**Springmvc执行基本的增删改查操作：**

1. 列出全部的员工

/\*\*

\* 将集合放到map里面，他会自动的放到request属性与里面

\* \*/

@RequestMapping("/emps")

**public** String list(Map<String, Object> map) { //列出全部的员工信息

map.put("employees",employeeDao.getAll()); //通过实际操作用类查询出所有员工。将所有员工保存在request请求域？然后跳转到显示页面。

**return** "list";

}

1. 实现增加新员工

@Controller

**public** **class** EmployeeHandler {

@Autowired

**private** EmployeeDao employeeDao; //员工实例操作类

@Autowired

**private** DepartmentDao departmentDao; //部门实际操作类

//执行添加

@RequestMapping(value="/emp",method=RequestMethod.***POST***)//添加新员工

**public** String save(Employee employee) { //将提交过来的表单进行自动封装成对象

employeeDao.save(employee); //调用真实操作方法中的保存

return "redirect:/emps"; //带redirect前缀

}

//准备添加

@RequestMapping(value="emp",method=RequestMethod.***GET***)

**public** String input(**Map<String, Object> map**) { //添加新员工的准备

map.put("departments", departmentDao.getDepartments()); //取得所有部门，增加页面进行下拉选择框的回显

**map.put("employee", new Employee()); //form标签表单提交，需要在属性域中有对应属性的bean**

**return** "input";

}

/\*\*

\* 将集合放到map里面，他会自动的放到request属性与里面

\* \*/

@RequestMapping("/emps")

**public** String list(**Map<String, Object> map**) { //列出全部的员工信息

map.put("employees",employeeDao.getAll()); //**将所有员工保存在request请求域？**

**return** "list";

}

}

添加页面：

<%@page import=*"java.util.HashMap"*%>

<%@page import=*"java.util.Map"*%>

<%@ page language=*"java"* contentType=*"text/html; charset=GBK"*

pageEncoding=*"GBK"*%>

<%@ taglib prefix=*"form"* uri=*"http://www.springframework.org/tags/form"*%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Insert title here</title>

</head>

<body>

<!-- **1.为什么要使用form标签**

**可以更快的开发出表单页面，而且可以更加方便的进行表单值回显**

**2.注意：**

**在表单中可以通过modelAttribute属性指定绑定的模型属性**

**若没有指定该属性，则默认从request域对象中读取名字为 command 的表单的bean**

**如果该属性也不存在，则会发生异常**

**也就是说当表单提交的时候，在请求的目标方法中要有一个bean，该bean具有表单中所有元素对应的属性，且该bean的名字要与表单中modelAttribute属性对应的名字一样。**

**若请求目标方法中没有，则在属性域中查找看看有没有这样的一个bean。因为springmvc认为表单一定是要进行回显的。即使你是第一次访问**

-->

<!--属性域中一定要有一个名字为employee的属性，不然会报异常-->

<form:form action=*"emp"* method=*"post"* **modelAttribute=*"employee"***>

<!-- path属性对应HTML表单标签的name属性值 -->

LastName:<form:input path=*"lastName"*/>

<br>

Email:<form:input path=*"email"*/>

<br>

<% //放在属性域中方便，下面的单选按钮组进行操作

Map<String,String> genders = **new** HashMap();

genders.put("1", "male");

genders.put("0", "Female");

**request.setAttribute("genders", genders);**

%>

**Gender:<form:radiobuttons delimiter=*"*<br>*"* path=*"gender"* items=*"*${genders }*"*/>**

<br>

**Department:<form:select path=*"department.id"* items=*"*${departments }*"***

**itemLabel=*"departmentName"* itemValue=*"id"*></form:select>**

<br>

<input type=*"submit"* value=*"submit"*>

</form:form>

</body>

</html>

**3.SpringMVC\_RESTRUL\_CRUD\_删除操作&amp;处理静态资源**

**执行删除操作时，由于超链接是使用的get请求，而删除操作需要使用DELETE请求，所以需要把get请求转换为post请求，然后由post请求转换为DELETE请求，这就要借助到jQuery。由于jQuery是外部静态资源，所以需要在springmvc配置文件中配置<mvc:default-servlet-handler/> 但是配置了<mvc:default-servlet-handler/> 以后会使requestMapping请求失效，所以需要配置：<mvc:annotation-driven></mvc:annotation-driven>（让requestMapping请求起作用） 由于删除操作需要传递一个id值，所以在handler中需要使用@pathvariable 来将传递的参数映射到handler中的请求方法中。**

