目录

[1 配置DispatcherServlet 3](#_Toc521592263)

[1.1 配置DispatcherServlet，截获特定的URL请求 3](#_Toc521592264)

[1.2 业务层和持久层的Spring配置 4](#_Toc521592265)

[1.3 配置Web层Spring容器 4](#_Toc521592266)

[1.4 DispatcherServlet内部逻辑 9](#_Toc521592267)

[2 控制器 10](#_Toc521592268)

[2.1 通过Reqest RUL进行映射 12](#_Toc521592269)

[2.2 请求处理方法签名 14](#_Toc521592270)

[2.2.1 使用@RequestParam绑定请求参数值 14](#_Toc521592271)

[2.2.2 使用@CookieValue绑定请求中Cookie值 15](#_Toc521592272)

[2.2.3 使用@RequestHeader绑定请求报文头的属性值 16](#_Toc521592273)

[2.2.4 使用命令/表单对象绑定请求参数值 16](#_Toc521592274)

[2.2.5 使用Servlet API 对象作为方法入参 18](#_Toc521592275)

[2.3 处理模型数据 21](#_Toc521592276)

[2.3.1 ModelAndView 21](#_Toc521592277)

[2.3.2 @ModeAttribute 22](#_Toc521592278)

[2.3.3 @SessionAttributes 24](#_Toc521592279)

[2.3.4 Map及Model 25](#_Toc521592280)

[2.3.5 逻辑视图 26](#_Toc521592281)

[3 文件上传 27](#_Toc521592282)

[4 输出 JSON 32](#_Toc521592283)

[5 自定义的数据类型转换 35](#_Toc521592284)

[5.1 Converter接口 35](#_Toc521592285)

[5.2 配置自定义的类型转换器 36](#_Toc521592286)

[6 配置过滤器 37](#_Toc521592287)

[6.1 实现一个过滤器 37](#_Toc521592288)

[6.2 Spring配置 39](#_Toc521592289)

[6.3 Web.xml配置 39](#_Toc521592290)

# 配置DispatcherServlet

DispatcherServlet是SpringMVC的灵魂与心脏，它负责接收HTTP请求并协助SpringMVC的各个组件完成请求处理工作。和任何Servlet一样，用户必须在web.xml中配置好DispatcherServlet。

## 配置DispatcherServlet，截获特定的URL请求

Web.xml:

|  |
| --- |
| <servlet>  <servlet-name>springmvc</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>/WEB-INF/webServletContext.xml</param-value>  </init-param>  </servlet>  <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <!—使用htm\html为后缀在Spring3.2中，将导致不能正确返回JSON,在3.1中可以。原因是Spring根据请求中URL含有html(htm)，认为请求的MIME类型为text/html而不是application/json. -->  <url-pattern>\*.htm</url-pattern>  </servlet-mapping> |

在<servlet-mapping>中指定其处理的URL。假设希望SpringMVC中的DispatcherServlet能截获并处理以.htm结束的URL请求，那么可以按以上方式进行配置。

## 业务层和持久层的Spring配置

Web.xml:

|  |
| --- |
| <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>classpath:applicationContext.xml</param-value>  </context-param>  <listener>  <listener-class>  org.springframework.web.context.ContextLoaderListener  </listener-class>  </listener> |

通过contextConfigLocation参数指定业务层Spring容器的配置文件（多个配置文件使用逗号分隔），ContextLoaderListener是一个ServletContextListener，它通过contextConfigLocation参数所指定的Spring配置文件启动“业务层”的Spring容器。

## 配置Web层Spring容器

Spring MVC配置：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:mvc=*"http://www.springframework.org/schema/mvc"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context-3.2.xsd*  *http://www.springframework.org/schema/mvc*  *http://www.springframework.org/schema/mvc/spring-mvc-3.2.xsd"*>  <!-- spring 3.0 的功能，使用MVC注解 -->  <mvc:annotation-driven/>  <!-- 扫描Spring MVC Controller 所在的包，以应用MVC注解。 -->  <context:component-scan base-package=*"com.profound.blog.controller"*/>  <!-- 配置SpringMVC视图解析器，将ModelAndView及字符串解析为具体的页面 -->  <bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*  p:viewClass=*"org.springframework.web.servlet.view.JstlView"*  p:suffix=*".jsp"*/>  <!-- 以JSON形式返回结果 -->  <bean class=*"org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter"*>  <property name=*"cacheSeconds"* value=*"0"*></property>  <property name=*"messageConverters"*>  <list>  <!—springVMC3 bean class=*"org.springframework.http.converter.json.MappingJacksonHttpMessageConverter"--!*>  <!—mvc 4 -->  <bean class=*"org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"*>  </bean>  </list>  </property>  </bean>  <!-- 文件上传 -->  <bean id=*"multipartResolver"* class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*  p:maxUploadSize=*"1024000000"*  p:uploadTempDir=*"/temp/"*  p:maxInMemorySize=*"10240"*  p:defaultEncoding=*"utf-8"*/>  </beans> |

