## 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）。

还记得DispatcherServlet吗？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

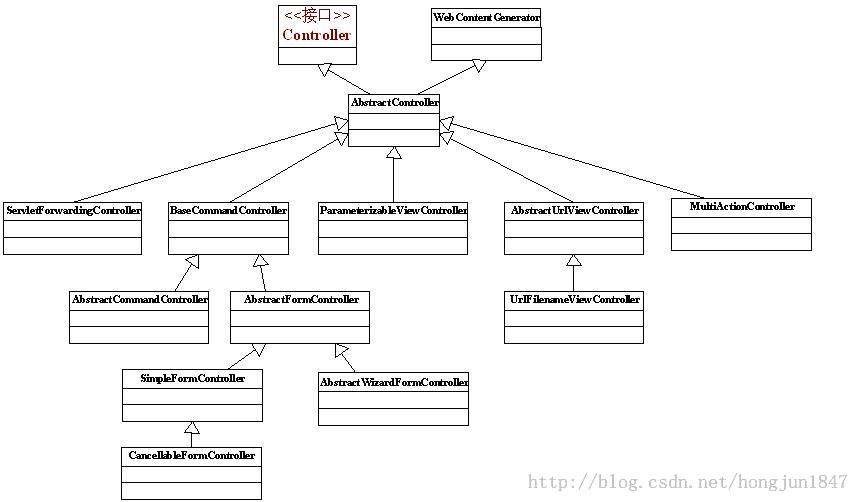
**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20534891) [copy](http://blog.csdn.net/hongjun1847/article/details/20534891)

[在CODE上查看代码片](https://code.csdn.net/snippets/218282)

1. **package** org.springframework.web.servlet.mvc;
2. **public** **interface** Controller {
3. ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) **throws** Exception;
4. }

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）。

还记得第二章的HelloWorld吗？我们的HelloWorldController实现Controller接口，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：



## 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）。

该抽象类默认被AbstractController和WebContentInterceptor继承。

## 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。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20534891) [copy](http://blog.csdn.net/hongjun1847/article/details/20534891)

[在CODE上查看代码片](https://code.csdn.net/snippets/218282)

1. **public** ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) **throws** Exception {
2. //委托给WebContentGenerator进行缓存控制
3. checkAndPrepare(request, response, **this** **instanceof** LastModified);
4. //当前会话是否应串行化访问.
5. **if** (**this**.synchronizeOnSession) {
6. HttpSession session = request.getSession(**false**);
7. **if** (session != **null**) {
8. Object mutex = WebUtils.getSessionMutex(session);
9. **synchronized** (mutex) {
10. **return** handleRequestInternal(request, response);
11. }
12. }
13. }
14. **return** handleRequestInternal(request, response);
15. }

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。

AbstractController使用方法：

首先让我们使用AbstractController来重写第二章的HelloWorldController：

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20534891) [copy](http://blog.csdn.net/hongjun1847/article/details/20534891)

[在CODE上查看代码片](https://code.csdn.net/snippets/218282)

1. **public** **class** HelloWorldController **extends** AbstractController {
2. @Override
3. **protected** ModelAndView handleRequestInternal(HttpServletRequest req, HttpServletResponse resp) **throws** Exception {
4. //1、收集参数
5. //2、绑定参数到命令对象
6. //3、调用业务对象
7. //4、选择下一个页面
8. ModelAndView mv = **new** ModelAndView();
9. //添加模型数据 可以是任意的POJO对象
10. mv.addObject("message", "Hello World!");
11. //设置逻辑视图名，视图解析器会根据该名字解析到具体的视图页面
12. mv.setViewName("hello");
13. **return** mv;
14. }
15. }

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20534891) [copy](http://blog.csdn.net/hongjun1847/article/details/20534891)

[在CODE上查看代码片](https://code.csdn.net/snippets/218282)

1. <!— 在chapter4-servlet.xml配置处理器 --**>**
2. **<bean** name="/hello" class="cn.javass.chapter4.web.controller.HelloWorldController"**/>**

从如上代码我们可以看出：

1、继承AbstractController

2、实现handleRequestInternal方法即可。

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：

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20534891) [copy](http://blog.csdn.net/hongjun1847/article/details/20534891)

[在CODE上查看代码片](https://code.csdn.net/snippets/218282)

1. **public** **class** HelloWorldWithoutReturnModelAndViewController **extends** AbstractController {
2. @Override
3. **protected** ModelAndView handleRequestInternal(HttpServletRequest req, HttpServletResponse resp) **throws** Exception {
5. resp.getWriter().write("Hello World!!");
6. //如果想直接在该处理器/控制器写响应 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
7. **return** **null**;
8. }
9. }

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20534891) [copy](http://blog.csdn.net/hongjun1847/article/details/20534891)

[在CODE上查看代码片](https://code.csdn.net/snippets/218282)

1. <!— 在chapter4-servlet.xml配置处理器 --**>**
2. **<bean** name="/helloWithoutReturnModelAndView" class="cn.javass.chapter4.web.controller.HelloWorldWithoutReturnModelAndViewController"**/>**

从如上代码可以看出如果想直接在控制器写出响应，只需要通过response写出，并返回null即可。

强制请求方法类型：

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20534891) [copy](http://blog.csdn.net/hongjun1847/article/details/20534891)

[在CODE上查看代码片](https://code.csdn.net/snippets/218282)

1. <!— 在chapter4-servlet.xml配置处理器 --**>**
2. **<bean** name="/helloWithPOST" class="cn.javass.chapter4.web.controller.HelloWorldController"**>**
3. **<property** name="supportedMethods" value="POST"**></property>**
4. **</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”。

比如注册/登录可能只允许POST请求。

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：

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20534891) [copy](http://blog.csdn.net/hongjun1847/article/details/20534891)

[在CODE上查看代码片](https://code.csdn.net/snippets/218282)

1. <!— 在chapter4-servlet.xml配置处理器 --**>**
2. **<bean** name="/helloRequireSession"
3. class="cn.javass.chapter4.web.controller.HelloWorldController"**>**
4. **<property** name="requireSession" value="true"**/>**
5. **</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。

客户端端缓存控制：

1、缓存5秒，cacheSeconds=5

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20534891) [copy](http://blog.csdn.net/hongjun1847/article/details/20534891)

[在CODE上查看代码片](https://code.csdn.net/snippets/218282)

1. **package** cn.javass.chapter4.web.controller;
2. //省略import
3. **public** **class** HelloWorldCacheController **extends** AbstractController {
4. @Override
5. **protected** ModelAndView handleRequestInternal(HttpServletRequest req, HttpServletResponse resp) **throws** Exception {
7. //点击后再次请求当前页面
8. resp.getWriter().write("<a href=''>this</a>");
9. **return** **null**;
10. }
11. }

