# Springmvc学习笔记

### 1.jar

spring-aop.jar

spring-bean.jar

spring-context.jar

spring-core.jar

spring-web.jar

spring-webmvc.jar

commons-logging.jar

报错NoClassDefFoundError：缺少jar

### 2.第一个SpringMVC程序

Servet - Springmvc

jsp ->Servlet (Springmvc)->Jsp

url

springmvc配置文件 springmvc.xml

选中常用的命名空间：beans aop context mvc

普通的servlet流程：

请求-url-pattern -交给对应的servlet去处理

如果现在想用springmvc，而不是普通的servlet，如何告知程序？-如何让springmvc 介入程序：

需要配置一个 Springmvc自带的servlet

通过以下配置，拦截所有请求，交给SpringMVC处理：

<servlet>

<servlet-name>springDispatcherServlet</servlet-name>

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

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:springmvc.xml</param-value>

</init-param>

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

</servlet>

<servlet-mapping>

<servlet-name>springDispatcherServlet</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

其中：

<url-pattern>.action</url-pattern>

/:一切请求 ，注意不是 /\*

/user:拦截以 /user开头的请求

/user/abc.do :只拦截该请求

.action:只拦截 .action结尾的请求

项目中同时兼容 springMVC和Servlet

<servlet-mapping>

<servlet-name>springDispatcherServlet</servlet-name>

<url-pattern>.action</url-pattern>

</servlet-mapping>

<servlet>

<servlet-name>springDispatcherServlet</servlet-name>

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

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:springmvc.xml</param-value>

</init-param>

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

</servlet>

通过

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:springmvc.xml</param-value>

</init-param>

指定springmvc配置文件的路径，如果要省略，必须放到 默认路径：

/WEB-INF/springDispatcherServlet-servlet.xml

映射是 去匹配@RequestMapping注解，可以和方法名、类名不一致

通过method指定 请求方式（get post delete put）

@RequestMapping(value="welcome",method=RequestMethod.POST)//映射

设置name="xxxx"的情况：

params= {"name2=zs","age!=23"}

name2:必须有name="name2"参数

age!=23 : a.如果有name="age"，则age值不能是23

b.没有age

!name2 ：不能name="name2"的属性

ant风格的请求路径

? 单字符

\* 任意个字符（0或多个）

\*\* 任意目录

@RequestMapping(value="welcome3/\*\*/test")

接受示例：

a href="welcome3/abc/xyz/abccc/test"

通过@PathVariable获取动态参数

<>

### 3.Springmvc处理模型数据

如果需要跳转时带数据:V,M,则可以使用以下方式

ModelAndView,ModelMap,Map,Model 数据放在了request作用域

@SessionAttributes @ModelAttribute

@RequestMapping(value="testModelAndView")

**public** ModelAndView testModelAndView() {

ModelAndView mv =**new** ModelAndView("success2");//view: views/success2.jsp

Student student=**new** Student();

student.setId(12);

student.setName("sbc");

mv.addObject("student", student);//相当于request.setAttribute("student", student);

**return** mv ;//succes.jsp

}

@RequestMapping(value="testModelMap")

**public** String testModelMap(ModelMap mm) {

Student student=**new** Student();

student.setId(12);

student.setName("sbc");

mm.put("student2", student);

**return** "success2" ;//forward: redirect: 指定跳转的方式,此种方式不会被视图解析器加上前缀和后缀

}

@RequestMapping(value="testMap")

**public** String testMap(Map<String, Object> mp) {

Student student=**new** Student();

student.setId(12);

student.setName("sbc");

mp.put("student3", student);

**return** "success2" ;//succes.jsp

}

@RequestMapping(value="testModel")

**public** String testModel(Model m) {

Student student=**new** Student();

student.setId(12);

student.setName("sbc");

m.addAttribute("student4",student);

**return** "success2" ;//succes.jsp

}

### 4自定义类型转换器的类(实现Converter接口)

#### 4.1编写

public class MyConverter implements Converter<String, Student> {

@Override

public Student convert(String source) {

//source接受前端传来的String:2-zs-23

String[] strs=source.split("-");

Student student=new Student();

student.setId(Integer.parseInt(strs[0]));

student.setName(strs[1]);

student.setAge(Integer.parseInt(strs[2]));

return student;

}

}

#### 4.2.配置

<!-- 1将 自定义转换器 纳入SpringIOC容器 -->

<bean id="myConverter" class="com.shibc.converter.MyConverter"></bean>

<!-- 2将myConverter再纳入 SpringMVC提供的转换器Bean -->

<bean id=*"conversionService"* class=*"org.springframework.context.support.ConversionServiceFactoryBean"*>

<property name=*"converters"*>

<set>

<ref bean=*"myConverter"*/>

</set>

</property>

</bean>

<!--此配置是SpringMVC的基础配置，很功能都需要通过该注解来协调 -->

<!-- 3将conversionService注册到annotation-driven中 -->

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

#### 4.3.测试转换器

@RequestMapping(value="testConverter")

public String testConverter(@RequestParam("studentInfo")Student student) {

System.out.println(student.getId()+","+student.getName()+","+student.getAge());

return "success";

}

其中@RequestParam("studentInfo")是触发转换器的桥梁:

@RequestParam("studentInfo")接受的数据是前端传递过来的:2-zs-23,但是需要将改数据赋值给修饰目标对象Student:因此Springmvc可以发现接受的数据和目标数据不一致

并且这两种数据分别是String 和 Student正好符合转换器(public Student convert(String source))

### 5.数据格式化

<!-- 配置 数据格式化 注解 所依赖的bean

<!-- 配置 数据格式化 注解 所依赖的bean

FormattingConversionServiceFactoryBean:既可以实现格式化、又可以实现类型转换

-->

<bean id=*"conversionService"* class=*"org.springframework.format.support.FormattingConversionServiceFactoryBean"*>

<property name=*"converters"*>

<set>

<ref bean=*"myConverter"*/>

</set>

</property>

</bean>

@DateTimeFormat(pattern = "yyyy-MM-dd")

@NumberFormat(pattern="###,#")

### 6.错误消息

public String testDateTimeFormat(Student student,BindingResult result,Map<String,Object> map) {

需要验证的数据是student中的birthday,springmvc要求如果验证失败则将错误消息自动放入该对象之后

即Student student,BindingResult result之间不能有参数

如果要将控制台的错误消息传到jsp中,则可以将错误消息对象放入request域中,然后在jsp冲request域获取

### 7.数据校验

JSR303

Hibernate Validator步骤:

7.1.jar

hibernate-validator.jar

classmate.jar

jboss-logging.jar

validatetion-api.jar

hibernate-validator-annotation-processor.jar

#### 7.2.配置

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

此时annotation-driven的作用:要实现Hibernate Validator/JSR303(或者其他各种校验),必须实现必须实现ValidatorFactory

LocalValidatorFactoryBean是必须实现ValidatorFactory的一个实现类

<mvc:annotation-driven ></mvc:annotation-driven>会在springmvc容器中,自动加载一个LocalValidatorFactoryBean,因此可以直接实现数据校验

#### 7.3.直接使用注解

public class Student {

@Past//当前时间以前

private Date birthday;

)

在校验的controller中,给校验的对象前增加@Valid

### ajax请求springmvc,并且JSON格式的数据

#### 8.1 Jar

Jackson-annotations.jar

Jackson-core.jar

Jackson-databind.jar

#### 8.2@ResponseBody

@ResponseBody修饰的方法,会将该方法的返回值 以一个json数组的形式返回给前台

后台:

@ResponseBody//告诉SpringMVC，此时的返回 不是一个 View页面，而是一个 ajax调用的返回值（Json数组）

@RequestMapping(value="testajax")

**public** List<Student2> testajax(String name) {//String name 是通过ajax以json格式传入的参数 参数Student2自动注入的Student2实体

List<Student2> students =**new** ArrayList<Student2>();

Student2 stu1=**new** Student2("shibc",25);

Student2 stu2=**new** Student2("ygf",26);

students.add(stu1);

students.add(stu2);

**return** students ;//succes.jsp

}

前台:服务端将返回值结果,以json数组形式传给result

直接post方式: $("#testAjax").click(**function**(){

$.post(

"testajax",

**function**(result){

**for**(**var** i=0;i<result.length ;i++){

alert(result[i].name +"-"+result[i].age);

}

}

)

});

传参方式:

$("#testAjax").click(**function**(){

$.ajax({

url:"testajax",

type:"post",

dateType:"json",

data:{"name":"shibc","age":888},

success:**function**(result){

**for**(**var** i=0;i<result.length ;i++){

alert(result[i].name +"-"+result[i].age);

}

}

});

});

### springMVC实现文件上传

和Servlet方式的本质一样，都是通过commons-fileupload.jar和commons-io.jar

SpringMVC可以简化文件上传的代码，但是必须满足条件：实现MultipartResolver接口 ；而该接口的实现类SpringMVC也已经提供了CommonsMultipartResolver

#### 9.1 jar

commons-fileupload.jar、commons-io.jar

#### 9.2 配置CommonsMultipartResolver

<!-- 配置CommonsMultipartResolver，用于实现文件上传

将其加入SpringIOC容器.

springIoc容器在初始化时，会自动寻找一个Id="multipartResolver"的bean，并将其加入Ioc容器

-->

<bean id=*"multipartResolver"* class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*>

<property name=*"defaultEncoding"* value=*"UTF-8"*></property>

<!-- 上传单个文件的最大值，单位Byte;如果-1，表示无限制 -->

<property name=*"maxUploadSize"*  value=*"104857600"*></property>

</bean>

#### 9.3处理方法

后端

@RequestMapping(value="testUpload")

**public** String testUpload(@RequestParam("desc")String desc,@RequestParam("file")MultipartFile file) **throws** IOException {

System.***out***.println("描述信息"+desc);

String filename=file.getOriginalFilename();//获取上传文件名和格式

InputStream input=file.getInputStream();

OutputStream out=**new** FileOutputStream("h:\\"+filename);

//放入缓冲流

BufferedInputStream reader=**new** BufferedInputStream(input);

BufferedOutputStream bos=**new** BufferedOutputStream(out);

**int** size=0;

**byte**[] buffer=**new** **byte**[10240];

**while**((size=reader.read(buffer))!=-1){

bos.write(buffer, 0, size);

}

reader.close();

bos.close();

**return** "success";

}

前端:

<form action=*"testUpload"* method=*"post"* enctype=*"multipart/form-data"*>

<input type=*"file"* name=*"file"* />

描述:<input name=*"desc"* type=*"text"* />

<input type=*"submit"* value=*"上传"*>

</form>

### 10.拦截器

拦截器的原理和过滤器相同。

SpringMVC：要想实现拦截器，必须实现一个接口HandlerInterceptor

#### 10.1编写拦截器

实现HandlerInterceptor接口

**public** **class** MyInterceptor **implements** HandlerInterceptor{

**public** **void** afterCompletion(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2, Exception arg3)

**throws** Exception {

System.***out***.println("第一个过滤器的渲染完毕后");

}

**public** **void** postHandle(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2, ModelAndView arg3)

**throws** Exception {

System.***out***.println("第一个过滤器的拦截相应");

}

**public** **boolean** preHandle(HttpServletRequest arg0, HttpServletResponse arg1, Object arg2) **throws** Exception {

System.***out***.println("第一个过滤器的拦截请求");

**return** **true**;

}

}

#### 10.2配置

<mvc:interceptors>

<!-- <bean class="com.shibc.Interceptor.MyInterceptor"></bean> 如果直接写bean表示拦截所有的请求-->

<!-- 配置具体的拦截路径 -->

<mvc:interceptor>

<!-- 指定拦截的路径,基于ant风格 -->

<mvc:mapping path=*"/\*\*"*/>

<!-- 指定拦不截的路径 -->

<mvc:exclude-mapping path=*"testUpload"*/>

<bean class=*"com.shibc.Interceptor.MyInterceptor"*></bean>

</mvc:interceptor>

<!-- 配置具体的拦截路径 -->

<mvc:interceptor>

<!-- 指定拦截的路径,基于ant风格 -->

<mvc:mapping path=*"/\*\*"*/>

<!-- 指定拦不截的路径 -->

<mvc:exclude-mapping path=*"testUpload"*/>

<bean class=*"com.shibc.Interceptor.MyInterceptor2"*></bean>

</mvc:interceptor>

</mvc:interceptors>

### 11异常处理

#### 11.1@ExceptionHandler

ExceptionHandlerExceptionResolver:主要提供了@ExceptionHandler注解，并通过该注解处理异常

方法中:

@ExceptionHandler({Exception.**class** })

**public** ModelAndView handlerArithmeticException2(Exception e) {

ModelAndView mv = **new** ModelAndView("error");

System.***out***.println(e +"============"+"该@ControllerAdvice中的异常处理方法，可以处理任何类中的异常");

mv.addObject("er", e) ;

**return** mv;

}

@ExceptionHandler标识的方法的参数 必须在异常类型(Throwable或其子类) ，不能包含其他类型的参数

异常处理路径：最短优先

如果有方法抛出一个ArithmeticException异常，而该类中 有2个对应的异常处理法你发：

@ExceptionHandler({Exception.class })

public ModelAndView handlerArithmeticException2(Exception e) {}

@ExceptionHandler({ArithmeticException.class })

public ModelAndView handlerArithmeticException1(Exception e) {}

则优先级： 最短优先。

#### 11.2自定义异常类@ControllerAdvice

@ControllerAdvice

**public** **class** MyExceptionHandler {//不是控制器，仅仅是 用于处理异常的类

@ExceptionHandler({Exception.**class** })

**public** ModelAndView handlerArithmeticException2(Exception e) {

ModelAndView mv = **new** ModelAndView("error");

System.***out***.println(e +"============"+"该@ControllerAdvice中的异常处理方法，可以处理任何类中的异常");

mv.addObject("er", e) ;

**return** mv;

}

}

注意:该类的@ControllerAdvice的注解必须在扫描包中

否则无效

<!-- 扫描 有注解的包 -->

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

总结：如果一个方法用于处理异常，并且只处理当前类中的异常：@ExceptionHandler

如果一个方法用于处理异常，并且处理所有类中的异常： 类前加@ControllerAdvice、 处理异常的方法前加@ExceptionHandler

#### 11.3自定义异常显示页面 @ResponseStatus

@ResponseStatus(value=HttpStatus.***FORBIDDEN***,reason="数组越界222!!!")

**public** **class** MyArrayIndexOutofBoundsException **extends** Exception {//自定义异常

}

@ResponseStatus也可以标志在方法前：

@RequestMapping("testMyException")

public String testMyException(@RequestParam("i") Integer i) throws MyArrayIndexOutofBoundsException {

if(i == 3) {

throw new MyArrayIndexOutofBoundsException();//抛出异常

}

return "success" ;

}

#### 11.4异常处理的实现类

DefaultHandlerExceptionResolver:SPringMVC在一些常见异常的基础上（300 500 405），新增了一些异常，例如：

\*@seeorg.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler

\* @see #handleNoSuchRequestHandlingMethod

\* @see #handleHttpRequestMethodNotSupported ：如果springmvc的处理方法限制为post方式，如果实际请求为get,则会触发此异常显示的页面

\* @see #handleHttpMediaTypeNotSupported

\* @see #handleMissingServletRequestParameter

\* @see #handleServletRequestBindingException

\* @see #handleTypeMismatch

\* @see #handleHttpMessageNotReadable

\* @see #handleHttpMessageNotWritable

\* @see #handleMethodArgumentNotValidException

\* @see #handleMissingServletRequestParameter

\* @see #handleMissingServletRequestPartException

\* @see #handleBindException

#### 11.5通过配置来实现异常的处理

<!-- SimpleMappingExceptionResolver:以配置的方式 处理异常 -->

<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">

<!-- 如果发生异常，异常对象会被保存在 exceptionAttribute的value值中；并且会放入request域中 ；异常变量的默认值是 exception-->

