SpringMVC

一、SpringMVC 基础

1. 课程目标

- 掌握 SpringMVC 执行流程
- 了解 SpringMVC 基础配置
- 掌握 SpringMVC 注解开发
- 掌握 SpringMVC 参数绑定

2. 知识架构树

- SpringMVC 执行流程
- SpringMVC 基础配置
- SpringMVC 注解开发
- SpringMVC 参数绑定

3. 理论知识

3.1 SpringMVC 执行流程

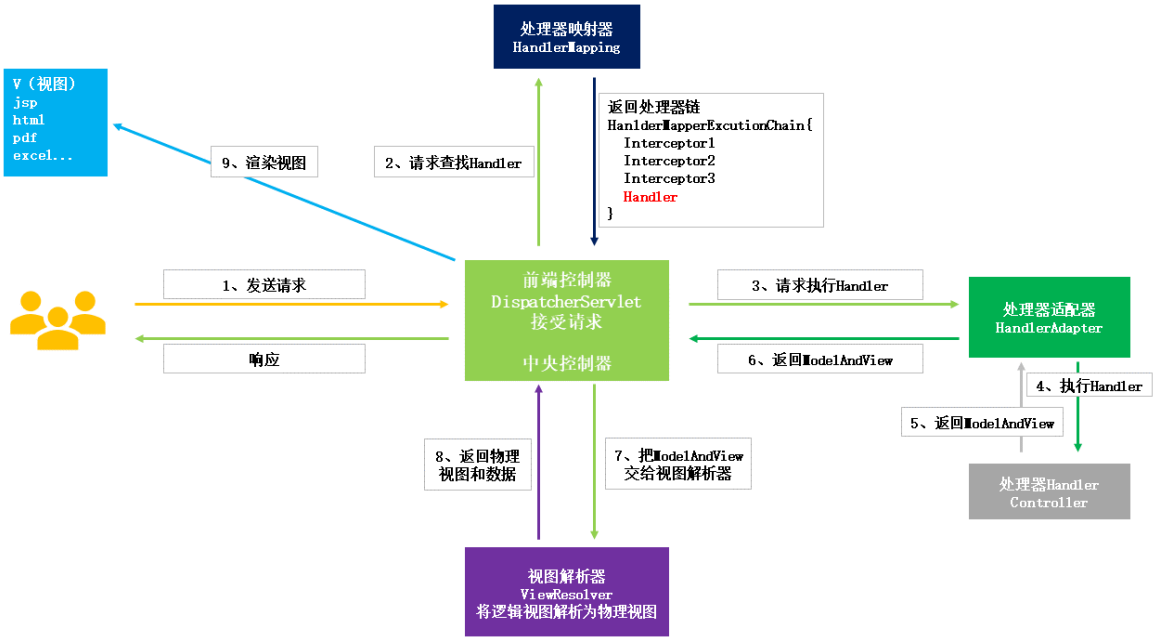
回顾：

```
Servlet 阶段代码的执行流程：
    页面 --> Servlet --> Service --> Dao --> DB
    页面 <-- Servlet <-- Service <-- Dao <-- |

页面到Servlet之间：生命周期
    页面 --> init --> Service --> doGet/doPost
```

```
需求：
    用户希望实现登录功能，在页面需要输入账号和密码。
    要将账号密码提交到Java 后台进行业务处理，判定登录的信息是否正确
    提交信息肯定有一个登录请求的地址：http://localhost:8080/login?user=admin&pass=admin
    后台想要处理账号密码，处理的功能肯定在一个类中。
    这个请求地址对应的一个类中某个方法。

业务角度看流程：
    拿着【/login】地址，找前端控制器（DispatcherServlet），这个路经你给我处理一下。作为管理者，它不回去解析这个路径，他就要找能解析的。
    前端控制器（DispatcherServlet）把【/login】地址给处理器映射器（HandlerMapping），找一下这个地址对应的是谁？找到之后就把结果反馈给前端控制器（DispatcherServlet）。
    前端控制器（DispatcherServlet）该处理这个类，但是心有余而力不足，找处理器适配器（HandlerAdapter）找合适的方法去执行登录请求
    执行完成之后将结果（ModelAndView）原路返回，给前端控制器（DispatcherServlet）。
    前端控制器（DispatcherServlet）把结果（ModelAndView）交给视图解析器(ViewResolver)去解析数据生成页面。
    最后把解析的结果给前端控制器（DispatcherServlet）把解析后的结果返回给用户。
```




```
SpringMVC基础组件：
    1、前端控制器(DispatcherServlet)：核心控制器，根据需求找对应的组件执行。
    2、处理器映射器(HandlerMapping)：根据用户的请求去匹配要执行指定的类（方法）。就是找路径和类（Controller）的对应关系。
    3、处理器适配器(HandlerAdapter)：根据要执行的类，匹配能执行的适配器。
    4、（*）处理器(Handler)：也就是控制层（Controller）业务需求的核心代码。
    5、视图解析器(ViewResolver)：把数据解析到页面，并做页面跳转。

1、用户发送请求到前端控制器（DispatcherServlet）,把请求交给处理器映射器（HandlerMapping）查找要执行的处理器
2、处理器映射器（HandlerMapping）返回执行器链（HandlerMappingExecutionChain）给前端控制器（DispatcherServlet）
3、前端控制器根据返回的执行器链给处理器适配器（HandlerAdapter）
4、处理器适配器（HandlerAdapter）请求执行处理器（Handler），返回ModelAndView给处理器适配器
5、处理器适配器（HandlerAdapter）把ModelAndView返回给前端控制器（DispatcherServlet）
6、前端控制器（DispatcherServlet）把ModelAndView交给视图解析器（ViewResolver）去解析，把逻辑视图解析为物理视图
7、视图解析器（ViewResolver）把解析好的物理视图和数据，返回给前端控制器（DispatcherServlet）
8、前端控制器（DispatcherServlet）把物理视图交给浏览器渲染
```

3.2 SpringMVC 基础配置

> Web 项目要求在服务器启动的时候加载所有的配置文件，以便于构建整体环境。由于 `web.xml` 是 Web 项目的核心配置文件，所以 SpringMVC 的配置要写在 `web.xml` 加载。由于 **SpringMVC 是一个Servlet的衍生产物**，所以要采用 Servlet 的配置方式。

导入依赖：

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.3.22</version>
</dependency>
自动依赖Spring其他的核心jar包
```

SpringMVC配置文件：spring-mvc.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!--2、处理器映射器：HandlerMapping-->
    <bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
    <!--3、处理器适配器：HandlerAdapter-->
    <bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>
    <!--4、处理器：Handler，自己编写的处理器Controller，name：浏览器的请求路径-->
    <bean name="/hello" class="com.soft.controller.HelloController"/>
    <!--5、视图解析器：ViewResolver-->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver"/>
</beans>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                             http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">
    <!--1、前端控制器：配置到web.xml-->
    <servlet>
        <servlet-name>SpringMVC</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!--
            web服务器启动时会默认从/WEB-INF/，找文件名是<servlet-name>-
servlet.xml
                SpringMVC -> SpringMVC-servlet.xml
            由于xml文件需要统一管理到resources目录下，所以需要使用<init-param>去初始
化加载
            自定义路径下的xml文件
```

```
        -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath*:spring-mvc.xml</param-value>
        </init-param>
        <!--加载的优先级顺序-->
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>SpringMVC</servlet-name>
        <!--
            路径的拦截就是看走不走DispatcherServlet，只有请求路径

            /：拦截所有的路径（不包含.jsp .html之类的路径），静态资源也会通过DispatcherServlet
            /*：拦截所有的路径
            *.后缀：拦截指定后缀的路径
        -->
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

控制器（类）：

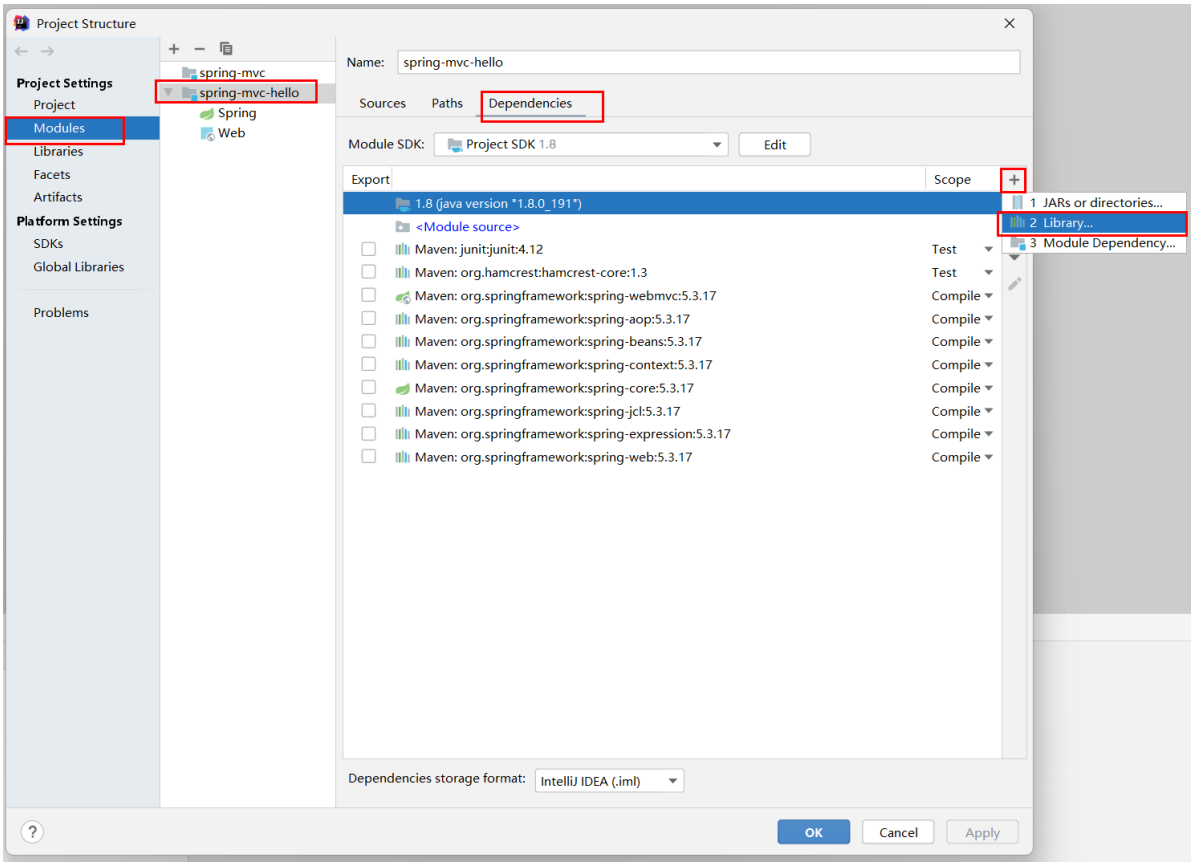
```
// 实现 Controller 接口目的是为了让处理器适配识别
public class HelloController implements Controller {
    /**
     * @param httpServletRequest 请求
     * @param httpServletResponse 响应
     * @return ModelAndView 模型（数据模型）和视图（逻辑视图）
     * @throws Exception
     */
    @Override
    public ModelAndView handleRequest(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse) throws Exception {