**<mvc:default-servlet-handler/> 的作用：**

**<!--**

**<mvc:default-servlet-handler/>将在springmvc上下文中定义一个DefaultServletHttpRequestHandler,**

**他会对进入DispatcherServlet的请求进行筛查，如果发现是没有经过映射的请求，就将该请求交由web服务器默认的Servlet处理，**

**如果不是静态资源的请求，才由DispatcherServlet继续处理。**

**一般web应用服务器默认的Servlet的名称都是default，父使用的web服务器默认的servlet名称不是default，则需要通过**

**default-servlet-name属性显示指定**

**-->**

**示例代码：**

**<!--**

**<mvc:default-servlet-handler/>将在springmvc上下文中定义一个DefaultServletHttpRequestHandler,**

**他会对进入DispatcherServlet的请求进行筛查，如果发现是没有经过映射的请求，就将该请求交由web服务器默认的Servlet处理，**

**如果不是静态资源的请求，才由DispatcherServlet继续处理。**

**一般web应用服务器默认的Servlet的名称都是default，父使用的web服务器默认的servlet名称不是default，则需要通过**

**default-servlet-name属性显示指定**

**-->**

**<mvc:default-servlet-handler/>**

**<!-- 让requestMpping起作用 -->**

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

<%@ page language=*"java"* contentType=*"text/html; charset=GBK"*

pageEncoding=*"GBK"*%>

<%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"*%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>显示全部员工</title>

**<!-- springmvc请求静态资源**

**1.为什么会有这样的问题：**

**优雅的REST风格的资源URL不希望带.html或.do等后缀**

**若将DispatcherServlet请求映射配置为/，**

**则springmvc将捕获web容器的所有请求，包括静态资源的请求，springmvc会将他们当成一个普通的请求处理，**

**因找出到对应处理器将导致错误**

**2.解决：**

**在springmvc的配置文件中配置一个<mvc:default-servlet-handler>**

**-->**

**<script type=*"text/javascript"* src=*"js/jquery-3.3.1.js"*></script>**

**<script type=*"text/javascript"* >**

**$(function(){**

**$(".delete").click(function(){**

**var href = $(this).attr("href");**

**$("form").attr("action",href).submit();**

**return false;**

**})**

**})**

**</script>**

</head>

<body>

<form action=*""* **method=*"post"***>

**<input type=*"hidden"* name=*"\_method"* value=*"DELETE"*/>**

</form>

<c:if test=*"*${**empty** requestScope.employees }*"*>

没有任何员工信息！

</c:if>

<c:if test=*"*${!**empty** requestScope.employees }*"*>

<table border=*"1"* cellpadding=*"10"* cellspacing=*"0"*>

<tr>

<th>ID</th>

<th>LastName</th>

<th>Email</th>

<th>Gender</th>

<th>Department</th>

<th>Edit</th>

<th>Delete</th>

</tr>

<c:forEach items=*"*${requestScope.employees}*"* var=*"emp"*>

<tr>

<td>${emp.id }</td>

<td>${emp.lastName }</td>

<td>${emp.email }</td>

<td>${emp.gender == 0?'Female':'Male' }</td>

<td>${emp.department.departmentName }</td>

<td><a href=*"#"*>Edit</a></td>

<!-- 必须要将超链接转为DELETE请求 -->

<td><a href=*"emp/*${emp.id}*"* class=*"delete"*>Delete</a></td>

</tr>

</c:forEach>

</table>

</c:if>

<br><br>

**<a href=*"emp"* class=*".delete"*>Add New Employee</a>**

</body>

</html>

**SpringMVC\_RESTRUL\_CRUD\_修改操作：**

**首先由一个超链接，点击后发出put请求，到handler类的具体方法中，handler类对应的处理方法通过id入参映射，查找到一个employee对象，并将该对象保存在map中（即request请求域中），然后查找到所有的部门信息，也保存在map中，然后通过return值，跳转到修改的页面进行表单回显，要求lastName字段不可做修改。**