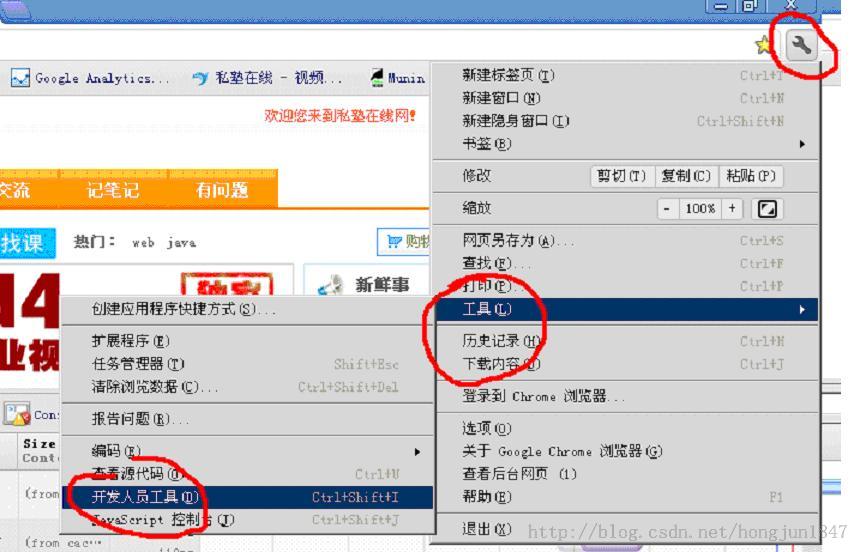
**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20534891) [copy](http://blog.csdn.net/hongjun1847/article/details/20534891)

[在CODE上查看代码片](https://code.csdn.net/snippets/218282)

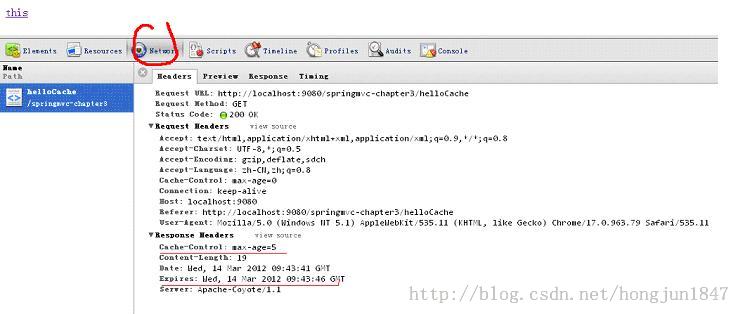
1. <!— 在chapter4-servlet.xml配置处理器 --**>**
2. **<bean** name="/helloCache"
3. class="cn.javass.chapter4.web.controller.HelloWorldCacheController"**>**
4. **<property** name="cacheSeconds" value="5"**/>**
5. **</bean>**

如上配置表示告诉浏览器缓存5秒钟：

开启chrome浏览器调试工具：



服务器返回的响应头如下所示：



添加了“Expires:Wed, 14 Mar 2012 09:38:32 GMT” 和“Cache-Control:max-age=5” 表示允许客户端缓存5秒，当你点“this”链接时，会发现如下：



而且服务器也没有收到请求，当过了5秒后，你再点“this”链接会发现又重新请求服务器下载新数据。

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，在未过期情况下一次请求也不发送）；

    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；

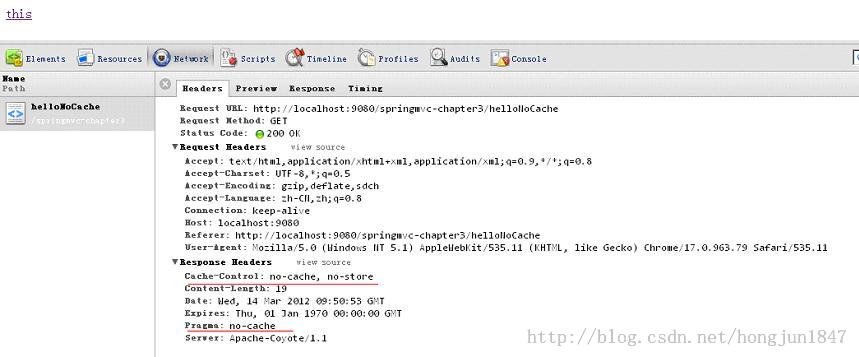
2、不缓存，cacheSeconds=0

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20534891) [copy](http://blog.csdn.net/hongjun1847/article/details/20534891)

[在CODE上查看代码片](https://code.csdn.net/snippets/218282)

1. <!— 在chapter4-servlet.xml配置处理器 --**>**
2. **<bean** name="/helloNoCache"
3. class="cn.javass.chapter4.web.controller.HelloWorldCacheController"**>**
4. **<property** name="cacheSeconds" value="0"**/>**
5. **</bean>**

以上配置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：

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20534891) [copy](http://blog.csdn.net/hongjun1847/article/details/20534891)

[在CODE上查看代码片](https://code.csdn.net/snippets/218282)

1. **package** cn.javass.chapter4.web.controller;
2. **public** **class** HelloWorldLastModifiedCacheController **extends** AbstractController **implements** LastModified {
3. **private** **long** lastModified;
4. **protected** ModelAndView handleRequestInternal(HttpServletRequest req, HttpServletResponse resp) **throws** Exception {
5. //点击后再次请求当前页面
6. resp.getWriter().write("<a href=''>this</a>");
7. **return** **null**;
8. }
9. **public** **long** getLastModified(HttpServletRequest request) {
10. **if**(lastModified == 0L) {
11. //TODO 此处更新的条件：如果内容有更新，应该重新返回内容最新修改的时间戳
12. lastModified = System.currentTimeMillis();
13. }
14. **return** lastModified;
15. }
16. }

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20534891) [copy](http://blog.csdn.net/hongjun1847/article/details/20534891)

[在CODE上查看代码片](https://code.csdn.net/snippets/218282)

1. <!— 在chapter4-servlet.xml配置处理器 --**>**
2. **<bean** name="/helloLastModified"
3. class="cn.javass.chapter4.web.controller.HelloWorldLastModifiedCacheController"**/>**

HelloWorldLastModifiedCacheController只需要实现LastModified接口的getLastModified方法，保证当内容发生改变时返回最新的修改时间即可。

分析：

（1、发送请求到服务器，如（[http://localhost:9080/springmvc-chapter4/helloLastModified](http://localhost:9080/springmvc-chapter3/helloLastModified)），则服务器返回的响应为：



（2、再次按F5刷新客户端，返回状态码304表示服务器没有更新过：



（3、重启服务器，再次刷新，会看到200状态码（因为服务器的lastModified时间变了）。

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：

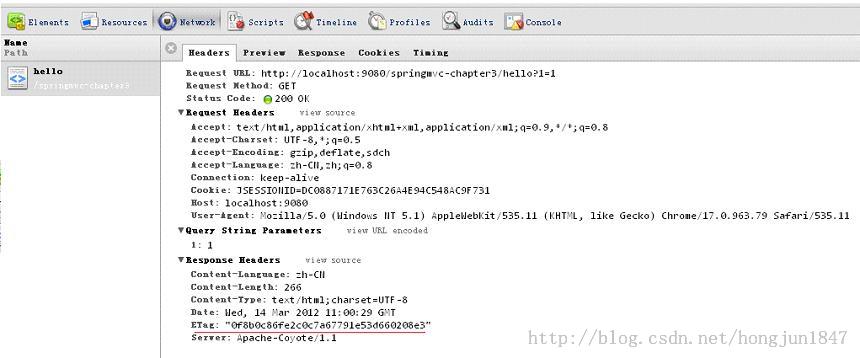
**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20534891) [copy](http://blog.csdn.net/hongjun1847/article/details/20534891)

[在CODE上查看代码片](https://code.csdn.net/snippets/218282)