DispatcherServlet处理所有被截获的请求。Web容器将做为“业务层”容器的子容器，即“Web层容器”可以访问“业务层”容器的Bean，而“业务层”容器不可以访问“Web层容器”的Bean。

“Web层容器”要DispatcherServlet来启动。遵循“契约优于配置”的原则。配置文件名为<servlet><servlet-name>springmvc</servlet-name></servlet>中，servlet-name的内容加上“-servlet.xml”如: springmvc-servlet.xml。并且保存在/WEB-INF/目录下。

Web.xml配置如下：

|  |
| --- |
| <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>\*.htm</url-pattern>  </servlet-mapping> |

如果要对DispatcherServlet的默认规则进行调整也可以。通过<servlet>的<init-param>指定。“contextConfigLocation”用来指定配置文件的位置。

如示例1：

|  |
| --- |
| <servlet>  <servlet-name>springmvc</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>/WEB-INF/webServletContext.xml</param-value>  </init-param>  </servlet>  <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <url-pattern>\*.htm</url-pattern>  </servlet-mapping> |

或如示例2：

|  |
| --- |
| <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:webServletContext.xml</param-value>  </init-param>  </servlet>  <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <url-pattern>\*.htm</url-pattern>  </servlet-mapping> |

可以在<servlet><load-on-startup>1</load-on-startup></servlet>中，将DispatcherServlet设置为第1个启动。

## DispatcherServlet内部逻辑

Spring如何将上下文中的SpringMVC组件装配到DispatchServlet？通过查看DispatcherServlet的initStrategies()代码，其中做了如下工作。

|  |
| --- |
| **protectedvoid** initStrategies(ApplicationContext context) {  initMultipartResolver(context);//初始化上传文件解析器  initLocaleResolver(context);//初始化本地化解析器  initThemeResolver(context);’//初始化主题解析器  initHandlerMappings(context);//初始化处理器解析器  initHandlerAdapters(context);//初始化处理器适配器  initHandlerExceptionResolvers(context);//初始化处理器异常解析器  initRequestToViewNameTranslator(context);//初始化请求到视图名翻译器  initViewResolvers(context);//初始化视图解析器  initFlashMapManager(context);//  } |

initStrategies()方法将在WebApplicationContext初始化后自动执行，此时Spring上下文中的Bean已经初始化完毕。该方法的工作是通过反映机制查找并装配Spring容器中用户显示自定义的组件Bean，如果找不到再装配默认的组件实例。

# 控制器

在SpringMVC中，任何一个使用了@Controller注解的类都可以成为控制器，接收RUL请求，并给予处理。

示例：

|  |
| --- |
| **package** com.profound.blog.controller;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpSession;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RequestMethod;  **import** com.profound.blog.Const;  **import** com.profound.blog.domain.RegUser;  **import** com.profound.blog.service.RegUserService;  @Controller  @RequestMapping("/user")  **public class** UserController {    @Autowired  **private** RegUserService userService;  @RequestMapping("/invalide")  **public** String invalide(HttpServletRequest req){  req.getSession().invalidate();  **return**"/Index";  }    @RequestMapping(value="/reg",  method={RequestMethod.*POST*,RequestMethod.*GET*})  **public** String reg(RegUser user, String valiCode, HttpServletRequest req){  String result = "RegOk";  HttpSession session = req.getSession();  String sessCode = (String)session.getAttribute(Const.*VALIDATE\_CODE*);  **if**(sessCode != **null**&& sessCode.equals(valiCode)){  RegUser u = **this**.userService.regUser(user);  session.setAttribute(Const.*USER\_INFO*, u);  }**else**{  result= "RegDetails";  req.setAttribute("msg", "验证码不正确");  }    **return** result;  }  } |

SpringMVC 通过@Controller注解即可将一个POJO转化为处理请求的控制器，通过@RequestMapping为控制器指定要处理哪些URL。示例中UserController中一个负责用户处理的控制器。

## 通过Reqest RUL进行映射

@RequestMapping使用value值指定请求的URL。（注：如果仅为value赋值，可以不写”value=”，直接写值。）

@RequestMapping可以注解在方法上，也可以注解在类上。当注解在类上时如示例所示，所请求的URL应是”类上的URL”/”方法的URL”。如用户注销的请求为：/user/ invalide.htm。