**当修改好信息后，点击按钮确认修改后，表单的提交方式为PUT，所以需要在form表单中增加一个隐藏域，name=”\_method”,value=”PUT”，即将post请求转化为PUT请求。提交到handler类中的相应的处理方法中。在处理方法中，由于要求lastName字段不能做修改，所以，需要用到@ModelAttribute注解的方法，模拟从数据库中根据id查询出要修改的一个员工，然后在处理方法中将表单提交的值赋值到该对象，然后返回list.jsp页面。**

**@ModelAttribute**

**public** **void** getEmployee(@RequestParam(value="id",required=**false**)

Integer id,**Map<String,Object> map**) { //在所有的请求方法执行前都会先执行@ModelAttribute注解修饰的方法

**if**(id != **null**) { //指的是通过隐藏域传递过来的id

map.put("employee", employeeDao.get(id));

}

}

@RequestMapping(value="/emp",method=RequestMethod.***PUT***)

**public** String update(**@ModelAttribute Employee employee**) { //确定修改操作

employeeDao.save(employee); //调用修改操作，但是LastName字段不可修改，所以此时就要使用@ModelAttribute注解，

//进行事先在数据库中取出一个对象，然后将表单中提交过来的值赋在他对应的属性上

**return** "redirect:/emps";

}

@RequestMapping(value="/emp/{id}" ,method=RequestMethod.***GET***) //准备修改员工信息

**public** String input(**@PathVariable("id") Integer id**,Map<String,Object> map) {

map.put("employee", employeeDao.get(id)); //为了修改的表单回显，将值放入request属性域中

map.put("departments", departmentDao.getDepartments());

**return** "input";

}

**自定义类型转换器：**

步骤：

1. 自定义一个转换器类，实现Convert<S,T>接口.在该类中编写源数据如何向目标数据转换。
2. 在handler中编写一个请求映射方法在该方法中进行转换后的数据保存.
3. 在springmvc的配置文件中，配置conversionservice 然后在，<mvc:annotation-driver>中注入conversion-service=“”。

范例：

**package** com.wp.springmvc.converter;

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

**import** org.springframework.stereotype.Controller;

**import** com.wp.springmvc.crud.entities.Department;

**import** com.wp.springmvc.crud.entities.Employee;

@Controller

**public** **class** EmployeeConverter **implements** Converter<String, Employee> {

@Override

**public** Employee convert(String source) {

**if**(source != **null**) {

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

**if**(vals != **null** && vals.length == 4) {

String lastName = vals[0];

String email = vals[1];

Integer gender = Integer.*parseInt*(vals[2]);

Department department = **new** Department();

department.setId(Integer.*parseInt*(vals[3]));

Employee employee = **new** Employee(**null**, lastName, email, gender, department);

System.***out***.println(source + "---convert---" + employee);

**return** employee;

}

}

**return** **null**;

}

}

@Controller

**public** **class** SpringTest {

@Autowired

**private** EmployeeDao employeeDao;

@RequestMapping("/testConversionServiceConver")

**public** String testConversionServiceConver(@RequestParam("employee") Employee employee) {

System.***out***.println("save: " + employee);

employeeDao.save(employee);

**return** "redirect:/emps";

}

}

<!-- 让requestMpping其作用 -->

<mvc:annotation-driven conversion-service=*"ConversionService"*></mvc:annotation-driven>

<!-- 配置conversionservice -->

<bean id=*"ConversionService"*

class=*"org.springframework.context.support.ConversionServiceFactoryBean"*>

<property name=*"converters"*>

<set>

<ref bean=*"employeeConverter"*/>

</set>

</property>

</bean>

**SpringMVC\_annotation-driven配置：**

**SpringMVC\_InitBinder注解：**

由@InitBinder标记的方法不能有返回值，他必须申明为void

@InitBinder方法的参数通常是WebDataBinder

**SpringMVC\_数据的格式化:**

类型的转化和数据的格式化一般都是一起进行的。

即由前台页面提交的数据格式（字符串），怎样才能转换为javabean中的数据格式。比如前台提交的出生日期：1995-10-04为字符串类型，怎样将该字符串转换为Javabean中的Date类型。