        // 模型视图对象
        ModelAndView mav = new ModelAndView();
        // 添加数据模型 req.setAttribute();
        mav.addObject("msg", "Hello SpringMVC By Xml");
        // 设置视图路径
        mav.setViewName("/index.jsp");

        return mav;
    }
}
```

HttpServletRequest 和 HttpServletResponse对象报错的解决方案：

- 方案一：在 pom.xm 中导入依赖
- 方案二（推荐）：在项目中导入 Tomcat



3.3 SpringMVC 注解配置

> 通过翻看底层源码 `DispatcherServlet.properties`，发现SpringMVC已经帮我们配置了默认组件，所以简化了开发流程，搭配注解的配置，是的配置代码大大减少。

```
# Default implementation classes for DispatcherServlet's strategy interfaces.
# Used as fallback when no matching beans are found in the DispatcherServlet context.
# Not meant to be customized by application developers.

org.springframework.web.servlet.LocaleResolver=org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver

org.springframework.web.servlet.ThemeResolver=org.springframework.web.servlet.theme.FixedThemeResolver

org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\

org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping,\
    org.springframework.web.servlet.function.support.RouterFunctionMapping

org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\
    org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\

org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter,\
    org.springframework.web.servlet.function.support.HandlerFunctionAdapter
```

```
org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.servlet.mvc.method.annotation.ExceptionHandlerExceptionResolver,\
org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\
org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver

org.springframework.web.servlet.RequestToViewNameTranslator=org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator

org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResourceViewResolver

org.springframework.web.servlet.FlashMapManager=org.springframework.web.servlet.support.SessionFlashMapManager
```

注解介绍：

```
@Controller：Spring 注解
    1、标识类是一个处理器
    2、管理所在类，在Spring 容器中创建对象（由于@Controller继承了@Component）

@RestController：SpringMVC 注解，@Controller 和 @ResponseBody 组合注解
    1、标识类是一个处理器
    2、管理所在类，在 Spring 容器中创建对象（由于@Controller继承了@Component）
    3、当前类中所有的方法都返回 JSON 数据

@RequestMapping：SpringMVC 注解，定义浏览器的请求路径，可以添加到类上和方法上
    value/path：请求的路径
    method：标识方法能接收什么样的请求方式，可以从枚举类RequestMethod中选择，GET/POST/DELETE/PUT...
    param：要求请求的参数必须携带指定参数，必要时还可以指定参数的名称

    @GetMapping：接收Get请求
    @PostMapping：接收Post请求
    @DeleteMapping：接收Delete请求
    @PutMapping：接收Put请求
```

开发流程：

1、编写控制类（处理器）

```
package com.soft.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

```

```java
@Controller
public class HelloController{

    @RequestMapping(value={"/hello", "/hi"}, method = {RequestMethod.GET, RequestMethod.POST})
    public ModelAndView hello(HttpServletRequest req, HttpServletResponse resp){
        ModelAndView mav = new ModelAndView();
        // 添加数据模型 req.setAttribute();
        mav.addObject("msg", "Hello SpringMVC By Annotation");
        // 设置视图路径
        mav.setViewName("/index.jsp");
        return mav;
    }
}
```

```
控制器方法编写：
    1、方法参数：取决于页面是否需要传递参数
    2、方法返回值：取决于需求
        1）只返回页面：返回值String，此时不建议添加@RestController和@ResponseBody两个注解
        2）有页面和数据：返回值ModelAndView
```

2、编写SpringMVC配置文件，由于是注解开发，只需要开启注解的开关即可

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <context:component-scan base-package="com.soft.controller"/>
</beans>
```

3、web.xml中配置SpringMVC的核心控制器以及加载SpringMVC的配置文件

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">

    <servlet>
        <servlet-name>SpringMVC</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
```

```
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath*:spring-mvc4.xml</param-value>
        </init-param>
        <!--加载的优先级顺序-->
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>SpringMVC</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

4、编写jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    ${msg}
</body>
</html>
```

3.4 SpringMVC 参数绑定

> 请求参数绑定就是将页面用户输入的数据，绑定到控制层对应的方法上。接受到参数之后可以进行业务处理。

3.4.1 ServletAPI

> 控制层方法通过`getParameter()`获取参数的值。可以获取 from 表单和 URL 传递的参数。
>
> 获取参数要求：
>
> 1、根据表单 name 属性的值获取
>
> 2、根据 URL 的 Key 获取

```
<form action="${pageContext.request.contextPath}/servletAPI" method="get">
    输入框：<input type="text" name="user"/><br>
    密码框：<input type="password" name="pass"/><br>
    <input type="submit" value="提交">
</form>

<a href="${pageContext.request.contextPath}/servletAPI?
user=1001&pass=123456">ServletAPI</a>
```

```
/**
 * HttpServletRequest:
 *  1、获取页面参数
 *  2、操作 Session
 */
@RequestMapping("/servletAPI")
```

```
public void servletAPI(HttpServletRequest req) {
    // 获取请求参数
    String user = req.getParameter("user");
    String pass = req.getParameter("pass");

    System.out.println(user);
    System.out.println(pass);
}
```

3.4.2 方法参数绑定：单一参数

> 将页面传递的参数绑定到控制层方法的普通参数上。控制层方法参数个数增加时，维护不便，代码可读性不强。
>
> 获取参数要求：
>
> 1、控制层方法参数名称和表单 name 属性的值一致，包括大小写。
>
> 2、控制层方法参数名称和URL路径 Key 一致，包括大小写。
>
> 3、页面和控制层方法参数无法对应时，可以使用`@RequestParam("页面参数名称")`映射。

```
<h1>请求参数绑定（单一参数：from提交）</h1>
<form action="${pageContext.request.contextPath}/baseArgs" method="get">
    输入框：<input type="text" name="user"/><br>
    密码框：<input type="password" name="pass"/><br>
    单选框：<input type="radio" name="sex" value="1"/>男<input type="radio"
name="sex" value="0"/>女<br>
    复选框：<input type="checkbox" name="hobby" value="1"/>运动
        <input type="checkbox" name="hobby" value="2"/>学习
        <input type="checkbox" name="hobby" value="3"/>游戏<br>
    下拉框： <select name="city">
                <option value="1">北京</option>
                <option value="2">河南</option>
                <option value="3">河北</option>
            </select><br>
    <input type="submit" value="提交">
</form>

<h2>请求参数绑定（单一参数：URL提交）</h2>
<a href="${pageContext.request.contextPath}/baseArgs?
user=1001&pass=123456&sex=1&radio=1&hobby=1&hobby=2&hobby=3&city=2">URL请求
</a>
```

