**4.1、Controller简介**

Controller控制器，是MVC中的**部分C**，为什么是部分呢？

因为此处的Controller控制器主要负责**功能处理**部分：

1、收集、验证请求参数并绑定到命令对象；

2、将命令对象交给业务对象，由业务对象处理并返回模型数据；

3、返回ModelAndView（Model部分是业务对象返回的模型数据，视图部分为逻辑视图名）。

DispatcherServlet主要负责**整体的控制流程的调度**部分：

1、负责将请求委托给控制器进行处理；

2、根据控制器返回的逻辑视图名选择具体的视图进行渲染（并把模型数据传入）。

因此MVC中**完整的C**（包含控制逻辑+功能处理）由（DispatcherServlet + Controller）组成。

因此此处的控制器是Web MVC中部分，也可以称为页面控制器、动作、处理器。

Spring Web MVC支持多种类型的控制器，比如实现Controller接口，从Spring2.5开始支持注解方式的控制器（如@Controller、@RequestMapping、@RequestParam、@ModelAttribute等），我们也可以自己实现相应的控制器（只需要定义相应的HandlerMapping和HandlerAdapter即可）。

因为考虑到还有部分公司使用继承Controller接口实现方式，因此我们也学习一下，虽然已经不推荐使用了。

 对于注解方式的控制器，后边会详细讲，在此我们先学习Spring2.5以前的Controller接口实现方式。

首先我们将项目springmvc-chapter2复制一份改为项目springmvc-chapter4，本章示例将放置在springmvc-chapter4中。

大家需要将项目springmvc-chapter4/ .settings/ org.eclipse.wst.common.component下的chapter2改为chapter4，否则上下文还是“springmvc-chapter2”。以后的每一个章节都需要这么做。

**4.2、Controller接口**

package org.springframework.web.servlet.mvc;

public interface Controller {

ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception;

}

这是控制器接口，此处只有一个方法handleRequest，用于进行请求的功能处理，处理完请求后返回ModelAndView（Model模型数据部分 和 View视图部分）。

 Spring默认提供了一些Controller接口的实现以方便我们使用，具体继承体系如图4-1：

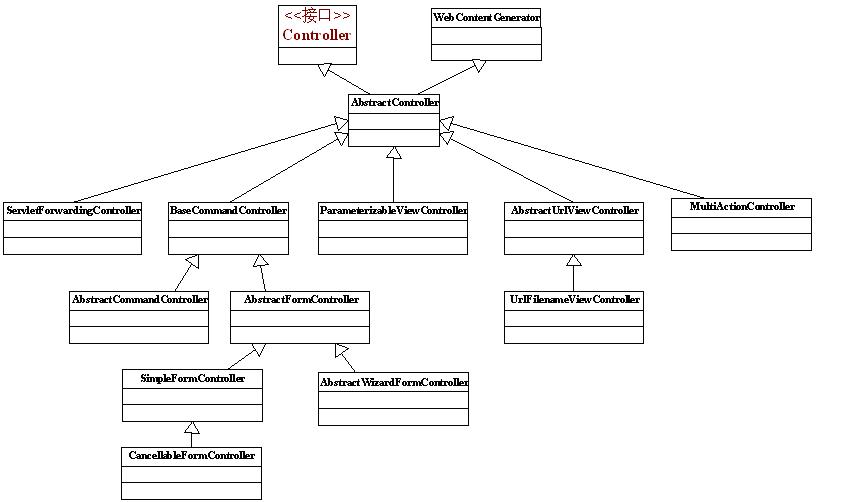


图4-1

**4.3、WebContentGenerator**

用于提供如**浏览器缓存控制**、**是否必须有session开启**、**支持的请求方法类型**（GET、POST等）等，该抽象类默认被AbstractController和WebContentInterceptor继承。该类主要有如下属性：

**Set<String>   supportedMethods：**

设置支持的请求方法类型，默认支持“GET”、“POST”、“HEAD”，如果我们想支持“PUT”，则可以加入该集合“PUT”。

**boolean requireSession = false：**

是否当前请求必须有session，如果此属性为true，但当前请求没有打开session将抛出HttpSessionRequiredException异常；

|  |
| --- |
| **boolean useExpiresHeader = true：**  是否使用HTTP1.0协议过期响应头：如果true则会在响应头添加：“Expires：”；需要配合cacheSeconds使用；  **boolean useCacheControlHeader = true：**  是否使用HTTP1.1协议的缓存控制响应头，如果true则会在响应头添加；需要配合cacheSeconds使用；  **boolean useCacheControlNoStore = true：**  是否使用HTTP 1.1协议的缓存控制响应头，如果true则会在响应头添加；需要配合cacheSeconds使用；  **private int cacheSeconds = -1：**  缓存过期时间，正数表示需要缓存，负数表示不做任何事情（也就是说保留上次的缓存设置）。 |
| 1、cacheSeconds**=**0时，则将设置如下响应头数据：          Pragma：no-cache             // HTTP 1.0的不缓存响应头          Expires：1L                  // useExpiresHeader=true时，HTTP 1.0          Cache-Control**：no-cache**// useCacheControlHeader=true时，HTTP 1.1          Cache-Control**：no-store**// useCacheControlNoStore=true时，该设置是防止Firefox缓存          2、cacheSeconds>0时，则将设置如下响应头数据：          Expires：System.currentTimeMillis() + **cacheSeconds** \* 1000L    // useExpiresHeader=true时，HTTP 1.0          Cache-Control**：max-age=cacheSeconds**              // useCacheControlHeader=true时，HTTP 1.1          3、cacheSeconds<0时，则什么都不设置，即保留上次的缓存设置。 |

|  |
| --- |
| 此处简单说一下以上响应头的作用，缓存控制已超出本书内容：  **HTTP1.0缓存控制响应头**    Pragma：no-cache：表示防止客户端缓存，需要强制从服务器获取最新的数据；    Expires：HTTP1.0响应头，本地副本缓存过期时间，如果客户端发现缓存文件没有过期则不发送请求，HTTP的日期时间必须是格林威治时间（GMT）， 如“Expires:Wed, 14 Mar 2012 09:38:32 GMT”；    **HTTP1.1缓存控制响应头**    Cache-Control**：**no-cache  强制客户端每次请求获取服务器的最新版本，不经过本地缓存的副本验证；    Cache-Control**：**no-store强制客户端不保存请求的副本，该设置是防止Firefox缓存    Cache-Control：max-age=[秒]    客户端副本缓存的最长时间，类似于HTTP1.0的Expires，只是此处是基于请求的相对时间间隔来计算，而非绝对时间。     还有相关缓存控制机制如Last-Modified（最后修改时间验证，客户端的上一次请求时间 在 服务器的最后修改时间 之后，说明服务器数据没有发生变化 返回304状态码）、ETag（没有变化时不重新下载数据，返回304）。 |

**4.4、AbstractController**

该抽象类实现了Controller，并继承了WebContentGenerator（具有该类的特性，具体请看4.3），该类有如下属性：

**boolean synchronizeOnSession = false：**表示该控制器是否在执行时同步session，从而保证该会话的用户串行访问该控制器。

|  |
| --- |
| **public** ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response){  // 委托给WebContentGenerator进行缓存控制  checkAndPrepare(request, response, **this** **instanceof** LastModified);  // 当前会话是否应串行化访问.  **if** (**this**.synchronizeOnSession) {  HttpSession session = request.getSession(**false**);  **if** (session != **null**) {  Object mutex = WebUtils.getSessionMutex(session);  **synchronized** (mutex) {  **return** handleRequestInternal(request, response);  }  }  }  **return** handleRequestInternal(request, response);  } |

可以看出AbstractController实现了一些特殊功能，如继承了WebContentGenerator缓存控制功能，并提供了可选的会话的串行化访问功能。而且提供了handleRequestInternal方法，因此我们应该在具体的控制器类中实现handleRequestInternal方法，而不再是handleRequest。

**直接通过response写响应**

如果我们想直接在控制器通过response写出响应呢，以下代码帮我们阐述：

|  |
| --- |
| **protected** ModelAndView handleRequestInternal(HttpServletRequest req, HttpServletResponse resp) {  resp.getWriter().write("Hello World!!");  //如果想直接在该控制器中写响应，可以通过返回null告诉DispatcherServlet自己已经写出响应了，不需要它进行视图解析  **return** **null**;  } |

**强制请求方法类型：**

|  |
| --- |
| <!— 在chapter4-servlet.xml配置处理器 -->  <bean name="/helloWithPOST" class="cn.javass.chapter4.web.controller.HelloWorldController">  <property name="supportedMethods" value="POST"></property>  </bean> |
| 以上配置表示只支持POST请求，如果是GET请求客户端将收到“HTTP Status 405 - Request method 'GET' not supported”。 |

**当前请求的session前置条件检查，如果当前请求无session将抛出HttpSessionRequiredException异常：**

|  |
| --- |
| <!— 在chapter4-servlet.xml配置处理器 -->  <bean name="/helloRequireSession"  class="cn.javass.chapter4.web.controller.HelloWorldController">  <property name="requireSession" value="true"/>  </bean> |
| 在进入该控制器时，一定要有session存在，否则抛出HttpSessionRequiredException异常。 |

**Session同步：**

即同一会话只能串行访问该控制器。

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

**注：下面提到一些关于缓存控制的一些特殊情况：**

|  |
| --- |
| 1、对于一般的页面跳转（如超链接点击跳转、通过js调用window.open打开新页面都是会使用浏览器缓存的，在未过期情况下会直接使用浏览器缓存的副本而不发送请求）；      2、对于刷新页面（如按F5键刷新），会再次发送一次请求到服务器的； |

**客户端端缓存控制：**

**1、缓存5秒，cacheSeconds=5**

|  |
| --- |
| <!— 在chapter4-servlet.xml配置处理器 -->  <bean name="/helloCache"  class="cn.javass.chapter4.web.controller.HelloWorldCacheController">  <property name="cacheSeconds" value="5"/>  </bean> |
| 服务器返回的响应头：  Cache-Control:max-age=5  Expires:Wed, 14 Mar 2012 09:38:32 GMT |
| 5秒内从缓存取，5秒后会重新发送请求 |

**2、不缓存，cacheSeconds=0**

|  |
| --- |
| <bean name="/helloNoCache"  class="cn.javass.chapter4.web.controller.HelloWorldCacheController">  <property name="cacheSeconds" value="0"/>  </bean> |
| Cache-Control: no-cache, no-store  Pragma: no-cache |
| 会要求浏览器每次都去请求服务器下载最新的数据 |

**3、cacheSeconds<0，将不添加任何数据**

响应头什么缓存控制信息也不加。

**4、Last-Modified缓存机制**

（1、在客户端第一次输入url时，服务器端会返回内容和状态码200表示请求成功并返回了内容；同时会添加一个“Last-Modified”的响应头表示此文件在服务器上的最后更新时间，如“Last-Modified:Wed, 14 Mar 2012 10:22:42 GMT”表示最后更新时间为（2012-03-14 10：22）；

（2、客户端第二次请求此URL时，客户端会向服务器发送请求头 “If-Modified-Since”，询问服务器该时间之后当前请求内容是否有被修改过，如“If-Modified-Since: Wed, 14 Mar 2012 10:22:42 GMT”，如果服务器端的内容没有变化，则自动返回 HTTP 304状态码（只要响应头，内容为空，这样就节省了网络带宽）。

**客户端强制缓存过期：**

（1、可以按ctrl+F5强制刷新（会添加请求头 HTTP1.0 Pragma:no-cache和 HTTP1.1 Cache-Control:no-cache、If-Modified-Since请求头被删除）表示强制获取服务器内容，不缓存。

（2、在请求的url后边加上时间戳来重新获取内容，加上时间戳后浏览器就认为不是同一份内容：

<http://sishuok.com/>?2343243243 和 [http://sishuok.com/?3433434](http://sishuok.com/?34334343)4 是两次不同的请求。

**Spring也提供了Last-Modified机制的支持，只需要实现LastModified接口，如下所示：**

|  |
| --- |
| **public** **class** HelloWorldLastModifiedCacheController **extends** AbstractController **implements** LastModified {  **protected** ModelAndView handleRequestInternal(HttpServletRequest req,  HttpServletResponse resp) **throws** Exception {  // 点击后再次请求当前页面  resp.getWriter().write("<a href=''>this</a>");  **return** **null**;  }  **private** **long** lastModified;  **public** **long** getLastModified(HttpServletRequest request) {  **if** (lastModified == 0L) {  lastModified = System.*currentTimeMillis*();  }  // **TODO** add 保证当内容发生改变时返回最新的修改时间  **return** lastModified;  }  } |
| **第一次请求,响应头中包含Last-Modified时间戳信息。**  **第二次请求, 请求头中会包含If\_Modified-Since时间戳信息，返回状态吗304表示服务器没有更新过。**  **重启服务器进行第三次请求，返回状态码200（因为服务器的lastModified变了）** |
| **Spring判断是否过期，通过如下代码，即请求的“If-Modified-Since” 大于等于当前的getLastModified方法的时间戳，则认为没有修改：**  **this.notModified = (ifModifiedSince >= (lastModifiedTimestamp / 1000 \* 1000));** |

**5、ETag（实体标记）缓存机制**

（1：浏览器第一次请求，服务器在响应时给请求URL标记，并在HTTP响应头中将其传送到客户端，类似服务器端返回的格式：“ETag:"0f8b0c86fe2c0c7a67791e53d660208e3"”

（2：浏览器第二次请求，客户端的查询更新格式是这样的：“If-None-Match:"0f8b0c86fe2c0c7a67791e53d660208e3"”，如果ETag没改变，表示内容没有发生改变，则返回状态304。

**Spring也提供了对ETag的支持，具体需要在web.xml中配置如下代码：**

|  |
| --- |
| <filter>  <filter-name>etagFilter</filter-name>  <filter-class>org.springframework.web.filter.ShallowEtagHeaderFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>etagFilter</filter-name>  <servlet-name>chapter4</servlet-name>  </filter-mapping> |
| 此过滤器只过滤到我们DispatcherServlet的请求。 |
| 1、发送请求到服务器，服务器返回的响应头中添加了（ETag:"0f8b0c86fe2c0c7a67791e53d660208e3"）  2、浏览器再次发送请求到服务器（按F5刷新），请求头中添加了（“If-None-Match: "0f8b0c86fe2c0c7a67791e53d660208e3"），  响应返回304代码，表示服务器没有修改，并且响应头再次添加了（ETag:"0f8b0c86fe2c0c7a67791e53d660208e3"）（每次都需要计算） |

**那服务器端是如何计算ETag的呢？**

|  |
| --- |
| protected String generateETagHeaderValue(byte[] bytes) {  StringBuilder builder = new StringBuilder("\"0");  DigestUtils.appendMd5DigestAsHex(bytes, builder);  builder.append('"');  return builder.toString();  } |
| bytes是response要写回到客户端的响应体（即响应的内容数据），是通过MD5算法计算的内容的摘要信息。也就是说如果服务器内容不发生改变，则ETag每次都是一样的，即服务器端的内容没有发生改变。 |

**此处只列举了部分缓存控制，详细介绍超出了本书的范围，强烈推荐：**[**http://www.mnot.net/cache\_docs/**](http://www.mnot.net/cache_docs/)**（中文版**[**http://www.chedong.com/tech/cache\_docs.html**](http://www.chedong.com/tech/cache_docs.html)**） 详细了解HTTP缓存控制及为什么要缓存。**

**缓存的目的是减少相应延迟 和 减少网络带宽消耗，比如css、js、图片这类静态资源应该进行缓存。**

**实际项目一般使用反向代理服务器（如nginx、apache等）进行缓存。**

## 4.5、ServletForwardingController

将接收到的请求转发到一个命名的servlet，具体示例如下：

|  |
| --- |
| Web.xml中配置的servlet:  <servlet>  <servlet-name>forwarding</servlet-name>  <servlet-class>cn.javass.chapter4.web.servlet.ForwardingServlet</servlet-class>  </servlet> |
| <bean name="/forwardToServlet"  class="org.springframework.web.servlet.mvc.ServletForwardingController">  <property name="servletName" value="forwarding"></property>  </bean> |
| 当我们请求/forwardToServlet时，会被转发到名字为“forwarding”的servlet处理 |

**4.12、ParameterizableViewController**

参数化视图控制器，不进行功能处理（即静态视图），根据参数的逻辑视图名直接选择需要展示的视图。

**Java代码  http://sishuok.com/forum/tinymce/tiny_mce/themes/advanced/img/trans.gif**

1. <bean name="/parameterizableView"
2. class="org.springframework.web.servlet.mvc.ParameterizableViewController">
3. <property name="viewName" value="success"/>
4. </bean>

该控制器接收到请求后直接选择参数化的视图，这样的好处是在配置文件中配置，从而避免程序的硬编码，比如像帮助页面等不需要进行功能处理，因此直接使用该控制器映射到视图。

**4.13、AbstractUrlViewController**

提供根据请求URL路径直接转化为逻辑视图名的支持基类，即不需要功能处理，直接根据URL计算出逻辑视图名，并选择具体视图进行展示：

**urlDecode：**是否进行url解码，不指定则默认使用服务器编码进行解码（如Tomcat默认ISO-8859-1）；

**urlPathHelper：**用于解析请求路径的工具类，默认为org.springframework.web.util.UrlPathHelper。

UrlFilenameViewController是它的一个实现者，因此我们应该使用UrlFilenameViewController。

**4.14、UrlFilenameViewController**

将请求的URL路径转换为逻辑视图名并返回的转换控制器，即不需要功能处理，直接根据URL计算出逻辑视图名，并选择具体视图进行展示：

根据请求URL路径计算逻辑视图名；

**Java代码  http://sishuok.com/forum/tinymce/tiny_mce/themes/advanced/img/trans.gif**

1. <bean name="/index1/\*"
2. class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
3. <bean name="/index2/\*\*"
4. class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
5. <bean name="/\*.html"
6. class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
7. <bean name="/index3/\*.html"
8. class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>

**/index1/\*：**可以匹配/index1/demo，但不匹配/index1/demo/demo，如/index1/demo逻辑视图名为demo；

**/index2/\*\*：**可以匹配/index2路径下的所有子路径，如匹配/index2/demo，或/index2/demo/demo，“/index2/demo”的逻辑视图名为demo，而“/index2/demo/demo”逻辑视图名为demo/demo；

**/\*.html：**可以匹配如/abc.html，逻辑视图名为abc，后缀会被删除（不仅仅可以是html）；

**/index3/\*.html：**可以匹配/index3/abc.html，逻辑视图名也是abc;

上述模式为Spring Web MVC使用的Ant-style 模式进行匹配的：

**Java代码  http://sishuok.com/forum/tinymce/tiny_mce/themes/advanced/img/trans.gif**

1. ?    匹配一个字符，如/index? 可以匹配 /index1 ， 但不能匹配 /index 或 /index12
2. \*    匹配零个或多个字符，如/index1/\*，可以匹配/index1/demo，但不匹配/index1/demo/demo
3. \*\*   匹配零个或多个路径，如/index2/\*\*：可以匹配/index2路径下的所有子路径，如匹配/index2/demo，或/index2/demo/demo
5. 如果我有如下模式，那Spring该选择哪一个执行呢？当我的请求为“/long/long”时如下所示：
6. /long/long
7. /long/\*\*/abc
8. /long/\*\*
9. /\*\*
10. Spring的AbstractUrlHandlerMapping使用：最长匹配优先；
11. 如请求为“/long/long” 将匹配第一个“/long/long”，但请求“/long/acd” 则将匹配 “/long/\*\*”，如请求“/long/aa/abc”则匹配“/long/\*\*/abc”，如请求“/abc”则将匹配“/\*\*”

UrlFilenameViewController还提供了如下属性：

**prefix：**生成逻辑视图名的前缀；

**suffix：**生成逻辑视图名的后缀；

**Java代码  http://sishuok.com/forum/tinymce/tiny_mce/themes/advanced/img/trans.gif**

1. protected String postProcessViewName(String viewName) {
2. return getPrefix() + viewName + getSuffix();
3. }

**Java代码  http://sishuok.com/forum/tinymce/tiny_mce/themes/advanced/img/trans.gif**

1. <bean name="/\*.htm" class="org.springframework.web.servlet.mvc.UrlFilenameViewController">
2. <property name="prefix" value="test"/>
3. <property name="suffix" value="test"/>
4. </bean>

当**prefix=“test”，suffix=“test”**，如上所示的**/\*.htm：**可以匹配如/abc.htm，但逻辑视图名将变为testabctest。

## 4.6、~~BaseCommandController~~

命令控制器通用基类，提供了以下功能支持：

1、数据绑定：请求参数绑定到一个command object（命令对象, 指绑定请求参数的任何POJO对象）。

   commandClass：表示命令对象实现类，如UserModel；

   commandName：表示放入请求的命令对象名字（默认command），request.setAttribute(commandName, commandObject);

 2、验证功能：提供Validator注册功能，注册的验证器会验证命令对象属性数据是否合法；

   validators：通过该属性注入验证器，验证器用来验证命令对象属性是否合法；

该抽象类没有没有提供流程功能，只是提供了一些公共的功能，实际使用时需要使用它的子类。

|  |
| --- |
| BaseCommandController  |-- AbstractCommandController  |-- AbstractFormController  |-- AbstractWizardFormController  |-- SimpleFormController  |-- CancellableFormController |

## 4.7、AbstractCommandController

命令控制器之一，可以实现该控制器来创建命令控制器，该控制器能把自动封装请求参数到一个命令对象，而且提供了验证功能。

|  |
| --- |
| public class UserModel {  private String username;  private String password;  //省略setter/getter  } |
| public class MyAbstractCommandController extends AbstractCommandController {  public MyAbstractCommandController() {  //设置命令对象实现类  setCommandClass(UserModel.class);  }  @Override  protected ModelAndView handle(HttpServletRequest req, HttpServletResponse resp,  Object command, BindException errors) throws Exception {  //将命令对象转换为实际类型  UserModel user = (UserModel) command;  ModelAndView mv = new ModelAndView();  mv.setViewName("abstractCommand");  mv.addObject("user", user);  return mv;  }  } |
| <bean name="/abstractCommand"  class="cn.javass.chapter4.web.controller.MyAbstractCommandController">  <!-- 也可以通过依赖注入 注入命令实现类 -->  <!-- property name="commandClass" value="cn.javass.chapter4.model.UserModel"/-->  </bean> |

**4.8、AbstractFormController**

用于支持带步骤的表单提交的命令控制器基类，使用该控制器可以完成：

1、定义表单处理（表单的渲染），并从控制器获取命令对象构建表单；

2、提交表单处理，当用户提交表单内容后，AbstractFormController可以将用户请求的数据绑定到命令对象，并可以验证表单内容、对命令对象进行处理。

**java代码：**

**Java代码  [收藏代码](javascript:void())**

1. @Override
2. rotected ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletResponse response)
3. **throws** Exception {
4. //1、是否是表单提交？ 该方法实现为（"POST".equals(request.getMethod())），即POST表示表单提交
5. **if** (isFormSubmission(request)) {
6. **try** {
7. Object command = getCommand(request);
8. ServletRequestDataBinder binder = bindAndValidate(request, command);
9. BindException errors = **new** BindException(binder.getBindingResult());
10. //表单提交应该放到该方法实现
11. **return** processFormSubmission(request, response, command, errors);
12. }
13. **catch** (HttpSessionRequiredException ex) {
14. //省略部分代码
15. **return** handleInvalidSubmit(request, response);
16. }
17. }
18. **else** {
19. //2、表示是表单展示，该方法又转调showForm方法，因此我们需要覆盖showForm来完成表单展示
20. **return** showNewForm(request, response);
21. }

**bindOnNewForm**：是否在进行表单展示时绑定请求参数到表单对象，默认false，不绑定；

**sessionForm**：session表单模式，如果开启（true）则会将表单对象放置到session中，从而可以跨越多次请求保证数据不丢失（多步骤表单常使用该方式，详解AbstractWizardFormController），默认false；

**Object formBackingObject(HttpServletRequest request)**：提供给表单展示时使用的表单对象（form object表单要展示的默认数据），默认通过commandName暴露到请求给展示表单；

**Map referenceData(HttpServletRequest request, Object command, Errors errors)**：展示表单时需要的一些引用数据（比如用户注册，可能需要选择工作地点，这些数据可以通过该方法提供），如：

**java代码：**

**Java代码  [收藏代码](javascript:void())**

1. **protected** Map referenceData(HttpServletRequest request) **throws** Exception {
2. Map model = **new** HashMap();
3. model.put("cityList", cityList);
4. **return** model;
5. }

    这样就可以在表单展示页面获取cityList数据。

SimpleFormController继承该类，而且提供了更简单的表单流程控制。

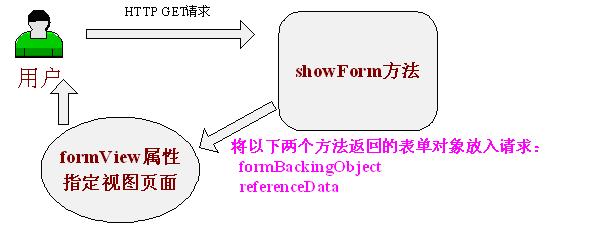
**4.9、SimpleFormController**

提供了更好的两步表单支持：

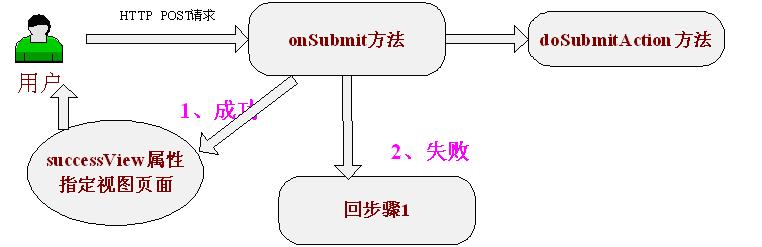
1、准备要展示的数据，并到表单展示页面；

2、提交数据数据进行处理。

第一步，展示：



第二步，提交表单：



接下来咱们写一个用户注册的例子学习一下：

**（1、控制器**

**java代码：**

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.javass.chapter4.web.controller;
2. //省略import
3. **public** **class** RegisterSimpleFormController **extends** SimpleFormController {
4. **public** RegisterSimpleFormController() {
5. setCommandClass(UserModel.**class**); //设置命令对象实现类
6. setCommandName("user");//设置命令对象的名字
7. }
8. //form object 表单对象，提供展示表单时的表单数据（使用commandName放入请求）
9. **protected** Object formBackingObject(HttpServletRequest request) **throws** Exception {
10. UserModel user = **new** UserModel();
11. user.setUsername("请输入用户名");
12. **return** user;
13. }
14. //提供展示表单时需要的一些其他数据
15. **protected** Map referenceData(HttpServletRequest request) **throws** Exception {
16. Map map = **new** HashMap();
17. map.put("cityList", Arrays.asList("山东", "北京", "上海"));
18. **return** map;
19. }
20. **protected** **void** doSubmitAction(Object command) **throws** Exception {
21. UserModel user = (UserModel) command;
22. //TODO 调用业务对象处理
23. System.out.println(user);
24. }
25. }

**setCommandClass和setCommandName：**分别设置了命令对象的实现类和名字；

**formBackingObject和referenceData：**提供了表单展示需要的视图；

**doSubmitAction：**用于执行表单提交动作，由onSubmit方法调用，如果不需要请求/响应对象或进行数据验证，可以直接使用doSubmitAction方法进行功能处理。

**（2、spring配置（chapter4-servlet.xml）**

**java代码：**

**Java代码  [收藏代码](javascript:void())**

1. <bean name="/simpleForm"
2. **class**="cn.javass.chapter4.web.controller.RegisterSimpleFormController">
3. <property name="formView" value="register"/>
4. <property name="successView" value="redirect:/success"/>
5. </bean>
6. <bean name="/success" **class**="cn.javass.chapter4.web.controller.SuccessController"/>

**formView：**表示展示表单时显示的页面；

**successView：**表示处理成功时显示的页面；“*redirect:/success*”表示成功处理后重定向到/success控制器；防止表单重复提交；

“*/success*” bean的作用是显示成功页面，此处就不列举了。

**（3、视图页面**

**java代码：**

**Java代码  [收藏代码](javascript:void())**

1. <!-- register.jsp 注册展示页面-->
2. <form method="post">
3. username:<input type="text" name="username" value="${user.username}"><br/>
4. password:<input type="password" name="username"><br/>
5. city:<select>
6. <c:forEach items="${cityList }" var="city">
7. <option>${city}</option>
8. </c:forEach>
9. </select><br/>
10. <input type="submit" value="注册"/>
11. </form>

此处可以使用${user.username}获取到formBackingObject设置的表单对象、使用${cityList}获取referenceData设置的表单支持数据；

到此一个简单的两步表单到此结束，但这个表单有重复提交表单的问题，而且表单对象到页面的绑定是通过手工绑定的，后边我们会学习spring标签库（提供自动绑定表单对象到页面）。

**4.10、CancellableFormController**

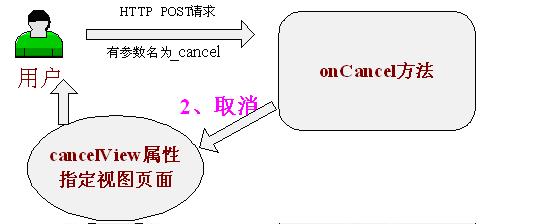
一个可取消的表单控制器，继承SimpleFormController，额外提供取消表单功能。

**1、表单展示：**和SimpleFormController一样；

**2、表单取消：**和SimpleFormController一样；

**3、表单成功提交：**取消功能处理方法为：onCancel(Object command)，而且默认返回cancelView属性指定的逻辑视图名。

   那如何判断是取消呢？如果请求中有参数名为“\_cancel”的参数，则表示表单取消。也可以通过cancelParamKey来修改参数名（如“\_cancel.x”等）。



**示例：**

**（1、控制器**

复制RegisterSimpleFormController一份命名为CanCancelRegisterSimpleFormController，添加取消功能处理方法实现：

**java代码：**

**Java代码  [收藏代码](javascript:void())**

1. @Override
2. **protected** ModelAndView onCancel(Object command) **throws** Exception {
3. UserModel user = (UserModel) command;
4. //TODO 调用业务对象处理
5. System.out.println(user);
6. **return** **super**.onCancel(command);
7. }

**onCancel：**在该功能方法内实现取消逻辑，父类的onCancel方法默认返回cancelView属性指定的逻辑视图名。

**（2、spring配置（chapter4-servlet.xml）**

**java代码：**

**Java代码  [收藏代码](javascript:void())**

1. <bean name="/canCancelForm"
2. **class**="cn.javass.chapter4.web.controller.CanCancelRegisterSimpleFormController">
3. <property name="formView" value="register"/>
4. <property name="successView" value="redirect:/success"/>
5. <property name="cancelView" value="redirect:/cancel"/>
6. </bean>
7. <bean name="/cancel" **class**="cn.javass.chapter4.web.controller.CancelController"/>

**cancelParamKey：**用于判断是否是取消的请求参数名，默认是\_cancel，即如果请求参数数据中含有名字\_cancel则表示是取消，将调用onCancel功能处理方法；

**cancelView：**表示取消时时显示的页面；“*redirect:/cancel*”表示成功处理后重定向到/cancel控制器；防止表单重复提交；

“*/cancel*” bean的作用是显示取消页面，此处就不列举了（详见代码）。

**（3、视图页面（修改register.jsp）**

**java代码：**

**Java代码  [收藏代码](javascript:void())**

1. <input type="submit" name="\_cancel" value="取消"/>

该提交按钮的作用是取消，因为name=*"\_cancel"，*即请求后会有一个名字为\_cancel的参数，因此会执行onCancel功能处理方法。

**（4、测试：**

在浏览器输入“http://localhost:9080/springmvc-chapter4/canCancelForm”，则首先到展示视图页面，点击“取消按钮”将重定向到“http://localhost:9080/springmvc-chapter4/cancel”，说明取消成功了。

实际项目可能会出现比如一些网站的完善个人资料都是多个页面（即多步），那应该怎么实现呢？接下来让我们看一下spring Web MVC提供的对多步表单的支持类AbstractWizardFormController。

**4.11、AbstractWizardFormController**

向导控制器类提供了多步骤（向导）表单的支持（如完善个人资料时分步骤填写基本信息、工作信息、学校信息等）

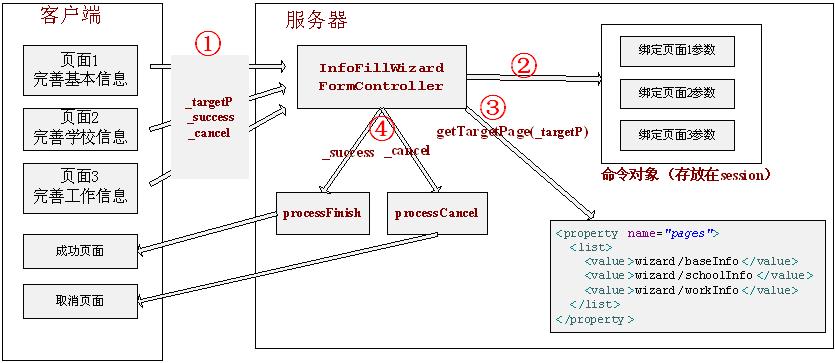
**假设现在做一个完善个人信息的功能，分三个页面展示：**

1、页面1完善基本信息；

2、页面2完善学校信息；

3、页面3完善工作信息。

这里我们要注意的是当用户跳转到页面2时页面1的信息是需要保存起来的，还记得AbstractFormController中的sessionForm吗？ 如果为true则表单数据存放到session中，哈哈，AbstractWizardFormController就是使用了这个特性。

  
**向导中的页码从0开始；**

**PARAM\_TARGET = "\_target"：**

用于选择向导中的要使用的页面参数名前缀，如“\_target0”则选择第0个页面显示，即图中的“wizard/baseInfo”，以此类推，如“\_target1”将选择第1页面，要得到的页码为去除前缀“\_target”后的数字即是；

**PARAM\_FINISH = "\_finish"：**

如果请求参数中有名为“\_finish”的参数，表示向导成功结束，将会调用processFinish方法进行完成时的功能处理；

**PARAM\_CANCEL = "\_cancel"：**

如果请求参数中有名为“\_cancel”的参数，表示向导被取消，将会调用processCancel方法进行取消时的功能处理；

**向导中的命令对象：**

向导中的每一个步骤都会把相关的参数绑定到命令对象，该表单对象默认放置在session中，从而可以跨越多次请求得到该命令对象。

接下来具体看一下如何使用吧。

**（1、修改我们的模型数据以支持多步骤提交：**

**Java代码  http://sishuok.com/forum/tinymce/tiny_mce/themes/advanced/img/trans.gif**

1. public class UserModel {
2. private String username;
3. private String password;
4. private String realname; //真实姓名
5. private WorkInfoModel workInfo;
6. private SchoolInfoModel schoolInfo;
7. //省略getter/setter
8. }

**Java代码  http://sishuok.com/forum/tinymce/tiny_mce/themes/advanced/img/trans.gif**

1. public class SchoolInfoModel {
2. private String schoolType; //学校类型：高中、中专、大学
3. private String schoolName; //学校名称
4. private String specialty; //专业
5. //省略getter/setter
6. }

**Java代码  http://sishuok.com/forum/tinymce/tiny_mce/themes/advanced/img/trans.gif**

1. public class WorkInfoModel {
2. private String city; //所在城市
3. private String job; //职位
4. private String year; //工作年限
5. //省略getter/setter
6. }

**（2、控制器**

**Java代码  http://sishuok.com/forum/tinymce/tiny_mce/themes/advanced/img/trans.gif**

1. package cn.javass.chapter4.web.controller;
2. //省略import
3. public class InfoFillWizardFormController extends AbstractWizardFormController {
4. public InfoFillWizardFormController() {
5. setCommandClass(UserModel.class);
6. setCommandName("user");
7. }
8. protected Map referenceData(HttpServletRequest request, int page) throws Exception {
9. Map map = new HashMap();
10. if(page==1) { //如果是填写学校信息页 需要学校类型信息
11. map.put("schoolTypeList", Arrays.asList("高中", "中专", "大学"));
12. }
13. if(page==2) {//如果是填写工作信息页 需要工作城市信息
14. map.put("cityList", Arrays.asList("济南", "北京", "上海"));
15. }
16. return map;
17. }
18. protected void validatePage(Object command, Errors errors, int page) {
19. //提供每一页数据的验证处理方法
20. }
21. protected void postProcessPage(HttpServletRequest request, Object command, Errors errors, int page) throws Exception {
22. //提供给每一页完成时的后处理方法
23. }
24. protected ModelAndView processFinish(HttpServletRequest req, HttpServletResponse resp, Object command, BindException errors) throws Exception {
25. //成功后的处理方法
26. System.out.println(command);
27. return new ModelAndView("redirect:/success");
28. }
29. protected ModelAndView processCancel(HttpServletRequest request, HttpServletResponse response, Object command, BindException errors) throws Exception {
30. //取消后的处理方法
31. System.out.println(command);
32. return new ModelAndView("redirect:/cancel");
33. }
34. }

**page页码：**是根据请求中以“\_target”开头的参数名来确定的，如“\_target0”，则页码为0；

**referenceData：**提供每一页需要的表单支持对象，如完善学校信息需要学校类型，page页码从0开始（而且根据请求参数中以“\_target”开头的参数来确定当前页码，如\_target1，则page=1）；

**validatePage：**验证当前页的命令对象数据，验证应根据page页码来分步骤验证；

**postProcessPage：**验证成功后的后处理；

**processFinish：**成功时执行的方法，此处直接重定向到/success控制器（详见CancelController）；

**processCancel：**取消时执行的方法，此处直接重定向到/cancel控制器（详见SuccessController）；

其他需要了解：

**allowDirtyBack和allowDirtyForward：**决定在当前页面验证失败时，是否允许向导前移和后退，默认false不允许；

**onBindAndValidate(HttpServletRequest request, Object command, BindException errors, int page)：**允许覆盖默认的绑定参数到命令对象和验证流程。

**（3、spring配置文件（chapter4-servlet.xml）**

**Java代码  http://sishuok.com/forum/tinymce/tiny_mce/themes/advanced/img/trans.gif**

1. <bean name="/infoFillWizard"
2. class="cn.javass.chapter4.web.controller.InfoFillWizardFormController">
3. <property name="pages">
4. <list>
5. <value>wizard/baseInfo</value>
6. <value>wizard/schoolInfo</value>
7. <value>wizard/workInfo</value>
8. </list>
9. </property>
10. </bean>

**pages：**表示向导中每一个步骤的逻辑视图名，当InfoFillWizardFormController的page=0，则将会选择“wizard/baseInfo”，以此类推，从而可以按步骤选择要展示的视图。

**（4、向导中的每一步视图**

**（4.1、基本信息页面(第一步) baseInfo.jsp：**

**Java代码  http://sishuok.com/forum/tinymce/tiny_mce/themes/advanced/img/trans.gif**

1. <form method="post">
2. 真实姓名:<input type="text" name="realname" value="${user.realname}"><br/>
3. <input type="submit" name="\_target1" value="下一步"/>
4. </form>

**当前页码为0；**

**name="\_target1"：**表示向导下一步要显示的页面的页码为1；

**（4.2、学校信息页面(第二步) schoolInfo.jsp：**

**Java代码  http://sishuok.com/forum/tinymce/tiny_mce/themes/advanced/img/trans.gif**

1. <form method="post">
2. 学校类型：<select name="schoolInfo.schoolType">
3. <c:forEach items="${schoolTypeList }" var="schoolType">
4. <option value="${schoolType }"
5. <c:if test="${user.schoolInfo.schoolType eq schoolType}">
6. selected="selected"
7. </c:if>
8. >
9. ${schoolType}
10. </option>
11. </c:forEach>
12. </select><br/>
13. 学校名称：<input type="text" name="schoolInfo.schoolName" value="${user.schoolInfo.schoolName}"/><br/>
14. 专业：<input type="text" name="schoolInfo.specialty" value="${user.schoolInfo.specialty}"/><br/>
15. <input type="submit" name="\_target0" value="上一步"/>
16. <input type="submit" name="\_target2" value="下一步"/>
17. </form>

**（4.3、工作信息页面(第三步) workInfo.jsp：**

**Java代码  http://sishuok.com/forum/tinymce/tiny_mce/themes/advanced/img/trans.gif**

1. <form method="post">
2. 所在城市：<select name="workInfo.city">
3. <c:forEach items="${cityList }" var="city">
4. <option value="${city }"
5. <c:if test="${user.workInfo.city eq city}">selected="selected"</c:if>
6. >
7. ${city}
8. </option>
9. </c:forEach>
10. </select><br/>
11. 职位：<input type="text" name="workInfo.job" value="${user.workInfo.job}"/><br/>
12. 工作年限：<input type="text" name="workInfo.year" value="${user.workInfo.year}"/><br/>
13. <input type="submit" name="\_target1" value="上一步"/>
14. <input type="submit" name="\_finish" value="完成"/>
15. <input type="submit" name="\_cancel" value="取消"/>
16. </form>

**当前页码为2；**

**name="\_target1"：上一步，**表示向导上一步要显示的页面的页码为1；

**name="\_finish"：向导完成，**表示向导成功，将会调用向导控制器的**processFinish方法**；

**name="\_cancel"：向导取消，**表示向导被取消，将会调用向导控制器的**processCancel方法**；

到此向导控制器完成，此处的向导流程比较简单，如果需要更复杂的页面流程控制，可以选择使用Spring Web Flow框架。

## 4.15、MultiActionController

之前学过的控制器【BaseCommandController及其子类，如AbstractCommandController、AbstractFormController等】只能实现一个功能（如新增），不能在一个控制器中实现多个功能（如用户增删改查），所以才有了MultiActionController。

#### 4.15.2 问题

1、  MultiActionController如何将不同的请求映射到不同的功能处理方法呢？

MethodNameResolver（方法名解析器）用于解析当前请求到需要执行的功能处理方法的方法名。

默认使用InternalPathMethodNameResolver，另外还提供了ParameterMethodNameResolver和PropertiesMethodNameResolver，当然我们也可以自己来实现。

2、那我们的功能处理方法应该怎么写呢？

public (ModelAndView | Map | String | void) actionName(HttpServletRequest request, HttpServletResponse response, [,HttpSession session] [,AnyObject]);

此处需要注意一下几点：

**1、返回值：即模型和视图部分；**

ModelAndView：模型和视图部分，之前已经见过了；

Map：只返回模型数据，逻辑视图名会根据RequestToViewNameTranslator实现类来计算；

String：只返回逻辑视图名；

void：表示该功能方法直接写出response响应（如果其他返回值类型（如Map）返回null则和void进行相同的处理）；

**2、actionName：**功能方法名字；由methodNameResolver根据请求信息解析功能方法名，通过反射调用；

**3、形参列表：**顺序固定，“[]”表示可选，我们来看看几个示例吧：

|  |
| --- |
| **//表示到新增页面**  public ModelAndView toAdd(HttpServletRequest request, HttpServletResponse response);  **//表示新增表单提交，在最后可以带着命令对象**  public ModelAndView add(HttpServletRequest request, HttpServletResponse response, UserModel user);  **//列表，但只返回模型数据，视图名会通过RequestToViewNameTranslator实现来计算**  public Map list(HttpServletRequest request, HttpServletResponse response);  **//文件下载，返回值类型为void，表示该功能方法直接写响应**  public void fileDownload(HttpServletRequest request, HttpServletResponse response)  **//第三个参数可以是session**  public ModelAndView sessionWith(HttpServletRequest request, HttpServletResponse response, HttpSession session);  **//如果第三个参数是session，那么第四个可以是命令对象，顺序必须是如下顺序**  public void sessionAndCommandWith(HttpServletRequest request, HttpServletResponse response, HttpSession session, UserModel user) |

**4、异常处理方法，**MultiActionController提供了简单的异常处理，即在请求的功能处理过程中遇到异常会交给异常处理方法进行处理，式如下所示：

public ModelAndView anyMeaningfulName(HttpServletRequest request, HttpServletResponse response, ExceptionClass exception)

MultiActionController会使用最接近的异常类型来匹配对应的异常处理方法，示例如下所示：

**//处理PayException**

public ModelAndView processPayException(HttpServletRequest request, HttpServletResponse response, PayException ex)

**//处理Exception**

public ModelAndView processException(HttpServletRequest request, HttpServletResponse response,  Exception ex)

#### 4.15.3 MultiActionController类实现

**类定义：**

public class MultiActionController extends AbstractController implements LastModified ，继承了AbstractController，并实现了LastModified接口，默认返回-1；

**核心属性：**

**delegate：**功能处理的委托对象，即我们要调用请求处理方法所在的对象，默认是this；

**methodNameResolver：**功能处理方法名解析器，即根据请求信息来解析需要执行的delegate的功能处理方法的方法名。

**核心方法：**

**Java代码  [收藏代码](javascript:void())**

1. //判断方法是否是功能处理方法
2. **private** **boolean** isHandlerMethod(Method method) {
3. //得到方法返回值类型
4. Class returnType = method.getReturnType();
5. //返回值类型必须是ModelAndView、Map、String、void中的一种，否则不是功能处理方法
6. **if** (ModelAndView.**class**.equals(returnType) || Map.**class**.equals(returnType) || String.**class**.equals(returnType) ||
7. **void**.**class**.equals(returnType)) {
8. Class[] parameterTypes = method.getParameterTypes();
9. //功能处理方法参数个数必须>=2，且第一个是HttpServletRequest类型、第二个是HttpServletResponse
10. //Controller接口的handleRequest(HttpServletRequest , HttpServletResponse )不是功能处理方法
11. **return** (parameterTypes.length >= 2 &&
12. HttpServletRequest.**class**.equals(parameterTypes[0]) &&
13. HttpServletResponse.**class**.equals(parameterTypes[1]) &&
14. !("handleRequest".equals(method.getName()) && parameterTypes.length == 2));
15. }
16. **return** **false**;
17. }

**Java代码  [收藏代码](javascript:void())**

1. //是否是异常处理方法
2. **private** **boolean** isExceptionHandlerMethod(Method method) {
3. //异常处理方法必须是功能处理方法 且 参数长度为3、第三个参数类型是Throwable子类
4. **return** (isHandlerMethod(method) &&
5. method.getParameterTypes().length == 3 &&
6. Throwable.**class**.isAssignableFrom(method.getParameterTypes()[2]));
7. }

**Java代码  [收藏代码](javascript:void())**

1. **private** **void** registerHandlerMethods(Object delegate) {
2. //缓存Map清空
3. **this**.handlerMethodMap.clear();
4. **this**.lastModifiedMethodMap.clear();
5. **this**.exceptionHandlerMap.clear();
7. //得到委托对象的所有public方法
8. Method[] methods = delegate.getClass().getMethods();
9. **for** (Method method : methods) {
10. //验证是否是异常处理方法，如果是放入exceptionHandlerMap缓存map
11. **if** (isExceptionHandlerMethod(method)) {
12. registerExceptionHandlerMethod(method);
13. }
14. //验证是否是功能处理方法，如果是放入handlerMethodMap缓存map
15. **else** **if** (isHandlerMethod(method)) {
16. registerHandlerMethod(method);
17. registerLastModifiedMethodIfExists(delegate, method);
18. }
19. }
20. }

**Java代码  [收藏代码](javascript:void())**

1. **protected** ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletResponse response)
2. **throws** Exception {
3. **try** {
4. //1、使用methodNameResolver 方法名解析器根据请求解析到要执行的功能方法的方法名
5. String methodName = **this**.methodNameResolver.getHandlerMethodName(request);
6. //2、调用功能方法（通过反射调用，此处就粘贴代码了）
7. **return** invokeNamedMethod(methodName, request, response);
8. }
9. **catch** (NoSuchRequestHandlingMethodException ex) {
10. **return** handleNoSuchRequestHandlingMethod(ex, request, response);
11. }
12. }

#### 4.15.4 MethodNameResolver

**1、InternalPathMethodNameResolver：**MultiActionController的默认实现，提供从请求URL路径解析功能方法的方法名，从请求的最后一个路径（/）开始，并忽略扩展名；如请求URL是“/user/list.html”，则解析的功能处理方法名为“list”，即调用list方法。该解析器还可以指定前缀和后缀，通过prefix和suffix属性，如指定prefix=”test\_”,则功能方法名将变为test\_list；

**2、ParameterMethodNameResolver：**提供从请求参数解析功能处理方法的方法名，并按照如下顺序进行解析：

（1、 methodParamNames：根据请求的参数名解析功能方法名（功能方法名和参数名同名）；

**Java代码  [收藏代码](javascript:void())**

1. <property name="methodParamNames" value="list,create,update"/>

  如上配置时，如果请求中含有参数名list、create、update时，则功能处理方法名为list、create、update，这种方式的可以在当一个表单有多个提交按钮时使用，不同的提交按钮名字不一样即可。

  ParameterMethodNameResolver也考虑到图片提交按钮提交问题：

    <input type="image" name="list"> 和submit类似可以提交表单，单击该图片后会发送两个参数“list.x=x轴坐标”和“list.y=y轴坐标”（如提交后会变为list.x=7&list.y=5）；因此我们配置的参数名（如list）在会加上“.x” 和 “.y”进行匹配。

**Java代码  [收藏代码](javascript:void())**

1. **for** (String suffix : SUBMIT\_IMAGE\_SUFFIXES)  {//SUBMIT\_IMAGE\_SUFFIXES {“.x”, “.y”}
2. **if** (request.getParameter(name + suffix) != **null**) {// name是我们配置的methodParamNames
3. **return** **true**;
4. }
5. }

（2、paramName：根据请求参数名的值解析功能方法名，默认的参数名是action，即请求的参数中含有“action=query”，则功能处理方法名为query；

（3、logicalMappings：逻辑功能方法名到真实功能方法名映射，如下所示：

**Java代码  [收藏代码](javascript:void())**

1. <property name="logicalMappings">
2. <props>
3. <prop key="doList">list</prop>
4. </props>
5. </property>

  即如果步骤1或2解析出逻辑功能方法名为doList（逻辑的），将会被重新映射为list功能方法名（真正执行的）。

（4、defaultMethodName：默认的方法名，当以上策略失败时默认调用的方法名。

**3、PropertiesMethodNameResolver：**提供自定义的从请求URL解析功能方法的方法名，使用一组用户自定义的模式到功能方法名的映射，映射使用Properties对象存放，具体配置示例如下：

**Java代码  [收藏代码](javascript:void())**

1. <bean id="propertiesMethodNameResolver"
2. **class**="org.springframework.web.servlet.mvc.multiaction.PropertiesMethodNameResolver">
3. <property name="mappings">
4. <props>
5. <prop key="/create">create</prop>
6. <prop key="/update">update</prop>
7. <prop key="/delete">delete</prop>
8. <prop key="/list">list</prop>
9. <!-- 默认的行为 -->
10. <prop key="/\*\*">list</prop>
11. </props>
12. </property>
13. </bean>

对于/create请求将调用create方法，Spring内部使用PathMatcher进行匹配（默认实现是AntPathMatcher）。

#### 4.15.5 RequestToViewNameTranslator

用于直接将请求转换为逻辑视图名。默认实现为DefaultRequestToViewNameTranslator。

**1、DefaultRequestToViewNameTranslator：**将请求URL转换为逻辑视图名，默认规则如下：

  http://localhost:9080/web上下文/list -------> 逻辑视图名为list

  http://localhost:9080/web上下文/list.html -------> 逻辑视图名为list(默认删除扩展名)

  http://localhost:9080/web上下文/user/list.html -------> 逻辑视图名为user/list

#### 4.15.6 示例

**（1、控制器UserController**

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.javass.chapter4.web.controller;
2. //省略import
3. **public** **class** UserController **extends** MultiActionController {
4. //用户服务类
5. **private** UserService userService;
6. //逻辑视图名 通过依赖注入方式注入，可配置
7. **private** String createView;
8. **private** String updateView;
9. **private** String deleteView;
10. **private** String listView;
11. **private** String redirectToListView;
12. //省略setter/getter
14. **public** String create(HttpServletRequest request, HttpServletResponse response, UserModel user) {
15. **if**("GET".equals(request.getMethod())) {
16. //如果是get请求 我们转向 新增页面
17. **return** getCreateView();
18. }
19. userService.create(user);
20. //直接重定向到列表页面
21. **return** getRedirectToListView();
22. }
23. **public** ModelAndView update(HttpServletRequest request, HttpServletResponse response, UserModel user) {
24. **if**("GET".equals(request.getMethod())) {
25. //如果是get请求 我们转向更新页面
26. ModelAndView mv = **new** ModelAndView();
27. //查询要更新的数据
28. mv.addObject("command", userService.get(user.getUsername()));
29. mv.setViewName(getUpdateView());
30. **return** mv;
31. }
32. userService.update(user);
33. //直接重定向到列表页面
34. **return** **new** ModelAndView(getRedirectToListView());
35. }

38. **public** ModelAndView delete(HttpServletRequest request, HttpServletResponse response, UserModel user) {
39. **if**("GET".equals(request.getMethod())) {
40. //如果是get请求 我们转向删除页面
41. ModelAndView mv = **new** ModelAndView();
42. //查询要删除的数据
43. mv.addObject("command", userService.get(user.getUsername()));
44. mv.setViewName(getDeleteView());
45. **return** mv;
46. }
47. userService.delete(user);
48. //直接重定向到列表页面
49. **return** **new** ModelAndView(getRedirectToListView());
50. }
52. **public** ModelAndView list(HttpServletRequest request, HttpServletResponse response) {
53. ModelAndView mv = **new** ModelAndView();
54. mv.addObject("userList", userService.list());
55. mv.setViewName(getListView());
56. **return** mv;
57. }
59. //如果使用委托方式，命令对象名称只能是command
60. **protected** String getCommandName(Object command) {
61. //命令对象的名字 默认command
62. **return** "command";
63. }
64. }

**增删改：**如果是GET请求方法，则表示到展示页面，POST请求方法表示真正的功能操作；

**getCommandName：**表示是命令对象名字，默认command，对于委托对象实现方式无法改变，因此我们就使用默认的吧。

**（2、spring配置文件chapter4-servlet.xml**

**Java代码  [收藏代码](javascript:void())**

1. <bean id="userService" **class**="cn.javass.chapter4.service.UserService"/>
2. <bean name="/user/\*\*" **class**="cn.javass.chapter4.web.controller.UserController">
3. <property name="userService" ref="userService"/>
4. <property name="createView" value="user/create"/>
5. <property name="updateView" value="user/update"/>
6. <property name="deleteView" value="user/delete"/>
7. <property name="listView" value="user/list"/>
8. <property name="redirectToListView" value="redirect:/user/list"/>
9. <!-- 使用PropertiesMethodNameResolver来解析功能处理方法名 -->
10. <!--property name="methodNameResolver" ref="propertiesMethodNameResolver"/-->
11. </bean>

**userService：**用户服务类，实现业务逻辑；

**依赖注入：**对于逻辑视图页面通过依赖注入方式注入，redirectToListView表示增删改成功后重定向的页面，防止重复表单提交；

**默认使用InternalPathMethodNameResolver解析请求URL到功能方法名。**

**（3、视图页面**

**（3.1、list页面（WEB-INF/jsp/user/list.jsp）**

**Java代码  [收藏代码](javascript:void())**

1. <a href="${pageContext.request.contextPath}/user/create">用户新增</a><br/>
2. <table border="1" width="50%">
3. <tr>
4. <th>用户名</th>
5. <th>真实姓名</th>
6. <th>操作</th>
7. </tr>
8. <c:forEach items="${userList}" var="user">
9. <tr>
10. <td>${user.username }</td>
11. <td>${user.realname }</td>
12. <td>
13. <a href="${pageContext.request.contextPath}/user/update?username=${user.username}">更新</a>
14. |
15. <a href="${pageContext.request.contextPath}/user/delete?username=${user.username}">删除</a>
16. </td>
17. </tr>
18. </c:forEach>
19. </table>

**（3.2、update页面（WEB-INF/jsp/user/update.jsp）**

**Java代码  [收藏代码](javascript:void())**

1. <form action="${pageContext.request.contextPath}/user/update" method="post">
2. 用户名： <input type="text" name="username" value="${command.username}"/><br/>
3. 真实姓名：<input type="text" name="realname" value="${command.realname}"/><br/>
4. <input type="submit" value="更新"/>
5. </form>

**（4、测试：**

**默认的InternalPathMethodNameResolver将进行如下解析：**

http://localhost:9080/springmvc-chapter4/user/list————>list方法名；

http://localhost:9080/springmvc-chapter4/user/create————>create方法名；

http://localhost:9080/springmvc-chapter4/user/update————>update功能处理方法名；

http://localhost:9080/springmvc-chapter4/user/delete————>delete功能处理方法名。

**我们可以将默认的InternalPathMethodNameResolver改为PropertiesMethodNameResolver：**

**Java代码  [收藏代码](javascript:void())**

1. <bean id="propertiesMethodNameResolver"
2. **class**="org.springframework.web.servlet.mvc.multiaction.PropertiesMethodNameResolver">
3. <property name="mappings">
4. <props>
5. <prop key="/user/create">create</prop>
6. <prop key="<span style="font-size: 1em; line-height: 1.5;">/user/</span><span style="font-size: 1em; line-height: 1.5;">update">update</prop></span>
7. <prop key="<span style="font-size: 1em; line-height: 1.5;">/user/</span><span style="font-size: 1em; line-height: 1.5;">delete">delete</prop></span>
8. <prop key="<span style="font-size: 1em; line-height: 1.5;">/user/</span><span style="font-size: 1em; line-height: 1.5;">list">list</prop></span>
9. <prop key="/\*\*">list</prop><!-- 默认的行为 -->
10. </props>
11. </property>
12. <property name="alwaysUseFullPath" value="false"/><!-- 不使用全路径 -->
13. </bean>
14. <bean name="/user/\*\*" **class**="cn.javass.chapter4.web.controller.UserController">
15. <!—省略其他配置，详见配置文件-->
16. <!-- 使用PropertiesMethodNameResolver来解析功能处理方法名 -->
17. <property name="methodNameResolver" ref="propertiesMethodNameResolver"/>
18. </bean>

**/\*\*表示默认解析到list功能处理方法。**

如上配置方式可以很好的工作，但必须继承MultiActionController，Spring Web MVC提供给我们无需继承MultiActionController实现方式，即使有委托对象方式，继续往下看吧。