<!--<property name="exceptionAttribute" value="exception"></property>-->

<property name="exceptionMappings">

<props>

<!-- 相当于catch(ArithmeticException ex){ 跳转：error } -->

<prop key="java.lang.ArithmeticException">

error

</prop>

<prop key="java.lang.NullPointerException">

error

</prop>

</props>

</property>

</bean>

### SSM整合

#### Jar

mybatis-spring.jar spring-tx.jar spring-jdbc.jar spring-expression.jar

spring-context-support.jar spring-core.jar spring-context.jar

spring-beans.jar spring-aop.jar spring-web.jar commons-logging.jar

commons-dbcp.jar ojdbc.jar mybatis.jar log4j.jar commons-pool.jar

spring-webmvc.jar

Spring - SpringMVC - MyBatis

1.

Spring - MyBatis : 需要整合：将MyBatis的SqlSessionFactory 交给Spring

2

Spring - SpringMVC ： 就是将Spring - SpringMVC 各自配置一遍

思路：

SqlSessionFactory -> SqlSession ->StudentMapper ->CRUD

可以发现 ，MyBatis最终是通过SqlSessionFactory来操作数据库，

Spring整合MyBatis 其实就是 将MyBatis的SqlSessionFactory 交给Spring

#### 12.1整合mybatis

<!-- 加载db.properties文件 -->

<bean id=*"config"* class=*"org.springframework.beans.factory.config.PreferencesPlaceholderConfigurer"*>

<property name=*"locations"*>

<array>

<value>classpath:db.properties</value>

</array>

</property>

</bean>

<!-- 配置配置数据库信息（替代mybatis的配置文件conf.xml） -->

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*>

<property name=*"driverClassName"* value=*"${driver}"*></property>

<property name=*"url"* value=*"${url}"*></property>

<property name=*"username"* value=*"${username}"*></property>

<property name=*"password"* value=*"${password}"*></property>

</bean>

<!-- conf.xml : 数据源,mapper.xml -->

<!-- 配置MyBatis需要的核心类：SqlSessionFactory -->

<!-- 在SpringIoc容器中 创建MyBatis的核心类 SqlSesionFactory -->

<bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

<!-- 加载mapper.xml路径 -->

<property name=*"mapperLocations"* value=*"classpath:org/shibc/dao/\*.xml"*></property>

</bean>

<!-- 将MyBatis的SqlSessionFactory 交给Spring -->

<bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>

<property name=*"sqlSessionFactoryBeanName"* value=*"sqlSessionFactory"*></property>

<property name=*"basePackage"* value=*"org.shibc.dao"*></property>

<!--上面basePackage所在的property的作用：

将org.lanqiao.mapper包中，所有的接口 产生与之对应的 动态代理对象

（对象名 就是 首字母小写的接口名） ：studentMapper.querystudentBYNO();

-->

</bean>

#### 12.2整合springmvc

<servlet>

<servlet-name>springDispatcherServlet</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> <!--加载spring和springmvc文件 -->

</init-param>

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

</servlet>

<!-- Map all requests to the DispatcherServlet for handling -->

<servlet-mapping>

<servlet-name>springDispatcherServlet</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

1、 用户发送请求至前端控制器DispatcherServlet。

2、 DispatcherServlet收到请求调用HandlerMapping处理器映射器。

3、 处理器映射器找到具体的处理器(可以根据xml配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。

4、 DispatcherServlet调用HandlerAdapter处理器适配器。

5、 HandlerAdapter经过适配调用具体的处理器(Controller，也叫后端控制器)。

6、 Controller执行完成返回ModelAndView。

7、 HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet。

8、 DispatcherServlet将ModelAndView传给ViewReslover视图解析器。

9、 ViewReslover解析后返回具体View。

10、DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。

11、 DispatcherServlet响应用户。