```
/*
 * SpringMVC自带类型转化器，可以将页面传递的参数转换成指定类型
 */
@RequestMapping("/baseArgs")
public void baseArgs(@RequestParam("user") String name, String pass,
                             Integer sex, String[] hobby, String option,
Integer number) {
    System.out.println(name);
    System.out.println(pass);
    System.out.println(sex);
    System.out.println(hobby);
    System.out.println(Arrays.toString(hobby));
```

```
    System.out.println(option);
    System.out.println(number);
}
```

3.4.3 方法参数绑定：对象参数

> 将页面传递的参数绑定到控制层方法的对象参数上。将需要的参数封装为实体类对象，便于传值的同时也便于代码维护，可读性更高。
>
> 获取参数要求：
>
> 1、表单 name 属性的值和参数中属性的 set 方法去掉 set 之后剩余内容首字母小写一致，包括大小写。
>
> 2、URL路径 Key 和参数中属性的 set 方法去掉 set 之后剩余内容首字母小写一致，包括大小写。

```
<h1>请求参数绑定（对象参数：from提交）</h1>
<form action="${pageContext.request.contextPath}/entityArgs" method="get">
    输入框：<input type="text" name="user"/><br>
    密码框：<input type="password" name="pass"/><br>
    单选框：<input type="radio" name="sex" value="1"/>男<input type="radio" name="sex" value="0"/>女<br>
    复选框：<input type="checkbox" name="hobby" value="1"/>运动
        <input type="checkbox" name="hobby" value="2"/>学习
        <input type="checkbox" name="hobby" value="3"/>游戏<br>
    下拉框： <select name="city">
                <option value="1">北京</option>
                <option value="2">河南</option>
                <option value="3">河北</option>
            </select><br>
    <input type="submit" value="提交">
</form>

<h2>请求参数绑定（对象参数：URL提交）</h2>
<a href="${pageContext.request.contextPath}/entityArgs?user=1001&pass=123456&sex=1&radio=1&hobby=1&hobby=2&hobby=3&city=2">URL请求</a>
```

```
/*
 * SpringMVC自带类型转化器，可以将页面传递的参数转换成指定类型
 */
@RequestMapping("/entityArgs")
public void entityArgs(Param param) {
    System.out.println(param);
}
```

```java
public class Param{
    private String user;
    private String pass;
    private String sex;
    private String[] hobby;
    private String city;

    // 省略set/get
}
```

3.4.4 方法参数绑定：Map 映射

> 将页面传递的参数绑定到控制层方法的Map参数上。表单组件的 name 属性值（URL参数的Key）会作为 Map 集合的 key 存储。需要注意的是，基于 Map 的特性，同一个 Key 只能存在一个，所以这种方式不适合复选框传值。
>
> 获取参数要求：Map 参数添加`@RequestParam`注解

```jsp
<h1>请求参数绑定（Map 映射：from提交）</h1>
<form action="${pageContext.request.contextPath}/mapArgs" method="get">
    输入框：<input type="text" name="user"/><br>
    密码框：<input type="password" name="pass"/><br>
    单选框：<input type="radio" name="sex" value="1"/>男<input type="radio" name="sex" value="0"/>女<br>
    复选框：<input type="checkbox" name="hobby" value="1"/>运动
        <input type="checkbox" name="hobby" value="2"/>学习
        <input type="checkbox" name="hobby" value="3"/>游戏<br>
    下拉框： <select name="city">
                <option value="1">北京</option>
                <option value="2">河南</option>
                <option value="3">河北</option>
            </select><br>
    <input type="submit" value="提交">
</form>

<h2>请求参数绑定（Map 映射：URL提交）</h2>
<a href="${pageContext.request.contextPath}/mapArgs?user=1001&pass=123456&sex=1&radio=1&hobby=1&hobby=2&hobby=3&city=2">URL请求</a>
```

```java
@RequestMapping("/mapArgs")
public void mapArgs(@RequestParam Map<String, Object> param) {
    System.out.println(param);
}
```

3.4.5 方法参数绑定：组合参数

> 组合参数适用于使用对象参数绑定之后，为了不破坏实体类的写法，还希望传递一些其他数据。组合参数写法是额外编写一个实体类，定义 Map 属性用于接收实体类以外的参数。

```jsp
<form action="${pageContext.request.contextPath}/mapArgs" method="get">
    输入框：<input type="text" name="user"/><br>
    密码框：<input type="password" name="pass"/><br>
```

```
    单选框：<input type="radio" name="sex" value="1"/>男<input type="radio" name="sex" value="0"/>女<br>
    复选框：<input type="checkbox" name="hobby" value="1"/>运动
        <input type="checkbox" name="hobby" value="2"/>学习
        <input type="checkbox" name="hobby" value="3"/>游戏<br>
    下拉框： <select name="city">
                <option value="1">北京</option>
                <option value="2">河南</option>
                <option value="3">河北</option>
            </select><br>

    <!-- 额外参数 -->
    额外参数1：<input type="checkbox" name="params[language]" value="1"/>Java
        <input type="checkbox" name="params[language]" value="2"/>C
        <input type="checkbox" name="params[language]" value="3"/>Python<br>

    额外参数2：<input type="number" name="params[year]"/><br>

    <input type="submit" value="提交">
</form>
```

```
@RequestMapping("/difArgs")
public void difArgs(Param param) {
    System.out.println(param);
}
```

```
public class Param extends BaseEntity{
    private String user;
    private String pass;
    private String sex;
    private String[] hobby;
    private String city;

    // 省略set/get
}

private class BaseEntity{
    Map<String, Object> params;

    // 省略set/get
}
```

3.4.6 方法参数绑定：URL占位符

> 将数据作为请求路径的一部分进行传递，而不是通过问号拼接的形式传递参数。这样可以保障一定的安全性，同时还支持问号拼接传值。
>
> 获取参数要求：
>
> 1、控制层请求路径是用`{name}`占位
>
> 2、控制层方法参数添加注解`@PathVariable`
>
> 3、占位符的名字和方法的参数名字一致，包括大小写，不一致时可以使用注解`@PathVariable("名字")`映射

```
<a href="${pageContext.request.contextPath}/urlArgs/1001?pass=123456&sex=1">URL请求</a>
```

```
@RequestMapping("/urlArgs/{user}")
public void urlArgs(@PathVariable String user, Param param) {
    System.out.println(user);
    System.out.println(param);
}
```

3.5 SpringMVC JSON 操作

3.5.1 返回 JSON 数据

> SpringMVC 返回 JSON 数据一般应用于异步请求。SpringMVC 对这类是数据的返回进行了封装，写法上也更加简单。只需要定义好返回值类型，搭配注解`@ResponseBody`即可返回数据

3.5.1.1 返回字面量数据

```
@GetMapping("/base")
@ResponseBody
public String base(){
    return "ok";
}
```

3.5.1.2 返回对象数据

> 返回对象数据需要借助其他依赖，同时开启配置

导入依赖

```
<dependency>
    <groupId>com.jwebmp.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>0.63.0.19</version>
</dependency>
```

```
@GetMapping("/obj")
@ResponseBody
public String obj(){
    Param param = new Param("1001", "123456");
    return param;
}
```

```
<mvc:annotation-driven/>
```

3.5.1.3 返回集合数据

```java
@GetMapping("/list")
@ResponseBody
public String list(){
    List<Param> list = new ArrayList();
    list.add(new Param("1001", "123456"));
    list.add(new Param("1002", "123456"));
    list.add(new Param("1003", "123456"));

    return list;
}
```

3.5.1.4 中文乱码

方案一： produces 属性

```java
@RequestMapping(value = "", method = {RequestMethod.GET}, produces = {"text/html;charset=utf-8"})
```

方案二：SpringMVC全局配置

```xml
<mvc:annotation-driven>
    <mvc:message-converters>
        <bean class="org.springframework.http.converter.StringHttpMessageConverter">
            <property name="supportedMediaTypes">
                <list>
                    <value>text/html;charset=UTF-8</value>
                    <value>application/json;charset=UTF-8</value>
                    <value>text/plain;charset=UTF-8</value>
                    <value>application/xml;charset=UTF-8</value>
                </list>
            </property>
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>
```

方案三：Response设置响应头信息

```java
response.setContentType("text/html;charset=utf-8");
```

3.5.2 接收 JSON 数据

> @RequestBody 主要用来接收前端传递给后端的 **JSON字符串** 中的数据的(请求体中的数据的)；
>
> GET方式无请求体，所以使用 @RequestBody 接收数据时，前端不能使用GET方式提交数据，而是用POST方式进行提交。
>
> @RequestBody 注解常用来处理 content-type 不是默认的 application/x-www-form-urlcoded 编码的内容(form表单和URL传值形式)。一般情况下来说常用其来处理 **application/json** 类型。(jquery异步请求默认是application/x-www-form-urlcoded)

