**Spring 3.1MVC 实战**

邮箱：86207344@qq.com

[一、spring3.1 mvc核心思想介绍 2](#_Toc325181267)

[二、spring3.1 mvc 框架的特点 3](#_Toc325181268)

[三、spring 3.1 MVC入门列子HelloWorld 3](#_Toc325181269)

[四、springMVC参数传递 6](#_Toc325181270)

[五、springMVC标签介绍 9](#_Toc325181271)

[（2） input 标签 10](#_Toc325181272)

[（3）checkbox 标签 11](#_Toc325181273)

[（4）radiobutton 标签 12](#_Toc325181274)

[六、springMVC拦截器 14](#_Toc325181275)

[七、spring3 MVC 类型转换 17](#_Toc325181276)

[八、json格式数据的输入和输出 21](#_Toc325181277)

[九、spring3mvc文件上传 23](#_Toc325181278)

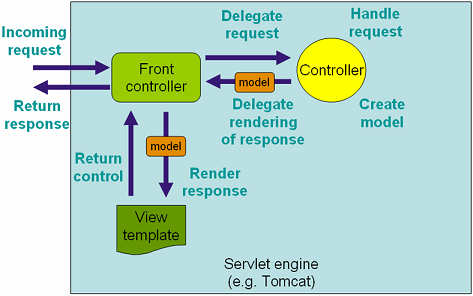
[十、spring mvc国际化和本地化 25](#_Toc325181279)

[十一、使用jsr303进行验证 29](#_Toc325181280)

# 一、spring3.1 mvc核心思想介绍

Spring MVC是spring的一个web组件，它为构建稳健的web应用提供了丰富的功能。

Spring MVC是基于每个逻辑和功能是高可配置的这样架构和设计的。当然spring MVC可以与其它流行的web框架像struts、webwork、javaserverface及tapestry实现无缝集成。我们看一下spring MVC的核心思想。



Spring请求的生命周期

为了便于理解，我把spring MVC的生命周期拆解为如下一个步骤。

总结一下springMVC几个关键的步骤，总共可以分为六个步骤，分别为：

1. 客户端向spring容器发起一个http请求
2. 发起的请求被前端控制起所拦截(DispatcherServlet)，前端控制器会去找恰当的映射处理器来处理这次请求。
3. 根据处理器映射（Handler Mapping）来选择并决定将请求发送给那一个控制器。
4. 在控制器中处理所发送的请求，并以modeandView（属性值和返回的页面）的形式返回给向前端控制器。
5. 前端控制器通过查询viewResolver对象来试着解决从控制返回的视图。
6. 如果前端控制找到对应的视图，则将视图返回给客户端，否则抛异常。

通过上面的图和springMVC生命周期的六个步骤，想必大家对springMVC的核心思想有个了大概的了解了，下面我们以实例为主，带领大家慢慢熟悉整个springMVC及如何使用springMVC。（本教程基于maven实现springMVC中的例子，所以大家得对maven需要有大概的了解）。

# 二、spring3.1 mvc 框架的特点

如果仅仅关注于web方面的支持，Spring有下面一些特点：

* 清晰的角色划分：控制器，验证器，命令对象，表单对象和模型对象；分发器，处理器映射和视图解析器；等等。
* 直接将框架类和应用类都作为JavaBean配置，包括通过应用上下文配置中间层引用，例如，从web控制器到业务对象和验证器的引用。
* 可适应性，但不具有强制性：根据不同的情况，使用任何你需要的控制器子类（普通控制器，命令，表单，向导，多个行为，或者自定义的），而不是要求任何东西都要从Action/ActionForm继承。
* 可重用的业务代码，而不需要代码重复：你可以使用现有的业务对象作为命令对象或表单对象，而不需要在ActionForm的子类中重复它们的定义。
* 可定制的绑定和验证：将类型不匹配作为应用级的验证错误，这可以保存错误的值，以及本地化的日期和数字绑定等，而不是只能使用字符串表单对象，手动解析它并转换到业务对象。
* 可定制的处理器映射，可定制的视图解析：灵活的模型可以根据名字/值映射，处理器映射和视图解析使应用策略从简单过渡到复杂，而不是只有一种单一的方法。
* 可定制的本地化和主题解析，支持JSP，无论有没有使用Spring标签库，支持JSTL，支持不需要额外过渡的Velocity，等等。
* 简单而强大的标签库，它尽可能地避免在HTML生成时的开销，提供在标记方面的最大灵活性。

# 三、spring 3.1 MVC入门列子HelloWorld

（1）在WEB-INF/web.xml中加入如下代码：

<servlet>

<servlet-name>dispatcherServlet</servlet-name>

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

<init-param>

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

<param-value>

classpath\*:spring-servlet.xml

</param-value>

</init-param>

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

</servlet>

<servlet-mapping>

<servlet-name>dispatcherServlet</servlet-name>

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

</servlet-mapping>

上述的配置的就是前段控制器，在servlet-mapping配置了\*.html，意味着所有以.html结尾的请求多会通过这个servlet，当dispatcherServlet启动时，他默认会在web-info目录下查找一个spring-servlet.xml的配置文件。上面我们通过显示指定了这个文件的位置，即在类路径底下的spring-servlet.xml.这个文件我们会在第二步点给他家做详细介绍。

（2）在类路径底下添加spring-servlet.xml文件，其内容如下：

<?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:p=*"http://www.springframework.org/schema/p"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.0.xsd"*>

