 ---------------------------------------------------

计划课程安排：

FUNDAMENTAL(7天)      该阶段已结束

OOP(11天)                        该阶段已结束

JAVASE01(10天)              该阶段已结束

JAVASE02(12天)               该阶段已结束

WEBBASIC(11天)        该阶段已结束

JQUERY(2天)                    该阶段已结束

DATABASE(5天)            该阶段已结束

JDBC(2天)                         该阶段已结束

SERVLETJSP(14天)        该阶段已结束

SPRINGMYBATIS01(8天) ←当前为该阶段第4天

AJAX(2天)

SPRINGMYBATIS02(12天) （该阶段已被取消）

LINUX(2天)                        （该阶段已被取消）

NGINX(2天)                      （该阶段已被取消）

REDIS(2天)                       （该阶段已被取消）

T-STORE(5天)                             （该阶段已被取消）

预计合计：101天

预计剩余：11天 （9天正课+2天自习）

注1：上述信息随时可能发生更改，仅供参考。

注2：被取消表示若按照正常流程就业则直接上班，不用再继续到来机构上课。

反之，则需要继续到机构上完全部课程。

---------------------------------------------------

****2019年6月21日****

****Java就业班-第92天****

**7:50实训室2-早自习**

昨日课程笔记：

### 1. SpringMVC框架

#### 1.1. 作用

SpringMVC框架解决了V与C的交互问题。

原生的`Servlet`就是控制器，使用Servlet主要存在的问题是实例太多，配置麻烦，管理难度大等一系列的问题。例如项目中有用户注册功能，则可能需要开发`UserRegisterServlet`，如果还有登录功能，则可能需要开发`UserLoginServlet`，几乎是每个功能需要有1个对应的`Servlet`，如果一个项目中有200个不同的功能，则需要200个`Servlet`，在更大的系统中，`Servlet`的数量就非常多，在实际运行时，在内存中的`Servlet`对象就会占据大量的内存空间！由于`Servlet`的数量很多，进而导致配置文件的配置信息会非常多，配置信息多了以后，就会引发管理难度大的问题。