如类上没有@RequestMapping注解，即注解只有方法上。则请求的URL，只包括”方法URL”。如用户注销的请求为： / invalide.htm。

@RequestMapping不但支持标准的URL，还支持Ant风格（即“?”、“\*”和“\*\*”的字符）和带有“{xxx}”占位符的URL。

Ant风格匹配符：

|  |  |
| --- | --- |
| **字符** | **含义** |
| **？** | 匹配一个字符 |
| **\*** | 匹配任意个字符 |
| **\*\*** | 匹配多层路径 |

示例：

|  |  |
| --- | --- |
| **URL** | **匹配的RUL** |
| **/user/\*/createUser** | /user/aaa/createUser、/user/bbbb/createUser |
| **/user/\*\*/createUser** | /user/createUser、/user/bbb/creatUser、/user/aaa/bb/createUser |
| **/user/createUser??** | /user/createUseraa、/user/createUserxf |
| **/user/{userId}** | /user/123、/user/456 |
| **/user/\*\*/{userId}** | /user/aa/bbb/123、/user/456、/user/aa/456 |
| **/company/{companyId}/user/{userId}/detail** | /company/123/user/456/detail |

带占位符的URL是Spring3.0的新功能，该功能在SpringMVC向REST目标挺进的发展过程中具有里程碑的意义。通过@PathVariable可以将URL中的占位符参数绑定到控制器的方法入参中。示例如下：

|  |
| --- |
| @RequestMapping(value="/user/reg/{id}", method={RequestMethod.*POST*,RequestMethod.*GET*})  **public** String reg(RegUser user, String valiCode,  HttpServletRequest req,  @PathVariable(“id”) String id){  System.*out*.println("--------------------id: "+id);  String result = "RegOk";  HttpSession session = req.getSession();  String sessCode = (String)session.getAttribute(Const.*VALIDATE\_CODE*);  **if**(sessCode != **null**&& sessCode.equals(valiCode)){  RegUser u = **this**.userService.regUser(user);  session.setAttribute(Const.*USER\_INFO*, u);  }**else**{  result= "/user/RegDetails";  req.setAttribute("msg", "验证码不正确");  }    **return** result;  } |

请求的RUL为：

|  |
| --- |
| <form action="<%=basePath %>user/reg/<%="ccc" %>.htm" > |

注：

|  |
| --- |
| **默认情况下，Java类的反射对象并未包含方法入参的名称，如果编译时打开debug开关（javac –debug=no）,则方法入参名也会记录到类的反射对象中。所以最好在@PathVariable中显示指明绑定的参数名以避免因编译方式不同造成参数绑定失败的隐患。** |

@RequestMapping的value、method、params、headers分别表示请求URL、请求方法、请求参数、报文头的映射条件。它们可以联合使用。

Params和headers分别通过请求参数及报文头属性进行映射，它们支持简单的表达式，下面以params表达式为例说明，headers可以参照params进行理解。

|  |  |
| --- | --- |
| **Params值** | **说明** |
| **params=”pwd”** | 表示请求的参数中必须含有pwd参数 |
| **Params=”!pwd2”** | 表示请求的参数中不可以含有pwd2参数 |
| **Params=”pwd != 123”** | 表示请求参数pwd的值不可以等于给定的值123。 |
| **Params={“pwd=123”,”pwd2”}** | 请求参数中必须含有pwd、pwd2两个参数，并且pwd参数的值为给定的值123。 |

Headers示例：

|  |
| --- |
| **headers={"Host","Referer=http://localhost:8088/springMVCblog/user/RegDetails.jsp"}** |
| **表示：http请求头中必须有host\referer两个请求头。并且referer请求头的值为给定的值。** |

## 请求处理方法签名

### 使用@RequestParam绑定请求参数值

|  |
| --- |
| @RequestMapping(value="/user/reg/{id}/{di}",  method={RequestMethod.*POST*,RequestMethod.*GET*},  params="pwd2", headers={"Host"})  **public** String reg(RegUser user,  @RequestParam(“valiCode”) String valiCode,  HttpServletRequest req,  @PathVariable("id") String id,  @PathVariable("di") String dii) |

@RequestParam用来指定请求参数做为方法的入参。其有三个参数。

|  |  |
| --- | --- |
| **参数** | **说明** |
| **value** | 参数名(最好指定请求的参数名) |
| **required** | 是否必须，默认为true,表示请求中必须含有对应的参数名，如果不存在将抛出异常。 |
| **defaultValue** | 默认参数值。设置该参数时，自动将required设为false。 |

### 使用@CookieValue绑定请求中Cookie值

|  |
| --- |
| @RequestMapping(value="/user/reg/{id}/{di}",  method={RequestMethod.*POST*,RequestMethod.*GET*},  params="pwd2", headers={"Host"})  **public** String reg(RegUser user,  @RequestParam("valiCode") String valiCode,  HttpServletRequest req,  @PathVariable("id") String id,  @PathVariable("di") String dii,  @CookieValue(value="JSESSIONID",required=**false**)String sessionId,  @CookieValue(value="userName", required=false) String userName) |