<context:component-scan

base-package=*"com.pango.spring.helloworld.controller"* />

<bean id=*"viewResolver"*

class=*"org.springframework.web.servlet.view.UrlBasedViewResolver"*>

<property name=*"viewClass"*

value=*"org.springframework.web.servlet.view.JstlView"* />

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

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

</bean>

</beans>

上面这个文件，我们定义了一个<context:component-scan

base-package=*"com.pango.spring.helloworld.controller"* />

这样的标签，定义了这个后，当spring在启动时，会加载*com.pango.spring.helloworld.controller这个包底下及子包底下的所有的组件（这就包的自动扫描机制，即spring会将标有@Controller @Component等类加载到spring容器管理中），后面我们还定义了*<bean id=*"viewResolver"*

class=*"org.springframework.web.servlet.view.UrlBasedViewResolver"*>

<property name=*"viewClass"*

value=*"org.springframework.web.servlet.view.JstlView"* />

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

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

</bean>

ViewResolver是一个试图解析器，就是我们第一部分提到的springMVC生命周期中的第五步，上面这段的配置的意思就是，当我们从后端控制器中返回的视图时，前端控制器就根据这一段配置来返回一个具体的视图，如后端控制返回的是一个hello，根据上面的配置，最后前端控制器会组并成这样的一个地址：/web-inf/jsp/hello.jsp,然后从/web-inf/jsp/这个目录下面查找一个hello.jsp返回客户端。第三部分我们看我们写得HelloworldController后台控制器。

（3）在包底下写一个HelloWorldController的类，其内容如下：

@Controller

**public** **class** HelloWorldController {

@RequestMapping(value="/hello")

**public** String sayHello(ModelMap modelMap){

modelMap.put("sayHello", "hello world");

**return** "/hello";

}

}

在这里简单介绍下上面的配置，后面我们会详细讲解各个参数：

* Controller即声明这个类是一个控制器，上面第二部分我们说明了，只要加了@Controller标示的，spring会通过自动扫描机制，将这个类纳入spring容器管理中。
* @RequestMapping(value="/hello")，这个定义的就是一个请求路径，只要符合/hello路径的多会交给这个控制器的sayhello方法来处理。
* 最后我们返回/hello的视图给客户端。

（4）好了，大功告成，我们再在web-info/jsp/目录下添加一个hello.jsp文件，就可以启动运行我们的第一个程序了。hello.jsp的内容如下：

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

pageEncoding=*"UTF-8"*%>

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

<!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=UTF-8"*>

<title>Insert title here</title>

</head>

<body>

<span>${sayHello}</span>

</body>

</html>

运行后访问ip;port/project/hello.html就可以看到我们预期的结果了。

# 四、springMVC参数传递

mvc结构中，v层不断有数据和c层交互，所以弄明白在springMVC中如何与后台进行数据交互是极其重要的，当然在下面我不会介绍每一个方法，只是对常用的方法，对于我这里没有涉及的方法大家可以参考spring官方的文档中springMVC这个章节。下面我们来看一幅图。

当我们向springMVC发起请求到视图返回前，spring MVC帮我们做了主要是上面几个步骤，通过数据绑定、数据类型转换、验证、结果绑定这几个步骤。

让我们看下实例：

@RequestMapping("/user/find")

**public** String findUserById(@RequestParam("userId") **int** userId,ModelMap modelMap){

modelMap.put("userId", userId);

**return** "/user";

}

* @RequestMapping("/user/find")，是对请求映射的说明，这个注解中主要包含四个属性，分别value、method、params、header，这四个参数分别表示：

Value：指定路径

Method：请求方式

Params：参数

Headers：请求头

后面三个就是对请求路径的一个限制条件。

SpringMVC对于路径的定义非常的灵活

以下URL都是合法的：

* /user/\*/createUser

匹配/user/aaa/createUser、/user/bbb/createUser等URL。

* /user/\*\*/createUser

匹配/user/createUser、/user/aaa/bbb/createUser等URL。

* /user/createUser??

匹配/user/createUseraa、/user/createUserbb等URL。

* /user/{userId}

匹配user/123、user/abc等URL。

* /user/\*\*/{userId}

匹配user/aaa/bbb/123、user/aaa/456等URL。

* company/{companyId}/user/{userId}/detail

匹配company/123/user/456/detail等的URL。

* 对RequestParam的介绍

@RequestParam有以下三个参数。

* value：参数名。
* required：是否必需，默认为true，表示请求中必须包含对应的参数名，如果不存在将抛出异常。
* defaultValue：默认参数名，设置该参数时，自动将required设为false。极少情况需要使用该参数，也不推荐使用该参数。

当发送请求时，请求参数中必须要包含userId这个参数，当不包含这个参数，请求将找不到这个映射。当属性required=true时，不包含这个参数将会抛异常，如果不能确定是否需要这个参数是我们可以写成，@RequestParam(value = "userId", **required = false**) 。

* 直接将属性映射到对象中

@RequestMapping("/user/find2")

**public** String find2UserById(User user,ModelMap modelMap){

modelMap.put("user", user);

**return** "/user";

}

Spring MVC按：

**“HTTP请求参数名 = 命令/表单对象的属性名”**

的规则，自动绑定请求数据，支持“级联属性名”，自动进行基本类型数据转换。

如：发起下面这个请求，springMVC会自动将id、name、password属性的值填充到user对象中。

<http://localhost:8080/springParam/user/save.html?id=12&name=marcle&password=123>

* SpringMVC以rest技术向springMVC传递参数

通过 REST 风格体系架构，请求和响应都是基于资源表示的传输来构建的。资源是通过全局 ID 来标识的，这些 ID 一般使用的是一个统一资源标识符（URI）。客户端应用使用 HTTP 方法（如，GET、POST、PUT 或 DELETE）来操作一个或多个资源。通常，GET 是用于获取或列出一个或多个资源，POST 用于创建，PUT 用于更新或替换，而 DELETE 则用于删除资源。

例如，GET http://host/context/employees/12345 将获取 ID 为 12345 的员工的表示。这个响应表示可以是包含详细的员工信息的 XML 或 ATOM，或者是具有更好 UI 的 JSP/HTML 页面。您看到哪种表示方式取决于服务器端实现和您的客户端请求的 MIME 类型。

RESTful Web Service 是一个使用 HTTP 和 REST 原理实现的 Web Service。通常，一个 RESTful Web Service 将定义基本资源 URI、它所支持的表示/响应 MIME，以及它所支持的操作。

Spring 3.0之后引入了对rest风格的支持。我们看实例

@RequestMapping("/user/find/{id}")

**public** String rest(@PathVariable **int** id,ModelMap modelMap){

User user = **new** User();

user.setName("marcle");

user.setPassword("123");

user.setId(id);

modelMap.put("user", user);

**return** "/user";

}

这里需要注意的地方时@RequestMapping("/user/find/{id}")和@PathVariable **int** id名称必须一样，否则会出现异常。

* 简单介绍返回视图的方式
* ModelAndView 形式返回

@RequestMapping("/user/save2")

**public** ModelAndView save2(User user,ModelMap modelMap){

ModelAndView mav = **new** ModelAndView();

mav.setViewName("/user");

mav.addObject("user", user);

**return** mav;

}

ModelAndView就是对返回到页面的值和视图进行封装。

* 直接字符串的形式返回，如”return “/user””,再把属性通过modelMap进行封装，modelMap存储的值属于request范围内，如果要发送服务器端请求，springMVC非常方便，你只要这样写即可return ”direct:user”.