```java
@RequestMapping(path="/sendJson2")
public void requestJson(@RequestBody Student student){
    System.out.println(student);
}
```

```jsp
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
    <script src="${pageContext.request.contextPath}/js/jquery-3.5.1.min.js">
</script>
</head>
<body>
    <button onclick="json2()">对象接收JSON数据</button>
    <script>
        function json2(){
            var obj = {
                'name': 'Tom',
                'pass': '123456',
                'sex': '男',
                'hobby': '1,2,3,4,6'
            };

            $.ajax({
                url:"/json/sendJson1",
                type:"post",  // 重点★
                contentType:"application/json;charset=utf-8", // 重点★
                data: JSON.stringify(obj),  // 把对象转换为字符串 重点★
                dataType:"json", // 方法返回值的类型
                success:function(data){
                    console.log(data);
                }
            });
        }
    </script>
</body>
</html>
```

```
控制器接收页面JSON数据(JSON格式的数据)
    控制器方法的参数添加：@RequestBody
    异步请求添加：contentType: "application/json;charset=utf-8"
```

3.6 SpringMVC 控制层方法

- 同步请求

```java
/**
 * 跳转JSP页面，不带参数（推荐）
 */
public String toPage(){
    return "/index.jsp";
}
/**
```

```java
 * 跳转JSP页面，带参数
 */
public ModelAndView mav(){
    ModelAndView mav = new ModelAndView();
    // 页面所需要的参数
    mav.addObject("msg", "");
    mav.setViewName("/index.jsp");
    return mav;
}
/**
 * 跳转JSP页面，带参数
 */
public String mav(Model model){
    // 页面所需要的参数
    model.addAttribute("msg", "");
    return "/index.jsp";
}
/**
 * 跳转JSP页面，带参数（推荐）
 */
public String mav(ModelMap mmap){
    // 页面所需要的参数
    mmap.put("msg", "");
    return "/index.jsp";
}
```

- 异步请求

```java
/**
 * 只返回数据
 */
@ResponseBody
public String backString(){
    return "结果！";
}
@ResponseBody
public CustomParam backObj(){
    return null;
}
@ResponseBody
public List<CustomParam> backList(){
    return null;
}
```

4. 综合实验

- 实现基础学生管理系统

5. 作业实践

- 创建 SpringMVC 项目
- 通过 SpringMVC 实现注册、登录、信息展示

二、SpringMVC 高级

1. 课程目标

2. 知识架构树

3. 理论知识

3.1 自定义类型转换器

> SpringMVC 底层已经封装了很多的类型转换器，也就是为什么我们页面上传的字符串可以使用Integer接收或者可以直接转换为数组的原因。
>
> 并不是所有类型的字符都可以正常转换，比如日期字符 “yyyy/mm/dd” 可以正常转换，而 “yyyy-mm-dd” 就不能转换。
>
> 怎么来决解这个问题，就需要自定义类型转换器来解决。

Converter.java

```
package org.springframework.core.convert.converter;

import org.springframework.lang.Nullable;

/**
 * A converter converts a source object of type {@code S} to a target of type {@code T}.
 *
 * <p>Implementations of this interface are thread-safe and can be shared.
 *
 * <p>Implementations may additionally implement {@link ConditionalConverter}.
 *
 * @author Keith Donald
 * @since 3.0
 * @param <S> the source type 源数据类型，页面传值的字符串
 * @param <T> the target type 目标类型，字符串类型要转成的目标类型
 */
@FunctionalInterface
public interface Converter<S, T> {

    /**
     * Convert the source object of type {@code S} to target type {@code T}.
     * @param source the source object to convert, which must be an instance of {@code S} (never {@code null})
     * @return the converted object, which must be an instance of {@code T} (potentially {@code null})
     * @throws IllegalArgumentException if the source cannot be converted to the desired target type
     */
    @Nullable
    T convert(S source);
}
```

StringToDateConverter.java

```
package com.soft.converter;
```

```java

import org.springframework.core.convert.converter.Converter;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * 字符类型转换日期类型
 * 通过调查源码得知，内置的转换器都实现了Converter<s, t>接口，所以要想让SpringMVC识别自定义的转换器，也需要实现这个接口。
 **/
public class StringToDateConverter implements Converter<String, Date> {
    /***
     * 字符类型转换为日期类型
     * @param s 页面传递的数据
     * @return
     */
    @Override
    public Date convert(String s) {

        // 判定字符为空时返回系统时间
        if(s == null || "".equals(s)){
            return null;
        }

        // 格式是：yyyy-mm-dd
        // 格式是：yyyy/mm/dd
        // 字符和日期怎么转换
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy/mm/dd");

        // 把字符中的横杠替换为斜杠，这样的目的可以让程序员之判定一种格式
        String replace = s.replace("-", "/");

        Date parse = null;

        try {
            parse = sdf.parse(replace);
        } catch (ParseException e) {
            return null;
        }

        return parse;
    }
}
```

SpringMVC是基于组件开发的一个框架，每个框架要想使用必须要先注册，所以自定义类之后要注册到SpringMVC服务中

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
```

```
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/mvc
        https://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <context:component-scan base-package="com.soft"/>

    <!-- 类型转换器服务工厂 -->
    <bean id="conversionService"
class="org.springframework.context.support.ConversionServiceFactoryBean">
        <property name="converters">
            <set>
                <!--把自定义类型转换器注册到服务中-->
                <bean class="com.soft.converter.StringToDateConverter"/>
            </set>
        </property>
    </bean>

    <mvc:annotation-driven conversion-service="conversionService"/>
</beans>
```

3.2 重定向和转发

```
转发（forward）:SpringMVC 默认的响应方式是转发（forward）。
特点：
    请求地址不发生改变
    可以携带数据（可以把数据放到域对象中传递）
    只能请求本项目中的路径

重定向(redirect)：在返回的请求路径前添加【redirect:】即可，
    源路径：【success.jsp】、【/login】
    重定向：【redirect:success.jsp】、【redirect:/login】
特点：
    请求地址发生改变
    不可以携带数据
    可以请求本项目和本项目以外的路径
    重定向不经过核心控制器
    重定向的路径不推荐写视图路径，/WEB-INF/目录之外的视图除外
        （重定向就相当于从浏览器直接请求，WEB-INF 中的内容是受保护的，不能访问）
```

3.3 文件上传下载

3.3.1 文件上传（服务器）

> 文件上传要求：
>
> 1、要求form表单的请求方式必须是POST 2、要求form表单添加属性，enctype=“multipart/form-data” 3、配置文件上传的解析器，id 属性的值必须是 `multipartResolver` 4、处理器的参数要求添加参数 `MultipartFile file`，同时参数名称和form表单的文件上传组件的 name 属性值一致，包括大小写

导入依赖

```xml
<dependency>
    <groupId>commons-fileupload</groupId>
    <artifactId>commons-fileupload</artifactId>
    <version>1.4</version>
</dependency>
```

配置文件上传解析器

```xml
<!--文件上传的解析器-->
<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!--编码格式-->
    <property name="defaultEncoding" value="utf-8"/>
    <!--文件上传的大小限制，以字节的形式配置，-1 表示没有限制-->
    <property name="maxUploadSize" value="-1"/>
</bean>
```

视图页面

```xml
<form action="${pageContext.request.contextPath}/upload" method="post"
enctype="multipart/form-data">
    <input type="text" name="user">
    <input type="file" name="file"/>
    <input type="submit" value="提交">
</form>
```

控制器

```java
// 表单提交文件组件的 name 属性的值要和参数的名字一致
@PostMapping("/upload")
public String upload(@RequestParam("file") MultipartFile multipartFile,
HttpServletRequest req) throws Exception {

    // 获取用户上传的文件名
    String fileName = multipartFile.getOriginalFilename();
    // 为了避免服务器重名，构建随机名字
    fileName = UUID.randomUUID().toString().replace("-", "") + "-" +
fileName;

    // 获取服务器路径
    String serverPath = req.getServletContext().getRealPath("/img/");
    // 文件在服务器上的路径
    String filePathInServer = serverPath + fileName;

    File serverFile = new File(path);
    // 判断文件是否存在，不存在就创建文件夹
    if(!serverFile.exists()){
        serverFile.mkdirs();
    }

    // 把文件写入到磁盘
    multipartFile.transferTo(new File(filePathInServer));

```

```
    return "success.jsp";
}
```

3.3.2 文件上传（远程服务器）

导入依赖

```
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-client</artifactId>
    <version>1.19.4</version>
</dependency>
<dependency>
    <groupId>com.sun.jersey</groupId>
    <artifactId>jersey-core</artifactId>
    <version>1.19.4</version>
</dependency>
```