使用@CookieValue可让处理方法入参绑定某个Cookie的值，它和@RequestParam拥有3个一样的参数。

### 使用@RequestHeader绑定请求报文头的属性值

请求报文包含了若干个报文头属性，服务器可以据此获取客户端的信息，通过@RequestHeader即可将报文头属性绑定到处理方法的入参中。

|  |
| --- |
| @RequestMapping(value="/user/reg/{id}/{di}",  method={RequestMethod.*POST*,RequestMethod.*GET*},  params="pwd2", headers={"Host"})  **public** String reg(RegUser user,  @RequestParam("valiCode") String valiCode,  HttpServletRequest req,  @PathVariable("id") String id,  @PathVariable("di") String dii,  @CookieValue(value="JSESSIONID", required=**false**) String sessionId,  @CookieValue(value="userName", required=**false**) String userName,  @RequestHeader(value="Referer", required=**false**) String referer,  HttpServletResponse response) |

使用@RequestHeader可让处理方法入参绑定某个Http请求头的值，它和@RequestParam拥有3个一样的参数。

### 使用命令/表单对象绑定请求参数值

所谓命令/表单对象并不需要实现任何接口，仅是一个拥有若干属性的POJO。Spring MVC 会按请求参数名和命令/表单对象属性名匹配的方式，自动为该对象填充属性值。

|  |
| --- |
| @RequestMapping(value="/user/reg/{id}/{di}",  method={RequestMethod.*POST*,RequestMethod.*GET*},  params="pwd2", headers={"Host"})  **public** String reg(RegUser user,  @RequestParam("valiCode") String valiCode,  HttpServletRequest req,  @PathVariable("id") String id,  @PathVariable("di") String dii,  @CookieValue(value="JSESSIONID", required=**false**) String sessionId,  @CookieValue(value="userName", required=**false**) String userName,  @RequestHeader(value="Referer", required=**false**) String referer,  HttpServletResponse response) |

HTML页面中，表单元素的name属性，为POJO类中定义的属性。并支持级联的属性名。

POJO对象定义：

|  |
| --- |
| **publicclass**RegUser**implements** Serializable{  **private** Long userID;  **private** String regName;  **private** String pwd;  **private** String email;  **private** String passwordAnswer;  **private** String passwordQuestion;  **private** String userName;  private Blog blog;  …  } |

表单元素的name属性:

|  |
| --- |
| <input type="text" name="regName" id="newName"/>  <input type="password" name="pwd" /> |

为级联属性赋值示例：<input type="text" name="blog.blogName" />

### 使用Servlet API 对象作为方法入参

在SpringMVC 中，控制器可以不依赖任何Servlet API，但是Spring MVC并不阻止我们使用Servlet API的类对象作为方法的入参。

示例1：Session为入参

|  |
| --- |
| @RequestMapping("/user/invalide")  **public** String invalide(**HttpSession session**){  session.invalidate();  **return**"/Index";  } |

示例2：Request为入参

|  |
| --- |
| @RequestMapping(value="/user/reg/{id}/{di}",  method={RequestMethod.*POST*,RequestMethod.*GET*},  params="pwd2", headers={"Host"})  **public** String reg(RegUser user,  @RequestParam("valiCode") String valiCode,  **HttpServletRequest req,**  @PathVariable("id") String id,  @PathVariable("di") String dii,  @CookieValue(value="JSESSIONID", required=**false**) String sessionId,  @CookieValue(value="userName", required=**false**) String userName,  @RequestHeader(value="Referer", required=**false**) String referer,  HttpServletResponse response){  System.*out*.println(referer);  System.*out*.println("---------userName in cookie:--------------" + userName);  Cookie k = **new** Cookie("userName", user.getRegName());  response.addCookie(k);  Cookie[] cs = req.getCookies();  **for**(Cookie c : cs){  System.*out*.println(c.getName() + ":" + c.getValue());  }  System.*out*.println(sessionId);  System.*out*.println(valiCode);  // referer = req.getHeader("Referer");  System.*out*.println("======referer: "+referer);    System.*out*.println("--------------------id: "+id);  System.*out*.println("--------------------di: "+dii);  String result = "RegOk";  **HttpSession session = req.getSession();**  String sessCode = (String)session.getAttribute(Const.*VALIDATE\_CODE*);  **if**(sessCode != **null**&& sessCode.equals(valiCode)){  RegUser u = **this**.userService.regUser(user);  session.setAttribute(Const.*USER\_INFO*, u);  }**else**{  result= "/user/RegDetails";  **req.setAttribute("msg", "验证码不正确");**  }    **return** result;  } |