还有一种传递参数的方法，我放在springMVC中的rest技术介绍

下面我们看看springMVC返回的过程

# 五、springMVC标签介绍

SpringMVC简单没几个标签，用起来还是非常好用的，在使用springMVC中的标签之前需要向每个jsp的头部引入标签支持<%@ taglib prefix=*"form"* uri=*"http://www.springframework.org/tags/form"* %>

。

1. form标签

这个标签会生成一个HTML的form标签，同时为内部标签的绑定暴露了一个绑定路径，它把命令对象（command object）放在pageContext中，这样内部的标签就可以访问这个 对象，这个库中的其它标签都是这个标签的嵌套标签。

如我们有个user的域对象，包含id、name、password属性，我们将把它当作返回index.jsp表单控制器的对象，如下面的代码：

<form:form action=*"*${ctx}*/user/save.*${ext}*"* method=*"post"* commandName=*"user"*>

<table>

<tr>

<td>Id:</td>

<td><form:input path =*"id"* /></td>

</tr>

<tr>

<td>name:</td>

<td><form:input path =*"name"* /></td>

</tr>

<tr>

<td>password:</td>

<td><form:input path =*"password"* /></td>

</tr>

<tr>

<td colspan=*"2"*><input type =*"submit"* value=*"Save"* /></td>

</tr>

</table>

</form:form>

上述的id、name、password由页面控制器放置在pageContext中，即在内部控制器方法中需要做这样的声明：

@RequestMapping(value="/user/save",method=RequestMethod.*GET*)

**public** String forSave(@ModelAttribute User user){

**return** "/index";

}

后台控制器中必须绑定这个@ModelAttribute User user命令行对象，而form下面的属性需要于这个user中的属性对应起来，否则将会抛异常。标签经过解析后生成的代码如下：

<form id="user" action="[/springTag/user/save.html](view-source:http://localhost:8080/springTag/user/save.html)" method="post">

<table>

<tr>

<td>Id:</td>

<td><input id="id" name="id" type="text" value="0"/></td>

</tr>

<tr>

<td>name:</td>

<td><input id="name" name="name" type="text" value=""/></td>

</tr>

<tr>

<td>password:</td>

<td><input id="password" name="password" type="text" value=""/></td>

</tr>

<tr>

<td colspan="2"><input type ="submit" value="Save Changes" /></td>

</tr>

</table>

</form>

### input 标签

使用时如上面的表达式，<form:input path =*"id"* />解析后会变成<input id="name" name="name" type="text" value=""/>可见用spring标签比传统的html简洁很多。

### （3）checkbox 标签

这个标签解析之后会变成html’中的type为checkbox的input元素，我们假设我们的用户有很多的参考东西，如信息的订阅、爱好、格言等，即如下面的域模型：

public class Preferences {

private boolean receiveNewsletter;

private String[] interests;

private String favouriteWord;

public boolean isReceiveNewsletter() {

return receiveNewsletter;

}

public void setReceiveNewsletter(boolean receiveNewsletter) {

this.receiveNewsletter = receiveNewsletter;

}

public String[] getInterests() {

return interests;

}

public void setInterests(String[] interests) {

this.interests = interests;

}

public String getFavouriteWord() {

return favouriteWord;

}

public void setFavouriteWord(String favouriteWord) {

this.favouriteWord = favouriteWord;

}

}

我们的相应的jsp文件可以写成：

<form:form action=*"*${ctx}*/pre/save.*${ext}*"* method=*"post"* commandName=*"preferences"*>