#### 4.15.7、委托方式实现

**（1、控制器UserDelegate**

   将UserController复制一份，改名为UserDelegate，并把继承MultiActionController去掉即可，其他无需改变。

**（2、spring配置文件chapter4-servlet.xml**

**Java代码  [收藏代码](javascript:void())**

1. <!—委托对象-->
2. <bean id="userDelegate" **class**="cn.javass.chapter4.web.controller.UserDelegate">
3. <property name="userService" ref="userService"/>
4. <property name="createView" value="user2/create"/>
5. <property name="updateView" value="user2/update"/>
6. <property name="deleteView" value="user2/delete"/>
7. <property name="listView" value="user2/list"/>
8. <property name="redirectToListView" value="redirect:/user2/list"/>
9. </bean>
10. <!—控制器对象-->
11. <bean name="/user2/\*\*"
12. **class**="org.springframework.web.servlet.mvc.multiaction.MultiActionController">
13. <property name="delegate" ref="userDelegate"/>
14. <property name="methodNameResolver" ref="parameterMethodNameResolver"/>
15. </bean>

**delegate：**控制器对象通过delegate属性指定委托对象，即实际调用delegate委托对象的功能方法。

**methodNameResolver：**此处我们使用ParameterMethodNameResolver解析器；

**Java代码  [收藏代码](javascript:void())**

1. <!—ParameterMethodNameResolver -->
2. <bean id="parameterMethodNameResolver"
3. **class**="org.springframework.web.servlet.mvc.multiaction.ParameterMethodNameResolver">
4. <!-- 1、根据请求参数名解析功能方法名 -->
5. <property name="methodParamNames" value="create,update,delete"/>
6. <!-- 2、根据请求参数名的值解析功能方法名 -->
7. <property name="paramName" value="action"/>
8. <!-- 3、逻辑方法名到真实方法名的映射 -->
9. <property name="logicalMappings">
10. <props>
11. <prop key="doList">list</prop>
12. </props>
13. </property>
14. <!—4、默认执行的功能处理方法 -->
15. <property name="defaultMethodName" value="list"/>
16. </bean>

**1、methodParamNames：**create,update,delete，当请求中有参数名为这三个的将被映射为功能方法名，如“<input type="submit" name="create" value="新增"/>”提交后解析得到的功能方法名为create;

**2、paramName：**当请求中有参数名为action，则将值映射为功能方法名，如“<input type="hidden"name="action" value="delete"/>”，提交后解析得到的功能方法名为delete；

**3、logicalMappings：**逻辑功能方法名到真实功能方法名的映射，如：

    http://localhost:9080/springmvc-chapter4/user2?action=doList；

    首先请求参数“action=doList”，则第二步解析得到逻辑功能方法名为doList；

    本步骤会把doList再转换为真实的功能方法名list。

**4、defaultMethodName：**以上步骤如果没有解析到功能处理方法名，默认执行的方法名。

**（3、视图页面**

**（3.1、list页面（WEB-INF/jsp/user2/list.jsp）**

**Java代码  [收藏代码](javascript:void())**

1. <a href="${pageContext.request.contextPath}/user2?action=create">用户新增</a><br/>
2. <table border="1" width="50%">
3. <tr>
4. <th>用户名</th>
5. <th>真实姓名</th>
6. <th>操作</th>
7. </tr>
8. <c:forEach items="${userList}" var="user">
9. <tr>
10. <td>${user.username }</td>
11. <td>${user.realname }</td>
12. <td>
13. <a href="${pageContext.request.contextPath}/user2?action=update&username=${user.username}">更新</a>
14. |
15. <a href="${pageContext.request.contextPath}/user2?action=delete&username=${user.username}">删除</a>
16. </td>
17. </tr>
18. </c:forEach>
19. </table>

**（3.2、update页面（WEB-INF/jsp/user2/update.jsp）**

**Java代码  [收藏代码](javascript:void())**

1. <form action="${pageContext.request.contextPath}/user2" method="post">
2. <input type="hidden" name="action" value="update"/>
3. 用户名： <input type="text" name="username" value="${command.username}"/><br/>
4. 真实姓名：<input type="text" name="realname" value="${command.realname}"/><br/>
5. <input type="submit" value="更新"/>
6. </form>

  通过参数name="action" value="update"来指定要执行的功能方法名update。

**（3.3、create页面（WEB-INF/jsp/user2/create.jsp）**

**Java代码  [收藏代码](javascript:void())**

1. <form action="${pageContext.request.contextPath}/user2" method="post">
2. 用户名： <input type="text" name="username" value="${command.username}"/><br/>
3. 真实姓名：<input type="text" name="realname" value="${command.realname}"/><br/>
4. <input type="submit" name="create" value="新增"/>
5. </form>

  通过参数name="create"来指定要执行的功能方法名create。

**（4、测试：**

**使用ParameterMethodNameResolver将进行如下解析：**

http://localhost:9080/springmvc-chapter4/user2?create      ————>create功能处理方法名（参数名映射）；

http://localhost:9080/springmvc-chapter4/user2?action=create————>create功能处理方法名（参数值映射）；

http://localhost:9080/springmvc-chapter4/user2?update      ————>update功能处理方法名；

http://localhost:9080/springmvc-chapter4/user2?action=update————>update功能处理方法名；

http://localhost:9080/springmvc-chapter4/user2?delete      ————>delete功能处理方法名；

http://localhost:9080/springmvc-chapter4/user2?action=delete————>delete功能处理方法名；

http://localhost:9080/springmvc-chapter4/user2?doList      ————>通过logicalMappings解析为list功能处理方法。

http://localhost:9080/springmvc-chapter4/user2?action=doList————>通过logicalMappings解析为list功能处理方法。

http://localhost:9080/springmvc-chapter4/user2————>默认的功能处理方法名list（默认）。

## 4.16、数据类型转换和数据验证



**流程：**

1、首先创建数据绑定器，在此此会创建ServletRequestDataBinder类的对象，并设置messageCodesResolver（错误码解析器）；

2、提供第一个扩展点，初始化数据绑定器，在此处我们可以覆盖该方法注册自定义的PropertyEditor（请求参数——>命令对象属性的转换）；

3、进行数据绑定，即请求参数——>命令对象的绑定；

4、提供第二个扩展点，数据绑定完成后的扩展点，此处可以实现一些自定义的绑定动作；

5、验证器对象的验证，验证器通过validators注入，如果验证失败，需要把错误信息放入Errors（此处使用BindException实现）；

6、提供第三个扩展点，此处可以实现自定义的绑定/验证逻辑；

7、将errors传入功能处理方法进行处理，功能方法应该判断该错误对象是否有错误进行相应的处理。

### 4.16.1、数据类型转换

请求参数（String）——>命令对象属性（可能是任意类型）的类型转换，即数据绑定时的类型转换，使用PropertyEditor实现绑定时的类型转换。

**一、Spring内建的PropertyEditor如下所示：**

|  |  |  |
| --- | --- | --- |
| **类名** | **说明** | **默认是否注册** |
| ByteArrayPropertyEditor | String<——>byte[] | √ |
| ClassEditor | String<——>Class  当类没有发现抛出IllegalArgumentException | √ |
| CustomBooleanEditor | String<——>Boolean  true/yes/on/1转换为true，false/no/off/0转换为false | √ |
| CustomCollectionEditor | 数组/Collection——>Collection  普通值——>Collection（只包含一个对象）  如String——>Collection  不允许Collection——>String（单方向转换） | √ |
| CustomNumberEditor | String<——>Number(Integer、Long、Double) | √ |
| FileEditor | String<——>File | √ |
| InputStreamEditor | String——>InputStream  单向的，不能InputStream——>String | √ |
| LocaleEditor | String<——>Locale，  （String的形式为[语言]\_[国家]\_[变量]，这与Local对象的toString()方法得到的结果相同） | √ |
| PatternEditor | String<——>Pattern | √ |
| PropertiesEditor | String<——>java.lang.Properties | √ |
| URLEditor | String<——>URL | √ |
| StringTrimmerEditor | 一个用于trim 的 String类型的属性编辑器  如默认删除两边的空格，charsToDelete属性：可以设置为其他字符  emptyAsNull属性：将一个空字符串转化为null值的选项。 | × |
| CustomDateEditor | String<——>java.util.Date | × |

**二、Spring内建的PropertyEditor支持的属性（符合JavaBean规范）操作：**

|  |  |
| --- | --- |
| **表达式** | **设值/取值说明** |
| username | 属性username  设值方法setUsername()/取值方法getUsername() 或 isUsername() |
| schooInfo.schoolType | 属性schooInfo的嵌套属性schoolType  设值方法getSchooInfo().setSchoolType()/取值方法getSchooInfo().getSchoolType() |
| hobbyList[0] | 属性hobbyList的第一个元素  索引属性可能是一个数组、列表、其它天然有序的容器。 |
| map[key] | 属性map（java.util.Map类型）  map中key对应的值 |

**三、示例：**

接下来我们写自定义的属性编辑器进行数据绑定：

**（1、模型对象：**

**java代码：**

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.javass.chapter4.model;
2. //省略import
3. **public** **class** DataBinderTestModel {
4. **private** String username;
5. **private** **boolean** bool;//Boolean值测试
6. **private** SchoolInfoModel schooInfo;
7. **private** List hobbyList;//集合测试，此处可以改为数组/Set进行测试
8. **private** Map map;//Map测试
9. **private** PhoneNumberModel phoneNumber;//String->自定义对象的转换测试
10. **private** Date date;//日期类型测试
11. **private** UserState state;//String——>Enum类型转换测试
12. //省略getter/setter
13. }
15. **package** cn.javass.chapter4.model;
16. //如格式010-12345678
17. **public** **class** PhoneNumberModel {
18. **private** String areaCode;//区号
19. **private** String phoneNumber;//电话号码
20. //省略getter/setter
21. }

**(2、PhoneNumber属性编辑器**

前台输入如010-12345678自动转换为PhoneNumberModel。

**java代码：**

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.javass.chapter4.web.controller.support.editor;
2. //省略import
3. **public** **class** PhoneNumberEditor **extends** PropertyEditorSupport {
4. Pattern pattern = Pattern.compile("^(\\d{3,4})-(\\d{7,8})$");
5. @Override
6. **public** **void** setAsText(String text) **throws** IllegalArgumentException {
7. **if**(text == **null** || !StringUtils.hasLength(text)) {
8. setValue(**null**); //如果没值，设值为null
9. }
10. Matcher matcher = pattern.matcher(text);
11. **if**(matcher.matches()) {
12. PhoneNumberModel phoneNumber = **new** PhoneNumberModel();
13. phoneNumber.setAreaCode(matcher.group(1));
14. phoneNumber.setPhoneNumber(matcher.group(2));
15. setValue(phoneNumber);
16. } **else** {
17. **throw** **new** IllegalArgumentException(String.format("类型转换失败，需要格式[010-12345678]，但格式是[%s]", text));
18. }
19. }
20. @Override
21. **public** String getAsText() {
22. PhoneNumberModel phoneNumber = ((PhoneNumberModel)getValue());
23. **return** phoneNumber == **null** ? "" : phoneNumber.getAreaCode() + "-" + phoneNumber.getPhoneNumber();
24. }
25. }

**PropertyEditorSupport：**一个PropertyEditor的支持类；

**setAsText：**表示将String——>PhoneNumberModel，根据正则表达式进行转换，如果转换失败抛出异常，则接下来的验证器会进行验证处理；

**getAsText：**表示将PhoneNumberModel——>String。

**（3、控制器**

需要在控制器注册我们自定义的属性编辑器。

此处我们使用AbstractCommandController，因为它继承了BaseCommandController，拥有绑定流程。

**java代码：**

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.javass.chapter4.web.controller;
2. //省略import
3. **public** **class** DataBinderTestController **extends** AbstractCommandController {
4. **public** DataBinderTestController() {
5. setCommandClass(DataBinderTestModel.**class**); //设置命令对象
6. setCommandName("dataBinderTest");//设置命令对象的名字
7. }
8. @Override
9. **protected** ModelAndView handle(HttpServletRequest req, HttpServletResponse resp, Object command, BindException errors) **throws** Exception {
10. //输出command对象看看是否绑定正确
11. System.out.println(command);
12. **return** **new** ModelAndView("bindAndValidate/success").addObject("dataBinderTest", command);
13. }
14. @Override
15. **protected** **void** initBinder(HttpServletRequest request, ServletRequestDataBinder binder) **throws** Exception {
16. **super**.initBinder(request, binder);
17. //注册自定义的属性编辑器
18. //1、日期
19. DateFormat df = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
20. CustomDateEditor dateEditor = **new** CustomDateEditor(df, **true**);
21. //表示如果命令对象有Date类型的属性，将使用该属性编辑器进行类型转换
22. binder.registerCustomEditor(Date.**class**, dateEditor);
23. //自定义的电话号码编辑器
24. binder.registerCustomEditor(PhoneNumberModel.**class**, **new** PhoneNumberEditor());
25. }
26. }

**initBinder:**第一个扩展点，初始化数据绑定器，在此处我们注册了两个属性编辑器；

**CustomDateEditor：**自定义的日期编辑器，用于在String<——>日期之间转换；

    binder.registerCustomEditor(Date.**class**, dateEditor)：表示如果命令对象是Date类型，则使用dateEditor进行类型转换；

**PhoneNumberEditor：**自定义的电话号码属性编辑器用于在String<——> PhoneNumberModel之间转换；

    binder.registerCustomEditor(PhoneNumberModel.**class**, **new**PhoneNumberEditor())：表示如果命令对象是PhoneNumberModel类型，则使用PhoneNumberEditor进行类型转换；

**（4、spring配置文件chapter4-servlet.xml**

**java代码：**

**Java代码  [收藏代码](javascript:void())**

1. <bean name="/dataBind"
2. **class**="cn.javass.chapter4.web.controller.DataBinderTestController"/>

**（5、视图页面（WEB-INF/jsp/bindAndValidate/success.jsp）**

**java代码：**

**Java代码  [收藏代码](javascript:void())**

1. EL phoneNumber:${dataBinderTest.phoneNumber}<br/>
2. EL state:${dataBinderTest.state}<br/>
3. EL date:${dataBinderTest.date}<br/>

视图页面的数据没有预期被格式化，如何进行格式化显示呢？请参考【第七章  注解式控制器的数据验证、类型转换及格式化】。

**（6、测试：**

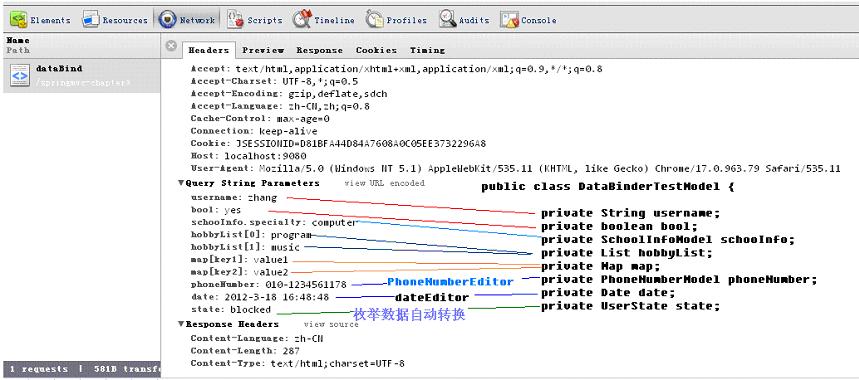
**1、在浏览器地址栏输入请求的URL，如**

http://localhost:9080/springmvc-chapter4/dataBind?username=zhang&bool=yes&schooInfo.specialty=computer&hobbyList[0]=program&hobbyList[1]=music&map[key1]=value1&map[key2]=value2&phoneNumber=010-12345678&date=2012-3-18 16:48:48&state=blocked

**2、控制器输出的内容：**

DataBinderTestModel [username=zhang, bool=true, schooInfo=SchoolInfoModel [schoolType=null, schoolName=null, specialty=computer], hobbyList=[program, music], map={key1=value1, key2=value2}, phoneNumber=PhoneNumberModel [areaCode=010, phoneNumber=12345678], date=Sun Mar 18 16:48:48 CST 2012, state=锁定]

类型转换如图所示：



**四、注册PropertyEditor**

**1、使用WebDataBinder进行控制器级别注册PropertyEditor（控制器独享）**

如“【三、示例】”中所使用的方式，使用WebDataBinder注册控制器级别的PropertyEditor，这种方式注册的PropertyEditor只对当前控制器独享，即其他的控制器不会自动注册这个PropertyEditor，如果需要还需要再注册一下。

**2、使用WebBindingInitializer批量注册PropertyEditor**

如果想在多个控制器同时注册多个相同的PropertyEditor时，可以考虑使用WebBindingInitializer。

示例：

（1、实现WebBindingInitializer

**java代码：**

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.javass.chapter4.web.controller.support.initializer;
2. //省略import
3. **public** **class** MyWebBindingInitializer **implements** WebBindingInitializer {
4. @Override
5. **public** **void** initBinder(WebDataBinder binder, WebRequest request) {
6. //注册自定义的属性编辑器
7. //1、日期
8. DateFormat df = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
9. CustomDateEditor dateEditor = **new** CustomDateEditor(df, **true**);
10. //表示如果命令对象有Date类型的属性，将使用该属性编辑器进行类型转换
11. binder.registerCustomEditor(Date.**class**, dateEditor);
12. //自定义的电话号码编辑器
13. binder.registerCustomEditor(PhoneNumberModel.**class**, **new** PhoneNumberEditor());
14. }
15. }

通过实现WebBindingInitializer并通过binder注册多个PropertyEditor。

（2、修改【三、示例】中的DataBinderTestController，注释掉initBinder方法；

（3、修改chapter4-servlet.xml配置文件：

**java代码：**

**Java代码  [收藏代码](javascript:void())**

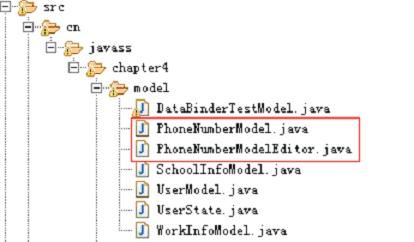
1. <!-- 注册WebBindingInitializer实现 -->
2. <bean id="myWebBindingInitializer" **class**="cn.javass.chapter4.web.controller.support.initializer.MyWebBindingInitializer"/>
3. <bean name="/dataBind" **class**="cn.javass.chapter4.web.controller.DataBinderTestController">
4. <!-- 注入WebBindingInitializer实现 -->
5. <property name="webBindingInitializer" ref="myWebBindingInitializer"/>
6. </bean>

（4、尝试访问“【三、示例】”中的测试URL即可成功。

使用WebBindingInitializer的好处是当你需要在多个控制器中需要同时使用多个相同的PropertyEditor可以在WebBindingInitializer实现中注册，这样只需要在控制器中注入WebBindingInitializer即可注入多个PropertyEditor。

**3、全局级别注册PropertyEditor（全局共享）**

只需要将我们自定义的PropertyEditor放在和你的模型类同包下即可，且你的Editor命名规则必须是“模型类名Editor”，这样Spring会自动使用标准JavaBean架构进行自动识别，如图所示：



此时我们把“DataBinderTestController”的“binder.registerCustomEditor(PhoneNumberModel.class, new PhoneNumberEditor());”注释掉，再尝试访问“【三、示例】”中的测试URL即可成功。

这种方式不仅仅在使用Spring时可用，在标准的JavaBean等环境都是可用的，可以认为是全局共享的（不仅仅是Spring环境）。

**PropertyEditor被限制为只能String<——>Object之间转换，不能Object<——>Object，Spring3提供了更强大的类型转换（TypeConversion）支持，它可以在任意对象之间进行类型转换，不仅仅是String<——>Object。**

如果我在地址栏输入错误的数据，即数据绑定失败，Spring Web MVC该如何处理呢？如果我输入的数据不合法呢？如用户名输入100个字符（超长了）那又该怎么处理呢？出错了需要错误消息，那错误消息应该是硬编码？还是可配置呢？

接下来我们来学习一下数据验证器进行数据验证吧。