控制器

```
// 表单提交文件组件的 name 属性的值要和参数的名字一致
@PostMapping("/upload")
public String upload(@RequestParam("file") MultipartFile multipartFile,
HttpServletRequest req) throws Exception {

    // 获取用户上传的文件名
    String fileName = multipartFile.getOriginalFilename();
    // 为了避免服务器重名，构建随机名字
    fileName = UUID.randomUUID().toString().replace("-", "") + "-" +
fileName;

    // 获取服务器路径
    String serverPath = "http://localhost:9090/FileServer/img/";
    // 文件在服务器上的路径
    String filePathInServer = serverPath + fileName;

    // 客户端
    Client client = Client.create();

    // 通过客户端获取的连接
    WebResource resource = client.resource(serverPath);

    resource.put(multipartFile.getBytes());

    return "success.jsp";
}
```

跨域上传文件需要配置 Tomcat 安装目录下的 web.xml

```
<servlet>
    <servlet-name>default</servlet-name>
    <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-
class>
    <init-param>
```

```
        <param-name>debug</param-name>
        <param-value>0</param-value>
    </init-param>
    <init-param>
        <param-name>listings</param-name>
        <param-value>false</param-value>
    </init-param>
    <!-- 跨域配置 -->
    <init-param>
        <param-name>readonly</param-name>
        <param-value>false</param-value>
    </init-param>
    <!-- 跨域配置 -->
    <load-on-startup>1</load-on-startup>
</servlet>
```

3.3.3 文件下载

> 把文件内容 **响应** 到浏览器的过程
>
> 要求：配置下载的头信息："**content-disposition**","**attachement;filename=xx.jpg**"

返回 ResponseEntity<byte[]> 对象

```
@RequestMapping("/download1")
public ResponseEntity<byte[]> download1(String file, HttpServletRequest request) throws IOException {
    // 服务路径
    String serverPath = request.getServletContext().getRealPath("/upload/");
    // 文件的完整路径
    String path = serverPath + file;

    // 把文件读取到字节数组中
    FileInputStream fis = new FileInputStream(new File(path));
    byte[] bytes = new byte[fis.available()];
    fis.read(bytes);

    // 设置下载的响应头信息
    HttpHeaders hh = new HttpHeaders();
    hh.setContentDispositionFormData("attachement", "new1.jpg");

    return new ResponseEntity<byte[]>(bytes,  hh, HttpStatus.OK);
}
```

通过 response 响应

```
@RequestMapping("/download2")
public void download2(String file, HttpServletRequest request, HttpServletResponse resp) throws IOException {
    // 服务路径
    String serverPath = request.getServletContext().getRealPath("/upload/");
    // 文件的完整路径
    String path = serverPath + file;

    // 把文件读取到字节数组中
```

```
    FileInputStream fis = new FileInputStream(new File(path));
    byte[] bytes = new byte[fis.available()];
    fis.read(bytes);

    // 设置下载的响应头信息
    resp.setHeader("content-disposition","attachement;filename=new2.jpg");

    ServletOutputStream outputStream = resp.getOutputStream();
    outputStream.write(bytes);
    outputStream.flush();
    outputStream.close();
}
```

3.4 异常处理器

> 在业务流程的处理中经常会发生异常，如果这些异常不进行处理，会直接展示在页面中，对用户来说是不有好的。为了在出现异常使给用户展示友好的页面，需要自定义处理异常。

1、自定义异常类，用于自定义异常

```
package com.soft.exception;

public class SysException extends Exception {

    private String message;

    @Override
    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public SysException(String message) {
        this.message = message;
    }
}
```

2、自定义异常处理器，用于在发生异常时候给用户一个友好的界面。

要求：实现HandlerExceptionResolver接口，重写resolveException方法

```
public class SysExceptionResolver implements HandlerExceptionResolver {
    /**
     * 异常处理
     * @param request Http请求
     * @param response Http响应
     * @param handler 核心代码抛出异常所在的方法
     * @param ex 核心代码抛出的异常
     * @return
     */
    @Override
```

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

        SysException se = null;

        // 判断抛出的异常是否是自定义异常
        if(ex instanceof SysException){
            // 把异常强转为自定义异常
            se = (SysException) ex;
        } else {
            // 创建自定义异常
            se = new SysException("系统维护中...");
        }

        ModelAndView mav = new ModelAndView();
        // 异常错误信息展示页面
        mav.setViewName("/page/500");
        // 异常错误信息
        mav.addObject("errMsg", se.getMessage());

        return mav;
    }
}
```

3、把异常处理器组件添加到SpringMVC中，在SpringMVC配置文件中直接使用 `<bean>` 标签配置即可或者在类上添加 `@Component` 注解

```
<!--配置自定义的常处理器-->
<bean class="com.soft.exception.SysExceptionResolver"/>
```

Spring异常	HTTP状态码
BindException	400 - Bad Request
ConversionNotSupportedException	500 - Internal Server Error
HttpMediaTypeNotAcceptableException	406 - Not Acceptable
HttpMediaTypeNotSupportedException	415 - Unsupported Media Type
HttpMessageNotReadableException	400 - Bad Request
HttpMessageNotWritableException	500 - Internal Server Error
HttpRequestMethodNotSupportedException	405 - Method Not Allowed
MethodArgumentNotValidException	400 - Bad Request
MissingServletRequestParameterException	400 - Bad Request
MissingServletRequestPartException	400 - Bad Request
NoSuchRequestHandlingMethodException	404 - Not Found
TypeMismatchException	400 - Bad Request

3.5 拦截器

3.5.1 拦截器和过滤器的区别

1. 过滤器：`Filter`；拦截器：`Interceptor`；
2. 过滤器（Filter）是 Servlet 的一个标准组件，任何的 web 项目都可以使用；
3. 拦截器（Interceptor）是 SpringMVC 的组件，只能在 SpringMVC 中使用；
4. **拦截器只拦截控制器的请求路径**，过滤器可以过滤所有的路径（控制层的路径和静态资源路径）；

3.5.2 自定义拦截器

1、编写自定义拦截器类，要求该类继承`HandlerInterceptorAdapter`类或者实现`HandlerInterceptor`接口，重写相应方法

```
// 在控制器方法执行之前执行
// 返回true：放行；返回false：不放行
default boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler);
// 在控制器方法执行之后，页面加载之前执行
default void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView);
// 页面加载之后执行
default void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex);
```

2、在springMVC配置文件中添加拦截器组件

```
<!--方式一：-->
<!--拦截器配置-->
<mvc:interceptors>
    <mvc:interceptor>
        <!--配置拦截路径-->
        <!--要拦截的路径，可以使用通配符-->
        <mvc:mapping path="/user/*"/>
        <!--不要拦截的路径-->
        <mvc:exclude-mapping path=""/>

        <!--引入自定义拦截器类，就是拦截的路径要走的那个类-->
        <bean class="com.soft.interceptor.MyInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>

<!--
    方式二：所有的路径都要经过拦截器
    编码设置
-->
<!--拦截器配置-->
<mvc:interceptors>
    <bean class="com.soft.interceptor.MyInterceptor"/>
</mvc:interceptors>
```

3.5.3 拦截器案例

3.5.3.1 编码格式

```
public class EncodingInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        request.setCharacterEncoding("UTF-8");
        response.setCharacterEncoding("UTF-8");
        return true;
    }

```

```
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        request.setCharacterEncoding("UTF-8");
        response.setCharacterEncoding("UTF-8");
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {
        request.setCharacterEncoding("UTF-8");
        response.setCharacterEncoding("UTF-8");
    }
}
```

3.5.3.2 登录/权限

```
/**
 * 会话拦截器，解决没有登录就不让操作的功能
 */
public class SessionInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        HttpSession session = request.getSession();
        Object login = session.getAttribute("login");
        if(login == null){
            response.sendRedirect("/login.jsp");
        } else {
            return true;
        }
        return false;
    }
}
```

3.5.3.3 防盗链

> 防止非法盗取连接对应的资源。比如：授权的视频网站，被三方的软件盗取链接，获取视频资源。
>
> 防盗链需要用到一个头信息【referer】，可以获取请求源的地址。

```
import org.springframework.web.servlet.HandlerInterceptor;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class LinkInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        String referer = request.getHeader("referer");
        System.out.println(referer);
        if(referer.startsWith("http://localhost:8080/")){
            return true;
```

```
            }
            response.sendRedirect("/login.jsp");
            return false;
        }
    }
```

3.5.3.4 表单重复提交

- 解决方案：

（1）form表单提交，在提交之后通过JS将提交按钮禁用（disabled），防止再次点击；

（2）在请求的路径上添加一个标识（令牌），每次提交数据校验标识是否合法（Token）；

- 实现思路：

```
需求：
    逛动物园
条件：
    1、买票，从官网的售票处（官方渠道）
    2、验票，官方的验票

总结：
    去动物园之前：买票
    进动物园验票：动物园工作人员进行验票
    票的来源：动物园官方发售的票

票：
    称为令牌(Token)
--------------------------------------------------------------------------
目的：逛动物园（核心代码：登录处理、注册处理、添加商品）