<table>

<tr>

<td>Subscribe to newsletter?:</td>

<%-- Approach 1: Property is of type java.lang.Boolean --%>

<td><form:checkbox path=*"receiveNewsletter"*/></td>

<td>&nbsp;</td>

</tr>

<tr>

<td>Interests:</td>

<td>

<%-- Approach 2: Property is of an array or of type java.util.Collection --%>

Quidditch: <form:checkbox path=*"interests"* value=*"Quidditch"*/>

Herbology: <form:checkbox path=*"interests"* value=*"Herbology"*/>

Defence Against the Dark Arts: <form:checkbox path=*"interests"*

value=*"Defence Against the Dark Arts"*/>

</td>

<td>&nbsp;</td>

</tr>

<tr>

<td>Favourite Word:</td>

<td>

<%-- Approach 3: Property is of type java.lang.Object --%>

Magic: <form:checkbox path=*"favouriteWord"* value=*"Magic"*/>

</td>

<td>&nbsp;</td>

</tr>

<tr>

<td colspan=*"2"*>

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

</td>

</tr>

</table>

</form:form>

如果有多个供选择的，在后台我们以数组的形式存储。

### （4）radiobutton 标签

解析后会变成html元素中type为radio的input元素

如下面的情况：

<tr>

<td>Sex:</td>

<td>Male: <form:radiobutton path="sex" value="M"/> <br/>

Female: <form:radiobutton path="sex" value="F"/> </td>

<td>&nbsp;</td>

</tr>

（5）password标签

解析后变成html元素中type为password的input元素，即为密码框。

<tr>

<td>Password:</td>

<td>

<form:password path="password" />

</td>

</tr>

（6）select标签

这个标签对应于html元素中的下拉框，即为select元素。

<tr>

<td>Skills:</td>

<td><form:select path=*"skills"* items=*"*${skills}*"* /></td>

<td></td>

</tr>

（7）option标签

<form:select path="house">

<form:option value="Gryffindor"/>

<form:option value="Hufflepuff"/>

<form:option value="Ravenclaw"/>

<form:option value="Slytherin"/>

</form:select>

（8）options标签

<form:select path="country">

<form:option value="-" label="--Please Select"/>

<form:options items="${countryList}" itemValue="code" itemLabel="name"/>

</form:select>

（9）textarea标签

<td><form:textarea path="notes" rows="3" cols="20" /></td>

（10）hidden标签

<form:hidden path="house" />

（11）errors标签

<form:form>

<table>

<tr>

<td>First Name:</td>

<td><form:input path="firstName" /></td>

*<%-- Show errors for firstName field --%>*

<td><form:errors path="firstName" /></td>

</tr>

<tr>

<td>Last Name:</td>

<td><form:input path="lastName" /></td>

*<%-- Show errors for lastName field --%>*

<td><form:errors path="lastName" /></td>

</tr>

<tr>

<td colspan="3">

<input type="submit" value="Save Changes" />

</td>

</tr>

</table>

</form:form>

# 六、springMVC拦截器

和Struts2一样，Spring MVC也可以使用拦截器对请求进行拦截处理，用户可以自定义拦截器来实现特定的功能，自定义的拦截器必须实现HandlerInterceptor接口。这个接口中定义了三个方法：preHandle()、postHandle()、afterCompletion()。

下面对代码中的三个方法进行解释。

preHandle()：这个方法在业务处理器处理请求之前被调用，在该方法中对用户请求request进行处理。如果程序员决定该拦截器对请求进行拦截处理后还要调用其他的拦截器，或者是业务处理器去进行处理，则返回true；如果程序员决定不需要再调用其他的组件去处理请求，则返回false。

postHandle()：这个方法在业务处理器处理完请求后，但是DispatcherServlet向客户端返回请求前被调用，在该方法中对用户请求request进行处理。

afterCompletion()：这个方法在DispatcherServlet完全处理完请求后被调用，可以在该方法中进行一些资源清理的操作。

下面通过一个例子来说明如何使用Spring MVC框架的拦截器。

要求编写一个拦截器，拦截所有不在工作时间的请求，把这些请求转发到一个特定的静态页面，而不对它们的请求进行处理。

首先编写TimeInterceptor.Java，代码如下：

**public** **class** TimeInterceptor **extends** HandlerInterceptorAdapter {

**private** **int** openingTime; // openingTime 属性指定上班时间

**private** **int** closingTime; // closingTime属性指定下班时间

**private** String outsideOfficeHoursPage; // outsideOfficeHoursPage属性指定错误

**public** **void** setOpeningTime(**int** openingTime) {

**this**.openingTime = openingTime;

}

**public** **void** setClosingTime(**int** closingTime) {

**this**.closingTime = closingTime;

}

**public** **void** setOutsideOfficeHoursPage(String outsideOfficeHoursPage) {

**this**.outsideOfficeHoursPage = outsideOfficeHoursPage;

}

// 重写 preHandle()方法，在业务处理器处理请求之前对该请求进行拦截处理

**public** **boolean** preHandle(HttpServletRequest request,

HttpServletResponse response, Object handler) **throws** Exception {

Calendar cal = Calendar.*getInstance*();

**int** hour = cal.get(Calendar.*HOUR\_OF\_DAY*); // 获取当前时间

**if** (openingTime <= hour && hour < closingTime) { // 判断当前是否处于工作 时间段内

**return** **true**;

} **else** {

response.sendRedirect(outsideOfficeHoursPage); // 返回提示页面

**return** **false**;

}

}

}

可以看出，上面的代码重载了preHandle()方法，该方法在业务处理器处理请求之前被调用。在该方法中，首先获得当前的时间，判断其是否在 openingTime和closingTime之间，如果在，返回true，这样才会调用业务控制器去处理该请求；否则直接转向一个静态页面，返回 false，这样该请求就不会被处理。

下面是在dispatcherServlet-servlet.xml中对拦截器进行的配置，代码如下：

<mvc:interceptors>

<mvc:interceptor>

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

<bean class=*"com.pango.spring.interceptor.TimeInterceptor"*>

<property name=*"openingTime"* value=*"12"*></property>

<property name=*"closingTime"* value=*"24"*></property>

<property name=*"outsideOfficeHoursPage"* value=*"outTime.html"*></property>

</bean>

</mvc:interceptor>

</mvc:interceptors>

可以看出，上面代码用bean标签去定义TimeInterceptor，令其id为officeHoursInterceptor，并给它的3个 属性赋值。在urlMapping中通过<property name="interceptors">去指定officeHoursInterceptor为一个拦截器，读者可以在<list> 和</list>之间定义多个拦截器。

outsideOfficeHours.html的代码很简单，只是输出一句提示语。

运行程序，在浏览器中随便访问一个页面，如果请求的时间在9点～18点之间，则该请求可以被处理；否则，返回一句提示语，如图23-5所示。

|  |
| --- |
| [164739714](http://images.51cto.com/files/uploadimg/20090927/164739714.jpg) |
| （点击查看大图）图23-5  请求被拦截效果图 |

说 明：在第22章中介绍过控制反转是Spring框架的核心思想，即用一个接口去定义一些操作，在接口的实现类中去重写这些操作，然后在Spring的配置 文件中去把该接口的实现类注入到应有框架中，这样就可以通过调用接口去调用接口的实现类。本节讲的拦截器就体现了这种思想，即实现 HandlerInterceptorAdapter接口，重写preHandle()方法并在配置文件中实现TimeInterceptor的注入。这 样当框架调用HandlerInterceptorAdapter时，就可以调用到TimeInterceptor类的preHandle()方法。

# 七、spring3 MVC 类型转换

Servlet中的输入参数为都是string类型，而spring mvc通过data bind机制将这些string 类型的输入参数转换为相应的command object（根据view和controller之间传输数据的具体逻辑，也可称为model attributes, domain model objects）。在这个转换过程中，spring实际是先利用java.beans.PropertyEditor中的 setAdText方法来把string格式的输入转换为bean属性，亦可通过继承java.beans.PropertyEditorSupport来实现自定义的PropertyEditors。