示例3：Response为入参

|  |
| --- |
| @RequestMapping("/user/canuse")  **Public void** canUse(@RequestParam("regName")String regName, **HttpServletResponse response**) **throws** IOException{  **boolean** result = **this**.userService.namePWDCanUse(regName);  String msg = "ok";  **if** (!result){  msg = "nook";  }  **PrintWriter writer = response.getWriter();**  writer.print(msg);  writer.flush();  writer.close();  } |

示例4：使用IO对象作为入参1

|  |
| --- |
| @RequestMapping("/user/canuse2")  **Public void** canUse2(@RequestParam("regName")String regName, **PrintWriter writer**) **throws** IOException{  **boolean** result = **this**.userService.namePWDCanUse(regName);  String msg = "ok";  **if** (!result){  msg = "nook";  }  writer.print(msg);  writer.flush();  writer.close();  } |

示例5：使用IO作为入参2

|  |
| --- |
| @RequestMapping("/user/getimg")  **publicvoid** getImg(HttpServletRequest req, **OutputStream out**) **throws** IOException{  String root = req.getServletContext().getRealPath("/");  String file = root+"/img/logon.jpg";  **FileCopyUtils.*copy*(new FileInputStream(file), out);**  } |

## 处理模型数据

对于MVC框架来说模型数据是最重要的，因为控制（C）是为了产生模型数据（M），而视图（V）则是为了渲染模型数据。

如何将模型数据暴露给视图是Spring MVC框架的一项重要工作，Spring MVC提供了多种途径输出模型数据：

|  |  |
| --- | --- |
| **模型输出途径** | **说明** |
| **ModelAndView** | 处理方法返回类型为ModelAndView时，方法体即可通过该对象添加模型数据。（实际上，ModelAndView对象是对模型对象和视图对象的封装。） |
| **@ModelAttribute** | 方法入参标注该注解后，入参的对象会被加入到模型（Model）中。 |
| **Map及Model** | 入参是org.springframework.ui.Model、org.springframework.ui.ModelMap、java.util.Map时，处理方法返回时，Map中的数据会自动添加到数据模型中。 |
| **@SessionAttributes** | 将模型中的某个属性暂时存到HttpSession中。 |

### ModelAndView

控制器处理方法的返回结果为ModelAndView，则其即包含视图信息，也包含模型数据信息，这样SpringMVC就可以使用视图对模型数据进行渲染。可以简单地将模型对数据看成一个Map<String,Object>对象。

ModelAndView示例：

|  |
| --- |
| @RequestMapping("/userList")  **public** ModelAndView listUser(@RequestParam("page") **int** page,  @RequestParam("count") **int** length){  **ModelAndView mv= new ModelAndView();**  List<RegUser> list = **this**.userService.listUser(length, page);  **mv.setViewName("/manage/UserList");** // 指定视图对象。  **mv.addObject("list", list);**//添加模型对象。  **return** mv;  } |

### @ModeAttribute

如果希望将方法的入参添加到模型中，仅需要在相应的入参前使用@ModelAttribute注解即可。

使用@ModelAttribute示例1

|  |
| --- |
| @RequestMapping("/logon")  **public** String reg(@ModelAttribute(value="userInfo") RegUser userInfo,String regName, String pwd){  String result = "/Index";  userInfo = **this**.userService.logon(regName, pwd);    **if**(**null** == userInfo){  result = "/user/Logon";  }    **return** result;  } |

使用@ModelAttribute示例2：

在reg()方法中，入参userInfo前面加有注解@ModelAttribute(“userInfo”),这时会首先从Model中找名字为”userInfo”的对象。如果没有则容器会创建一个作为入参。然后才会将表单发送的参数设置入对象中。

方法getUserInfo()含有注解@ModelAttribute(“userInfo”),容器会从getUserInfo()中获取对象，以”userInfo”为名，加入Mode中。这一过程在调用处理方法之前完成。

|  |
| --- |
| @ModelAttribute("userInfo")  **public** RegUser getUserInfo(){  RegUser u = **new** RegUser();  u.setBirthday(**new** Date(System.*currentTimeMillis*()));  **return** u;  }    @RequestMapping("/logon")  public String reg(@ModelAttribute(value="userInfo") RegUser userInfo,String regName, String pwd,Model mo){  System.out.println("-------------------containsAttribute:userInfo:"+mo.containsAttribute("userInfo")); // 输出：为true.  String result = "/Index";  Date birthday = userInfo.getBirthday();  userInfo = this.userService.logon(regName, pwd);  userInfo.setBirthday(birthday);  if(null == userInfo){  result = "/user/Logon";  }    return result;  } |

### @SessionAttributes

当我们希望通过Session共享方法入参数据（即将数据放入Session中）。可以在**控制器类上**使用@SessionAttributes注解。SpringMVC会将**模型中**对应的属性存入到Session中。