1. 在javabean中的对应属性上面用@DateTimeFormat或@NumberFormat或其他注解进行标注输入要转换的格式类型。
2. 在表单中按照Javabean中标注的数据格式进行输入。

Eg:

@Past

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

**private** Date birth;

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

**private** Float salary;

**若格式化失败呢（即类型转换出错）？**

可以在handler中请求映射的方法中添加一个参数BindingResult，该参数是用来接收错误的集合。可以在方法中判断该集合中的数量是否大于0。若大于零则表示出错了。可以打印出错信息。

范例：

@RequestMapping(value="/emp",method=RequestMethod.***POST***)//添加新员工

**public** String save(Employee employee**,BindingResult result**) {

System.***out***.println("employee: " + employee);

//若数据类型转换出错了

**if**(**result.getErrorCount() > 0)** {

System.***out***.println("出错了！");

**for**(**FieldError error:result.getFieldErrors()**) {

System.***out***.println(error.getField() + ":" + error.getDefaultMessage());

}

}

employeeDao.save(employee); //调用真实操作方法中的保存

**return** "redirect:/emps";

}

**SpringMVC\_JSR303数据校验：**

JSR 303是java为bean数据合法校验的一个标准。

Hibernate Validator是JSR的一个参考时实现。

1. 加入JSR 303的jar包。
2. 在配置文件中配置
3. 在处理方法的入参上标注@valid注解即可让Springmvc在完成数据绑定后执行数据校验工作
4. SpringMVC是通过对处理方法签名的规约来保存校验结果的：前一个表单/命令对象的校验结果保存到随后的入参中，这个保存校验结果的入参必须是BindingResult或Errors类型。

**3.数据校验**

**1）.如何校验？注解？**

**①.使用JSRR 303验证标准**

**②.加入hibernate validate 验证框架的jar包**

**③.在springmvc的配置文件中添加<mvc:annotation-drivern>**

**④.需要在bean的属性上添加对应的注解**

**⑤.在目标方法bean类型的前面(即入参的前面)添加@valid**

**2）.验证出错转向到哪一个页面？**

**注意：在处理方法入参中，需要验证的bean和处理错误结果（BandingResult两个参数需要挨着）**

**3).错误消息如何显示，如何把错误消息进行国际化？**

比如在employee原子类中加入相关验证注解：

**private** Integer id;

**@NotEmpty** //该注解是为了数据校验

**private** String lastName;

**@Email**

**private** String email;

//1 male, 0 female

**private** Integer gender;

**private** Department department;

**@Past**

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

**private** Date birth;

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

**private** Float salary;

在处理方法的入参中加入注解：

@RequestMapping(value="/emp",method=RequestMethod.***POST***)//添加新员工

**public** String save(**@Valid Employee employee**,**BindingResult result**,

Map<String,Object> map) { //入参这里的employee如果在request属性域中没有改对象，则新创建一个，然后放在request域中。

System.***out***.println("employee: " + employee);

//若数据类型转换出错了

**if**(result.getErrorCount() > 0) {

System.***out***.println("出错了！");

**for**(FieldError error:result.getFieldErrors()) {

System.***out***.println(error.getField() + ":" + error.getDefaultMessage());

}

//若验证出错则转向定制的页面

**map.put("departments", departmentDao.getDepartments());** //查询出全部的部门信息，将部门信息放在request请求域,为了表单回显

return "input"; //返回增加页面，进行表单回显

}

employeeDao.save(employee); //若数据校验没有错误，调用真实操作方法中的保存

**return** "redirect:/emps";

}

**SpringMVC\_错误消息的显示及国际化：**

1. 先搞一个资源文件，资源文件的key命名规则为：**原子类中的属性校验注解类型名.**

**处理方法中的入参对象名（若用了@modelAttribute修饰，则为@modelAttribute修饰的value值名）.要校验的属性名。**

1. 在springmvc的配置文件中配置国际化资源文件。

范例：

NotEmpty.employee.lastName=LastName:\u4E0D\u80FD\u4E3A\u7A7A

Email.employee.email=Email:\u683C\u5F0F\u4E0D\u5408\u6CD5

Past.employee.birth=Birth:\u4E0D\u662F\u4E00\u4E2A\u5408\u6CD5\u7684\u65F6\u95F4

typeMismatch.employee.birth=Birth:\u4E0D\u662F\u4E00\u4E2A\u5408\u6CD5\u7684\u65E5\u671F