自定义完毕propertyEditor后，有以下几种方式来注册自定义的customer propertyEditor. （我只实现了第二种转换方式，至于其它方法大家可以自己尝试）

* **直接将自定义的propertyEditor放到需要处理的java bean相同的目录下**

名称和java Bean相同但后面带Editor后缀。

例如需要转换的java bean 名为User,则在相同的包中存在UserEditor类可实现customer propertyEditor的自动注册。

* **利用@InitBinder来注册customer propertyEditor**

这个在之前的笔记中已经介绍过了，即在controller类中增加一个使用@InitBinder标注的方法，在其中注册customer Editor

Java代码  [收藏代码](javascript:void())

**public** **class** BaseController {

@InitBinder

**public** **void** initBinder(WebDataBinder binder) {

binder.registerCustomEditor(Date.**class**, **new** CustomDateEditor(**true**));

}

}

* **继承 WebBindingInitializer 接口来实现全局注册**   
  使用@InitBinder只能对特定的controller类生效，为注册一个全局的customer Editor，可以实现接口WebBindingInitializer 。

Java代码  [收藏代码](javascript:void())

**public** **class** CustomerBinding **implements** WebBindingInitializer {

**public** **void** initBinder(WebDataBinder binder, WebRequest request) {

// **TODO** Auto-generated method stub

SimpleDateFormat dateFormat = **new** SimpleDateFormat("yyyy-MM-dd");

dateFormat.setLenient(**false**);

binder.registerCustomEditor(Date.**class**, **new** CustomDateEditor(dateFormat, **false**));

}

}

并修改 spring-servlet xml配置文件

Xml代码  [收藏代码](javascript:void())

<bean

        class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">

        <property name="webBindingInitializer">

            <bean

                class="net.zhepu.web.customerBinding.CustomerBinding" />

        </property>

    </bean>

但这样一来就无法使用mvc:annotation-driven  了。

**使用conversion-service来注册自定义的converter**

DataBinder实现了PropertyEditorRegistry, TypeConverter这两个interface，而在spring mvc实际处理时，返回值都是return binder.convertIfNecessary(见HandlerMethodInvoker中的具体处理逻辑)。因此可以使用customer conversionService来实现自定义的类型转换。

Xml代码  [收藏代码](javascript:void())

<bean id=*"conversionService"*

class=*"org.springframework.format.support.FormattingConversionServiceFactoryBean"*>

<property name=*"converters"*>

<list>

<bean class=*"net.zhepu.web.customerBinding.CustomerConverter"* />

</list>

</property>

</bean>

需要修改spring-servlet xml配置文件中的annotation-driven,增加属性conversion-service指向新增的conversionService bean。

Xml代码  [收藏代码](javascript:void())

<mvc:annotation-driven validator="validator"

conversion-service="conversionService" />

**对于第二种方式实现如下**

Date类型编辑器

**public** **class** CustomDateEditor **extends** PropertyEditorSupport {

**private** **static** **final** Map<String, String> *dateMap*;

**static** {

*dateMap* = **new** HashMap<String, String>();

*dateMap*.put("yyyy-MM-dd", "\\d{4}-\\d{2}-\\d{2}");

*dateMap*.put("yyyy-MM-dd hh:mm:ss", "\\d{4}-\\d{2}-\\d{2} \\d{2}:\\d{2}:\\d{2}");

*dateMap*.put("yyyy年MM月dd日", "\\d{4}年\\d{2}月\\d{2}日");

}

**private** **final** **boolean** allowEmpty;

**public** CustomDateEditor(**boolean** allowEmpty) {

**this**.allowEmpty = allowEmpty;

}

@Override

**public** **void** setAsText(String text) **throws** IllegalArgumentException {

**if** (**this**.allowEmpty && !StringUtils.*hasText*(text)) {

// Treat empty String as null value.

setValue(**null**);

} **else** {

**try** {

**boolean** flag = **false**;

**for** (String dateFormatStr : *dateMap*.keySet()) {

**if** (text.matches(*dateMap*.get(dateFormatStr))) {

flag = **true**;

System.*out*.println(text);

DateFormat dateFormat = **new** SimpleDateFormat(dateFormatStr);

setValue(dateFormat.parse(text));

**break**;

}

}

**if** (!flag) {

//throw new IllegalArgumentException("Could not parse date: " + text);

}

} **catch** (ParseException ex) {

//throw new IllegalArgumentException("Could not parse date: " + ex.getMessage(), ex);

}

}

}

@Override

**public** String getAsText() {

Date value = (Date) getValue();

DateFormat dateFormat = **new** SimpleDateFormat("yyyy-MM-dd");

**return** value != **null** ? dateFormat.format(value) : "";

}

}

@InitBinder来注册customer propertyEditor

@InitBinder

**public** **void** initBinder(WebDataBinder binder) {

binder.registerCustomEditor(Date.**class**, **new** CustomDateEditor(**true**));

}

**最后讲讲对于requestBody或httpEntity中数据的类型转换**

Spring MVC中对于requestBody中发送的数据转换不是通过databind来实现，而是使用HttpMessageConverter来实现具体的类型转换。

例如，之前提到的json格式的输入，在将json格式的输入转换为具体的model的过程中，spring mvc首先找出request header中的contenttype，再遍历当前所注册的所有的HttpMessageConverter子类， 根据子类中的canRead()方法来决定调用哪个具体的子类来实现对requestBody中的数据的解析。如果当前所注册的 httpMessageConverter中都无法解析对应contexttype类型，则抛出 HttpMediaTypeNotSupportedException （http 415错误）。

那么需要如何注册自定义的messageConverter呢，很不幸，在spring 3.0.5中如果使用annotation-driven的配置方式的话，无法实现自定义的messageConverter的配置，必须老老实实的自己 定义AnnotationMethodHandlerAdapter的bean定义，再设置其messageConverters以注册自定义的 messageConverter。

Xml代码  [收藏代码](javascript:void())

<mvc:annotation-driven>

    <mvc:message-converters>

        <bean class="org.springframework.http.converter.StringHttpMessageConverter"/>

        <bean class="org.springframework.http.converter.ResourceHttpMessageConverter"/>

        <bean class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter"/>

    </mvc:message-converters>

</mvc:annotation-driven>

# 八、json格式数据的输入和输出

Spring mvc处理json需要使用jackson的类库，因此为支持json格式的输入输出需要先修改pom.xml增加jackson包的引用

Xml代码  [收藏代码](javascript:void())

<dependency>

<groupId>org.codehaus.jackson</groupId>

<artifactId>jackson-core-lgpl</artifactId>

<version>1.8.0</version>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>org.codehaus.jackson</groupId>

<artifactId>jackson-mapper-asl</artifactId>

<version>1.8.0</version>

</dependency>

在spring-servlet.xml中必须加入这段代码：<mvc:annotation-driven />

根据前面的分析，在spring mvc中解析输入为json格式的数据有两种方式   
1:使用@RequestBody来设置输入

Java代码  [收藏代码](javascript:void())

@RequestMapping("/json1")

@ResponseBody

**public** JsonResult testJson1(@RequestBody User u){

log.info("get json input from request body annotation");

log.info(u.getUserName());

**return** **new** JsonResult(**true**,"return ok");

}

2:使用HttpEntity来实现输入绑定

Java代码  [收藏代码](javascript:void())

  @RequestMapping("/json2")

**public** ResponseEntity<JsonResult> testJson2(HttpEntity<User> u){

log.info("get json input from HttpEntity annotation");

log.info(u.getBody().getUserName());

ResponseEntity<JsonResult> responseResult = **new** ResponseEntity<JsonResult>( **new** JsonResult(**true**,"return ok"),HttpStatus.*OK*);

**return** responseResult;

对应Json格式的输出也对应有两种方式

1：使用@responseBody来设置输出内容为context body

@RequestMapping(value="/kfc/brands/{name}", method = RequestMethod.*GET*)

**public** @ResponseBody List<Shop> getShopInJSON(@PathVariable String name) {

List<Shop> shops = **new** ArrayList<Shop>();

Shop shop = **new** Shop();

shop.setName(name);

shop.setStaffName(**new** String[]{"mkyong1", "mkyong2"});

shops.add(shop);

Shop shop2 = **new** Shop();

shop2.setName(name);

shop2.setStaffName(**new** String[]{"mktong1", "mktong2"});

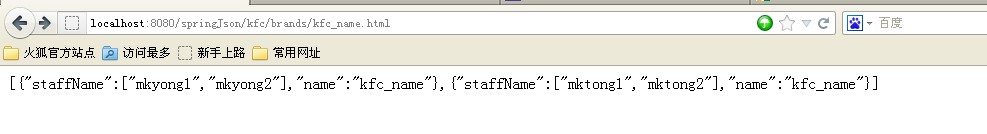
shops.add(shop2);

**return** shops;

}

当我们在地址栏中输入：<http://localhost:8080/springJson/kfc/brands/kfc_name.html>

服务器端会返回给我们jason格式的数据，这样我们就可以省去手工繁琐的组并了



2：返回值设置为ResponseEntity<?>类型，以返回context body

@RequestMapping("/json2")

**public** ResponseEntity<JsonResult> testJson2(HttpEntity<User> u){

log.info("get json input from HttpEntity annotation");

log.info(u.getBody().getUserName());

ResponseEntity<JsonResult> responseResult = **new** ResponseEntity<JsonResult>( **new** JsonResult(**true**,"return ok"),HttpStatus.*OK*);

**return** responseResult;

}

# 九、spring3mvc文件上传

Spring mvc使用jakarta的commons fileupload来支持文件上传，因此我们需要在pom.xml中导入所依赖的两个包：

<dependency>

<groupId>commons-io</groupId>

<artifactId>commons-io</artifactId>

<version>2.0.1</version>

</dependency>

<dependency>

<groupId>commons-fileupload</groupId>

<artifactId>commons-fileupload</artifactId>

<version>1.2.2</version>

</dependency>

在spring-servlet.xml中加入以下这段代码：

<bean id=*"multipartResolver"*

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

<!-- one of the properties available; the maximum file size in bytes -->

<property name=*"maxUploadSize"* value=*"100000"* />

</bean>

其中的property中可以限制最大和最小文件上传。

在客户端的代码如下：

<form method=*"post"* action=*"*${ctx}*/user/upload.*${ext}*"* enctype=*"multipart/form-data"*>

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

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

<input type=*"submit"*/>

</form>

服务器端的代码如下：

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

**public** String handleFormUpload(@RequestParam("name") String name,

@RequestParam("file") MultipartFile file, HttpServletRequest request)

**throws** IOException {

String filePath = request.~~getRealPath~~("/");

**if** (!file.isEmpty()) {

String fileName = file.getOriginalFilename();

System.*out*.println(filePath + "/" + fileName);

**byte**[] bytes = file.getBytes();

FileOutputStream output = **new** FileOutputStream(**new** File(filePath

+ fileName));

output.write(bytes);

output.close();

**return** "redirect:/success.jsp";

} **else** {

**return** "redirect:/failure.jsp";

}

}

# 十、spring mvc国际化和本地化

何为国际化，简单来说就是在那个国家显示哪个国家的语言，在计算机中，国际化和本地化意味着计算机软件要适应不同的语言和地区的差异。国际化就是设计为了适应不同地区和语言的差异而工程不需要做任何改动。

这一节的目的就是在springMVC中增加国际化和本地化的应用，我们将在这一节实现三种语言可以相互切换的国际化和本地化。

（1）我们在resources下面添加三个property文件，分别为：messages\_de.properties、messages\_en.properties、messages\_zh.properties，文件的命名规则：messages\_语言.properties