注意：当Session中没有对应的属性时，Model不能从Session中获取指定的属性，这时将需要一个获取Model属性的方法。

|  |
| --- |
| @Controller  @SessionAttributes(value={"userInfo"})  **publicclass** UserController {    @RequestMapping("/user/getimg")  **publicvoid** getImg(HttpServletRequest req, OutputStream out, Model m) **throws** IOException{  String root = req.getServletContext().getRealPath("/");  // 修改了Model中与Session对应的属性，会被自动的更新到Session中。  RegUser u = (RegUser) m.asMap().get("userInfo");  u.setRegName("yyyy");  String file = root+"/img/logon.jpg";  FileCopyUtils.*copy*(**new** FileInputStream(file), out);  }    // 为Model提供属性值  @ModelAttribute("userInfo")  **public** RegUser getUserInfo(){  RegUser u = **new** RegUser();  u.setBirthday(**new** Date(System.*currentTimeMillis*()));  **return** u;  }    @RequestMapping("/logon")  **public** String reg(@ModelAttribute(value="userInfo") RegUser userInfo,String regName, String pwd,Model mo){  System.*out*.println("-------------------containsAttribute:userInfo:"+mo.containsAttribute("userInfo"));  String result = "/Index";  Date birthday = userInfo.getBirthday();  userInfo = **this**.userService.logon(regName, pwd);  userInfo.setBirthday(birthday);  **if**(**null** == userInfo){  result = "/user/Logon";  }    **return** result;  }      } |

### Map及Model

SpringMVC 在内部使用一个org.springframework.ui.Model接口存储模型数据。它的功能类似java.util.Map。org.springframework.ui.ModelMap实现了Map接口。而org.springframework.ui.ExtendedModelMap扩展了ModelMap同时实现了Model接口。

示例：

|  |
| --- |
| @RequestMapping("/userList2")  **public** String listUser2(@RequestParam("page") **int** page,  @RequestParam("count") **int** length,  ModelMap mm){  String result = "/manage/UserList";    List<RegUser> list = **this**.userService.listUser(length, page);    mm.addAttribute("list", list);//添加模型对象。    **return** result;  } |

### 逻辑视图

控制器方法返回的字符串的值会被当成逻辑视图名进行处理。但如果字符串带“forward:”或“redirect:”前缀时，Spring MVC将对它们进行特殊的处理。将“forward:”或“redirect:”当成指示符，其后的字符串作为URL处理。

示例：

|  |
| --- |
| @RequestMapping("/logon")  public String logon(@ModelAttribute(value="userInfo") RegUser userInfo,String regName, String pwd,Model mo){  System.out.println("-------------------containsAttribute:userInfo:"+mo.containsAttribute("userInfo"));  **String result = "redirect:/Index.jsp";**  Date birthday = userInfo.getBirthday();  userInfo = this.userService.logon(regName, pwd);  if(userInfo != null){  userInfo.setBirthday(birthday);  }    if(null == userInfo){  //result = "/user/Logon";  **result="forward:/logonFaild.htm";**  }    return result;  }    @RequestMapping**("/logonFaild")**  public String logonFaild(@ModelAttribute(value="userInfo") RegUser userInfo,String regName, String pwd,Model mo){  return "/user/NewFile";  } |

# 文件上传

SpringMVC为文件上传提供了直接的支持，这种支持是通过即插即用的MultipartResolver实现的。Sprig使用Jakarta Commons FileUpdate技术实现了一个MultipartResolver实现类：CommosMultipartResolver。

SpringMVC 上下文中默认没有装配MultipartResolver，因此默认情况下没有装配MultipartResolver。因此默认情况下不能处理文件的上传工作。如果想使用Spring上传功能，需要先在上下文中配置MultipartResolver。

|  |
| --- |
| <!-- 文件上传 -->  <bean id=*"multipartResolver"* class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*  p:maxUploadSize=*"1024000000"*  p:uploadTempDir=*"/temp/"*  p:maxInMemorySize=*"10240"*  p:defaultEncoding=*"utf-8"*  /> |

maxUploadSize:最大允许上传的文件的大小，单位为字节。

uploadTempDir:上传文件的临时目录

maxInMemorySize:最大使用内存，单位字节。

defaultEncoding:默认的请求编码格式，默认为ISO-9959-1。

示例：

|  |
| --- |
| <formaction="apply.htm" method="post" enctype="multipart/form-data">  <tablewidth="80%" border="1">  <tr>  <tdwidth="22%">博客名：</td>  <tdwidth="78%"><inputtype="text" name="blogName"></td>  </tr>  <tr>  <td>昵称：</td>  <td><inputtype="text" name="nickName"/></td>  </tr>  <tr>  <td>照片：</td>  <td><inputtype="file" name="photo"/></td>  </tr>  <tr>  <td>个人简介：</td>  <td><textareaname="description"></textarea></td>  </tr>  <tr>  <td>评价：</td>  <td><textareaname="affiche" ></textarea></td>  </tr>  <tr>  <tdcolspan="2"> ${msg } <inputtype="submit"/></td>  </tr>  </table>  </form> |