过程：
    买门票（生成令牌）：处理核心代码之前先生成令牌
    进动物园时（校验令牌）：对令牌进行校验

总结：
    Java 后台生成令牌，存储起来，给用户一个额外的拷贝
    用户提交数据的时候带着令牌提交
    Java 后台验证令牌真伪
--------------------------------------------------------------------------

需求：
    登录操作(用于在登录页面输入用户信息之后，点击登录进行用户信息及验证的过程)
分析：
    1、在登录的时候需要携带令牌（进动物园的时候，需要携带门票进行校验）也就意味着登录的时候就已经有了令牌了。
    2、登录的时候是从登录页面点击的提交，也就意味着，登录页面是有令牌的。
    3、登陆页面的令牌肯定是从后台生成之后才有的。所以要先走后台生成令牌，然后跳转登录页面，此时登录页面才会有令牌。

模块划分：
    1、跳转登录页面：生成令牌
    2、登录功能：对令牌进行校验和对用户信息的校验
--------------------------------------------------------------------------
```

```
1、买票（生成令牌）
    1）售票窗口购买（真）
        正常流程生成的令牌
    2）其他渠道购买（假）
        自己编造的令牌
        过期令牌

2、验票（校验令牌）
    1）票为真，放行
        页面传递令牌和服务器令牌对比为真
    2）票为假，拦截
        页面传递令牌为空
        页面传递令牌和服务器令牌对比为假
            假令牌
            真令牌但已过期
        服务器令牌不存在
```

- 令牌注解

```java
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Token {
    TokenType value();
}
```

注解值：通过枚举类固定值的范围

```java
public enum TokenType {
    CREATE, VALIDATE
}
```

- 登录相关操作

```java
@Controller
@RequestMapping("/login")
public class LoginController {

    // 去登录页：去动物园，要买票
    @RequestMapping(value = "/toLogin")
    @Token(TokenType.CREATE)
    public String toLogin(){
        return "/login.jsp";
    }

    // 登录：过闸机验票
    @RequestMapping("/login")
    @Token(TokenType.VALIDATE)
    public String login(String token){
        return "/success.jsp";
    }
}
```

```java
package com.soft.interceptor;
```

```java

import com.soft.annotation.Token;
import com.soft.annotation.TokenType;
import org.springframework.web.method.HandlerMethod;
import org.springframework.web.servlet.HandlerInterceptor;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.util.UUID;

public class TokenInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

        // 处理器的方法对象
        HandlerMethod handlerMethod = null;
        HttpSession session = request.getSession();
        if(handler != null) {

            handlerMethod = (HandlerMethod) handler;
            // 获取指定方法上的注解
            Token annotation = handlerMethod.getMethodAnnotation(Token.class);

            // 创建令牌
            if (annotation.value().equals(TokenType.CREATE)) {
                // 创建令牌UUID、随机数
                String uuid = UUID.randomUUID().toString();

                // 生成的令牌要存起来，方便验证时验证

                session.setAttribute("token", uuid);

                // 直接放行，去到指定的页面
                return true;
            } else if (annotation.value().equals(TokenType.VALIDATE)) {
                // 获取页面提交的令牌：逛动物园之前购买的票
                String token = request.getParameter("token");
                // 获取上次一请求的路径
                String referer = request.getHeader("Referer");
                // 是否验证通过标识，ture表示通过，false表示不通过。
                boolean flg = false;

                // 页面提交的令牌是空的
                if (token == null || "".equals(token)) {
                    // 重新生成令牌
                    flg = false;
                } else {
                    // 获取服务器生成的令牌
                    Object obj = session.getAttribute("token");

                    // 服务器的令牌为空

```

```
                    if (obj == null) {
                        // 重新生成令牌
                        flg = false;
                    } else {
                        String sessionToken = (String) obj;

                        // 验证令牌
                        if (token.equals(sessionToken)) {
                            // 验证成功，销毁票根
                            request.getSession().removeAttribute("token");
                            flg = true;
                            return true;
                        } else {
                            // 重新生成令牌
                            flg = false;
                        }
                    }
                }
                if(!flg){
                    session.setAttribute("msg", "登录超时请重新登录！！");
                    response.sendRedirect(referer);
                }
            }
        }
        return false;
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex)
        throws Exception {
        request.getSession().removeAttribute("msg");
    }
}
```

3.6 国际化

> 一个项目根据不同语言选择，显示不同的内容。
>
> 国际化语言标识：
>
> ```
> zh_CN：中文
>
> ja_JP：日文
>
> en_US：英文
> ```

1、编写国际化资源文件，文件名：前缀_语言标识.properties，properties文件中的所有key值保持一致

message_en_US.properties

```
user=account
pass=password
submit=submit
```

message_ja_JP.properties

```
user=\u30a2\u30ab\u30a6\u30f3\u30c8
pass=\u30d1\u30b9\u30ef\u30fc\u30c9
submit=\u30b5\u30d6\u30df\u30c3\u30c8
```

message_zh_CN.properties

```
user=\u8d26\u53f7
pass=\u5bc6\u7801
submit=\u63d0\u4ea4
```

2、管理国际化资源文件

```
<!--
    把语言资源文件管理起来
    id值必须是：messageSource
-->
<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="defaultEncoding" value="utf-8"/>
    <!--资源文件的前缀-->
    <property name="basename" value="message"/>
</bean>
```

3、设定国际化的实现方式

```
<!-- 以下方式二选一-->
<!--基于session的国际化-->
<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.SessionLocaleResolver"/>
<!--基于cookie的国际化-->
<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>
```

4、配置国际化拦截器

```
<!--拦截器配置-->
<mvc:interceptors>
    <!--配置国际化拦截器-->
    <bean
class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
        <!--
            paramName：默认值是local，用于接收页面传递的语言标识
        -->
        <property name="paramName" value="lang"/>
    </bean>
</mvc:interceptors>
```

5、静态页面，导入spring标签，使用<spring:message code="key" />读取properties文件的key

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
```

```jsp
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<html>
<head>
    <title>Title</title>
    <link type="text/css" rel="stylesheet"
href="${pageContext.request.contextPath}/css/index.css">
</head>
<body>
    <h2>Hello World!</h2>
    <a href="?lang=zh_CN">中文</a>
    <a href="?lang=ja_JP">日文</a>
    <a href="?lang=en_US">英文</a>

    <form action="${pageContext.request.contextPath}/login/login"
method="get">
        <spring:message code="user"/><input type="text" name="user">
        <spring:message code="pass"/><input type="text" name="pass">
        <input type="submit" value="<spring:message code="submit"/>">
    </form>
</body>
</html>
```

注意：不能直接请求jsp页面，需要通过后台转发到jsp页面session才能起作用。

应用场景：页面语言、提示信息获取

3.7 静态资源配置

> 静态资源：js、css、jpg、mp3、mp4等
>
> 由于在配置SpringMVC时，拦截的路径是 **/** ，覆盖了tomcat中加载静态资源的路径，所以需要额外配置静态资源加载。如果配置SpringMVC时拦截的路径是*.后缀的方式，则不用考虑静态资源问题。
>
> 静态资源要放到/WEB-INF/同一级目录下
>
> **Tomcat安装目录下Web.xml的部分配置：**

```xml
<!-- The mapping for the default servlet -->
<servlet-mapping>
<servlet-name>default</servlet-name>
<url-pattern>/</url-pattern>
</servlet-mapping>

<servlet>
<servlet-name>default</servlet-name>
<servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-
class>
<init-param>
  <param-name>debug</param-name>
  <param-value>0</param-value>
</init-param>
<init-param>
  <param-name>listings</param-name>
  <param-value>false</param-value>
</init-param>
```

```
<init-param>
  <param-name>readonly</param-name>
  <param-value>false</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
```

方案一（推荐）：在SpringMVC配置文件中配置静态资源

```
<!--加载静态资源-->
<mvc:default-servlet-handler/>
```

方案二：在SpringMVC配置文件中配置静态资源路径和映射

```
<!--
    mapping：页面请求资源的路径，可以使用通配符 /tupian/a.jpg
    location：项目中资源文件的实际目录 /img/a.jpg
-->
<mvc:resources mapping="/img/*" location="/img/"/>
<mvc:resources mapping="/css/*" location="/css/"/>
```

方案三：在web.xml中配置静态资源

```
<servlet-mapping>
    <!-- servlet-name必须是default，指向tomcat中全局配置文件配置的servlet类 -->
    <servlet-name>default</servlet-name>
    <!-- *.后缀的形式添加资源文件类型 -->
    <url-pattern>*.png</url-pattern>
    <url-pattern>*.js</url-pattern>
    <url-pattern>*.css</url-pattern>
    <url-pattern>*.gif</url-pattern>
    <url-pattern>*.jpg</url-pattern>
</servlet-mapping>
```