三个文件的内容如下：

* messages\_de.properties

label.firstname=Vorname

label.lastname=Familiename

label.email=Email

label.telephone=Telefon

label.addcontact=Addieren Kontakt

label.title=spring mvc Internationalization (i18n) / Localization

* messages\_en.properties

label.firstname=First Name

label.lastname=Last Name

label.email=Email

label.telephone=Telephone

label.addcontact=Add Contact

label.title=spring mvc Internationalization (i18n) / Localization

* messages\_zh.properties（经过转换后的中文）

label.firstname=\u59D3

label.lastname=\u540D\u5B57

label.email=\u7535\u5B50\u90AE\u4EF6

label.telephone=\u7535\u8BDD

label.addcontact=\u8054\u7CFB\u65B9\u5F0F

label.title=spring mvc \u56FD\u9645\u5316\u548C\u672C\u5730\u5316\u652F\u6301

（2）spring-servet.xml文件的配置

<!-- 为了使用国际化信息源，Spring MVC 必须实现MessageSource接口。当然框架内部有许多内置的实现类。我们需要做的是注册一个MessageSource类型的Bean。Bean 的名称必须为messageSource，从而方便DispatcherServlet自动检测它。每个DispatcherServlet只能注册一个 信息源-->

<bean id=*"messageSource"*

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

<property name=*"basename"* value=*"classpath:messages"* />

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

</bean>

<!—session 解析区域 -->

<bean id=*"localeResolver"*

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

<!-- property name="defaultLocale" value="en"/> -->

</bean>

<!-- 修改用户的区域需要调用区域修改拦截器 LocaleChangeInterceptor。如下所设定设定paramName属性来设定拦截请求中的特定 参数（这里是language）确定区域。既然是拦截器那就需要注册到拦截器 Bean 中，这里是注册到了DefaultAnnotationHandlerMapping Bean中 -->

<bean id=*"localeChangeInterceptor"*

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

<property name=*"paramName"* value=*"lang"* />

</bean>

<!--

<bean id="localeResolver"

class="org.springframework.web.servlet.i18n.CookieLocaleResolver">

<property name="defaultLocale" value="en" />

</bean>

-->

<bean id=*"handlerMapping"*

class=*"org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping"*>

<property name=*"interceptors"*>

<ref bean=*"localeChangeInterceptor"* />

</property>

</bean>

（3）在jsp目录下面创建一个contact.jsp文件

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

pageEncoding=*"UTF-8"*%>

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

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

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

<c:set var=*"ctx"* value=*"*${pageContext.request.contextPath}*"* />

<c:set var=*"ext"* value=*"html"* />

<html>

<head>

<title>Spring 3 MVC Series - Contact Manager</title>

</head>

<body>

<h3><spring:message code=*"label.title"*/></h3>

<span style="float: *right*">

<a href=*"*${ctx}*/language.*${ext}*?local=en"*>英文</a>

|

<a href=*"*${ctx}*/language.*${ext}*?local=de"*>德文</a>

<a href=*"*${ctx}*/language.*${ext}*?local=zh"*>中文</a>

</span>

<form:form method=*"post"* action=*"addContact.html"* commandName=*"contact"*>

<table>

<tr>

<td><form:label path=*"firstname"*><spring:message code=*"label.firstname"*/></form:label></td>

<td><form:input path=*"firstname"* /></td>

</tr>

<tr>

<td><form:label path=*"lastname"*><spring:message code=*"label.lastname"*/></form:label></td>

<td><form:input path=*"lastname"* /></td>

</tr>

<tr>

<td><form:label path=*"lastname"*><spring:message code=*"label.email"*/></form:label></td>

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

</tr>

<tr>

<td><form:label path=*"lastname"*><spring:message code=*"label.telephone"*/></form:label></td>

<td><form:input path=*"telephone"* /></td>

</tr>

<tr>

<td colspan=*"2"*>

<input type=*"submit"* value=*"*<spring:message code=*"label.addcontact"*/>*"*/>

</td>

</tr>

</table>

</form:form>

</body>

</html>

其中<spring:message>标签结合 ResourceBundleMessageSource 的功能，在网页上显示 messages.properties 中的文字讯息。

（4）创建LanguageController

@Controller

**public** **class** LanguageController {

@Autowired

**private** SessionLocaleResolver localeResolver;

@RequestMapping("/forLanguage")

**public** String forLanguage(@ModelAttribute Contact contact){

**return** "/contact";

}

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

**public** ModelAndView changeLocal(@ModelAttribute Contact contact,HttpServletRequest request,@RequestParam String local,HttpServletResponse response){

**if**("zh".equals(local)){

localeResolver.setLocale(request, response, Locale.*CHINA*);

}**else** **if**("en".equals(local)) {

localeResolver.setLocale(request, response, Locale.*ENGLISH*);

}**else** **if**("de".equals(local)){

localeResolver.setLocale(request, response, Locale.*GERMAN*);

}

**return** **new** ModelAndView("/contact");

}

}

其中红色部分就是对语言的设置

效果如下图：





# 十一、使用jsr303进行验证

JSR 303 – Bean Validation 是一个数据验证的规范，2009 年 11 月确定最终方案。2009 年 12 月 Java EE 6 发布，Bean Validation 作为一个重要特性被包含其中，Spring MVC在使用了<mvc:annotation-driven> 后，如果路径中有jsr 303的实现，将自动提供对jsr 303验证方式的支持。

* 引入hibernate-validator，hibernate-validator对jsr 303做了实现

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-validator</artifactId>

<version>4.2.0.Final</version>

</dependency>

* 新增一个pojo bean ,增加jsr 303格式的验证annotation

**public** **class** Contact {

**private** Long id = 0L;

@Size(min = 1)

**private** String firstName;

@NotNull

@Size(min = 1)

**private** String lastName;

@Past

**private** Date dateOfBirth;

**private** **boolean** married;

@Min(0)

@Max(20)