1. **<filter>**
2. **<filter-name>**etagFilter**</filter-name>**
3. **<filter-class>**org.springframework.web.filter.ShallowEtagHeaderFilter**</filter-class>**
4. **</filter>**
5. **<filter-mapping>**
6. **<filter-name>**etagFilter**</filter-name>**
7. **<servlet-name>**chapter4**</servlet-name>**
8. **</filter-mapping>**
10. 此过滤器只过滤到我们DispatcherServlet的请求。

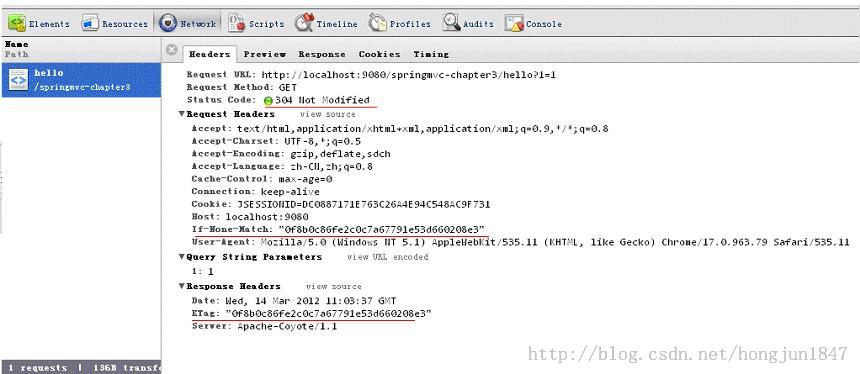
分析：

1）：发送请求到服务器：“[http://localhost:9080/springmvc-chapter4/hello](http://localhost:9080/springmvc-chapter3/hello)”，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？

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20534891) [copy](http://blog.csdn.net/hongjun1847/article/details/20534891)

[在CODE上查看代码片](https://code.csdn.net/snippets/218282)

1. **protected** String generateETagHeaderValue(**byte**[] bytes) {
2. StringBuilder builder = **new** StringBuilder("\"0");
3. DigestUtils.appendMd5DigestAsHex(bytes, builder);
4. builder.append('"');
5. **return** builder.toString();
6. }

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/>（中文版<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：

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20539723) [copy](http://blog.csdn.net/hongjun1847/article/details/20539723)

[在CODE上查看代码片](https://code.csdn.net/snippets/218419)

1. **package** cn.javass.chapter4.web.servlet;
2. **public** **class** ForwardingServlet **extends** HttpServlet {
3. @Override
4. **protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp)
5. **throws** ServletException, IOException {
7. resp.getWriter().write("Controller forward to Servlet");
9. }
10. }

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20539723) [copy](http://blog.csdn.net/hongjun1847/article/details/20539723)

[在CODE上查看代码片](https://code.csdn.net/snippets/218419)

1. **<servlet>**
2. **<servlet-name>**forwarding**</servlet-name>**
3. **<servlet-class>**cn.javass.chapter4.web.servlet.ForwardingServlet**</servlet-class>**
4. **</servlet>**

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20539723) [copy](http://blog.csdn.net/hongjun1847/article/details/20539723)

[在CODE上查看代码片](https://code.csdn.net/snippets/218419)

1. <!— 在chapter4-servlet.xml配置处理器 --**>**
2. **<bean** name="/forwardToServlet"
3. class="org.springframework.web.servlet.mvc.ServletForwardingController"**>**
4. **<property** name="servletName" value="forwarding"**></property>**
5. **</bean>**

当我们请求/forwardToServlet时，会被转发到名字为“forwarding”的servlet处理，该sevlet的servlet-mapping标签配置是可选的。

## 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，非GoF里的命令设计模式），这里的命令对象是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。

## 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。

1、创建命令类（就是普通的JavaBean类/POJO）

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20539723) [copy](http://blog.csdn.net/hongjun1847/article/details/20539723)

[在CODE上查看代码片](https://code.csdn.net/snippets/218419)

1. **package** cn.javass.chapter4.model;
2. **public** **class** UserModel {
3. **private** String username;
4. **private** String password;
5. //省略setter/getter
6. }

2、实现控制器

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20539723) [copy](http://blog.csdn.net/hongjun1847/article/details/20539723)

[在CODE上查看代码片](https://code.csdn.net/snippets/218419)

1. **package** cn.javass.chapter4.web.controller;
2. //省略import
3. **public** **class** MyAbstractCommandController **extends** AbstractCommandController {
4. **public** MyAbstractCommandController() {
5. //设置命令对象实现类
6. setCommandClass(UserModel.**class**);
7. }
8. @Override
9. **protected** ModelAndView handle(HttpServletRequest req, HttpServletResponse resp, Object command, BindException errors) **throws** Exception {
10. //将命令对象转换为实际类型
11. UserModel user = (UserModel) command;
12. ModelAndView mv = **new** ModelAndView();
13. mv.setViewName("abstractCommand");
14. mv.addObject("user", user);
15. **return** mv;
16. }
17. }

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20539723) [copy](http://blog.csdn.net/hongjun1847/article/details/20539723)

[在CODE上查看代码片](https://code.csdn.net/snippets/218419)

1. <!— 在chapter4-servlet.xml配置处理器 --**>**
2. **<bean** name="/abstractCommand"
3. class="cn.javass.chapter4.web.controller.MyAbstractCommandController"**>**
4. <!-- 也可以通过依赖注入 注入命令实现类 -->
5. <!-- property name="commandClass" value="cn.javass.chapter4.model.UserModel"/-->
6. **</bean>**

**[plain]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20539723) [copy](http://blog.csdn.net/hongjun1847/article/details/20539723)

[在CODE上查看代码片](https://code.csdn.net/snippets/218419)

1. <!— WEB-INF/jsp/abstractCommand.jsp视图下的主要内容 -->
3. ${user.username }-${user.password }

当我们在浏览器中输入“[http://localhost:9080/springmvc-chapter4/abstractCommand?username=123&password=123](http://localhost:9080/springmvc-chapter3/abstractCommand?username=123&password=123)”，会自动将请求参数username和password绑定到命令对象；绑定时按照JavaBean命名规范绑定；



## 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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20539723) [copy](http://blog.csdn.net/hongjun1847/article/details/20539723)

[在CODE上查看代码片](https://code.csdn.net/snippets/218419)

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20539723) [copy](http://blog.csdn.net/hongjun1847/article/details/20539723)

[在CODE上查看代码片](https://code.csdn.net/snippets/218419)

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：

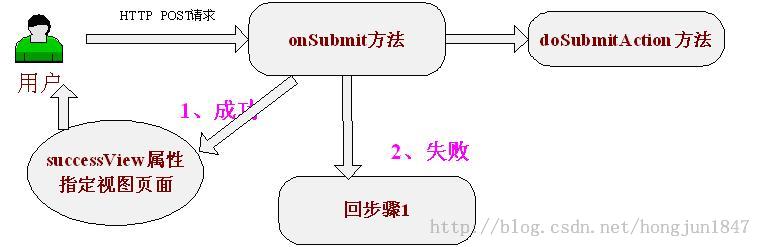
1、准备要展示的数据，并到表单展示页面；

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20539723) [copy](http://blog.csdn.net/hongjun1847/article/details/20539723)

[在CODE上查看代码片](https://code.csdn.net/snippets/218419)

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）

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20539723) [copy](http://blog.csdn.net/hongjun1847/article/details/20539723)

[在CODE上查看代码片](https://code.csdn.net/snippets/218419)

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

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20539723) [copy](http://blog.csdn.net/hongjun1847/article/details/20539723)

[在CODE上查看代码片](https://code.csdn.net/snippets/218419)

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20539723) [copy](http://blog.csdn.net/hongjun1847/article/details/20539723)

[在CODE上查看代码片](https://code.csdn.net/snippets/218419)

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）

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20539723) [copy](http://blog.csdn.net/hongjun1847/article/details/20539723)

[在CODE上查看代码片](https://code.csdn.net/snippets/218419)

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）

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20539723) [copy](http://blog.csdn.net/hongjun1847/article/details/20539723)

[在CODE上查看代码片](https://code.csdn.net/snippets/218419)

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）

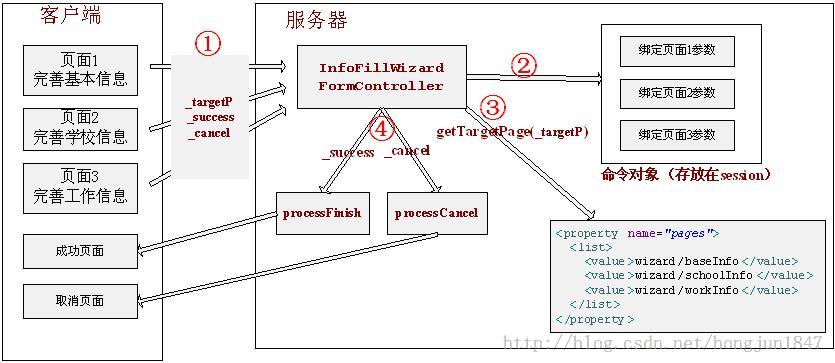
假设现在做一个完善个人信息的功能，分三个页面展示：

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20543119) [copy](http://blog.csdn.net/hongjun1847/article/details/20543119)

[在CODE上查看代码片](https://code.csdn.net/snippets/218502)

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20543119) [copy](http://blog.csdn.net/hongjun1847/article/details/20543119)

[在CODE上查看代码片](https://code.csdn.net/snippets/218502)

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20543119) [copy](http://blog.csdn.net/hongjun1847/article/details/20543119)

[在CODE上查看代码片](https://code.csdn.net/snippets/218502)

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20543119) [copy](http://blog.csdn.net/hongjun1847/article/details/20543119)

[在CODE上查看代码片](https://code.csdn.net/snippets/218502)

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）

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20543119) [copy](http://blog.csdn.net/hongjun1847/article/details/20543119)

[在CODE上查看代码片](https://code.csdn.net/snippets/218502)

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：

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20543119) [copy](http://blog.csdn.net/hongjun1847/article/details/20543119)

[在CODE上查看代码片](https://code.csdn.net/snippets/218502)

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：

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20543119) [copy](http://blog.csdn.net/hongjun1847/article/details/20543119)

[在CODE上查看代码片](https://code.csdn.net/snippets/218502)

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：

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20543119) [copy](http://blog.csdn.net/hongjun1847/article/details/20543119)

[在CODE上查看代码片](https://code.csdn.net/snippets/218502)

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。

## 4.12、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。

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20543119) [copy](http://blog.csdn.net/hongjun1847/article/details/20543119)

[在CODE上查看代码片](https://code.csdn.net/snippets/218502)

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；

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20543119) [copy](http://blog.csdn.net/hongjun1847/article/details/20543119)

[在CODE上查看代码片](https://code.csdn.net/snippets/218502)

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：

**[plain]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20543119) [copy](http://blog.csdn.net/hongjun1847/article/details/20543119)

[在CODE上查看代码片](https://code.csdn.net/snippets/218502)

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20543119) [copy](http://blog.csdn.net/hongjun1847/article/details/20543119)

[在CODE上查看代码片](https://code.csdn.net/snippets/218502)

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

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20543119) [copy](http://blog.csdn.net/hongjun1847/article/details/20543119)

[在CODE上查看代码片](https://code.csdn.net/snippets/218502)

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。

## 4.12、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。

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544045) [copy](http://blog.csdn.net/hongjun1847/article/details/20544045)

[在CODE上查看代码片](https://code.csdn.net/snippets/218513)

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；

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544045) [copy](http://blog.csdn.net/hongjun1847/article/details/20544045)

[在CODE上查看代码片](https://code.csdn.net/snippets/218513)

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：

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544045) [copy](http://blog.csdn.net/hongjun1847/article/details/20544045)

[在CODE上查看代码片](https://code.csdn.net/snippets/218513)

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544045) [copy](http://blog.csdn.net/hongjun1847/article/details/20544045)

[在CODE上查看代码片](https://code.csdn.net/snippets/218513)

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

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544045) [copy](http://blog.csdn.net/hongjun1847/article/details/20544045)

[在CODE上查看代码片](https://code.csdn.net/snippets/218513)

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。

## 4.15、MultiActionController

之前学过的控制器如AbstractCommandController、SimpleFormController等一般对应一个功能处理方法（如新增），如果我要实现比如最简单的用户增删改查（CRUD Create-Read-Update-Delete），那该怎么办呢？

### 4.15.1 解决方案

1、每一个功能对应一个控制器，如果是CRUD则需要四个控制器，但这样我们的控制器会暴增，肯定不可取；

2、使用Spring Web MVC提供的MultiActionController，用于支持在一个控制器里添加多个功能处理方法，即将多个请求的处理方法放置到一个控制器里，这种方式不错。

### 4.15.2 问题

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

Spring Web MVC提供了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实现类，另外还提供了ParameterMethodNameResolver和PropertiesMethodNameResolver，当然我们也可以自己来实现，稍候我们仔细研究下它们是如何工作的。

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]);

哦，原来如此，我们只需要按照如上格式写我们的功能处理方法即可；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，稍候讨论；

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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. //且不能Controller接口的handleRequest(HttpServletRequest request, HttpServletResponse response)，这个方法是由系统调用
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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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. }

接下来，我们看一下MultiActionController如何使用MethodNameResolver来解析请求到功能处理方法的方法名。

### 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）；

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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：

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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：

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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）

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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）

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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：

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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. **<prop** key="/\*\*"**>**list**</prop>**<!-- 默认的行为 -->
10. **</props>**
11. **</property>**
12. **</bean>**
13. **<bean** name="/user/\*\*" class="cn.javass.chapter4.web.controller.UserController"**>**
14. <!—省略其他配置，详见配置文件--**>**
15. <!-- 使用PropertiesMethodNameResolver来解析功能处理方法名 -->
16. **<property** name="methodNameResolver" ref="propertiesMethodNameResolver"**/>**
17. **</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

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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；

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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）

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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）

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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）

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20544423) [copy](http://blog.csdn.net/hongjun1847/article/details/20544423)

[在CODE上查看代码片](https://code.csdn.net/snippets/218542)

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20548623) [copy](http://blog.csdn.net/hongjun1847/article/details/20548623)

[在CODE上查看代码片](https://code.csdn.net/snippets/218681)

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20548623) [copy](http://blog.csdn.net/hongjun1847/article/details/20548623)

[在CODE上查看代码片](https://code.csdn.net/snippets/218681)

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20548623) [copy](http://blog.csdn.net/hongjun1847/article/details/20548623)

[在CODE上查看代码片](https://code.csdn.net/snippets/218681)

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

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20548623) [copy](http://blog.csdn.net/hongjun1847/article/details/20548623)

[在CODE上查看代码片](https://code.csdn.net/snippets/218681)

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

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

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20548623) [copy](http://blog.csdn.net/hongjun1847/article/details/20548623)

[在CODE上查看代码片](https://code.csdn.net/snippets/218681)

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：

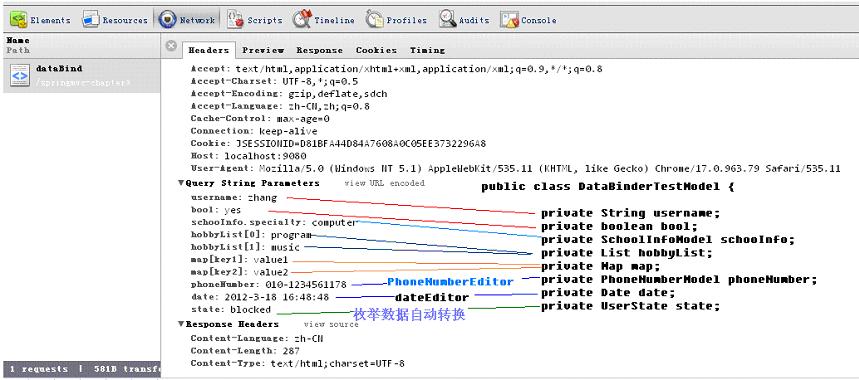
1、在浏览器地址栏输入请求的URL，如

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20548623) [copy](http://blog.csdn.net/hongjun1847/article/details/20548623)

[在CODE上查看代码片](https://code.csdn.net/snippets/218681)

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]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20548623) [copy](http://blog.csdn.net/hongjun1847/article/details/20548623)

[在CODE上查看代码片](https://code.csdn.net/snippets/218681)

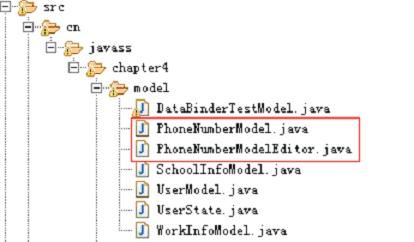
1. <!-- 注册WebBindingInitializer实现 -->
2. <bean id="myWebBindingInitializer" **class**="cn.javass.chapter4.web.controller.support.initializer.MyWebBindingInitializer"/>
3. <bean name="/dataBind" **class**="cn.javass.chapter4.web.controller.DataBinderTestController">
4. <!-- 注入WebBindingInitializer实现 -->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



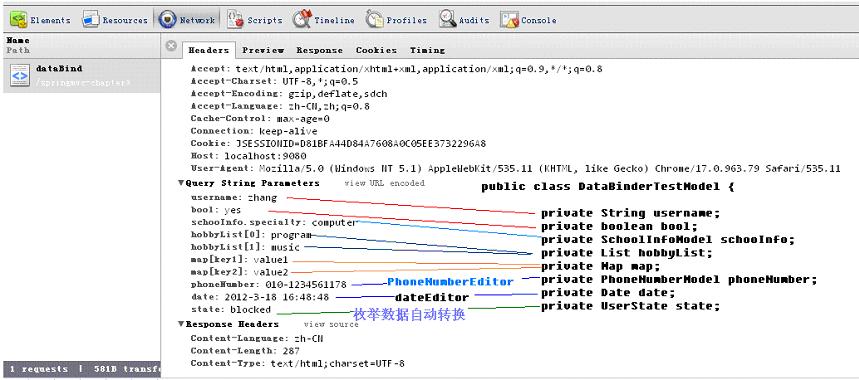
此时我们把“DataBinderTestController”的“binder.registerCustomEditor(PhoneNumberModel.class, new PhoneNumberEditor());”注释掉，再尝试访问“【三、示例】”中的测试URL即可成功。

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（Type Conversion）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。



**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

### 4.16.2、数据验证

1、数据绑定失败：比如需要数字却输入了字母；

2、数据不合法：可以认为是业务错误，通过自定义验证器验证，如用户名长度必须在5-20之间，我们却输入了100个字符等；

3、错误对象：当我们数据绑定失败或验证失败后，错误信息存放的对象，我们叫错误对象，在Spring Web MVC中Errors是具体的代表者；线程不安全对象；

4、错误消息：是硬编码，还是可配置？实际工作应该使用配置方式，我们只是把错误码（errorCode）放入错误对象，在展示时读取相应的错误消息配置文件来获取要显示的错误消息(errorMessage)；

4.16.2.1、验证流程

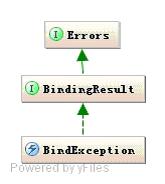


1、首先进行数据绑定验证，如果验证失败会通过MessageCodesResolver生成错误码放入Errors错误对象；

2、数据不合法验证，通过自定义的验证器验证，如果失败需要手动将错误码放入Errors错误对象；

4.16.2.2、错误对象和错误消息

错误对象的代表者是Errors接口，并且提供了几个实现者，在Spring Web MVC中我们使用的是如下实现：



相关的错误方法如下：

Errors：存储和暴露关于数据绑定错误和验证错误相关信息的接口，提供了相关存储和获取错误消息的方法：

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20549515) [copy](http://blog.csdn.net/hongjun1847/article/details/20549515)

[在CODE上查看代码片](https://code.csdn.net/snippets/218716)

1. **package** org.springframework.validation;
2. **public** **interface** Errors {
3. //=========================全局错误消息（验证/绑定对象全局的）=============================
4. //注册一个全局的错误码（）
5. **void** reject(String errorCode);
6. //注册一个全局的错误码，当根据errorCode没有找到相应错误消息时，使用defaultMessage作为错误消息
7. **void** reject(String errorCode, String defaultMessage);
8. //注册一个全局的错误码，当根据errorCode没有找到相应错误消息时（带错误参数的），使用defaultMessage作为错误消息
9. **void** reject(String errorCode, Object[] errorArgs, String defaultMessage);
10. //=========================全局错误消息（验证/绑定整个对象的）=============================
11. //=========================局部错误消息（验证/绑定对象字段的）=============================
12. //注册一个对象字段的错误码，field指定验证失败的字段名
13. **void** rejectValue(String field, String errorCode);
14. **void** rejectValue(String field, String errorCode, String defaultMessage);
15. **void** rejectValue(String field, String errorCode, Object[] errorArgs, String defaultMessage);
16. //=========================局部错误消息（验证/绑定对象字段的）=============================
17. **boolean** hasErrors();      ////是否有错误
18. **boolean** hasGlobalErrors(); //是否有全局错误
19. **boolean** hasFieldErrors();  //是否有字段错误
20. Object getFieldValue(String field); //返回当前验证通过的值，或验证失败时失败的值；
21. }

getFieldValue：可以得到验证失败的失败值，这是其他Web层框架很少支持的，这样就可以给用户展示出错时的值（而不是空或其他的默认值等）。

BindingResult：代表数据绑定的结果，继承了Errors接口。

BindException：代表数据绑定的异常，它继承Exception，并实现了BindingResult，这是内部使用的错误对象。

示例：

（1、控制器

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20549515) [copy](http://blog.csdn.net/hongjun1847/article/details/20549515)

[在CODE上查看代码片](https://code.csdn.net/snippets/218716)

1. **package** cn.javass.chapter4.web.controller;
2. //省略import
3. **public** **class** ErrorController **extends** AbstractCommandController {
4. **public** ErrorController() {
5. setCommandClass(DataBinderTestModel.**class**);
6. setCommandName("command");
7. }
8. @Override
9. **protected** ModelAndView handle(HttpServletRequest req, HttpServletResponse resp, Object command, BindException errors) **throws** Exception {
10. //表示用户名不为空
11. errors.reject("username.not.empty");
12. //带有默认错误消息
13. errors.reject("username.not.empty1", "用户名不能为空1");
14. //带有参数和默认错误消息
15. errors.reject("username.length.error", **new** Object[]{5, 10});
17. //得到错误相关的模型数据
18. Map model = errors.getModel();
19. **return** **new** ModelAndView("bindAndValidate/error", model);
20. }
21. }

errors.reject("username.not.empty")：注册全局错误码“username.not.empty”，我们必须提供messageSource来提供错误码“username.not.empty”对应的错误信息（如果没有会抛出NoSuchMessageException异常）；

errors.reject("username.not.empty1", "用户名不能为空1")：注册全局错误码“username.not.empty1”，如果从messageSource没没有找到错误码“username.not.empty1”对应的错误信息，则将显示默认消息“用户名不能为空1”；

errors.reject("username.length.error", new Object[]{5, 10})：错误码为“username.length.error”，而且错误信息需要两个参数，如我们在我们的配置文件中定义“用户名长度不合法，长度必须在{0}到{1}之间”，则实际的错误消息为“用户名长度不合法，长度必须在5到10之间”

errors.getModel()：当有错误信息时，一定将errors.getModel()放入我们要返回的ModelAndView中，以便使用里边的错误对象来显示错误信息。

（2、spring配置文件chapter4-servlet.xml

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20549515) [copy](http://blog.csdn.net/hongjun1847/article/details/20549515)

[在CODE上查看代码片](https://code.csdn.net/snippets/218716)

1. **<bean** id="messageSource"
2. class="org.springframework.context.support.ReloadableResourceBundleMessageSource"**>**
3. **<property** name="basename" value="classpath:messages"**/>**
4. **<property** name="fileEncodings" value="utf-8"**/>**
5. **<property** name="cacheSeconds" value="120"**/>**
6. **</bean>**
8. **<bean** name="/error" class="cn.javass.chapter4.web.controller.ErrorController"**/>**

messageSource：用于获取错误码对应的错误消息的，而且bean名字默认必须是messageSource。

messages.properties（需要执行NativeToAscii）

**[plain]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20549515) [copy](http://blog.csdn.net/hongjun1847/article/details/20549515)

[在CODE上查看代码片](https://code.csdn.net/snippets/218716)

1. username.not.empty=用户名不能为空
2. username.length.error=用户名长度不合法，长度必须在{0}到{1}之间

（3、视图页面（WEB-INF/jsp/bindAndValidate/error.jsp）

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20549515) [copy](http://blog.csdn.net/hongjun1847/article/details/20549515)

[在CODE上查看代码片](https://code.csdn.net/snippets/218716)

1. <%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
2. <!-- 表单的默认命令对象名为command -->
3. <form:form commandName="command">
4. <form:errors path="\*"></form:errors>
5. </form:form>

form标签库：此处我们使用了spring的form标签库；

<form:form commandName="command">:表示我们的表单标签，commandName表示绑定的命令对象名字，默认为command；

<form:errors path="\*"></form:errors>：表示显示错误信息的标签，如果path为“\*”表示显示所有错误信息。

接下来我们来看一下 数据绑定失败和数据不合法时，如何处理。

4.16.2.3、数据绑定失败

如我们的DataBinderTestModel类：

bool：boolean类型，此时如果我们前台传入非兼容的数据，则会数据绑定失败；

date：Date类型，此时如果我们前台传入非兼容的数据，同样会数据绑定失败；

phoneNumber：自定义的PhoneNumberModel类型，如果如果我们前台传入非兼容的数据，同样会数据绑定失败。

示例：

（1、控制器，DataBinderErrorTestController。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20549515) [copy](http://blog.csdn.net/hongjun1847/article/details/20549515)

[在CODE上查看代码片](https://code.csdn.net/snippets/218716)

1. **package** cn.javass.chapter4.web.controller;
2. //省略import
3. **public** **class** DataBinderErrorTestController **extends** SimpleFormController {
4. **public** DataBinderErrorTestController() {
5. setCommandClass(DataBinderTestModel.**class**);
6. setCommandName("dataBinderTest");
7. }
8. @Override
9. **protected** ModelAndView showForm(HttpServletRequest request, HttpServletResponse response, BindException errors) **throws** Exception {
10. //如果表单提交有任何错误都会再回到表单展示页面
11. System.out.println(errors);
12. **return** **super**.showForm(request, response, errors);
13. }
14. @Override
15. **protected** **void** doSubmitAction(Object command) **throws** Exception {
16. System.out.println(command); //表单提交成功（数据绑定成功）进行功能处理
17. }
18. @Override
19. **protected** **void** initBinder(HttpServletRequest request, ServletRequestDataBinder binder) **throws** Exception {
20. **super**.initBinder(request, binder);
21. //注册自定义的属性编辑器
22. //1、日期
23. DateFormat df = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
24. CustomDateEditor dateEditor = **new** CustomDateEditor(df, **true**);
25. //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
26. binder.registerCustomEditor(Date.**class**, dateEditor);
28. //自定义的电话号码编辑器
29. binder.registerCustomEditor(PhoneNumberModel.**class**, **new** PhoneNumberEditor());
30. }
31. }

此处我们使用SimpleFormController；

showForm：展示表单，当提交表单有任何数据绑定错误会再回到该方法进行表单输入（在此处我们打印错误对象）；

doSubmitAction：表单提交成功，只要当表单的数据到命令对象绑定成功时，才会执行；

（2、spring配置文件chapter4-servlet.xml

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20549515) [copy](http://blog.csdn.net/hongjun1847/article/details/20549515)

[在CODE上查看代码片](https://code.csdn.net/snippets/218716)

1. **<bean** name="/dataBindError"
2. class="cn.javass.chapter4.web.controller.DataBinderErrorTestController"**>**
3. **<property** name="formView" value="bindAndValidate/input"**/>**
4. **<property** name="successView" value="bindAndValidate/success"**/>**
5. **</bean>**

（3、视图页面（WEB-INF/jsp/bindAndValidate/ input.jsp）

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20549515) [copy](http://blog.csdn.net/hongjun1847/article/details/20549515)

[在CODE上查看代码片](https://code.csdn.net/snippets/218716)

1. **<**%@ page language="java" contentType="text/html; charset=UTF-8"   pageEncoding="UTF-8"%**>**
2. **<**%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
3. <!-- 表单的命令对象名为dataBinderTest -->
4. **<form:form** commandName="dataBinderTest"**>**
5. **<form:errors** path="\*" cssStyle="color:red"**></form:errors><br/><br/>**
6. bool:**<form:input** path="bool"**/><br/>**
7. phoneNumber:**<form:input** path="phoneNumber"**/><br/>**
8. date:**<form:input** path="date"**/><br/>**
9. **<input** type="submit" value="提交"**/>**
10. **</form:form>**

此处一定要使用form标签库，借此我们可以看到它的强大支持（别的大部分Web框架所不具备的，展示用户验证失败的数据）。

<form:form commandName="dataBinderTest">：指定命令对象为dataBinderTest，默认command；

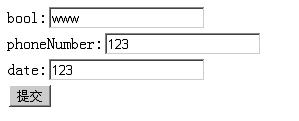
<form:errors path="\*" cssStyle="color:red"></form:errors>：显示错误消息，当提交表单有错误时展示错误消息（数据绑定错误/数据不合法）；

<form:input path="bool"/>：等价于（<input type=’text’>），但会从命令对象中取出bool属性进行填充value属性，或如果表单提交有错误会从错误对象取出之前的错误数据（而非空或默认值）；

<input type="submit" value="提交"/>：spring没有提供相应的提交按钮，因此需要使用html的。

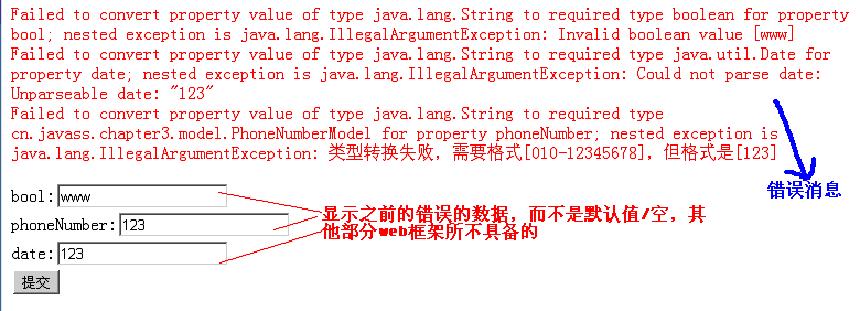
（4、测试

在地址栏输入如下地址：http://localhost:9080/springmvc-chapter4/dataBindError



全部是错误数据，即不能绑定到我们的命令对象；

当提交表单后，我们又回到表单输入页面，而且输出了一堆错误信息



1、错误消息不可读；

2、表单元素可以显示之前的错误的数据，而不是默认值/空；

（5、问题

这里最大的问题是不可读的错误消息，如何让这些错误消息可读呢？

首先我们看我们的showForm方法里输出的“errors”错误对象信息：

**[plain]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20549515) [copy](http://blog.csdn.net/hongjun1847/article/details/20549515)

[在CODE上查看代码片](https://code.csdn.net/snippets/218716)

1. org.springframework.validation.BindException: org.springframework.validation.BeanPropertyBindingResult: 3 errors
3. Field error in object 'dataBinderTest' on field 'bool': rejected value [www]; codes [typeMismatch.dataBinderTest.bool,typeMismatch.bool,typeMismatch.boolean,typeMismatch]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [dataBinderTest.bool,bool]; arguments []; default message [bool]]; default message [Failed to convert property value of type 'java.lang.String' to required type 'boolean' for property 'bool'; nested exception is java.lang.IllegalArgumentException: Invalid boolean value [www]]
5. Field error in object 'dataBinderTest' on field 'date': rejected value [123]; codes [typeMismatch.dataBinderTest.date,typeMismatch.date,typeMismatch.java.util.Date,typeMismatch]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [dataBinderTest.date,date]; arguments []; default message [date]]; default message [Failed to convert property value of type 'java.lang.String' to required type 'java.util.Date' for property 'date'; nested exception is java.lang.IllegalArgumentException: Could not parse date: Unparseable date: "123"]
7. Field error in object 'dataBinderTest' on field 'phoneNumber': rejected value [123]; codes [typeMismatch.dataBinderTest.phoneNumber,typeMismatch.phoneNumber,typeMismatch.cn.javass.chapter4.model.PhoneNumberModel,typeMismatch]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [dataBinderTest.phoneNumber,phoneNumber]; arguments []; default message [phoneNumber]]; default message [Failed to convert property value of type 'java.lang.String' to required type 'cn.javass.chapter4.model.PhoneNumberModel' for property 'phoneNumber'; nested exception is java.lang.IllegalArgumentException: 类型转换失败，需要格式[010-12345678]，但格式是[123]]

数据绑定失败（类型不匹配）会自动生成如下错误码（错误码对应的错误消息按照如下顺序依次查找）：

1、typeMismatch.命令对象名.属性名

2、typeMismatch.属性名

3、typeMismatch.属性全限定类名（包名.类名）

4、typeMismatch

⊙内部使用MessageCodesResolver解析数据绑定错误到错误码，默认DefaultMessageCodesResolver，因此想要详细了解如何解析请看其javadoc；

⊙建议使用第1个进行错误码的配置。

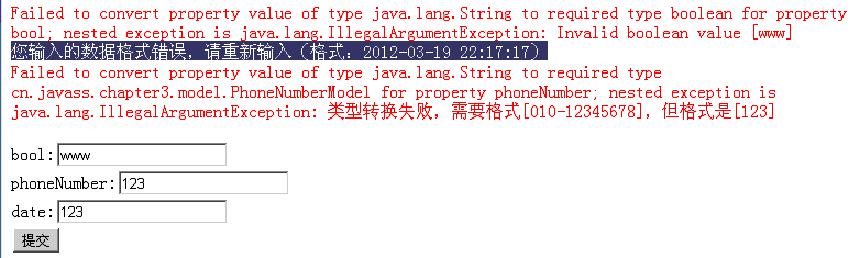
因此修改我们的messages.properties添加如下错误消息（需要执行NativeToAscii）：

**[plain]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20549515) [copy](http://blog.csdn.net/hongjun1847/article/details/20549515)

[在CODE上查看代码片](https://code.csdn.net/snippets/218716)

1. typeMismatch.dataBinderTest.date=您输入的数据格式错误，请重新输入（格式：2012-03-19 22:17:17）
2. #typeMismatch.date=2
3. #typeMismatch.java.util.Date=3
4. #typeMismatch=4

再次提交表单我们会看到我们设置的错误消息：



到此，数据绑定错误我们介绍完了，接下来我们再看一下数据不合法错误。

4.16.2.4、数据不合法

1、比如用户名长度必须在5-20之间，而且必须以字母开头，可包含字母、数字、下划线；

2、比如注册用户时 用户名已经存在或邮箱已经存在等；

3、比如去一些论坛经常会发现，您发的帖子中包含×××屏蔽关键字等。

还有很多数据不合法的场景，在此就不罗列了，对于数据不合法，Spring Web MVC提供了两种验证方式：

◆编程式验证器验证

◆声明式验证

先从编程式验证器开始吧。

4.16.2.4.1、编程式验证器

一、验证器接口

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20549515) [copy](http://blog.csdn.net/hongjun1847/article/details/20549515)

[在CODE上查看代码片](https://code.csdn.net/snippets/218716)

1. **package** org.springframework.validation;
2. **public** **interface** Validator {
3. **boolean** supports(Class<?> clazz);
4. **void** validate(Object target, Errors errors);
5. }

Validator接口：验证器，编程实现数据验证的接口；

supports方法：当前验证器是否支持指定的clazz验证，如果支持返回true即可；

validate方法：验证的具体方法，target参数表示要验证的目标对象（如命令对象），errors表示验证出错后存放错误信息的错误对象。

示例：

（1、验证器实现

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20549515) [copy](http://blog.csdn.net/hongjun1847/article/details/20549515)

[在CODE上查看代码片](https://code.csdn.net/snippets/218716)

1. **package** cn.javass.chapter4.web.controller.support.validator;
2. //省略import
3. **public** **class** UserModelValidator **implements** Validator {
4. **private** **static** **final** Pattern USERNAME\_PATTERN = Pattern.compile("[a-zA-Z]\\w{4,19}");
5. **private** **static** **final** Pattern PASSWORD\_PATTERN = Pattern.compile("[a-zA-Z0-9]{5,20}");
6. **private** **static** **final** Set<String> FORBINDDDEN\_WORD\_SET = **new** HashSet<String>();
7. **static** {
8. FORBINDDDEN\_WORD\_SET.add("fuc k"); //删掉空格
9. FORBINDDDEN\_WORD\_SET.add("admin");
10. }
11. @Override
12. **public** **boolean** supports(Class<?> clazz) {
13. **return** UserModel.**class** == clazz;//表示只对UserModel类型的目标对象实施验证
14. }
15. @Override
16. **public** **void** validate(Object target, Errors errors) {
17. //这个表示如果目标对象的username属性为空，则表示错误（简化我们手工判断是否为空）
18. ValidationUtils.rejectIfEmpty(errors, "username", "username.not.empty");
20. UserModel user = (UserModel) target;
22. **if**(!USERNAME\_PATTERN.matcher(user.getUsername()).matches()) {
23. errors.rejectValue("username", "username.not.illegal");//如果用户名不合法
24. }
26. **for**(String forbiddenWord : FORBINDDDEN\_WORD\_SET) {
27. **if**(user.getUsername().contains(forbiddenWord)) {
28. errors.rejectValue("username", "username.forbidden", **new** Object[]{forbiddenWord}, "您的用户名包含非法关键词");//用户名包含屏蔽关键字
29. **break**;
30. }
31. }
32. **if**(!PASSWORD\_PATTERN.matcher(user.getPassword()).matches()) {
33. errors.rejectValue("password","password.not.illegal", "密码不合法");//密码不合法
34. }
35. }
36. }

supports方法：表示只对UserModel类型的对象验证；

validate方法：数据验证的具体方法，有如下几个验证：

    1、用户名不合法（长度5-20，以字母开头，随后可以是字母、数字、下划线）

    USERNAME\_PATTERN.matcher(user.getUsername()).matches() //使用正则表达式验证

    errors.rejectValue("username", "username.not.illegal");//验证失败为username字段添加错误码

    2、屏蔽关键词：即用户名中含有不合法的数据（如admin）

    user.getUsername().contains(forbiddenWord) //用contains来判断我们的用户名中是否含有非法关键词

    errors.rejectValue("username", "username.forbidden", new Object[]{forbiddenWord}, "您的用户名包含非法关键词");//验证失败为username字段添加错误码（参数为当前屏蔽关键词）（默认消息为"您的用户名包含非法关键词"）

    3、密码不合法

    在此就不罗列代码了；

    4、ValidationUtils

    ValidationUtils.rejectIfEmpty(errors, "username", "username.not.empty");

    表示如果目标对象的username属性数据为空，则添加它的错误码；

    内部通过（value == null || !StringUtils.hasLength(value.toString())）实现判断value是否为空，从而简化代码。

（2、spring配置文件chapter4-servlet.xml

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20549515) [copy](http://blog.csdn.net/hongjun1847/article/details/20549515)

[在CODE上查看代码片](https://code.csdn.net/snippets/218716)

1. **<bean** id="userModelValidator"
2. class="cn.javass.chapter4.web.controller.support.validator.UserModelValidator"**/>**
3. **<bean** name="/validator"
4. class="cn.javass.chapter4.web.controller.RegisterSimpleFormController"**>**
5. **<property** name="formView" value="registerAndValidator"**/>**
6. **<property** name="successView" value="redirect:/success"**/>**
7. **<property** name="validator" ref="userModelValidator"**/>**
8. **</bean>**

此处使用了我们第4.9节创建的RegisterSimpleFormController。

（3、错误码配置（messages.properties），需要执行NativeToAscii

**[plain]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20549515) [copy](http://blog.csdn.net/hongjun1847/article/details/20549515)

[在CODE上查看代码片](https://code.csdn.net/snippets/218716)

1. username.not.empty=用户名不能为空
2. username.not.illegal=用户名错误，必须以字母开头，只能出现字母、数字、下划线，并且长度在5-20之间
3. username.forbidden=用户名中包含非法关键词【{0}】
4. password.not.illegal=密码长度必须在5-20之间

（4、视图页面（/WEB-INF/jsp/registerAndValidator.jsp）

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20549515) [copy](http://blog.csdn.net/hongjun1847/article/details/20549515)

[在CODE上查看代码片](https://code.csdn.net/snippets/218716)

1. **<**%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%**>**
2. **<**%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %**>**
3. **<form:form** commandName="user"**>**
5. **<form:errors** path="\*" cssStyle="color:red"**></form:errors><br/>**
7. username:**<form:input** path="username"**/>**
8. **<form:errors** path="username" cssStyle="color:red"**></form:errors>**
9. **<br/>**
11. password:**<form:password** path="password"**/>**
12. **<form:errors** path="password" cssStyle="color:red"**></form:errors>**
13. **<br/>**
14. **<input** type="submit" value="注册"**/>**
15. **</form:form>**

form:errors path="username"：表示只显示username字段的错误信息；

（5、测试

地址：http://localhost:9080/springmvc-chapter4/validator



当我们输入错误的数据后，会报错（form:errors path="\*"显示所有错误信息，而form:errors path="username"只显示该字段相关的）。

问题：

如MultiActionController控制器相关方法没有提供给我们errors对象（Errors），我们应该怎么进行错误处理呢？

此处给大家一个思路，errors本质就是一个Errors接口实现，而且在页面要读取相关的错误对象，该错误对象应该存放在模型对象里边，因此我们可以自己创建个errors对象并将其添加到模型对象中即可。

此处我们复制4.15节的UserController类为UserAndValidatorController，并修改它的create（新增）方法添加如下代码片段：

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20549515) [copy](http://blog.csdn.net/hongjun1847/article/details/20549515)

[在CODE上查看代码片](https://code.csdn.net/snippets/218716)

1. BindException errors = **new** BindException(user, getCommandName(user));
2. //如果用户名为空
3. **if**(!StringUtils.hasLength(user.getUsername())) {
4. errors.rejectValue("username", "username.not.empty");
5. }
6. **if**(errors.hasErrors()) {
7. **return** **new** ModelAndView(getCreateView()).addAllObjects(errors.getModel());
8. }

√ new BindException(user, getCommandName(user))：使用当前的命令对象，和命令对象的名字创建了一个BindException作为errors；

√StringUtils.hasLength(user.getUsername())：如果用户名为空就是用errors.rejectValue("username", "username.not.empty");注入错误码；

√errors.hasErrors()：表示如果有错误就返回到新增页面并显示错误消息；

√ModelAndView(getCreateView()).addAllObjects(errors.getModel())：此处一定把errors对象的模型数据放在当前的ModelAndView中，作为当前请求的模型数据返回。

在浏览器地址栏输入：http://localhost:9080/springmvc-chapter4/userAndValidator/create 到新增页面



用户名什么都不输入，提交后又返回到新增页面 而且显示了错误消息说明我们的想法是正确的。

4.16.2.4.2、声明式验证器

从Spring3开始支持JSR-303验证框架，支持XML风格和注解风格的验证，目前在@RequestMapping时才能使用，也就是说基于Controller接口的实现不能使用该方式（但可以使用编程式验证，有需要的可以参考hibernate validator实现），我们将在第七章详细介绍。

到此Spring2风格的控制器我们就介绍完了，以上控制器从spring3.0开始已经不推荐使用了（但考虑到还有部分公司使用这些@Deprecated类，在此也介绍了一下），而是使用注解控制器实现（@Controller和@RequestMapping）。