3.8 RestFul 风格

> Restful（Representational State Transfer：**表现层状态转化**）一种软件架构风格、设计风格，而不是标准，只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。
>
> 基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。
>
> Restful 是一种互联网应用程序的 API 设计理念：URL 定位资源，用 HTTP 动词（GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS, TRACE）描述操作。
>
> **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性的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。

> **状态转换(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"状态转换"(State Transfer)。而这种转换是建立在表现层之上的，所以就是"表现层状态转换"。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。他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。

```
传统方式操作资源：通过URL指定动作
    查询(get)：http://localhost:8080/users/queryUser?id=xxx
    添加(post)：http://localhost:8080/users/addUser
    删除(get)：http://localhost:8080/users/delUser?id=xxx
    修改(get/post)：http://localhost:8080/users/editUser

Restful风格操作资源：通过HTTP的状态已经可以表明操作，所以没有必要再通过URL指定动作
    查询列表(get)：http://localhost:8080/users/
    查询某个用户(get)：http://localhost:8080/users/1001
    添加用户信息(post)： http://localhost:8080/users/
    修改用户全部信息(put)： http://localhost:8080/users/
    修改用户部分信息(patch)： http://localhost:8080/users/
    删除用户信息(delete)： http://localhost:8080/users/1001
```

> 传统的操作从功能角度看是没有问题的，但也存在一些问题。每次请求的接口或者地址，都在做描述，例如查询的时候用了queryUser，新增的时候用了addUser，修改的时候用了editUser，其实完全没有这个必要。使用了GET请求，就是查询，使用POST请求，就是新增的请求，PUT就是修改，DELETE 就是删除，我的意图很明显，完全没有必要做描述，这就是为什么有了RestFul风格。
>
> 通过 GET、 POST、 PUT、 PATCH、 DELETE 等请求方式对服务端的资源进行操作。其中，GET 用于查询资源，POST 用于创建资源，PUT 用于更新服务端的资源的全部信息，PATCH 用于更新服务端的资源的部分信息，DELETE 用于删除服务端的资源。

3.9 Excel 工具类

> 管理类系统中，例如：电商后台，有大批量的信息需要录入、修改；单个录入不仅麻烦、耗时耗力，信息的正确率也不能够保证。所以对于数据的批量操作显得尤为重要。其中**Exel**作为原始数据编辑工具，如果能对其操作，将简化大部分工作量。

导包：

```xml
<!-- excel xls -->
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi</artifactId>
    <version>4.1.1</version>
</dependency>
<!-- excel xlsx -->
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi-ooxml</artifactId>
    <version>4.1.1</version>
</dependency>
```

Excel 导入思路：

```
导入数据的核心思路，将本地的数据，保存到数据库中；

    本地数据：Excel（文件：xls、xlsx）
    文件要解析：通过 Java 代码拿到文件Excel 文件解析 （POI技术）
    Java 代码如何拿到 Excel：文件上传（满足文件上传的要求）
    数据库要接收数据的类型：集合/数组
    解析目的：Excel（Excel组成：sheet、行、单元格） 转换为 集合，借助工具类

    数据如何进入数据库：批量插入
        插入n条，要用到批量插入的语句
            插入的数据应该是List集合（解析excel形成的）
        数据库的方法的入参肯定是一个集合，集合中一般都是实体类
```

```
工具类要解析 Excel 文件：解析的是流

工具类如何写？
    参数：流、Excel 文件类型、List<T> 中要存放的泛型
    返回值：List<T>
```

Excel导出思路：

1、文件下载（需要满足的条件） 2、Excel的文件类型（xls、xlsx） 3、创建Excel（Excel组成：sheet、行、单元格）

```
导入的核心思路，将数据库的数据保存到本地的文件；
    数据库的数据：集合
    构建文件：Excel（文件：xls、xlsx），借助工具类
    下载文件：将构建好的文件通过流下载本地

工具类如何写？
    参数：List集合、文件类型、sheet名字、实体类的类型（构建表头）
    返回值：流
```

代码实现：

```java
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.ss.usermodel.*;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.*;

/**
 * Excel导入
 *
 * Maven依赖
```

```
 *      <dependency>
 *          <groupId>org.apache.poi</groupId>
 *          <artifactId>poi</artifactId>
 *          <version>4.1.1</version>
 *      </dependency>
 *      <dependency>
 *          <groupId>org.apache.poi</groupId>
 *          <artifactId>poi-ooxml</artifactId>
 *          <version>4.1.1</version>
 *      </dependency>
 **/
public class ExcelUtil {
    // Excel后缀
    public static final String SUFFIX_XLS = ".xls";
    public static final String SUFFIX_XLSX = ".xlsx";

    // 默认文件名
    private static String fileName_default = "excel_export";
    // 默认sheet名
    private static String sheetName_default = "sheet1";

    // 标题数组
    private static String[] title;
    // 数据总行数
    private static Integer rowCount;

    /**
     * 读取Excel文件的数据，可以把数据封装为List<T>类型
     *
     * 要求:
     *      1、excel第一行必须是标题
     *      2、excel所有单元格的格式必须是字符
     *
     * @param suffix excel文件后缀名 xls、xlsx
     * @param inputStream 解析文件就是解析输入流
     * @param clazz 类对象 集合中的泛型
     * @return List<T>
     * @throws IOException
     */
    public static <T> List<T> importExcel(String suffix, InputStream
inputStream, Class<T> clazz) throws IOException,
            IllegalAccessException, InstantiationException,
InvocationTargetException {

        // 判断后缀
        if(suffix == null || "".equalsIgnoreCase(suffix)){
            throw new RuntimeException("不能识别文件类型！");
        }

        // 判断输入流
        if(inputStream == null){
            throw new RuntimeException("文件不能为空！");
        }

        // 判断映射实体类类型
```

```
        if(clazz == null){
            throw new RuntimeException("对象映射错误！");
        }

        // Excel对象
        Workbook book = null;
        // 封装数据集合
        List<T> list = null;
        // 类对象生成的实例
        Object obj = null;

        // 2017以前版本：xls
        if(SUFFIX_XLS.equalsIgnoreCase(suffix)){
            book = new HSSFWorkbook(inputStream);
        // 2017以后版本：xlsx
        } else if(SUFFIX_XLSX.equalsIgnoreCase(suffix)){
            book = new XSSFWorkbook(inputStream);
        // 文件类型不能匹配
        } else {
            throw new RuntimeException("不能识别文件类型！");
        }

        // 获取工作簿，默认获取第一个工作簿对象
        Sheet sheet = book.getSheetAt(0);

        // 获取工作簿共有多少行数据
        rowCount = sheet.getLastRowNum();

        // 遍历数据行，开始封装数据
        for(int i = 0; i < rowCount; i++){
            /*
                由于要求第一行是标题，所以除了第一行不创建实体类对象以后每一行都创建
                这里利用了对象类型是值引用的特性，可以先把对象添加到List中
            */
            if(i > 0){
                obj = clazz.newInstance();
                if(list == null){
                    list = new ArrayList<T>();
                }
                list.add((T) obj);
            }

            // 获取行对象
            Row row = sheet.getRow(i);

            // 获取总有多少列数据
            short lastCellNum = row.getLastCellNum();

            // 遍历列
            for(int j = 0; j < lastCellNum; j++){
                // 单元格对象
                Cell cell = row.getCell(j);
                // 获取单元格数据
                String value = cell.getStringCellValue();

```

```
                // 第一行：标题
                if(i == 0){
                    // 存放标题
                    if(title == null){
                        title = new String[lastCellNum];
                    }
                    title[j] = value.toLowerCase();
                } else {

                    // 映射实体类set方法的集合
                    Map<String, Method> map = loadClass(clazz);

                    String methodName = "set" + title[j];
                    Method method = map.get(methodName);

                    if(method != null){
                        method.invoke(obj, value);
                    }
                }
            }
        }

        close(book, inputStream, null);
        return list;
    }


    /**
     * Excel导出，导出xlsx版本，支持2017以上版本
     * 使用默认文件名：excel_export.xlsx
     *
     * @param list 结果集合，封装实体类
     * @param clazz 实体类对象
     * @param resp  response
     * @throws IOException InvocationTargetException IllegalAccessException
     */
    public static <T> void exportExcel(List<T> list, Class<T> clazz,
HttpServletResponse resp) throws IOException, InvocationTargetException,
IllegalAccessException {
        exportExcel(null, list, clazz, resp);
    }

    /**
     * Excel导出，导出xlsx版本，支持2017以上版本
     * 自定义文件名
     *
     * @param fileName 文件名
     * @param list 结果集合，封装实体类
     * @param clazz 实体类对象
     * @param resp  response
     * @throws IOException InvocationTargetException IllegalAccessException
     */
    public static <T> void exportExcel(String fileName, List<T> list,
Class<T> clazz, HttpServletResponse resp) throws IOException,
InvocationTargetException, IllegalAccessException {
```

```
178
179         // 判断文件名
180         if(fileName == null || "".equals(fileName)){
181             fileName = fileName_default;
182         }
183 
184         // 创建Excel对象
185         Workbook book = new XSSFWorkbook();
186 
187         // 创建工作簿
188         Sheet sheet = book.createSheet(sheetName_default);
189 
190         // 获取对象中所有的公共属性
191         Field[] fields = clazz.getDeclaredFields();
192 
193         // 行数
194         int r = 0;
195 
196         // 遍历数据结果集
197         for(int i = 0; i < list.size();){
198             // 创建行对象
199             Row row = sheet.createRow(r);
200 
201             // 遍历对象中的属性
202             for (int j = 0; j <= fields.length; j++) {
203                 // 获取属性名
204                 String fieldName = fields[j].getName().toLowerCase();
205                 String value = "";
206 
207                 // 标题
208                 if(r == 0) {
209                     // 标题数组
210                     if (title == null) {
211                         title = new String[fields.length];
212                     }
213                     title[j] = fieldName;
214 
215                     // 标题
216                     value = fieldName;
217                 } else {
218                     // list集合中的单个对象
219                     T t = list.get(i);
220 
221                     // 获取类对象中所有公共的方法
222                     Map<String, Method> map = loadClass(clazz);
223                     Method method = map.get("get" + fieldName);
224 
225                     if(method != null){
226                         value = (String)method.invoke(t);
227                     }
228                 }
229 
230                 // 单元格对象
231                 Cell cell = row.createCell(j);
232                 // 单元格赋值
```

```
                cell.setCellValue(value);
                // 单元格样式
                CellStyle cellStyle = book.createCellStyle();
                // 四面边框实线
                cellStyle.setBorderTop(BorderStyle.THIN);
                cellStyle.setBorderRight(BorderStyle.THIN);
                cellStyle.setBorderBottom(BorderStyle.THIN);
                cellStyle.setBorderLeft(BorderStyle.THIN);
                // 设置单元格样式
                cell.setCellStyle(cellStyle);
            }
            // 判断行数，如果行数大于1，读取的数据下标才开始累加
            if(r > 0){
                i++;
            }
            // 行累加
            r++;
        }

        // 设置响应头信息
        resp.setHeader("content-disposition","attachement;filename=" + fileName + SUFFIX_XLSX);
        // 把Excel响应到页面
        ServletOutputStream outputStream = resp.getOutputStream();
        book.write(resp.getOutputStream());
        // 释放资源
        close(book, null, outputStream);
    }

    /**
     * 释放资源
     * @param book
     * @param inputStream
     */
    private static void close(Workbook book, InputStream inputStream, OutputStream outputStream) {
        try {
            if(book != null){
                book.close();
            }

            if(inputStream != null){
                inputStream.close();
            }

            if(outputStream != null){
                outputStream.close();
            }
        } catch (IOException e) {
            throw new RuntimeException("资源释放失败! ");
        }
    }

    /**
     * 解析类对象，返回set方法的Map映射集合
```

```
286         * @param clazz
287         * @return
288         */
289        private static <T> Map<String, Method> loadClass(Class<T> clazz){
290            // 判断类对象是否为空
291            if(clazz == null){
292                throw new RuntimeException("对象映射错误! ");
293            }
294
295            // 存储方法的Map集合
296            Map<String, Method> map = new HashMap<String, Method>();
297
298            // 获取类中所有的方法对象
299            Method[] methods = clazz.getMethods();
300
301            // 遍历
302            for (Method method : methods) {
303                String name = method.getName().toLowerCase();
304                map.put(name, method);
305            }
306
307            return map;
308        }
309
310        /**
311         * 获取Excel标题数组
312         * @return
313         */
314        public static String[] getTitle() {
315            return title;
316        }
317
318        /**
319         * 获取Excel总数据行数
320         * @return
321         */
322        public static Integer getRowCount() {
323            return rowCount;
324        }
325    }
```

4. 综合实验

- 完善学生管理系统

5. 作业实践

- 通过自定义类型转换器解决日期上传问题
- 通过文件上传导入学生信息
- 通过文件下载导出学生信息
- 通过异常处理器处理系统异常
- 通过拦截器配置权限和编码格式

三、SSM 整合

> Struts2 + Spring + MyBatis **SpringMVC + Spring + MyBatis**
>
> SSH：Struts2 + Spring + Hibernate SSI：Struts2 + Spring + iBatis

1、每个技术需要用到的配置：

```
SpringMVC：前后台交互的桥梁
    web.xml web项目入口
    spring-mvc.xml 配置控制层注解扫描、视图解析器、拦截器、异常处理器、类型转换器等组件

Spring：容器，用于框架之间的整合
    spring-config.xml 配置业务层、持久层注解扫描、增强通知等
    spring-dao.xml 配置数据库、事务、MyBatis整合

MyBatis：持久层框架，和数据库交互
    数据库配置
    映射文件
    实体类别名
    ......
```

2、配置思路

- 从前向后配置

```
1、配置和页面相关的内容：SpringMVC
    控制层扫描（标配）
    静态资源处理（标配）
    拦截器（选配）
    异常处理器（选配）
    类型转换器（选配）
    文件上传解析器（选配）

2、业务层：Service
    Spring 管理

3、持久层：MyBatis
    数据源配置
    映射文件
```

- 从后向前配置

```
3、配置和页面相关的内容：SpringMVC
    控制层扫描（标配）
    静态资源处理（标、选配）
    拦截器（选配）
    异常处理器（选配）
    类型转换器（选配）
    文件上传解析器（选配）

2、业务层：Service
    Spring 管理

1、持久层：MyBatis
    数据源配置
    映射文件
```

3、由于是web项目，希望在tomcat启动时加载所有的配置，所以需要在web.xml中配置。通过web容器加载SpringMVC容器和Spring容器

maven依赖

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
    <!--spring-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.2.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-tx</artifactId>
        <version>5.2.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.2.1.RELEASE</version>
    </dependency>
    <!--文件上传依赖-->
    <dependency>
        <groupId>commons-fileupload</groupId>
        <artifactId>commons-fileupload</artifactId>
        <version>1.4</version>
    </dependency>
    <!--mybatis依赖-->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.3</version>
    </dependency>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>2.0.1</version>
    </dependency>
    <!--Oracle数据库依赖-->
    <dependency>
        <groupId>ojdbc</groupId>
        <artifactId>ojdbc</artifactId>
        <version>6.0</version>
```

```
    </dependency>
    <!-- 阿里数据库连接池 -->
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.1.21</version>
    </dependency>
</dependencies>
```

spring-mvc.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/mvc
        https://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!--控制器扫描：主要扫描@Controller-->
    <context:component-scan base-package="com.soft">
        <!--白名单-->
        <context:include-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
    </context:component-scan>

    <mvc:annotation-driven/>

    <!--配置视图解析器-->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <!--前缀-->
        <property name="prefix" value=""/>
        <!--后缀-->
        <property name="suffix" value=".jsp"/>
    </bean>

    <!--
        文件上传的解析器
        id值必须是：multipartResolver
    -->
    <bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
        <!--上传文件的大小限制，以字节为单位，如果是-1，则表示不限制大小-->
        <property name="maxUploadSize" value="-1"/>
        <property name="defaultEncoding" value="UTF-8"/>
    </bean>
</beans>
```

spring-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/mvc
        https://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!--业务层和持久层的扫描-->
    <context:component-scan base-package="com.soft">
        <!--黑名单-->
        <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
    </context:component-scan>
</beans>
```

spring-dao.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/mvc
        https://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!--数据源-->
    <context:property-placeholder location="classpath:jdbc.properties"/>
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${oracle.driver}"/>
        <property name="url" value="${oracle.url}"/>
        <property name="username" value="${oracle.username}"/>
        <property name="password" value="${oracle.password}"/>
    </bean>
    <!--MyBatis-->
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <!--引入数据库-->
        <property name="dataSource" ref="dataSource"/>
```

```
        <property name="configLocation" value="classpath:mybatis-config.xml"/>
        <!--映射文件-->
        <property name="mapperLocations" value="classpath*:com/soft/mapper/*.xml"/>
        <!--别名-->
        <property name="typeAliasesPackage" value="com.soft.entity"/>
        <!--插件-->
        <property name="plugins">
            <bean class="com.github.pagehelper.PageInterceptor"></bean>
        </property>
    </bean>
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"/>
        <property name="basePackage" value="com.soft.mapper"/>
    </bean>
    <!--事务-->
    <bean class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>
    <tx:annotation-driven/>
</beans>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">

    <!--加载springMVC的配置-->
    <servlet>
        <servlet-name>SpringMVC</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath*:spring-mvc.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>SpringMVC</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <!--
        spring的配置
        /WEB-INF/applicationContext.xml
    -->
    <listener>
```

```
        <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <!--加载自定义路径下的配置文件-->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath*:spring-*.xml</param-value>
    </context-param>

    <!--字符编码-->
    <filter>
        <filter-name>encoding</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>encoding</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```