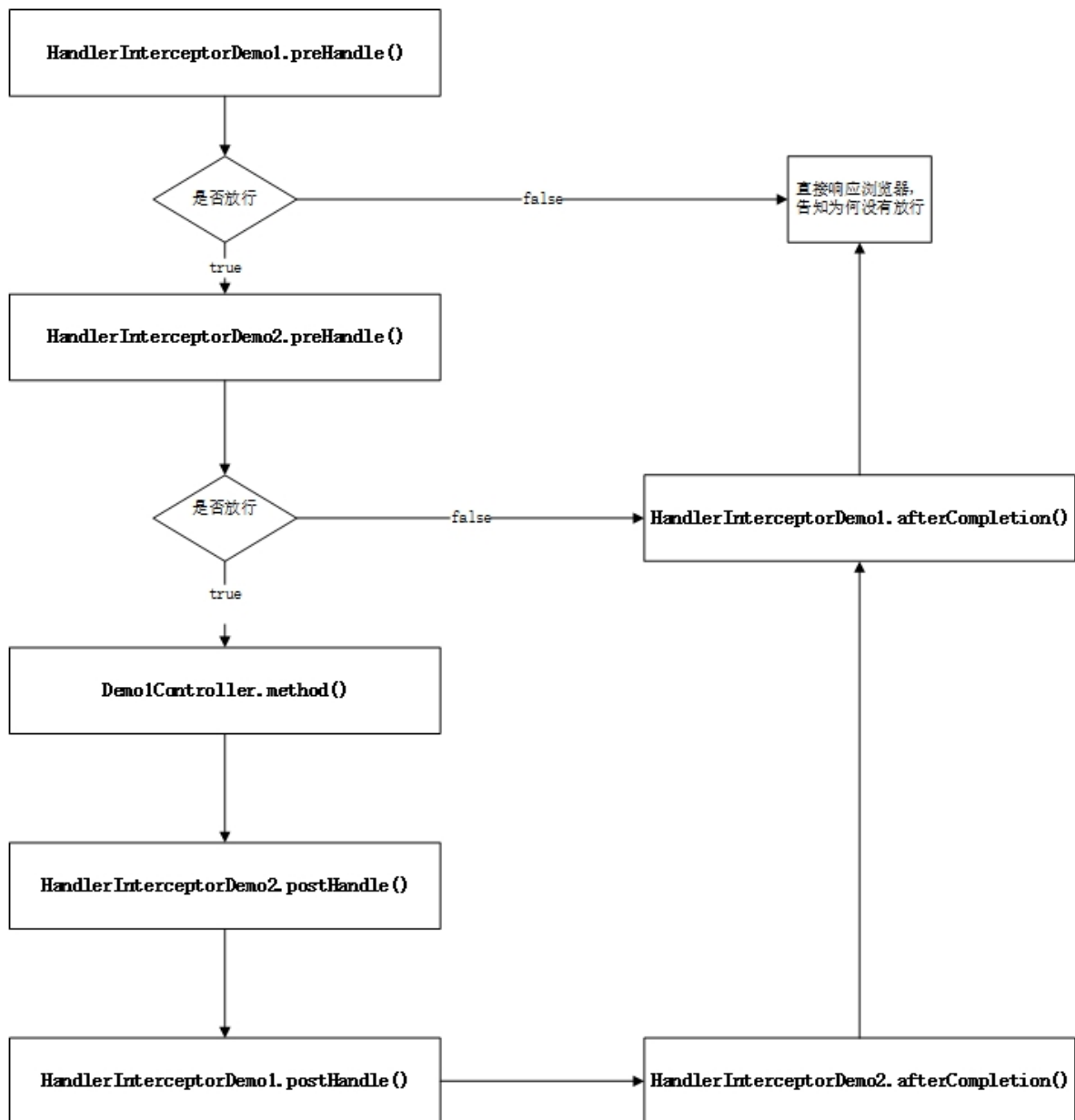
```
preHandle 执行了...
demo01 fun01()...
postHandle 执行了...
afterCompletion 执行了...
```

3.5 多个拦截器执行顺序

回想多个过滤器的执行顺序:

1. 如果采用配置文件方式配置过滤器, 那么就按照过滤器的配置先后顺序执行
2. 如果采用注解方式配置过滤器, 那么就按照类名的排序执行

我们可以配置多个拦截器, 所以就存在一个优先级问题了. 多个拦截器的优先级是按照配置的顺序决定的。



- 配置顺序

```

<!--配置拦截器-->
<mvc:interceptors>
  <mvc:interceptor>
    <!--用于指定拦截的路径-->
    <mvc:mapping path="/*"/>
    <!--用于指定忽略(不拦截的路径)-->
    <mvc:exclude-mapping path="/demo01/fun02"/>
    <bean id="interceptor01"
class="com.itheima.web.interceptor.Interceptor01"></bean>
  </mvc:interceptor>
  <mvc:interceptor>
    <!--用于指定拦截的路径-->
    <mvc:mapping path="/*"/>
    <!--用于指定忽略(不拦截的路径)-->
    <mvc:exclude-mapping path="/demo01/fun02"/>
    <bean id="interceptor02"
class="com.itheima.web.interceptor.Interceptor02"></bean>
  </mvc:interceptor>
</mvc:interceptors>

```

- 结果

```
Interceptor01 preHandle 执行了...
Interceptor02 preHandle 执行了...
demo01 fun01()...
Interceptor02 postHandle 执行了...
Interceptor01 postHandle 执行了...
Interceptor02 afterCompletion 执行了...
Interceptor01 afterCompletion 执行了...
```

4.小结

第四章-将面面项目改成SSM

第一步-引入依赖

1. spring相关依赖
 1. spring-webmvc
 2. spring-jdbc
 3. aop依赖
2. mybatis整合spring的依赖
3. log相关依赖
4. jackson相关的依赖
5. druid连接池依赖
6. 并且去掉heimaMvc的依赖

```
<properties>
  <spring.version>5.0.2.RELEASE</spring.version>
  <!-- 日志打印框架 -->
  <slf4j.version>1.6.6</slf4j.version>
  <log4j.version>1.2.12</log4j.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${spring.version}</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-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>
<!-- log start -->
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>${log4j.version}</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>${slf4j.version}</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>${slf4j.version}</version>
</dependency>
<!-- log end -->
<!--
      关键：mybatis整合spring的依赖
-->
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis-spring</artifactId>
  <version>1.3.0</version>
</dependency>
<!--druid-->
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.5</version>
</dependency>
<!--SpringAOP相关的坐标-->
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.8.7</version>
</dependency>

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

```

第二步-修改mybatis的核心配置文件

去掉environments、mappers等等配置，保留别名包扫描

第三步-编写spring整合mybatis的配置文件

1. 整合mybatis
2. 配置事务管理者
3. 加载事务注解驱动

第三步-编写spring的主配置文件

1. 包扫描
2. 加载mvc注解驱动

第四步-修改web.xml配置

1. 将heimaMVC中的DispatcherServlet的配置改成SpringMVC中的DispatcherServlet的配置
2. 将解决乱码的过滤器的配置改成SpringMVC中的解决乱码的过滤器配置

第五步-修改各个controller的代码

1. 将原本heimaMvc的Controller注解改成springMvc中的RestController注解
2. 使用Autowired注解注入Service对象
3. 在各个Controller类上添加RequestMapping注解
4. 将Controller类的各个方法上的RequestMapping注解改成SpringMVC中的RequestMapping注解，并且修改路径
5. 将各个方法的获取请求参数的方式改成SpringMVC中的获取请求参数
6. 将各个方法的响应数据给客户端的方式改成SpringMVC的响应数据方式

第六步-修改各个Service的代码

1. 使用Service注解进行IOC
2. 使用Autowired注解注入Dao对象
3. 将方法中所有涉及到SqlSession的代码全部删掉，直接就能使用Dao对象
4. 要进行事务控制的类或者方法上添加Transactional注解

第七步-修改文件上传的代码

1. 将原来的获取上传文件的代码改成SpringMVC的文件上传代码
2. 在创建spring文件上传的配置文件，配置文件解析器

第八步-在spring的主配置文件中引入其它的spring配置文件