<!-- 配置国际化资源文件 -->

<bean id=*"messageSource"*

class=*"org.springframework.context.support.ResourceBundleMessageSource"*>

<property name=*"basename"* value=*"i18n"*></property>

</bean>

**SpringMVC\_返回JSON：**

步骤：

1. 编写一个jQuery函数，进行发送请求和接受返回数据。
2. 编写处理方法，进行返回数据并且一定要用@ResponseBody 注解标注。
3. 导入相关jar包。

范例：

@ResponseBody

@RequestMapping("/testJson")

**public** Collection<Employee> testJson() {

**return** employeeDao.getAll(); //返回所有的员工

}

<script type=*"text/javascript"* src=*"js/jquery-3.3.1.js"*></script>

<script type=*"text/javascript"*>

$(**function**() {

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

**var** url = **this**.href;

**var** args = {};

**$.post(url,args,function(data)**{ //接受返回的数据

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

**var** id = data[i].id;

**var** lastName = data[i].lastName;

alert(id + ": " + lastName);

}

});

**return** **false**;

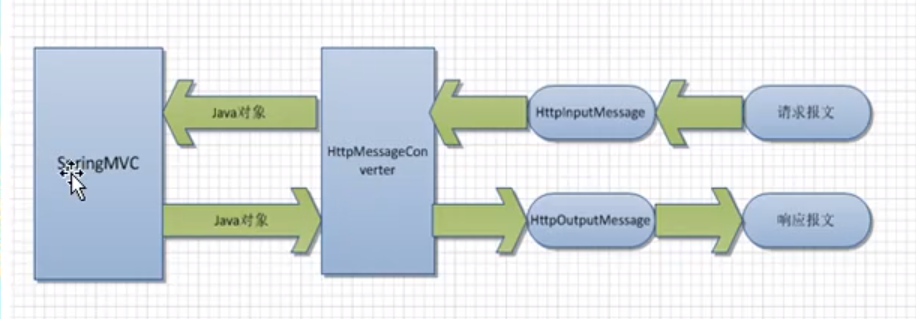
});

})

</script>

**SpringMVC\_HttpMessageConverter原理：**

其实json数据格式的转换原理就是一个HttpMessageConverter。如下：



**SpringMVC\_使用HttpMessageConverter：**

使用HttpMessageConverter<T>将请求信息转化并绑定到处理方法的入参中，或将响应的结果转为对应类型的响应信息，Spring提供了两种途径：

1. 使用@RequestBody / @ResponseBody对处理方法进行标注。
2. 使用HttpEntity<T> / ResponseEntity<T>作为处理方法的入参或返回值。

**当控制器处理方法使用到@RequestBody或ResponseBody或HttpEntity<T> / RespnoseEntity<T>时，Spring首先根据请求头或响应头的Accept属性选择匹配的HTTPMessageConvert,进而根据参数类型或泛型类型的过滤得到匹配的HttpMessageConverter，若找不到可用的HttpMessageConverter将报错。**

**@RequestBody和@ResponseBody不需要成对出现。**

范例：

**@ResponseBody**

@RequestMapping("/testHttpMessageConverter")

**public** String testHttpMessageConverter(**@RequestBody String body**) { //要求传过来的参数或文件内容.模拟文件上传

System.***out***.println(body);

**return** "helloworld!" + **new** Date(); //将这个字符串返回给客户端

}

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

File:<input type=*"file"* name=*"file"*>

Desc:<input type=*"text"* name=*"desc"*>

<input type=*"submit"* value=*"submit"*>

</form>

使用ResponseEntity：模拟文件下载

@RequestMapping("/testResponseEntity")