**private** **int** children;

}

* 在controller 类中的handler method中，对需要验证的对象前增加@Valid 标志

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

**public** String addContact(@ModelAttribute @Valid Contact contact,BindingResult result){

**if**(result.hasErrors()){

**return** "/contact";

}

**return** "/contact";

}

* 在jsp页面中添加contact.jsp

<form:form action=*"*${ctx }*/contact/add.*${ext}*"* method=*"post"*

commandName=*"contact"*>

<table border=*"1"*>

<tr>

<th>&nbsp;</th>

<th><spring:message code=*"editcontact.heading"* /></th>

</tr>

<tr>

<td bgcolor=*"cyan"*><spring:message

code=*"editcontact.label.firstname"* /></td>

<td><form:input path=*"firstName"* size=*"40"* /><font

color=*"#FF0000"*><form:errors path=*"firstName\*"* /></font></td>

</tr>

<tr>

<td bgcolor=*"cyan"*><spring:message

code=*"editcontact.label.lastname"* /></td>

<td><form:input path=*"lastName"* size=*"40"* /><font

color=*"#FF0000"*><form:errors path=*"lastName\*"* /></font></td>

</tr>

<tr>

<td bgcolor=*"cyan"*><spring:message code=*"editcontact.label.dob"* /></td>

<td><form:input path=*"dateOfBirth"* size=*"40"* /><font

color=*"#FF0000"*><form:errors path=*"dateOfBirth\*"* /></font></td>

</tr>

<tr>

<td bgcolor=*"cyan"*><spring:message

code=*"editcontact.label.married"* /></td>

<td><form:checkbox path=*"married"* /><font color=*"#FF0000"*><form:errors

path=*"married"* /></font></td>

</tr>

<tr>

<td bgcolor=*"cyan"*><spring:message

code=*"editcontact.label.children"* /></td>

<td><form:input path=*"children"* size=*"5"* /><font

color=*"#FF0000"*><form:errors path=*"children\*"* /></font></td>

</tr>

<tr>

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

value=*"*<spring:message code=*"editcontact.button.save"*/>*"* /></td>

<td><input type=*"reset"*

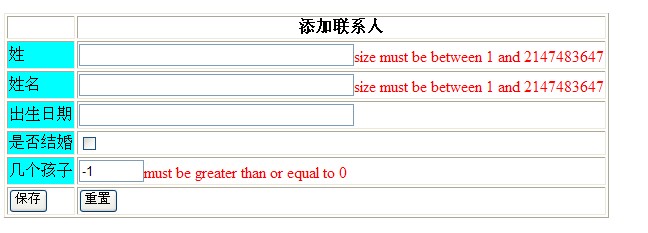
value=*"*<spring:message code=*"editcontact.button.reset"*/>*"* /></td>

</tr>

</table>

</form:form>

* 结果



使用jsr 303非常简单吧，有些人就问了，可以不可以自定义错误信息，当然是可以的，下面我就通过自定义错误来实现对contact的校验。

@Size(min = 1, message = "Contact first name is required.")

**private** String firstName;

@NotNull(message = "Contact cannot be left empty.")

@Size(min = 1, message = "Contact last name is required.")

**private** String lastName;

@Past(message = "Contact date of birth must be a date in the past.")

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

**private** Date dateOfBirth;

**private** **boolean** married;

@Min(value = 0, message = "A contact cannot have fewer than 0 children")

@Max(value = 20, message = "A contact cannot have more than 20 children")

**private** **int** children;

结果



只要将错误信息写到注解后面的message中即可，简单吧，我们再来看看jsr 303 主要的注解有哪些？

**表 1. Bean Validation 中内置的 constraint**

|  |  |
| --- | --- |
| **注 解** | **功能说明** |
| **@Null** | **被注释的元素必须为 null** |
| **@NotNull** | **被注释的元素必须不为 null** |
| **@AssertTrue** | **被注释的元素必须为 true** |
| **@AssertFalse** | **被注释的元素必须为 false** |
| **@Min(value)** | **被注释的元素必须是一个数字，其值必须大于等于指定的最小值** |
| **@Max(value)** | **被注释的元素必须是一个数字，其值必须小于等于指定的最大值** |
| **@DecimalMin(value)** | **被注释的元素必须是一个数字，其值必须大于等于指定的最小值** |
| **@DecimalMax(value)** | **被注释的元素必须是一个数字，其值必须小于等于指定的最大值** |
| **@Size(max, min)** | **被注释的元素的大小必须在指定的范围内** |
| **@Digits (integer, fraction)** | **被注释的元素必须是一个数字，其值必须在可接受的范围内** |
| **@Past** | **被注释的元素必须是一个过去的日期** |
| **@Future** | **被注释的元素必须是一个将来的日期** |
| @Pattern(value) | 被注释的元素必须符合指定的正则表达式 |

**表 2. Hibernate Validator 附加的 constraint**

|  |
| --- |
| @Email 被注释的元素必须是电子邮箱地址 |
| @Length 被注释的字符串的大小必须在指定的范围内 |
| @NotEmpty 被注释的字符串的必须非空 |
| @Range 被注释的元素必须在合适的范围内 |

有人写就问了，那么可以不可以自定义注释类型呢？答案当然是可以的。

* 自定义jsr303注释类型

（1）@Age是一个定制化的 constraint，由两个内置的 constraint 组合而成。代码如下

@Max(130)

@Min(1)

@Constraint(validatedBy = {})

@Documented

@Target( { ElementType.*ANNOTATION\_TYPE*, ElementType.*METHOD*, ElementType.*FIELD* })

@Retention(RetentionPolicy.*RUNTIME*)

**public** **@interface** Age {

String message() **default** "年龄填写不正确";

Class<?>[] groups() **default** {};

Class<? **extends** Payload>[] payload() **default** {};

}

（2）status是一个重新写得注释类型

**@Status 的 annotation 部分**

@Constraint(validatedBy = {StatusValidator.**class**})

@Documented

@Target( { ElementType.*ANNOTATION\_TYPE*, ElementType.*METHOD*, ElementType.*FIELD* })

@Retention(RetentionPolicy.*RUNTIME*)

**public** **@interface** Status {

String message() **default** "状态选择不正确";

Class<?>[] groups() **default** {};

Class<? **extends** Payload>[] payload() **default** {};

}

**@Status 的 constraint validator 部分**

**public** **class** StatusValidator **implements** ConstraintValidator<Status, Integer> {

**private** **final** **int**[] ALL\_STATUS = {1, 2, 3};

**public** **void** initialize(Status arg0) {

// **TODO** Auto-generated method stub

}

**public** **boolean** isValid(Integer value, ConstraintValidatorContext arg1) {

**if**(Arrays.*asList*(ALL\_STATUS).contains(value)) {

**return** **true**;

} **else**{

**return** **false**;

}

}

}

如果大家对于注释怎么写不够了解，请参考其它相关文档。

在属性上增加age、status

@Age

**private** **int** age;

@Status

**private** **int** status;

运行结果如下