|  |
| --- |
| package com.profound.blog.controller;  import java.io.File;  import java.io.Reader;  import javax.servlet.http.HttpSession;  import org.apache.commons.fileupload.FileItem;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.context.annotation.Scope;  import org.springframework.stereotype.Controller;  import org.springframework.ui.ModelMap;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestParam;  import org.springframework.web.multipart.commons.CommonsMultipartFile;  import com.profound.blog.Const;  import com.profound.blog.domain.Blog;  import com.profound.blog.domain.RegUser;  import com.profound.blog.service.BlogService;  @Controller  @Scope("prototype")  public class BlogController {  @Autowired  private BlogService blogService;    @RequestMapping(value="/blog/apply")  public String apply(@RequestParam("blogName") String blogName,  @RequestParam("nickName") String nickName,  @RequestParam("description") String description,  @RequestParam("affiche") String affiche,  HttpSession session, ModelMap mm,  @RequestParam(required=false, value="photo") CommonsMultipartFile photo){// @RequestParam()注解是必须的，否则不能初始化CommonsMultipartFile对象。    // 无论成功还是失败都返回申请页面，给出信息提示。  String view = "/blog/ApplyBlog";  // 1、检查是否已经登录，只有登录后，才可进行后续的操作\将用户信息设置到blog对象中。  RegUser user = (RegUser) session.getAttribute(Const.USER\_INFO);  if(null != user){  try {  // 2、将上传的文件存储到指定的目录中，之后，将文件名设值到blog对象中，以便进行存储到数据库中。  String root = session.getServletContext().getRealPath("/");    String fileName = photo.getOriginalFilename();    File newFile = new File(root, fileName);    FileItem fileItem = photo.getFileItem();  fileItem.write(newFile);      // 3、将blog对象存入数据库  Blog blog = new Blog();  blog.setBlogName(blogName);  blog.setNickName(nickName);  blog.setRegUser(user);  blog.setPhoto(fileName);  user.setBlog(blog);  blog.setDescription(description);  blog.setAffiche(affiche);    this.blogService.apply(user);  mm.put("msg", "ok");    } catch (Exception e) {  e.printStackTrace();  }  }else{  view = "/user/Logon"; // 没在登录，转到登录页面。  }    return view;  }  } |

注：在上传处理方法中，将非文件表单域转换为Blog对象，没有成功，导致HTTP 400错误。

# 输出JSON-1(Spring 5)

引入Alibaba 的Fastjson 包，无需任何配置。在Controller 中使用@ResponseBody注解。

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>1.2.62</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-annotations</artifactId>

<version>2.10.0</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-core</artifactId>

<version>2.10.0</version>

</dependency>

<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind -->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.10.0</version>

</dependency>

# 输出 JSON-2（Spring3）

需使用：jackson包。

SpringMVC配置：

|  |
| --- |
| <!-- 以JSON形式返回结果 -->  <bean class=*"org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter"*>  <property name=*"cacheSeconds"* value=*"0"*></property>  <property name=*"messageConverters"*>  <list>  <bean class=*"org.springframework.http.converter.json.MappingJacksonHttpMessageConverter"*>  </bean>  </list>  </property>  </bean> |

控制器处理方法：使用@ResponseBody将方法返回结果，作为响应的返回内容。

|  |
| --- |
| package com.profound.blog.controller;  import java.util.List;  import javax.servlet.http.HttpServletResponse;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Controller;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.ResponseBody;  import com.profound.blog.domain.Country;  import com.profound.blog.service.CountryService;  @Controller  @RequestMapping("/country")  public class CountryController {  @Autowired  private CountryService countryService;    @RequestMapping("/listjson")  @ResponseBody  public List<Country> listCountry(HttpServletResponse response){  return this.countryService.listAll();  }    } |

请求：

|  |
| --- |
| http://localhost:8088/springMVCblog/country/listjson.do |

|  |
| --- |
| 注：使用htm\html为后缀在Spring3.2中，将导致不能正确返回JSON,在3.1中可以。原因是Spring根据请求中URL含有html(htm)，认为请求的MIME类型为text/html而不是application/json。可使用“.do”，“.action”，“.shtml”。  <servlet>  <servlet-name>springmvc</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>/WEB-INF/webServletContext.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> |