#### 1.2. SpringMVC中的核心组件

- `DispatcherServlet`：前端控制器，用于接收所有请求，并负责分发；

- `HandlerMapping`：根据请求路径映射控制器或控制器的方法，确定请求路径与控制器或控制器中的方法的对应关系；

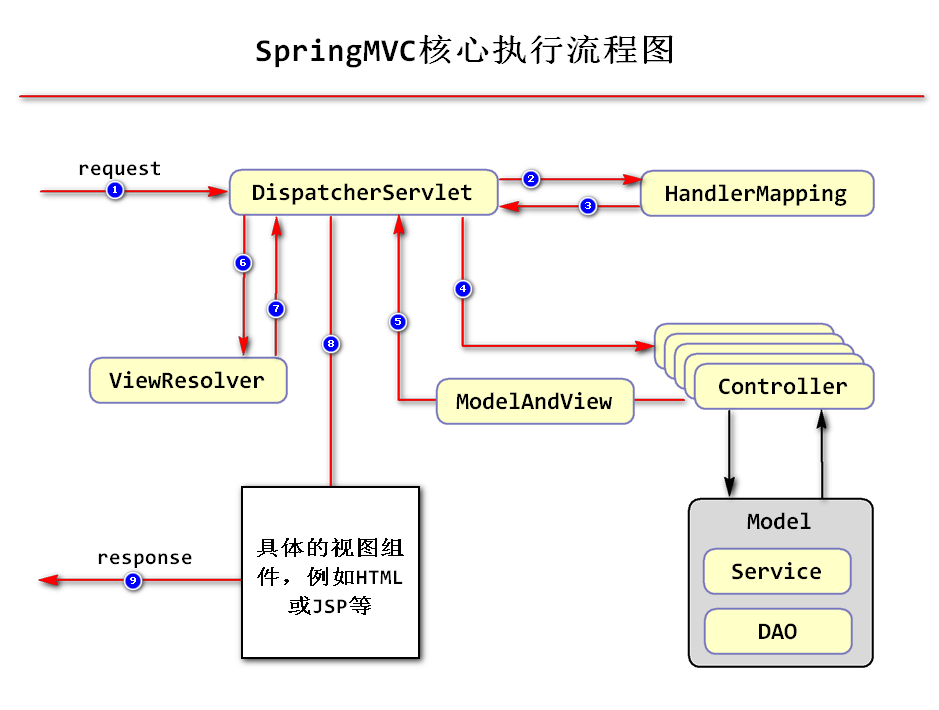
- `Controller`：实际处理请求的组件；

- `ModelAndView`：控制器的返回结果，包括处理完成后的数据，及最终应该响应给客户端的视图名称；

- `ViewResolver`：根据视图名称得到具体的视图组件。

具体的执行流程图：

![](01.png)



### 2. SpringMVC HelloWorld

#### 2.1. 目标

在浏览器中通过`http://localhost:8080/项目名称/hello.do`可以访问某个JSP显示的页面，页面中显示\*\*Hello, SpringMVC!!!\*\*字样。

#### 2.2. 创建项目

创建\*\*Maven Project\*\*，\*\*Artifact Id\*\*为`xx.xxxx.spring`，\*\*Artifact Id\*\*为`SPRINGMVC01`，\*\*Packaing\*\*必须选择`war`。

创建完成后，首先生成\*\*web.xml\*\*文件。

然后，在\*\*pom.xml\*\*中添加`spring-webmvc`的依赖。

然后，将Spring的配置文件复制到\*\*src/main/resources\*\*下。

另外，还需要添加Tomcat运行环境。

#### 2.3. 配置DispatcherServlet

打开\*\*web.xml\*\*，在配置文件中对`DispatcherServlet`进行配置，使之可以处理所有以`.do`结尾的请求：

<servlet>

<servlet-name>SpringMVC</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>SpringMVC</servlet-name>

<url-pattern>\*.do</url-pattern>

</servlet-mapping>

SpringMVC框架是基于Spring框架的，后续开发的组件也应该随着项目启动就完成初始化，例如自定义的控制器类等等，也就意味着当项目启动时，就应该加载Spring的配置文件(spring.xml)。

在`DispatcherServlet`的父类`FrameworkServlet`中定义了名为`contextConfigLocation`的属性，当`DispatcherServlet`初始化时，会自动读取该属性的值，而该属性的值应该是配置文件的路径，则`DispatcherServlet`就会自动加载配置文件！

所以，还应该对`DispatcherSerlvet`的`contextConfigLocation`属性进行配置，值是Spring的配置文件，并保证`DispatcherServlet`是默认启动的：

<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.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>SpringMVC</servlet-name>

<url-pattern>\*.do</url-pattern>

</servlet-mapping>

完成以上配置后，如果需要检测配置是否成功，可以在Spring的配置文件中添加组件扫描：

<context:component-scan base-package="xx.xxxx.spring" />

然后，创建`xx.xxxx.spring.User`类，为该类添加注解，并自定义构造方法：

@Component

public class User {

public User() {

System.out.println("初始化User对象。");

}

}

当把项目部署到Tomcat时，Tomcat启动就会初始化`DispatcherServlet`，由于配置了初始化参数`contextConfigLocation`，就会加载Spring的配置文件，在配置文件中执行组件扫描，扫描到`User`时，由于`User`类添加了注解，Spring就会创建`User`类的对象，也就是执行了无参数的构造方法，则会执行以上代码中的输出语句！总的来说，启动Tomcat就可以看到以上输出语句！

#### 2.4. 通过控制器接收请求

自定义控制器类`xx.xxxx.spring.HelloController`，在类之前添加`@Controller`注解：

@Controller

public class HelloController {

}

然后，在类中添加方法以处理请求，关于方法的设计：

1. 应该使用`public`权限；

2. 暂时使用`String`作用方法的返回值类型；

3. 方法的名称可以自由定义；

4. 方法可以没有参数。

则可以添加：

public String showHello() {

}

然后，在方法之前使用`@RequestMapping("请求路径")`来配置请求路径：

@RequestMapping("hello.do")

public String showHello() {

}

则实现了请求路径与处理请求的方法的映射关系，即：当请求`hello.do`时，就会自动调用`showHello()`方法！

可以在以上方法中输出日志，以观察运行效率，暂时不关心返回值，所以，测试运行时页面将提示404错误，暂时是正常的：

@RequestMapping("hello.do")

public String showHello() {

System.out.println("HelloController.showHello()");

return null;

}

最后，重新启动项目，通过`http://localhost:8080/SPRINGMVC01/hello.do`进行访问，在浏览器中会提示404，并在Eclipse的控制台中可以看到以上输出语句的内容！

#### 2.5. 显示页面

首先，在\*\*webapp/WEB-INF/hello.jsp\*\*页面，用于显示。

在控制器处理请求的方法中，返回的`String`值表示“视图名称”，将根据视图解析器`ViewResolver`进行解析，得到以上创建的JSP文件。

实际使用的`InternalResourceViewResolver`的工作原理就是以\*\*webapp\*\*作为根路径，“前缀 + 控制器返回的字符串 + 后缀”得到文件位，所以，可以配置为：

<!-- 视图解析器 -->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<!-- 前缀 -->

<property name="prefix" value="/WEB-INF/"></property>

<!-- 后缀 -->

<property name="suffix" value=".jsp"></property>

</bean>

然后，控制器的方法必须返回`"hello"`，这样，前缀与返回值与后缀拼接起来，就可以得到`/WEB-INF/hello.jsp`。

最后，在浏览器再次访问，可以看到JSP页面设计的内容。

#### 2.6. 作业

新创建`SPRINGMVC02`项目，要求通过`http://localhost:8080/SPRINGMVC02/reg.do`可以显示用户注册页面，页面中至少包括用户名、密码、年龄、手机号码、邮箱这5个输入框，及1个提交按钮；要求通过`http://localhost:8080/SPRINGMVC02/login.do`可以显示用户登录页面，页面中至少包括用户名、密码这2个输入框，及1个提交按钮；

提示：只需要创建1个控制器类，该类中添加2个处理请求的方法。

**其他知识补充：**

### 8. 自动装配的@Autowired与@Resource的区别

`@Autowired`是Spring框架的注解，而`@Resource`是JavaEE中的注解。

`@Resource`注解是优先byName实现自动装配的，要求被装配的属性名与某个bean的id保持一致；也可以配置该属性的属性，以明确的指定需要装配的bean id，例如`@Resource(name="subUserDao")`，如果byName装配失败，还会尝试byType自动装配，此时通过byType自动装配时，要求匹配类型的bean必须有且仅有1个，如果没有，或超过1个，都会直接报错！

`@Autowired`注解是优先byType实现自动装配的，此时，如果匹配类型的对象有多个，则会尝试byName，如果能够根据byName装配成功，则装配结束，如果byName装配失败，会检查匹配类型的对象的数量，如果数量为1，则直接装配，否则，将报告错误！

**8:30实训室2-早自习**

今日晨讲主题——HashMap的使用方法复习

**17:50实训室2-下课时间**

今日课程笔记：

### 1. 接收请求参数

#### 1.1. (不推荐) 使用HttpServletRequest

在处理请求的方法的参数列表中添加`HttpServletRequest`参数，然后，在处理过程中，调用`request`的`getParameter()`方法即可获取各请求参数的值：

@RequestMapping("handle\_register.do")

public String handleRegister(HttpServletRequest request) {

System.out.println("UserController.handleRegister()");

String username = request.getParameter("username");

String password = request.getParameter("password");

Integer age = Integer.valueOf(request.getParameter("age"));

String phone = request.getParameter("phone");

String email = request.getParameter("email");

System.out.println("\tusername=" + username);

System.out.println("\tpassword=" + password);

System.out.println("\tage=" + age);

System.out.println("\tphone=" + phone);

System.out.println("\temail=" + email);

return null;

}

以上做法的不足有：

1. 如果期望的数据类型不是`String`，需要自行转换！

2. 获取数据的操作比较繁琐；

3. 不便于执行单元测试。

#### 1.2. (推荐) 将请求参数设计为处理请求的方法的参数

当需要获取请求参数时，直接将它们添加到处理请求的方法的参数列表中即可：

@RequestMapping("handle\_register.do")

public String handleRegister(

String username, String password,

Integer age, String phone, String email) {

System.out.println("UserController.handleRegister()");

System.out.println("\tusername=" + username);

System.out.println("\tpassword=" + password);

System.out.println("\tage=" + (age + 1));

System.out.println("\tphone=" + phone);

System.out.println("\temail=" + email);

return null;

}

这种操作要求请求参数的名称与处理请求的方法的参数名称保持一致！如果名称不一致，则处理请求的方法中的参数值将是`null`。

在设计处理请求的方法的参数列表时，各参数的类型可以直接声明为期望的数据类型，例如期望名为`age`的量是`Integer`类型，则直接声明为`Integer`即可，无需在方法中自行转换！

这种做法的缺陷在于：不适合处理过多的请求参数。

#### 1.3. (推荐) 使用封装的类型作为处理请求的方法的参数

当请求参数较多时，可以将这些参数封装到1个自定义的类型中：

public class User {

private String username;

private String password;

private Integer age;

private String phone;

private String email;

}

然后，将自定义的数据类型作为处理请求的方法的参数即可：

@RequestMapping("handle\_register.do")

public String handleRegister(User user) {

System.out.println("UserController.handleRegister()");

System.out.println("\t" + user);

return null;

}

同样，这种做法仍要求请求参数的名称，与封装的类型中的属性名称保持一致！

这种做法还有1个优点：当请求参数的数量发生变化时，也许处理请求的方法的参数列表可以不用调整，而只调整自定义的封装类型即可。

#### 1.4. 如何选取各种获取请求参数的方式

第1种做法是始终不推荐的，即完全不使用第1种做法。

如果参数的数量较多，应该优先选取第3种做法；如果参数的数量可能发生变化，应该优先选取第3种做法；如果参数的数量较少且固定，应该优先选取第2种做法。

另外，第2种做法和第3种做法可以混合在一起使用！

### 2. 重定向

在处理请求的方法中，返回`String`类型的结果时，返回值使用`redirect:`作为前缀，则表示重定向！在`redirect:`右侧的必须是重定向到的目标的相对定位或绝对定义的URL，例如：

return "redirect:xxx.do";

假设当前用户的注册一定能成功，且注册成功后显示登录页面，则：

@RequestMapping("handle\_register.do")

public String handleRegister(User user) {

System.out.println("UserController.handleRegister()");

System.out.println("\t" + user);

return "redirect:login.do";

}

### 3. 转发数据

#### 3.1. (不推荐) 通过HttpServletRequest参数封装转发的数据

在SpringMVC中，如果方法的返回值类型是`String`类型，默认转发！

假设在控制器的方法中处理登录时，只有\*\*root/1234\*\*是正确的用户名和密码，登录成功后重定向到主页，而登录失败，将错误信息转发到\*\*/WEB-INF/error.jsp\*\*错误页面。

当需要转发时，可以在处理请求的方法中添加`HttpServletRequest`参数，在使用时，调用参数对象的`setAttribute()`方法即可封装转发的数据，然后，执行转发即可(无需获取转发器等)：

@RequestMapping("handle\_login.do")

public String handleLogin(String username, String password, HttpServletRequest request) {

System.out.println("UserController.handleLogin();");

System.out.println("\tusername=" + username);

System.out.println("\tpassword=" + password);

// 定义错误提示信息

String errorMessage;

// 判断用户名是否正确

if ("root".equals(username)) {

// 用户名正确，继续判断密码是否正确

if ("1234".equals(password)) {

// 密码也正确，则登录成功，重定向到主页

return "redirect:index.do";

} else {

// 密码错误，先准备好错误提示

errorMessage = "登录失败！密码错误！";

// 将错误提示封装到请求对象中

request.setAttribute("msg", errorMessage);

// 执行转发

return "error";

}

} else {

// 用户名错误

errorMessage = "登录失败！用户名不存在！";

request.setAttribute("msg", errorMessage);

return "error";

}

}

#### 3.2. (更不推荐) 使用ModelAndView

可以使用`ModelAndView`作为处理请求的方法的返回值类型，在该类型的对象中，设置其`viewName`属性即可确定需要转发到的视图名称，另外还使用了`Map<String, ?>`类型的数据作为需要转发的数据：

@RequestMapping("handle\_login.do")

public ModelAndView handleLogin(String username, String password) {

System.out.println("UserController.handleLogin();");

System.out.println("\tusername=" + username);

System.out.println("\tpassword=" + password);

// 准备返回值对象

ModelAndView mav;

// 封装转发的数据的Map

Map<String, Object> model = new HashMap<String, Object>();

// 定义错误提示信息

String errorMessage;

// 判断用户名是否正确

if ("root".equals(username)) {

// 用户名正确，继续判断密码是否正确

if ("1234".equals(password)) {

// 密码也正确，则登录成功，重定向到主页

return null;

} else {

// 密码错误，先准备好错误提示

errorMessage = "登录失败！密码错误！";

// 将错误提示封装到请求对象中

model.put("msg", errorMessage);

mav = new ModelAndView("error", model);

// 执行转发

return mav;

}

} else {

// 用户名错误

errorMessage = "登录失败！用户名不存在！";

model.put("msg", errorMessage);

mav = new ModelAndView("error", model);

return mav;

}

}

#### 3.3. (推荐) 使用ModelMap封装转发的数据

使用方式与使用`HttpServletRequest`是相同的：

@RequestMapping("handle\_login.do")

public String handleLogin(String username, String password, ModelMap modelMap) {

System.out.println("UserController.handleLogin();");

System.out.println("\tusername=" + username);

System.out.println("\tpassword=" + password);

// 定义错误提示信息

String errorMessage;

// 判断用户名是否正确

if ("root".equals(username)) {

// 用户名正确，继续判断密码是否正确

if ("1234".equals(password)) {

// 密码也正确，则登录成功，重定向到主页

return null;

} else {

// 密码错误，先准备好错误提示

errorMessage = "[ModelMap] 登录失败！密码错误！";

// 将错误提示封装到请求对象中

modelMap.addAttribute("msg", errorMessage);

// 执行转发

return "error";

}

} else {

// 用户名错误

errorMessage = "[ModelMap] 登录失败！用户名不存在！";

modelMap.addAttribute("msg", errorMessage);

return "error";

}

}

### 4. 关于@RequestMapping注解

在处理请求的方法之前添加`@RequestMapping`用于配置请求路径与处理响应的方法之间的映射。

该注解也可以添加在类的声明之前，例如：

@RequestMapping("user")

@Controller

public class UserController {

}

添加在类之前的`@RequestMapping`注解用于配置请求路径中的层次！将作用于当前类中配置的所有请求路径！原有的例如`login.do`的请求路径就会变成`user/login.do`。

无论是添加在类之前配置路径，还是在方法之前配置路径，在配置时，路径的左右两侧都可以添加`/`符号，例如：

@RequestMapping("/user/")

在使用该注解时，对左右两侧的`/`是没有要求的，例如在类和方法之前分别配置：

user login.do

/user /login.do

/user/ /login.do

/user/ login.do

以上各种配置方式都是正确的！在实际使用时，应该保持自己的代码风格统一！

在实际应用中，推荐为每一个类都添加该注解！

配置注解中的`value`属性或`path`属性，可以配置请求路径与处理请求的方法的映射，且值可以是数组类型：

@RequestMapping(name="hahaha", path={"reg.do", "register.do"})

可以在注解中配置`method`属性，以限制某个请求路径的允许使用的若干种请求方式，例如：

@RequestMapping(path="handle\_login.do", method=RequestMethod.POST)

或：

@RequestMapping(path="handle\_login.do", method={RequestMethod.POST, RequestMethod.GET})

如果使用错误的请求方式发出请求，则会导致405错误：

HTTP Status 405 - Request method 'GET' not supported

### 5. 关于@RequestParam注解

`@RequestParam`是添加在请求参数之前的注解！

使用该注解可以解决客户端提交的请求参数名称，与服务器端的处理请求的方法的参数名称不统一的问题：

@RequestParam("uname") String username

当添加以上注解后，默认情况下，该参数就是必须提交的，如果请求中没有提交该参数，就会出现400错误：

HTTP Status 400 - Required String parameter 'uname' is not present

这种表现是由于该属性中的`required`属性决定的，在该注解的源代码：

boolean required() default true;

所以，如果并不强制要求客户端提交该参数，可以：

@RequestParam(name="uname", required=false)

另外，还可以配置`defaultValue`属性，用于配置默认值，即当客户端没有提交该请求参数时，视为客户端提交了某个值：

@RequestParam(name="uname", required=false, defaultValue="JSD1902")

当然，在设置`defaultValue`时，需要显式的将`required`设置为`false`。

所以，通常`@RequestParam`注解的使用场景：

1. 客户端提交的请求参数名称与服务器端处理请求的方法的参数名称不一致时；

2. 强制要求客户端提交某些参数时；

3. 需要为某些请求参数设置默认值时。

**21:03宿舍-休息**

昨晚还是没能得到良好的休息，今天上课几乎睡了一整天的觉，直到下午6点才恢复正常。

晚自习进行了结业考试，其实和之前的一般笔试考试没有太大的区别，惟一的区别就是昨天的模拟面试官（推荐就业负责人）负责监考，这次考试就没办法作弊了。

不过一套题目写下来我觉得也没有什么特别大的必要作弊，并且考前负责人有提到及格分为50分（100分制）。

想要及格基本上没有什么难度。

明天似乎没有什么特别的安排，今天中午似乎班上还进行了一次就业动员，但是因为我在睡觉所以也不清楚具体说了什么……应该也不是什么重要的事情吧。

后天周六是第一次招聘会（第二次在下周四），到时候应该可以看到各种企业到机构来进行招人面试，虽然不知道我这种水平能不能上，但体验一把应该是没什么问题的。

话说回来，还有2个星期不到就算毕业了，我还是什么都不会，说实话我现在非常慌，这些天的课程笔记都没有仔细研究学习，水平不说进步，甚至都倒退了不少，归根结底还是因为我已经太久没有写代码了，当初的动力到如今似乎消减了不少，人也开始越来越懒惰了。

貌似这个情况也不只是发生在我身上，班上的其他学员基本上也处于一种半放弃状态，原本期望的学习环境如今已经成为一种累赘了。

时间也是越来越少了，原本给自己定下的练习计划也是一遍又一遍的推翻直至彻底搁浅，现在唯一能做的事情似乎就剩刷面试题了。

事到如今日记也没有什么内容可记录的了，临近的结束比我预料的要提前了一个月，我只能说无论是哪个方面，我都还没有准备好。

随着我对行业的调查越来越深入，了解的情况越来越多，越发的对这个行业感到莫名的恐惧，这种感觉就好像是前进的道路上铺满了尸体，并且这些尸体还不断地劝阻着后来人，仿佛前方就是一个不断吐出尸体的无尽深渊，充满了神秘与未知的同时还不断闪烁着幻想中的画面……毕竟我还没有入行，不过也快了吧。

今日感想总结：今晚是选择逃避还是前进呢？我想放弃思考，一切靠自己主动实在是太难了。

今日评分：？/10分

今天就到这里，明天继续投稿。

感谢支持！

================================================================

****Acer程序员群：790482850****

****up主的java学习日记word文件及全部代码（随日记同步更新）：****

Address：github.com/wt62635