**public** ResponseEntity<**byte**[]> testResponseEntity(HttpSession session) **throws** IOException{

**byte**[] body = **null**;

ServletContext servletContext = session.getServletContext();

InputStream in = **servletContext.getResourceAsStream("/files/demo.txt");** //文件原地址

body = **new** **byte**[in.available()];

in.read(body); //读到字节数组里面

HttpHeaders headers = **new** HttpHeaders(); //设置响应头

headers.add("Content-Disposition", "attachment;filename=abc.txt"); //设置向客户端返回的响应头

HttpStatus statusCode = HttpStatus.***OK***; //设置响应码

ResponseEntity<**byte**[]> response = **new** ResponseEntity<**byte**[]>(body,headers,statusCode);

**return** response;

}

**SpringMVC\_国际化\_概述：**

**<!-- 关于国际化：**

**1.在页面上能够根据浏览器语言设计的情况对文本（不是内容）,时间，数值进行本地化处理**

**2.在bean中获取国际化资源文件Locale对应的消息**

**2.可以通过超链接切换Locale，而不再依赖于浏览器的语言设置情况。**

**解决：**

**1.使用JSTL的fmt标签**

**2.bean中注入ResourceBundleMessageSource的示例，使用其对应的getMessage方法即可**

**3.配置LocalResovler和LocaleChangeInterceptor**

**-->**

**SpringMVC\_国际化\_前两个问题：**

第一个问题：

配置springmvc的配置文件：

<!-- <mvc:view-controller path="/i18n" view-name="i18n"/> -->

<mvc:view-controller path=*"/i18n2"* view-name=*"i18n2"*/>

以上的参数中分别对应访问的名称和视图的名称

第二个问题：

**在bean中获取国际化资源文件Locale对应的消息：**

@RequestMapping("/i18n")

**public** String testI18n(Locale locale) {

String val = messageSource.getMessage("i18n.user", **null**,locale); //取得i18n.user的值

System.***out***.println(val);

**return** "i18n";

}

第三个问题：

编写配置文件：

<!-- 配置SessionLocalResolver -->

<bean id=*"localeResolver"*

class=*"org.springframework.web.servlet.i18n.SessionLocaleResolver"*>

</bean>

<!-- 配置LocalChanceInterceptor -->

<mvc:interceptors>

<bean class=*"org.springframework.web.servlet.i18n.LocaleChangeInterceptor"*></bean>

</mvc:interceptors>

添加两个超链接：

<a href=*"****i18n?locale=zh\_CH****"*>中文</a>

<br><br>

<a href=*"****i18n?locale=en\_US****"*>英文</a>

<br><br>

点击中文便会以中文显示，点击英文便会以英文显示。

**SpringMVC\_文件上传：**

步骤：

1.在配置文件中配置： MultipartResovler

2.编写文件上传表单

3.编写请求处理方法

范例：

<!-- 配饰MultipartResovler 实现文件上传-->

<bean id=*"multipartResolver"*

class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*>

<property name=*"defaultEncoding"* value=*"GBK"*></property>

<property name=*"maxUploadSize"* value=*"1024000"*></property>

</bean>

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

File:<input type=*"file"* name=*"file"*>

Desc:<input type=*"text"* name=*"desc"*>

<input type=*"submit"* value=*"submit"*>

</form>

@RequestMapping("/testFileUpload")

**public** String testFileUpload(**@RequestParam("desc") String desc,**

**@RequestParam("file") MultipartFile file**) **throws** IOException {

System.***out***.println("desc : " + desc);

System.***out***.println("file name : " + file.getOriginalFilename());

System.***out***.println("InputStream : " + file.getInputStream());

**return** "success";

}

**SpringMVC\_第一个自定义的拦截器：**

**SpringMVC\_拦截器的配置：**

步骤：

1. 先自定义一个拦截器类，要实现HandlerInterceptor接口
2. 在配置文件中配置<mvc:interceptors>节点中配置一个拦截器的bean。

范例：

**package** com.wp.springmvc.interceptor;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

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

**import** org.springframework.web.servlet.ModelAndView;

**public** **class** FirstInterceptor **implements** HandlerInterceptor {

/\*\*

\* 在渲染视图之后别调用

\*

\* 释放资源

\* \*/

@Override

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

**throws** Exception {

System.***out***.println("[FirstInterceptor] afterConpletion");

}

/\*\*

\* 调用目标方法之后，但渲染视图之前 被调用

\*

\* 可以对请求域中的属性或视图做出修改

\* \*/

@Override

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

**throws** Exception {

System.***out***.println("[FirstInterceptor] postHandle");

}

/\*\*

\* 该方法在目标方法之前被调用，

\* 若返回值为true，则继续调用后续的拦截器和目标方法

\* 若返回值为false，则不会再调用拦截器和目标方法

\*

\* 可以考虑做权限，日志，事务等等

\* \*/

@Override

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

System.***out***.println("[FirstInterceptor] preHandle");

**return** **true**;

}

}

<mvc:interceptors>

<!-- 配置自定义的拦截器 -->

<bean class=*"com.wp.springmvc.interceptor.FirstInterceptor"*></bean>

<!-- 配置LocalChanceInterceptor 用于国际化拦截器 -->

<bean class=*"org.springframework.web.servlet.i18n.LocaleChangeInterceptor"*></bean>

</mvc:interceptors>

**SpringMVC\_多个拦截方法的执行顺：**

<mvc:interceptors>

<!-- 配置自定义的拦截器 -->

<bean class=*"com.wp.springmvc.interceptor.FirstInterceptor"*></bean>

<!-- 配置拦截器作用的（不作用的）路径 -->

**<mvc:interceptor>**

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

**<bean class=*"com.wp.springmvc.interceptor.SecondInterceptor"*></bean>**

**</mvc:interceptor>**

<!-- 配置LocalChanceInterceptor 用于国际化拦截器 -->

<bean class=*"org.springframework.web.servlet.i18n.LocaleChangeInterceptor"*></bean>

</mvc:interceptors>

可以配置拦截器对哪些请求或操作进行拦截或不进行拦截。

**SpringMVC\_多个拦截方法的执行顺序：**

1. 对preHandle按配置的顺序执行，对postHandler和afterCommplete按配置的反序执行。

**SpringMVC\_异常处理\_ExceptionHandler注解：**

Springmvc通过HandlerExceptionResovler处理程序的异常，包括Handler映射，数据绑定以及目标方法执行时发生的异常。

Springmvc提供的HandlerExceptionResolver的实现类。

**ExceptionHandlerExceptionResovler:**

主要处理Handler中用@ExceptionHandler注解定义的方法。

@ExceptionHandler注解定义的方法优先级问题：例如发生的是NullPointerException,但是声明的异常有RuntimeException和Exception，此时会根据异常的最近继承关系找到继承深度最浅的哪个@ExceptionHandler注解方法，即标记了RuntimeException的方法

ExceptionHandlerMethodResolver内部若找不到@Exception注解的话，会找@ControllerAdvice中的@Exception注解方法。

如何将异常信息带到页面进行显示：

在处理异常的方法中使用ModelAndView对象去将异常信息带到页面进行显示。

范例：

/\*\*

**\* 在handler类中标记的@ExceptionHandler只能处理当前handler类中方法出现的异常，可以自定义一个更**

**\* 全局的异常处理**

**\* 1.在@ExceptionHandler方法的入参中加入Exception类型的参数，该参数即对应发生的异常对象**

**\* 2.@ExceptionHandler方法的入参中不能传入Map。若希望把异常信息传到页面上，需要使用ModelAndView作为返回值**

**\* 3.@ExceptionHandler标记的异常有优先级的问题**

**\* 4.@ControllerAdvice:如果在当期那Handler中找不到@ExceptionHandler 方法来处理当前方法出现的异常，**

**\* 则去@ControllerAdvice标记的类中查找@ExceptionHandler 标记的方法来处理异常。**

\* \*/

/\* @ExceptionHandler({RuntimeException.class})

public ModelAndView handlerArithmeticException2(Exception ex) {

System.out.println("[出异常了：] " + ex);

ModelAndView mv = new ModelAndView("error"); //使用modelandview把错误信息显示到页面上。

mv.addObject("exception",ex); //将异常保存到属性域中

return mv;

}

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

public ModelAndView handlerArithmeticException(Exception ex) {

System.out.println("出异常了： " + ex);

**ModelAndView mv = new ModelAndView("error");** //使用modelandview把错误信息显示到页面上。

**mv.addObject("exception",ex);** //将异常保存到属性域中

return **mv**;

}\*/

//**在**@ExceptionHandler **标记的异常处理方法中，不能使用map进行将异常信息保存到属性域中。只能使用ModelAndView。**

@RequestMapping("/testExceptionHandlerExceptionResolver")

**public** String testExceptionHandlerExceptionResolver(@RequestParam("i") **int** i) { //异常测试

System.***out***.println("result: " + (10/i));

**return** "success";

}

**package** com.wp.springmvc.exception;

**import** org.springframework.web.bind.annotation.ControllerAdvice;

**import** org.springframework.web.bind.annotation.ExceptionHandler;

**import** org.springframework.web.servlet.ModelAndView;

@ControllerAdvice

**public** **class** TestExceptionHandler **extends** **Exception**{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

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

**public** String handlerArithmeticException(Exception ex) {

System.***out***.println("----->出异常了： " + ex);

ModelAndView mv = **new** ModelAndView("error"); //使用modelandview把错误信息显示到页面上。

mv.addObject("exception",ex); //将异常保存到属性域中

**return** "mv";

}

}

**SpringMVC\_异常处理：**

ResponseStatusExceptionResolver:

在异常及异常父类中找到@ResponseStatus注解，然后使用这个注解的属性进行处理。

电影以一个@ResponseStatus注解修饰的异常类

范例：

[//](mailto:/@ResponseStatus(value=HttpStatus.NOT_FOUND,reason=\"测试。。。\"))**[@ResponseStatus(value=HttpStatus.NOT\_FOUND,reason="测试。。。")](mailto:/@ResponseStatus(value=HttpStatus.NOT_FOUND,reason=\"测试。。。\")) //修饰方法**

@RequestMapping("/UserNameNotMatchPasswordException")

**public** String testUserNameNotMatchPasswordException(@RequestParam("i") **int** i) {

**if**(i == 13) {

**throw** **new** UserNameNotMatchPasswordException();

}**else** {

System.***out***.println("UserNameNotMatchPasswordException.....");

}

**return** "success";

}

**@ResponseStatus(value=HttpStatus.*FORBIDDEN*,reason="用户名和密码不匹配")//用在类上进行修饰类**

**public** **class** UserNameNotMatchPasswordException **extends** RuntimeException{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

}

**SpringMVC\_异常处理\_DefaultHandlerExceptionResolver:**

@RequestMapping(value="/testDefaultHandlerExceptionResolver",method=RequestMethod.***POST***)

**public** String testDefaultHandlerExceptionResolver() {

System.***out***.println("testDefaultHandlerExceptionResolver.....");

**return** "success";

}

**SimpleMappingExceptionResolver:**

如果希望对所有的异常进行统一处理，可以使用SimpleMappingExceptionResolver，他将异常类名映射为视图名称，即发生异常时使用对应的视图报告异常。（即：可以在xml文件中配置，出现哪个异常，转向哪个页面）

范例：

<!-- 配置使用 SimpleMappingExceptionResolver 来映射异常 -->

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

<property name=*"exceptionMappings"*>

<props>

<prop

<!--配置集体异常类的全类名和异常后跳转的页面。可配置多个具体异常-->

key=*"java.lang.ArrayIndexOutOfBoundsException"*>error</prop>

</props>

</property>

<!-- 配置保存的异常的名字，以便在页面中通过名字取得异常信息 。默认的是名字为exception-->

<property name=*"exceptionAttribute"* value=*"ex"*></property>

</bean>

@RequestMapping("/SimpleMappingExceptionResolver")

**public** String testSimpleMappingExceptionResolver(@RequestParam("i") **int** i) {

String [] vals = **new** String[10];

System.***out***.println(vals[i]); //发生数组下标越界异常

**return** "success";

}

**SpringMVC\_运行流程图解：**