# 自定义的数据类型转换

在用户输入日期的情况下，日期可以是多种格式。

springMVC可以为如下日期格式自动转换为日期类型：

“3/3/1990”、“1990/3/3”。

但是对中文常用的“1990-3-3”确不能正确的转换（java.util.Date），而抛出异常。

要正确的进行日期类型的数据转换，必须自定义数据类型转换器。

下面将以此为例，说明自定义数据类型转换的使用与配置。

本例适用的spring版本为3.2和3.1（未进行验证，参考网络上的文章可适用于3.1。在spring3.2.9上验证通过。）

## Converter接口

org.springframework.core.convert.converter.Converter接口。要自定义一个类型转换器需实现Converter接口。

|  |
| --- |
| package com.profound.util;  import java.text.SimpleDateFormat;  import java.util.Date;  import org.springframework.core.convert.converter.Converter;  public class DataConverter implements Converter<String, Date> {  @Override  public Date convert(String arg0) {  SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");  format.setLenient(true);  Date date = null;  try {  date = format.parse(arg0);  } catch (Exception e) {  e.printStackTrace();  }  return date;  }  } |

## 配置自定义的类型转换器

|  |
| --- |
| <!-- spring 3.0 的功能，使用MVC注解 要加上conversion-service属性 -->  <mvc:annotation-driven conversion-service="conversionService"/>  <!-- 添加指定的数据类型转换器 -->  <bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean">  <property name="converters">  <list>  <!-- 自定义的类型转换器 -->  <bean class="com.profound.util.DataConverter"></bean>  </list>  </property>  </bean> |

# 配置过滤器

SpringMVC通过DispatchServlet对Action的请求进行了管理。但是没有对过滤器进行管理。如果我们要在过滤器中使用Spring管理的Bean，并使Spring能够帮我们进行注入，就要使用DelegatingFilterProxy。使用DelegatingFilterProxy来代理我们自己定义的Filter。

比如我们要在过滤器中实现用户自动登录功能。通过过滤器对所有的用户请求进行过滤，当可以从Cookie中获得满足登录的条件时，为用户实现自动登录。

## 实现一个过滤器

首先我们必须定义一个过滤器，实现javax.servlet.Filter接口。这与我们在使用Servlet来开发并无区别。但为了使Spring的Bean容器来管理这个Filter。我们需给它加上相应的注解（当然也可以在XML文件中进行配置）。

|  |
| --- |
| package com.profound.blog.filter;  import java.io.IOException;  import javax.servlet.Filter;  import javax.servlet.FilterChain;  import javax.servlet.FilterConfig;  import javax.servlet.ServletException;  import javax.servlet.ServletRequest;  import javax.servlet.ServletResponse;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Component;  import org.springframework.web.context.ContextLoader;  import com.profound.blog.service.RegUserService;  @Component  public class AutoLogonFilter implements Filter {  @Autowired  RegUserService regUserService ;  public void setRegUserService(RegUserService regUserService) {  this.regUserService = regUserService;  }  public void destroy() {  // TODO Auto-generated method stub  }  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {  // = ContextLoader.getCurrentWebApplicationContext().getBean(RegUserService.class);  System.out.println("#############"+regUserService );  chain.doFilter(request, response);  }    public void init(FilterConfig fConfig) throws ServletException {  // TODO Auto-generated method stub  }  } |

## Spring配置

服务器在加载Web.xml文件的配置时，先加载监听器再加载过滤器和Servlet。所以要在监听器中实现对我们自定义的Filter Bean的管理，先于Filter代理的加载。因此，在Spring的Bean定义文件中，过滤器与业务方法和DAO一同被扫描管理。在Web.xml文件中，定义为ContextLoaderListener来加载这个父级Bean容器。

|  |
| --- |
| <context:component-scan base-package="com.profound.blog.dao"/>  <context:component-scan base-package="com.profound.blog.service"/>  <context:component-scan base-package="com.profound.blog.filter"/>  <!--如果不扫描filter所在的包，就按下面这样定义吧 -->  <!-- <bean name="autoLogonFilter" -->  <!-- class="com.profound.blog.filter.AutoLogonFilter" -->  <!-- p:regUserService-ref="regUserService" -->  <!-- ></bean> --> |

## Web.xml配置

在Web.xml文件中，要配置Filter代理。

|  |
| --- |
| <filter>  <!-- filter-name 必须与Spring 管理的过滤器bean名字相同 -->  <filter-name>autoLogonFilter</filter-name>  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>  <!-- <init-param> -->  <!-- <param-name>targetBeanName</param-name> -->  <!-- <param-value>autoLogonFilter</param-value> -->  <!-- </init-param> -->  <!-- <init-param> -->  <!-- <param-name>contextAttribute</param-name> -->  <!-- <param-value>session</param-value> -->  <!-- </init-param> -->  <init-param>  <param-name>targetFilterLifecycle</param-name>  <!-- 为true，表示执行过滤器的生命周期方法 -->  <param-value>true</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>autoLogonFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |