**6.1、注解式控制器简介**

一、Spring2.5之前，我们都是通过实现Controller接口或其实现来定义我们的处理器类。已经@Deprecated。

二、Spring2.5引入注解式处理器支持，通过@Controller 和 @RequestMapping注解定义我们的处理器类。并且提供了一组强大的注解：

需要通过处理器映射DefaultAnnotationHandlerMapping和处理器适配器AnnotationMethodHandlerAdapter来开启支持@Controller 和 @RequestMapping注解的处理器。

@Controller：用于标识是处理器类；

@RequestMapping：请求到处理器功能方法的映射规则；

@RequestParam：请求参数到处理器功能处理方法的方法参数上的绑定；

@ModelAttribute：请求参数到命令对象的绑定；

@SessionAttributes：用于声明session级别存储的属性，放置在处理器类上，通常列出模型属性（如@ModelAttribute）对应的名称，则这些属性会透明的保存到session中；

@InitBinder：自定义数据绑定注册支持，用于将请求参数转换到命令对象属性的对应类型；

三、Spring3.0引入RESTful架构风格支持(通过@PathVariable注解和一些其他特性支持),且又引入了更多的注解支持：

@CookieValue：cookie数据到处理器功能处理方法的方法参数上的绑定；

@RequestHeader：请求头（header）数据到处理器功能处理方法的方法参数上的绑定；

@RequestBody：请求的body体的绑定（通过HttpMessageConverter进行类型转换）；

@ResponseBody：处理器功能处理方法的返回值作为响应体（通过HttpMessageConverter进行类型转换）；

@ResponseStatus：定义处理器功能处理方法/异常处理器返回的状态码和原因；

@ExceptionHandler：注解式声明异常处理器；

@PathVariable：请求URI中的模板变量部分到处理器功能处理方法的方法参数上的绑定，从而支持RESTful架构风格的URI；

四、Spring3.1使用新的HandlerMapping 和 HandlerAdapter来支持@Contoller和@RequestMapping注解处理器。

新的@Contoller和@RequestMapping注解支持类：处理器映射RequestMappingHandlerMapping 和 处理器适配器RequestMappingHandlerAdapter组合来代替Spring2.5开始的处理器映射DefaultAnnotationHandlerMapping和处理器适配器AnnotationMethodHandlerAdapter，提供更多的扩展点。

接下来，我们一起开始学习基于注解的控制器吧。

**6.2、入门**

（1、控制器实现

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20552433) [copy](http://blog.csdn.net/hongjun1847/article/details/20552433)

[在CODE上查看代码片](https://code.csdn.net/snippets/218840)

1. **package** cn.javass.chapter6.web.controller;
2. //省略import
3. @Controller         // 或 @RequestMapping               //①将一个POJO类声明为处理器
4. **public** **class** HelloWorldController {
5. @RequestMapping(value = "/hello")                  //②请求URL到处理器功能处理方法的映射
6. **public** ModelAndView helloWorld() {
7. //1、收集参数
8. //2、绑定参数到命令对象
9. //3、调用业务对象
10. //4、选择下一个页面
11. ModelAndView mv = **new** ModelAndView();
12. //添加模型数据 可以是任意的POJO对象
13. mv.addObject("message", "Hello World!");
14. //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
15. mv.setViewName("hello");
16. **return** mv;                                         //○3 模型数据和逻辑视图名
17. }
18. }

1 可以通过在一个POJO类上放置@Controller或@RequestMapping，即可把一个POJO类变身为处理器；

2 @RequestMapping(value = "/hello") 请求URL(/hello) 到 处理器的功能处理方法的映射；

3 模型数据和逻辑视图名的返回。

现在的处理器无需实现/继承任何接口/类，只需要在相应的类/方法上放置相应的注解说明下即可，非常方便。

（2、Spring配置文件chapter6-servlet.xml

（2.1、HandlerMapping和HandlerAdapter的配置

如果您使用的是Spring3.1之前版本，开启注解式处理器支持的配置为：DefaultAnnotationHandlerMapping和AnnotationMethodHandlerAdapter。

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20552433) [copy](http://blog.csdn.net/hongjun1847/article/details/20552433)

[在CODE上查看代码片](https://code.csdn.net/snippets/218840)

1. <!—Spring3.1之前的注解 HandlerMapping --**>**
2. **<bean**
3. class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping"**/>**
5. <!—Spring3.1之前的注解 HandlerAdapter --**>**
6. **<bean**
7. class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter"**/>**

如果您使用的Spring3.1开始的版本，建议使用RequestMappingHandlerMapping和RequestMappingHandlerAdapter。

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20552433) [copy](http://blog.csdn.net/hongjun1847/article/details/20552433)

[在CODE上查看代码片](https://code.csdn.net/snippets/218840)

1. <!--Spring3.1开始的注解 HandlerMapping -->
2. **<bean**
3. class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"**/>**
4. <!--Spring3.1开始的注解 HandlerAdapter -->
5. **<bean**
6. class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"**/>**

下一章我们介绍DefaultAnnotationHandlerMapping和AnnotationMethodHandlerAdapter 与RequestMappingHandlerMapping和RequestMappingHandlerAdapter 的区别。

（2.2、视图解析器的配置

还是使用之前的org.springframework.web.servlet.view.InternalResourceViewResolver。

（2.3、处理器的配置

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20552433) [copy](http://blog.csdn.net/hongjun1847/article/details/20552433)

[在CODE上查看代码片](https://code.csdn.net/snippets/218840)

1. <!-- 处理器 -->
2. **<bean** class="cn.javass.chapter6.web.controller.HelloWorldController"**/>**

只需要将处理器实现类注册到spring配置文件即可，spring的DefaultAnnotationHandlerMapping或RequestMappingHandlerMapping能根据注解@Controller或@RequestMapping自动发现。

（2.3、视图页面（/WEB-INF/jsp/hello.jsp）

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20552433) [copy](http://blog.csdn.net/hongjun1847/article/details/20552433)

[在CODE上查看代码片](https://code.csdn.net/snippets/218840)

1. **<**%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%**>**
2. <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd"**>**
3. **<html>**
4. **<head>**
5. **<meta** http-equiv="Content-Type" content="text/html; charset=UTF-8"**>**
6. **<title>**Hello World**</title>**
7. **</head>**
8. **<body>**
9. ${message}
10. **</body>**
11. **</html>**

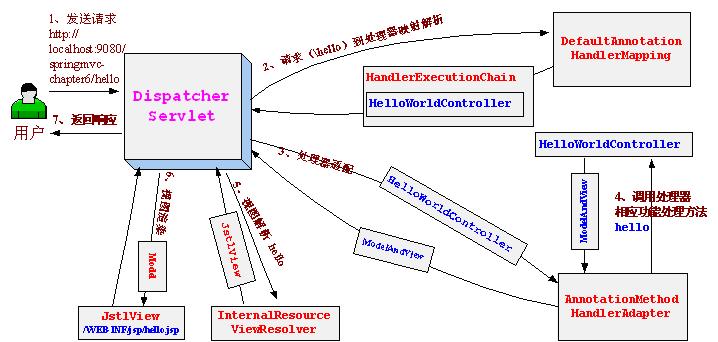
${message}：表示显示由HelloWorldController处理器传过来的模型数据。

（4、启动服务器测试

地址栏输入<http://localhost:9080/springmvc-chapter6/hello>，我们将看到页面显示“Hello World!”，表示成功了。

整个过程和我们第二章中的Hello World 类似，只是处理器的实现不一样。接下来我们来看一下具体流程吧。

**6.3、运行流程**



和第二章唯一不同的两处是：

1、HandlerMapping实现：使用DefaultAnnotationHandlerMapping（spring3.1之前）或RequestMappingHandlerMapping（spring3.1）替换之前的BeanNameUrlHandlerMapping。

注解式处理器映射会扫描spring容器中的bean，发现bean实现类上拥有@Controller或@RequestMapping注解的bean，并将它们作为处理器。

2、HandlerAdapter实现：使用AnnotationMethodHandlerAdapter（spring3.1之前）或RequestMappingHandlerAdapter（spring3.1）替换之前的SimpleControllerHandlerAdapter。

注解式处理器适配器会通过反射调用相应的功能处理方法（方法上拥有@RequestMapping注解）。

好了到此我们知道Spring如何发现处理器、如何调用处理的功能处理方法了，接下来我们详细学习下如何定义处理器、如何进行请求到功能处理方法的定义。

**6.4、处理器定义**

**6.4.1、@Controller**

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20552433) [copy](http://blog.csdn.net/hongjun1847/article/details/20552433)

[在CODE上查看代码片](https://code.csdn.net/snippets/218840)

1. @Controller
2. **public** **class** HelloWorldController {
3. ……
4. }

推荐使用这种方式声明处理器，它和我们的@Service、@Repository很好的对应了我们常见的三层开发架构的组件。

**6.4.2、@RequestMapping**

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20552433) [copy](http://blog.csdn.net/hongjun1847/article/details/20552433)

[在CODE上查看代码片](https://code.csdn.net/snippets/218840)

1. @RequestMapping
2. **public** **class** HelloWorldController {
3. ……
4. }

这种方式也是可以工作的，但如果在类上使用@ RequestMapping注解一般是用于窄化功能处理方法的映射的，详见6.4.3。

**6.4.3、窄化请求映射**

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20552433) [copy](http://blog.csdn.net/hongjun1847/article/details/20552433)

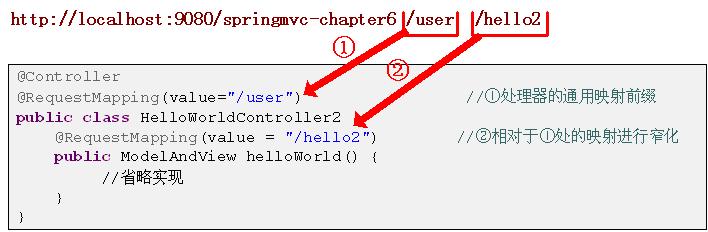
[在CODE上查看代码片](https://code.csdn.net/snippets/218840)

1. **package** cn.javass.chapter6.web.controller;
2. @Controller
3. @RequestMapping(value="/user")                 //①处理器的通用映射前缀
4. **public** **class** HelloWorldController2 {
5. @RequestMapping(value = "/hello2")        //②相对于①处的映射进行窄化
6. **public** ModelAndView helloWorld() {
7. //省略实现
8. }
9. }

①类上的@RequestMapping(value="/user") 表示处理器的通用请求前缀；

②处理器功能处理方法上的是对①处映射的窄化。

因此[http://localhost:9080/springmvc-chapter6/hello2 无法映射到HelloWorldController2](http://localhost:9080/springmvc-chapter6/hello2%20%E6%97%A0%E6%B3%95%E6%98%A0%E5%B0%84%E5%88%B0HelloWorldController2)的 helloWorld功能处理方法；而<http://localhost:9080/springmvc-chapter6/user/hello2>是可以的。



窄化请求映射可以认为是方法级别的@RequestMapping继承类级别的@RequestMapping。

窄化请求映射还有其他方式，如在类级别指定URL，而方法级别指定请求方法类型或参数等等，后续会详细介绍。

到此，我们知道如何定义处理器了，接下来我们需要学习如何把请求映射到相应的功能处理方法进行请求处理。

**6.5、请求映射**

处理器定义好了，那接下来我们应该定义功能处理方法，接收用户请求处理并选择视图进行渲染。首先我们看一下图6-1:



http请求信息包含六部分信息：

①请求方法，如GET或POST，表示提交的方式；

②URL，请求的地址信息；

③协议及版本；

④请求头信息（包括Cookie信息）；

⑤回车换行（CRLF）；

⑥请求内容区（即请求的内容或数据），如表单提交时的参数数据、URL请求参数（?abc=123 ？后边的）等。

想要了解HTTP/1.1协议，请访问<http://tools.ietf.org/html/rfc2616>。

那此处我们可以看到有①、②、④、⑥一般是可变的，因此我们可以这些信息进行请求到处理器的功能处理方法的映射，因此请求的映射分为如下几种：

URL路径映射：使用URL映射请求到处理器的功能处理方法；

请求方法映射限定：如限定功能处理方法只处理GET请求；

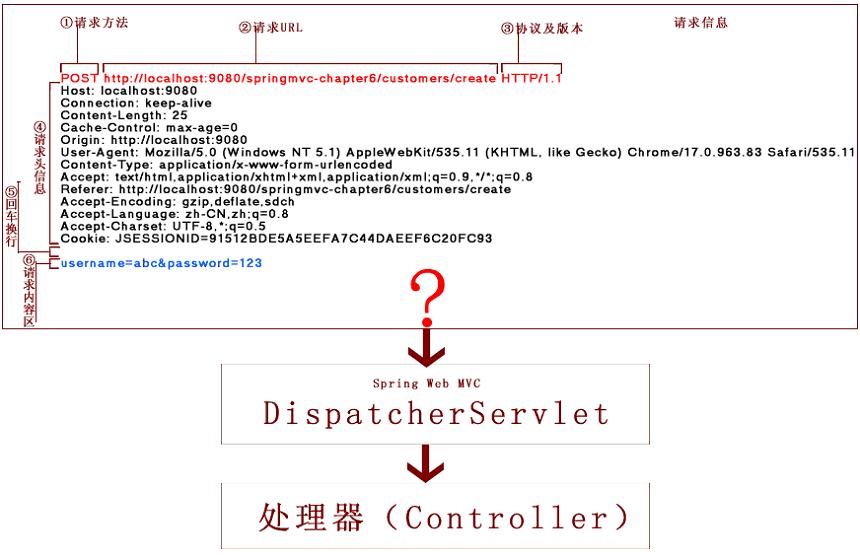
请求参数映射限定：如限定只处理包含“abc”请求参数的请求；

请求头映射限定：如限定只处理“Accept=application/json”的请求。

接下来看看具体如何映射吧。

## 6.5、请求映射

处理器定义好了，那接下来我们应该定义功能处理方法，接收用户请求处理并选择视图进行渲染。首先我们看一下图6-1:



http请求信息包含六部分信息：

①请求方法，如GET或POST，表示提交的方式；

②URL，请求的地址信息；

③协议及版本；

④请求头信息（包括Cookie信息）；

⑤回车换行（CRLF）；

⑥请求内容区（即请求的内容或数据），如表单提交时的参数数据、URL请求参数（?abc=123 ？后边的）等。

想要了解HTTP/1.1协议，请访问<http://tools.ietf.org/html/rfc2616>。

那此处我们可以看到有①、②、④、⑥一般是可变的，因此我们可以这些信息进行请求到处理器的功能处理方法的映射，因此请求的映射分为如下几种：

URL路径映射：使用URL映射请求到处理器的功能处理方法；

请求方法映射限定：如限定功能处理方法只处理GET请求；

请求参数映射限定：如限定只处理包含“abc”请求参数的请求；

请求头映射限定：如限定只处理“Accept=application/json”的请求。

接下来看看具体如何映射吧。

### 

6.5.1、URL路径映射

#### 6.5.1.1、普通URL路径映射

@RequestMapping(value={"/test1", "/user/create"})：多个URL路径可以映射到同一个处理器的功能处理方法。

#### 6.5.1.2、URI模板模式映射

@RequestMapping(value="/users/{userId}")：{×××}占位符， 请求的URL可以是 “/users/123456”或

“/users/abcd”，通过6.6.5讲的通过@PathVariable可以提取URI模板模式中的{×××}中的×××变量。

@RequestMapping(value="/users/{userId}/create")：这样也是可以的，请求的URL可以是“/users/123/create”。

@RequestMapping(value="/users/{userId}/topics/{topicId}")：这样也是可以的，请求的URL可以是“/users/123/topics/123”。

#### 6.5.1.3、Ant风格的URL路径映射

@RequestMapping(value="/users/\*\*")：可以匹配“/users/abc/abc”，但“/users/123”将会被【URI模板模式映射中的“/users/{userId}”模式优先映射到】【详见4.14的最长匹配优先】。

@RequestMapping(value="/product?")：可匹配“/product1”或“/producta”，但不匹配“/product”或“/productaa”;

@RequestMapping(value="/product\*")：可匹配“/productabc”或“/product”，但不匹配“/productabc/abc”;

@RequestMapping(value="/product/\*")：可匹配“/product/abc”，但不匹配“/productabc”;

@RequestMapping(value="/products/\*\*/{productId}")：可匹配“/products/abc/abc/123”或“/products/123”，也就是Ant风格和URI模板变量风格可混用;

此处需要注意的是【4.14中提到的最长匹配优先】，Ant风格的模式请参考4.14。

#### 6.5.1.4、正则表达式风格的URL路径映射

从Spring3.0开始支持正则表达式风格的URL路径映射，格式为{变量名:正则表达式}，这样我们就可以通过6.6.5讲的通过@PathVariable提取模式中的{×××：正则表达式匹配的值}中的×××变量了。

@RequestMapping(value="/products/{categoryCode:\\d+}-{pageNumber:\\d+}")：可以匹配“/products/123-1”，但不能匹配“/products/abc-1”，这样可以设计更加严格的规则。

正则表达式风格的URL路径映射是一种特殊的URI模板模式映射：

URI模板模式映射是{userId}，不能指定模板变量的数据类型，如是数字还是字符串；

正则表达式风格的URL路径映射，可以指定模板变量的数据类型，可以将规则写的相当复杂。

#### 6.5.1.5、组合使用是“或”的关系

如 @RequestMapping(value={"/test1", "/user/create"}) 组合使用是或的关系，即“/test1”或“/user/create”请求URL路径都可以映射到@RequestMapping指定的功能处理方法。

以上URL映射的测试类为：cn.javass.chapter6.web.controller.mapping.MappingController.java。

到此，我们学习了Spring Web MVC提供的强大的URL路径映射，而且可以实现非常复杂的URL规则。Spring Web MVC不仅仅提供URL路径映射，还提供了其他强大的映射规则。接下来我们看一下请求方法映射限定吧。

6.5.2、请求方法映射限定

一般我们熟悉的表单一般分为两步：第一步展示，第二步提交，如4.9、SimpleFormController那样，那如何通过@RequestMapping来实现呢？

#### 6.5.2.1、请求方法映射限定

我们熟知的，展示表单一般为GET请求方法；提交表单一般为POST请求方法。但6.5.1节讲的URL路径映射方式对任意请求方法是全盘接受的，因此我们需要某种方式来告诉相应的功能处理方法只处理如GET请求方法的请求或POST请求方法的请求。

接下来我们使用@RequestMapping来实现SimpleFormController的功能吧。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20553493) [copy](http://blog.csdn.net/hongjun1847/article/details/20553493)

[在CODE上查看代码片](https://code.csdn.net/snippets/218879)

1. **package** cn.javass.chapter6.web.controller.method;
2. //省略import
3. @Controller
4. @RequestMapping("/customers/\*\*")                                     //①处理器的通用映射前缀
5. **public** **class** RequestMethodController {
6. @RequestMapping(value="/create", method = RequestMethod.GET)//②类级别的@RequestMapping窄化
7. **public** String showForm() {
8. System.out.println("===============GET");
9. **return** "customer/create";
10. }
11. @RequestMapping(value="/create", method = RequestMethod.POST)//③类级别的@RequestMapping窄化
12. **public** String submit() {
13. System.out.println("================POST");
14. **return** "redirect:/success";
15. }
16. }

 ①处理器的通用映射前缀（父路径）：表示该处理器只处理匹配“/customers/\*\*”的请求；

②对类级别的@RequestMapping进行窄化，表示showForm可处理匹配“/customers/\*\*/create”且请求方法为“GET”的请求；

③对类级别的@RequestMapping进行窄化，表示submit可处理匹配“/customers/\*\*/create”且请求方法为“POST”的请求。

#### 6.5.2.2、组合使用是“或”的关系

@RequestMapping(value="/methodOr", method = {RequestMethod.POST, RequestMethod.GET})：即请求方法可以是 GET 或 POST。

提示：

1、一般浏览器只支持GET、POST请求方法，如想浏览器支持PUT、DELETE等请求方法只能模拟，稍候章节介绍。

2、除了GET、POST，还有HEAD、OPTIONS、PUT、DELETE、TRACE。

3、DispatcherServlet默认开启对 GET、POST、PUT、DELETE、HEAD的支持；

4、如果需要支持OPTIONS、TRACE，请添加DispatcherServlet在web.xml的初始化参数：dispatchOptionsRequest 和 dispatchTraceRequest 为true。

请求方法的详细使用请参考RESTful架构风格一章。

以上请求方法映射限定测试类为：cn.javass.chapter6.web.controller.method.RequestMethodController。

6.5.3、请求参数数据映射限定

#### 6.5.3.1、请求数据中有指定参数名

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20553493) [copy](http://blog.csdn.net/hongjun1847/article/details/20553493)

[在CODE上查看代码片](https://code.csdn.net/snippets/218879)

1. **package** cn.javass.chapter6.web.controller.parameter;
2. //省略import
3. @Controller
4. @RequestMapping("/parameter1")                                      //①处理器的通用映射前缀
5. **public** **class** RequestParameterController1 {
6. //②进行类级别的@RequestMapping窄化
7. @RequestMapping(params="create", method=RequestMethod.GET)
8. **public** String showForm() {
9. System.out.println("===============showForm");
10. **return** "parameter/create";
11. }
12. //③进行类级别的@RequestMapping窄化
13. @RequestMapping(params="create", method=RequestMethod.POST)
14. **public** String submit() {
15. System.out.println("================submit");
16. **return** "redirect:/success";
17. }
18. }

②@RequestMapping(params="create", method=RequestMethod.GET) ：表示请求中有“create”的参数名且请求方法为“GET”即可匹配，如可匹配的请求URL“http://×××/parameter1?create”；

③@RequestMapping(params="create", method=RequestMethod.POST)：表示请求中有“create”的参数名且请求方法为“POST”即可匹配；

此处的create请求参数名表示你请求的动作，即你想要的功能的一个标识，常见的CRUD(增删改查)我们可以使用如下请求参数名来表达：

◇（create请求参数名 且 GET请求方法） 新增页面展示、（create请求参数名 且 POST请求方法）新增提交；

◇（update请求参数名 且 GET请求方法） 新增页面展示、（update请求参数名 且 POST请求方法）新增提交；

◇（delete请求参数名 且 GET请求方法） 新增页面展示、（delete请求参数名 且 POST请求方法）新增提交；

◇（query请求参数名 且 GET请求方法） 新增页面展示、（query请求参数名 且 POST请求方法） 新增提交；

◇（list请求参数名 且 GET请求方法） 列表页面展示；

◇（view请求参数名 且 GET请求方法） 查看单条记录页面展示。

#### 6.5.3.2、请求数据中没有指定参数名

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20553493) [copy](http://blog.csdn.net/hongjun1847/article/details/20553493)

1. //请求参数不包含 create参数名
2. @RequestMapping(params="!create", method=RequestMethod.GET)//进行类级别的@RequestMapping窄化

 @RequestMapping(params="!create", method=RequestMethod.GET)：表示请求中没有“create”参数名且请求方法为“GET”即可匹配，如可匹配的请求URL“http://×××/parameter1?abc”。

#### 6.5.3.3、请求数据中指定参数名=值

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20553493) [copy](http://blog.csdn.net/hongjun1847/article/details/20553493)

[在CODE上查看代码片](https://code.csdn.net/snippets/218879)

1. **package** cn.javass.chapter6.web.controller.parameter;
2. //省略import
3. @Controller
4. @RequestMapping("/parameter2")                      //①处理器的通用映射前缀
5. **public** **class** RequestParameterController2 {
6. //②进行类级别的@RequestMapping窄化
7. @RequestMapping(params="submitFlag=create", method=RequestMethod.GET)
8. **public** String showForm() {
9. System.out.println("===============showForm");
10. **return** "parameter/create";
11. }
12. //③进行类级别的@RequestMapping窄化
13. @RequestMapping(params="submitFlag=create", method=RequestMethod.POST)
14. **public** String submit() {
15. System.out.println("===============submit");
16. **return** "redirect:/success";
17. }
18. }

②@RequestMapping(params="submitFlag=create", method=RequestMethod.GET)：表示请求中有“submitFlag=create”请求参数且请求方法为“GET”即可匹配，如请求URL为[http://×××/parameter2?submitFlag=create](http://localhost:9080/springmvc-chapter6/parameter?submitFlag=toCreate)；

③@RequestMapping(params="submitFlag=create", method=RequestMethod.POST)：表示请求中有“submitFlag=create”请求参数且请求方法为“POST”即可匹配；

此处的submitFlag=create请求参数表示你请求的动作，即你想要的功能的一个标识，常见的CRUD(增删改查)我们可以使用如下请求参数名来表达：

◇（submitFlag=create请求参数名 且 GET请求方法） 新增页面展示、（submitFlag=create请求参数名 且 POST请求方法） 新增提交；

◇（submitFlag=update请求参数名 且 GET请求方法） 新增页面展示、（submitFlag=update请求参数名 且 POST请求方法） 新增提交；

◇（submitFlag=delete请求参数名 且 GET请求方法） 新增页面展示、（submitFlag=delete请求参数名 且 POST请求方法） 新增提交；

◇（submitFlag=query请求参数名 且 GET请求方法） 新增页面展示、（submitFlag=query请求参数名 且 POST请求方法） 新增提交；

◇（submitFlag=list请求参数名 且 GET请求方法） 列表页面展示；

◇（submitFlag=view请求参数名 且 GET请求方法） 查看单条记录页面展示。

#### 6.5.3.4、请求数据中指定参数名!=值

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20553493) [copy](http://blog.csdn.net/hongjun1847/article/details/20553493)

1. //请求参数submitFlag 不等于 create
2. @RequestMapping(params="submitFlag!=create", method=RequestMethod.GET)

 @RequestMapping(params="submitFlag!=create", method=RequestMethod.GET)：表示请求中的参数“submitFlag!=create”且请求方法为“GET”即可匹配，如可匹配的请求URL“http://×××/parameter1?submitFlag=abc”。

#### 6.5.3.5、组合使用是“且”的关系

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20553493) [copy](http://blog.csdn.net/hongjun1847/article/details/20553493)

1. @RequestMapping(params={"test1", "test2=create"})  //②进行类级别的@RequestMapping窄化

 @RequestMapping(params={"test1", "test2=create"})：表示请求中的有“test1”参数名 且 有“test2=create”参数即可匹配，如可匹配的请求URL“http://×××/parameter3?test1&test2=create。

以上请求参数数据映射限定测试类为：cn.javass.chapter6.web.controller.method包下的RequestParameterController1、RequestParameterController2、RequestParameterController3。

6.5.4、请求头数据映射限定

#### 6.5.4.1、准备环境

浏览器：建议chrome最新版本；

插件：ModHeader

安装地址：<https://chrome.google.com/webstore/detail/idgpnmonknjnojddfkpgkljpfnnfcklj>

插件安装步骤：

1、打开<https://chrome.google.com/webstore/detail/idgpnmonknjnojddfkpgkljpfnnfcklj>，如图6-2



图6-2

2、点击“添加至chrome”后弹出“确认安装”对话框，点击“安装”按钮即可，如图6-3：



图6-3

3、安装成功后，在浏览器右上角出现如图6-4的图标表示安装成功：

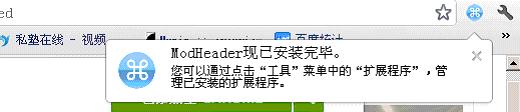


图6-4

4、鼠标右击右上角的“Modify Header”图标，选择选项，打开如图6-5：

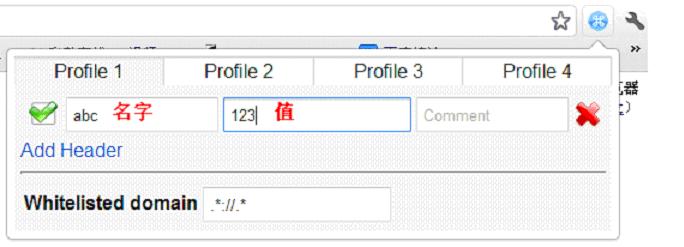


图6-5

7、修改完成后，输入URL请求，你可以在chrome的“开发人员工具的”网络选项卡下，看到如图6-7的信息表示添加请求头成功了：

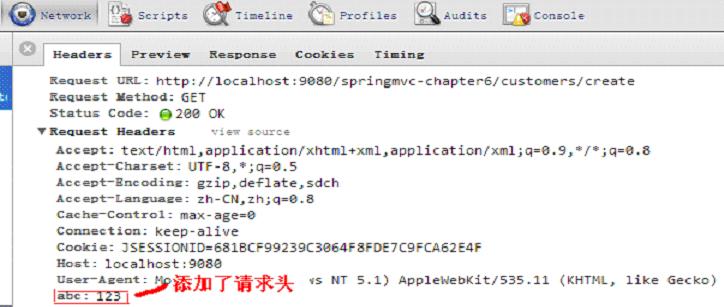


图6-7

到此我们的工具安装完毕，接下来看看如何使用请求头数据进行映射限定。

#### 6.5.4.2、请求头数据中有指定参数名

@RequestMapping(value="/header/test1", headers = "Accept")：表示请求的URL必须为“/header/test1”

且 请求头中必须有Accept参数才能匹配。

@RequestMapping(value="/header/test1", headers = "abc")：表示请求的URL必须为“/header/test1”

且 请求头中必须有abc参数才能匹配，如图6-8时可匹配。

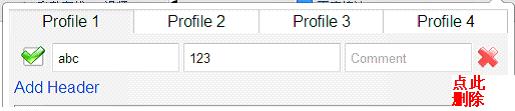


图6-8

#### 6.5.4.3、请求头数据中没有指定参数名

@RequestMapping(value="/header/test2", headers = "!abc")：表示请求的URL必须为“/header/test2”

且 请求头中必须没有abc参数才能匹配。（将Modify Header的abc参数值删除即可）。



#### 6.5.4.4、请求头数据中指定参数名=值

@RequestMapping(value="/header/test3", headers = "Content-Type=application/json")：表示请求的URL必须为“/header/test3” 且 请求头中必须有“Content-Type=application/json”参数即可匹配。（将Modify Header的Content-Type参数值改为“application/json”即可）；



当你请求的URL为“/header/test3” 但 如果请求头中没有或不是“Content-Type=application/json”参数（如“text/html”其他参数），将返回“HTTP Status 415”状态码【表示不支持的媒体类型(Media Type)，也就是MIME类型】，即我们的功能处理方法只能处理application/json的媒体类型。

@RequestMapping(value="/header/test4", headers = "Accept=application/json")：表示请求的URL必须为“/header/test4” 且 请求头中必须有“Accept =application/json”参数即可匹配。（将Modify Header的Accept参数值改为“application/json”即可）；



当你请求的URL为“/header/test4” 但 如果请求头中没有“Accept=application/json”参数（如“text/html”其他参数），将返回“HTTP Status 406”状态码【不可接受，服务器无法根据Accept头的媒体类型为客户端生成响应】，即客户只接受“application/json”媒体类型的数据，即我们的功能处理方法的响应只能返回“application/json”媒体类型的数据。

@RequestMapping(value="/header/test5", headers = "Accept=text/\*") ：表示请求的URL必须为“/header/test5” 且 请求头中必须有如“Accept=text/plain”参数即可匹配。（将Modify Header的Accept参数值改为“text/plain”即可）；

Accept=text/\*：表示主类型为text，子类型任意，如“text/plain”、“text/html”等都可以匹配。

@RequestMapping(value="/header/test6", headers = "Accept=\*/\*") ：表示请求的URL必须为“/header/test6” 且 请求头中必须有任意Accept参数即可匹配。（将Modify Header的Accept参数值改为“text/html”或“application/xml”等都可以）。

Accept=\*/\*：表示主类型任意，子类型任意，如“text/plain”、“application/xml”等都可以匹配。

#### 6.5.4.5、请求头数据中指定参数名!=值

@RequestMapping(value="/header/test7", headers = "Accept!=text/vnd.wap.wml")：表示请求的URL必须为“/header/test7” 且 请求头中必须有“Accept”参数但值不等于“text/vnd.wap.wml”即可匹配。

#### 6.5.4.6、组合使用是“且”的关系

@RequestMapping(value="/header/test8", headers = {"Accept!=text/vnd.wap.wml","abc=123"})：表示请求的URL必须为“/header/test8” 且 请求头中必须有“Accept”参数但值不等于“text/vnd.wap.wml”且 请求中必须有参数“abc=123”即可匹配。

注：Accept:text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

如果您的请求中含有Accept：“\*/\*”，则可以匹配功能处理方法上的如“text/html”、“text/\*”，“application/xml”等。

6.6.5、生产者、消费者限定

#### 6.6.5.1、基本概念

首先让我们看一下通过HTTP协议传输的媒体类型及如何表示媒体类型：

一、Media Type：

互联网媒体类型，一般就是我们所说的MIME类型，用来确定请求的内容类型或响应的内容类型。

 写道

**[plain]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20555477) [copy](http://blog.csdn.net/hongjun1847/article/details/20555477)

[在CODE上查看代码片](https://code.csdn.net/snippets/218982)

1. 媒体类型格式：type/subtype(;parameter)?
2. type主类型，任意的字符串，如text，如果是\*号代表所有；
3. subtype 子类型，任意的字符串，如html，如果是\*号代表所有；
4. parameter 可选，一些参数，如Accept请求头的q参数， Content-Type的 charset参数。

详见http://tools.ietf.org/html/rfc2616#section-3.7

 常见媒体类型：

**[plain]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20555477) [copy](http://blog.csdn.net/hongjun1847/article/details/20555477)

[在CODE上查看代码片](https://code.csdn.net/snippets/218982)

1. text/html ： HTML格式          text/plain ：纯文本格式             text/xml ：XML格式
2. image/gif ：gif图片格式          image/jpeg ：jpg图片格式          image/png：png图片格式
4. application/x-www-form-urlencoded ： <form encType=””>中默认的encType，form表单数据被编码为key/value格式发送到服务器（表单默认的提交数据的格式）。
5. multipart/form-data ： 当你需要在表单中进行文件上传时，就需要使用该格式；
7. application/xhtml+xml ：XHTML格式               application/xml     ： XML数据格式
8. application/atom+xml  ：Atom XML聚合格式    application/json    ： JSON数据格式
9. application/pdf       ：pdf格式                        application/msword  ： Word文档格式
10. application/octet-stream ： 二进制流数据（如常见的文件下载）。

在如tomcat服务器的 “conf/web.xml”中指定了扩展名到媒体类型的映射，在此我们可以看到服务器支持的媒体类型。

二、Content-Type：内容类型，即请求/响应的内容区数据的媒体类型；

2.1、请求头的内容类型，表示发送到服务器的内容数据的媒体类型；

request中设置请求头“Content-Type: application/x-www-form-urlencoded”表示请求的数据为key/value数据；

（1、控制器cn.javass.chapter6.web.controller.consumesproduces.contenttype.RequestContentTypeController

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20555477) [copy](http://blog.csdn.net/hongjun1847/article/details/20555477)

[在CODE上查看代码片](https://code.csdn.net/snippets/218982)

1. @RequestMapping(value = "/ContentType", method = RequestMethod.GET)
2. **public** String showForm() **throws** IOException {
3. //form表单，使用application/x-www-form-urlencoded编码方式提交表单
4. **return** "consumesproduces/Content-Type";
5. }
7. @RequestMapping(value = "/ContentType", method = RequestMethod.POST,
8. headers = "Content-Type=application/x-www-form-urlencoded")
9. **public** String request1(HttpServletRequest request) **throws** IOException {
10. //①得到请求的内容区数据的类型
11. String contentType = request.getContentType();
12. System.out.println("========ContentType:" + contentType);
13. //②得到请求的内容区数据的编码方式，如果请求中没有指定则为null
14. //注意，我们的CharacterEncodingFilter这个过滤器设置了编码(UTF-8)
15. //编码只能被指定一次，即如果客户端设置了编码，则过滤器不会再设置
16. String characterEncoding = request.getCharacterEncoding();
17. System.out.println("========CharacterEncoding:" + characterEncoding);
19. //③表示请求的内容区数据为form表单提交的参数，此时我们可以通过request.getParameter得到数据（key=value）
20. System.out.println(request.getParameter("realname"));
21. System.out.println(request.getParameter("username"));
22. **return** "success";
23. }

showForm功能处理方式：展示表单，且form的enctype="application/x-www-form-urlencoded"，在提交时请求的内容类型头为“Content-Type:application/x-www-form-urlencoded”；

request1功能处理方法：只对请求头为“Content-Type:application/x-www-form-urlencoded”的请求进行处理（即消费请求内容区数据）;

      request.getContentType()：可以得到请求头的内容区数据类型（即Content-Type头的值）

      request.getCharacterEncoding()：如“Content-Type:application/json;charset=GBK”,则得到的编码为“GBK”，否则如果你设置过滤器（CharacterEncodingFilter）则得到它设置的编码，否则返回null。

      request.getParameter()：因为请求的内容区数据为application/x-www-form-urlencoded格式的数据，因此我们可以通过request.getParameter()得到相应参数数据。

request中设置请求头“Content-Type:application/json;charset=GBK”表示请求的内容区数据为json类型数据，且内容区的数据以GBK进行编码；

（1、控制器cn.javass.chapter6.web.controller.consumesproduces.contenttype.RequestContentTypeController

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20555477) [copy](http://blog.csdn.net/hongjun1847/article/details/20555477)

[在CODE上查看代码片](https://code.csdn.net/snippets/218982)

1. @RequestMapping(value = "/request/ContentType", method = RequestMethod.POST,
2. headers = "Content-Type=application/json")
3. **public** String request2(HttpServletRequest request) **throws** IOException {
4. //①表示请求的内容区数据为json数据
5. InputStream is = request.getInputStream();
6. **byte** bytes[] = **new** **byte**[request.getContentLength()];
7. is.read(bytes);
8. //②得到请求中的内容区数据（以CharacterEncoding解码）
9. //此处得到数据后你可以通过如json-lib转换为其他对象
10. String jsonStr = **new** String(bytes, request.getCharacterEncoding());
11. System.out.println("json data：" + jsonStr);
12. **return** "success";
13. }

request2功能处理方法：只对请求头为“Content-Type:application/json”的进行请求处理（即消费请求内容区数据）;

      request.getContentLength()：可以得到请求头的内容区数据的长度；

      request.getCharacterEncoding()：如“Content-Type:application/json;charset=GBK”,则得到的编码为“GBK”，否则如果你设置过滤器（CharacterEncodingFilter）则得到它设置的编码，否则返回null。

      我们得到json的字符串形式后就能很简单的转换为JSON相关的对象。

（2、客户端发送json数据请求

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20555477) [copy](http://blog.csdn.net/hongjun1847/article/details/20555477)

[在CODE上查看代码片](https://code.csdn.net/snippets/218982)

1. //请求的地址
2. String url = "http://localhost:9080/springmvc-chapter6/request/ContentType";
3. //①创建Http Request(内部使用HttpURLConnection)
4. ClientHttpRequest request =
5. **new** SimpleClientHttpRequestFactory().
6. createRequest(**new** URI(url), HttpMethod.POST);
7. //②设置请求头的内容类型头和内容编码（GBK）
8. request.getHeaders().set("Content-Type", "application/json;charset=gbk");
9. //③以GBK编码写出请求内容体
10. String jsonData = "{\"username\":\"zhang\", \"password\":\"123\"}";
11. request.getBody().write(jsonData.getBytes("gbk"));
12. //④发送请求并得到响应
13. ClientHttpResponse response = request.execute();
14. System.out.println(response.getStatusCode());

此处我们使用Spring提供的Http客户端API SimpleClientHttpRequestFactory创建了请求并设置了请求的Content-Type和编码并在响应体中写回了json数据（即生产json类型的数据），此处是硬编码，实际工作可以使用json-lib等工具进行转换。

具体代码在cn.javass.chapter6.web.controller.consumesproduces.contenttype.RequestContentTypeClient。

2.2、响应头的内容类型，表示发送到客户端的内容数据类型，和请求头的内容类型类似，只是方向相反。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20555477) [copy](http://blog.csdn.net/hongjun1847/article/details/20555477)

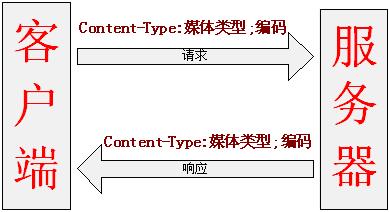
[在CODE上查看代码片](https://code.csdn.net/snippets/218982)

1. @RequestMapping("/response/ContentType")
2. **public** **void** response1(HttpServletResponse response) **throws** IOException {
3. //①表示响应的内容区数据的媒体类型为html格式，且编码为utf-8(客户端应该以utf-8解码)
4. response.setContentType("text/html;charset=utf-8");
5. //②写出响应体内容
6. response.getWriter().write("<font style='color:red'>hello</font>");
7. }

如上所示，通过response.setContentType("text/html;charset=utf-8") 告诉客户端响应体媒体类型为html，编码为utf-8，大家可以通过chrome工具查看响应头为“Content-Type:text/html;charset=utf-8”，还一个“Content-Length:36”表示响应体大小。

代码在cn.javass.chapter6.web.controller.consumesproduces.contenttype.ResponseContentTypeController。

如上代码可以看出Content-Type可以指定请求/响应的内容体的媒体格式和可选的编码方式。如图6-9



①客户端—发送请求—服务器：客户端通过请求头Content-Type指定内容体的媒体类型（即客户端此时是生产者），服务器根据Content-Type消费内容体数据（即服务器此时是消费者）；

②服务器—发送请求—客户端：服务器生产响应头Content-Type指定的响应体数据（即服务器此时是生产者），客户端根据Content-Type消费内容体数据（即客户端此时是消费者）。

问题：

①服务器端可以通过指定【headers = "Content-Type=application/json"】来声明可处理（可消费）的媒体类型，即只消费Content-Type指定的请求内容体数据；

②客户端如何告诉服务器端它只消费什么媒体类型的数据呢？即客户端接受（需要）什么类型的数据呢？服务器应该生产什么类型的数据？此时我们可以请求的Accept请求头来实现这个功能。

三、Accept：用来指定什么媒体类型的响应是可接受的，即告诉服务器我需要什么媒体类型的数据，此时服务器应该根据Accept请求头生产指定媒体类型的数据。

 2.1、json数据

（1、服务器端控制器

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20555477) [copy](http://blog.csdn.net/hongjun1847/article/details/20555477)

[在CODE上查看代码片](https://code.csdn.net/snippets/218982)

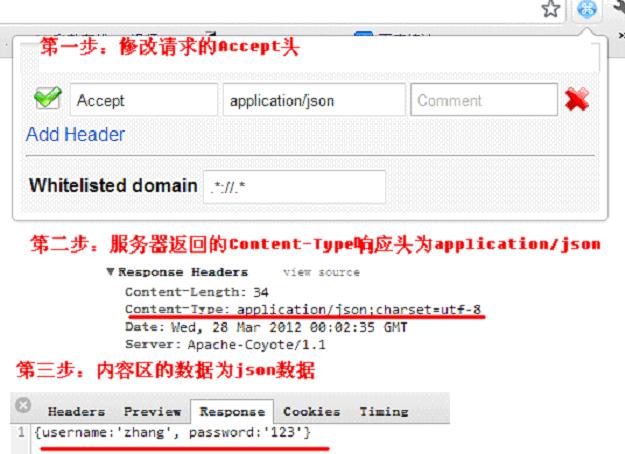
1. @RequestMapping(value = "/response/ContentType", headers = "Accept=application/json")
2. **public** **void** response2(HttpServletResponse response) **throws** IOException {
3. //①表示响应的内容区数据的媒体类型为json格式，且编码为utf-8(客户端应该以utf-8解码)
4. response.setContentType("application/json;charset=utf-8");
5. //②写出响应体内容
6. String jsonData = "{\"username\":\"zhang\", \"password\":\"123\"}";
7. response.getWriter().write(jsonData);
8. }

服务器根据请求头“Accept=application/json”生产json数据。

（2、客户端端接收服务器端json数据响应

使用浏览器测试(Ajax场景使用该方式)

请求地址为：http://localhost:9080/springmvc-chapter6/response/ContentType，且把修改请求头Accept改为“Accept=application/json”:



大家可以下载chrome的JSONView插件来以更好看的方式查看json数据，安装地址：https://chrome.google.com/webstore/detail/chklaanhfefbnpoihckbnefhakgolnmc

使用普通客户端测试（服务器之间通信可使用该方式）

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20555477) [copy](http://blog.csdn.net/hongjun1847/article/details/20555477)

[在CODE上查看代码片](https://code.csdn.net/snippets/218982)

1. **private** **static** **void** jsonRequest() **throws** IOException, URISyntaxException {
2. //请求的地址
3. String url = "http://localhost:9080/springmvc-chapter6/response/ContentType";
4. //①创建Http Request(内部使用HttpURLConnection)
5. ClientHttpRequest request =
6. **new** SimpleClientHttpRequestFactory().
7. createRequest(**new** URI(url), HttpMethod.POST);
8. //②设置客户端可接受的媒体类型（即需要什么类型的响应体数据）
9. request.getHeaders().set("Accept", "application/json");
10. //③发送请求并得到响应
11. ClientHttpResponse response = request.execute();
12. //④得到响应体的编码方式
13. Charset charset = response.getHeaders().getContentType().getCharSet();
14. //⑤得到响应体的内容
15. InputStream is = response.getBody();
16. **byte** bytes[] = **new** **byte**[(**int**)response.getHeaders().getContentLength()];
17. is.read(bytes);
18. String jsonData = **new** String(bytes, charset);
19. System.out.println("charset : " + charset + ", json data : " + jsonData);
20. }

request.getHeaders().set("Accept", "application/json")：表示客户端只接受（即只消费）json格式的响应数据；

response.getHeaders()：可以得到响应头，从而可以得到响应体的内容类型和编码、内容长度。

2.2、xml数据

（1、服务器端控制器

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20555477) [copy](http://blog.csdn.net/hongjun1847/article/details/20555477)

[在CODE上查看代码片](https://code.csdn.net/snippets/218982)

1. @RequestMapping(value = "/response/ContentType", headers = "Accept=application/xml")
2. **public** **void** response3(HttpServletResponse response) **throws** IOException {
3. //①表示响应的内容区数据的媒体类型为xml格式，且编码为utf-8(客户端应该以utf-8解码)
4. response.setContentType("application/xml;charset=utf-8");
5. //②写出响应体内容
6. String xmlData = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>";
7. xmlData += "<user><username>zhang</username><password>123</password></user>";
8. response.getWriter().write(xmlData);
9. }

和生产json数据唯一不同的两点：请求头为“Accept=application/xml”，响应体数据为xml。

（2、客户端端接收服务器端xml数据响应

使用浏览器测试(Ajax场景使用该方式)

请求地址为：http://localhost:9080/springmvc-chapter6/response/ContentType，且把修改请求头Accept改为“Accept=application/xml”,和json方式类似，此处不再重复。

使用普通客户端测试（服务器之间通信可使用该方式）

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20555477) [copy](http://blog.csdn.net/hongjun1847/article/details/20555477)

[在CODE上查看代码片](https://code.csdn.net/snippets/218982)

1. **private** **static** **void** xmlRequest() **throws** IOException, URISyntaxException {
2. //请求的地址
3. String url = "http://localhost:9080/springmvc-chapter6/response/ContentType";
4. //①创建Http Request(内部使用HttpURLConnection)
5. ClientHttpRequest request =
6. **new** SimpleClientHttpRequestFactory().
7. createRequest(**new** URI(url), HttpMethod.POST);
8. //②设置客户端可接受的媒体类型（即需要什么类型的响应体数据）
9. request.getHeaders().set("Accept", "application/xml");
10. //③发送请求并得到响应
11. ClientHttpResponse response = request.execute();
12. //④得到响应体的编码方式
13. Charset charset = response.getHeaders().getContentType().getCharSet();
14. //⑤得到响应体的内容
15. InputStream is = response.getBody();
16. **byte** bytes[] = **new** **byte**[(**int**)response.getHeaders().getContentLength()];
17. is.read(bytes);
18. String xmlData = **new** String(bytes, charset);
19. System.out.println("charset : " + charset + ", xml data : " + xmlData);
20. }

request.getHeaders().set("Accept", "application/xml")：表示客户端只接受（即只消费）xml格式的响应数据；

response.getHeaders()：可以得到响应头，从而可以得到响应体的内容类型和编码、内容长度。

许多开放平台，都提供了同一种数据的多种不同的表现形式，此时我们可以根据Accept请求头告诉它们我们需要什么类型的数据，他们根据我们的Accept来判断需要返回什么类型的数据。

实际项目使用Accept请求头是比较麻烦的，现在大多数开放平台（国内的新浪微博、淘宝、腾讯等开放平台）使用如下两种方式：

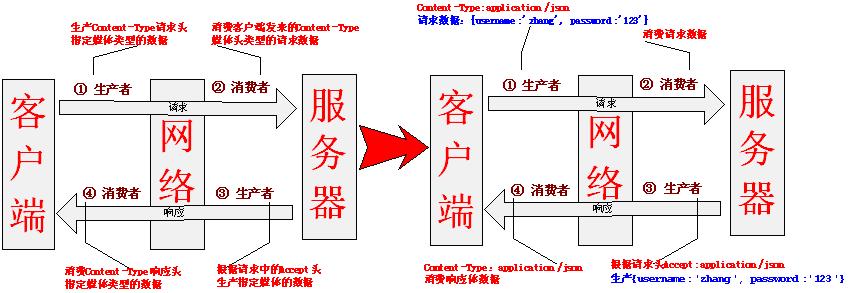
扩展名：如response/ContentType.json response/ContentType.xml方式，使用扩展名表示需要什么类型的数据；

参数：如response/ContentType?format=json response/ContentType?format=xml，使用参数表示需要什么类型的数据；

也就是说，目前我们可以使用如上三种方式实现来告诉服务器我们需要什么类型的数据，但麻烦的是现在有三种实现方式，难道我们为了支持三种类型的数据就要分别进行三种实现吗？当然不要这么麻烦，后续我们会学ContentNegotiatingViewResolver，它能帮助我们做到这一点。

#### 6.6.5.2、生产者消费者流程图

生产者消费者流程，如图6-10：



从图6-10可以看出：

请求阶段：客户端是生产者【生产Content-Type媒体类型的请求内容区数据】，服务器是消费者【消费客户端生产的Content-Type媒体类型的请求内容区数据】；

响应阶段：服务器是生产者【生产客户端请求头参数Accept指定的响应体数据】，客户端是消费者【消费服务器根据Accept请求头生产的响应体数据】。

如上生产者/消费者写法无法很好的体现我们分析的生产者/消费者模式，Spring3.1为生产者/消费者模式提供了简化支持，接下来我们学习一下如何在Spring3.1中来实现生产者/消费者模式吧。

#### 6.6.5.3、生产者、消费者限定

Spring3.1开始支持消费者、生产者限定，而且必须使用如下HandlerMapping和HandlerAdapter才支持：

**[html]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20555477) [copy](http://blog.csdn.net/hongjun1847/article/details/20555477)

[在CODE上查看代码片](https://code.csdn.net/snippets/218982)

1. <!--Spring3.1开始的注解 HandlerMapping -->
2. **<bean**
3. class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"**/>**
4. <!--Spring3.1开始的注解 HandlerAdapter -->
5. **<bean**
6. class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"**/>**

一、功能处理方法是消费者

@RequestMapping(value = "/consumes", consumes = {"application/json"})：此处使用consumes来指定功能处理方法能消费的媒体类型，其通过请求头的“Content-Type”来判断。

此种方式相对使用@RequestMapping的“headers = "Content-Type=application/json"”更能表明你的目的。

服务器控制器代码详解cn.javass.chapter6.web.controller.consumesproduces.ConsumesController；

客户端代码类似于之前的Content-Type中的客户端，详见ConsumesClient.java代码。

二、功能处理方法是生产者

@RequestMapping(value = "/produces", produces = "application/json")：表示将功能处理方法将生产json格式的数据，此时根据请求头中的Accept进行匹配，如请求头“Accept:application/json”时即可匹配;

@RequestMapping(value = "/produces", produces = "application/xml")：表示将功能处理方法将生产xml格式的数据，此时根据请求头中的Accept进行匹配，如请求头“Accept:application/xml”时即可匹配。

此种方式相对使用@RequestMapping的“headers = "Accept=application/json"”更能表明你的目的。

服务器控制器代码详解cn.javass.chapter6.web.controller.consumesproduces.ProducesController；

客户端代码类似于之前的Content-Type中的客户端，详见ProducesController.java代码。

当你有如下Accept头：

①Accept：text/html,application/xml,application/json

      将按照如下顺序进行produces的匹配 ①text/html ②application/xml ③application/json

②Accept：application/xml;q=0.5,application/json;q=0.9,text/html

      将按照如下顺序进行produces的匹配 ①text/html ②application/json ③application/xml

      q参数为媒体类型的质量因子，越大则优先权越高(从0到1)

③Accept：\*/\*,text/\*,text/html

      将按照如下顺序进行produces的匹配 ①text/html ②text/\* ③\*/\*

即匹配规则为：最明确的优先匹配。

代码详见ProducesPrecedenceController1、ProducesPrecedenceController2、ProducesPrecedenceController3。

Accept详细信息，请参考<http://tools.ietf.org/html/rfc2616#section-14.1>。

三、窄化时是覆盖 而 非继承

如类级别的映射为 @RequestMapping(value="/narrow", produces="text/html")，方法级别的为@RequestMapping(produces="application/xml")，此时方法级别的映射将覆盖类级别的，因此请求头“Accept:application/xml”是成功的，而“text/html”将报406错误码，表示不支持的请求媒体类型。

详见cn.javass.chapter6.web.controller.consumesproduces.NarrowController。

只有生产者/消费者 模式 是 覆盖，其他的使用方法是继承，如headers、params等都是继承。

四、组合使用是“或”的关系

@RequestMapping(produces={"text/html", "application/json"}) ：将匹配“Accept:text/html”或“Accept:application/json”。

五、问题

消费的数据，如JSON数据、XML数据都是由我们读取请求的InputStream并根据需要自己转换为相应的模型数据，比较麻烦；

生产的数据，如JSON数据、XML数据都是由我们自己先把模型数据转换为json/xml等数据，然后输出响应流，也是比较麻烦的。

Spring提供了一组注解（@RequestBody、@ResponseBody）和一组转换类（HttpMessageConverter）来完成我们遇到的问题，详见6.6.8节。

到目前为止，请求已经能交给我们的处理器进行处理了，接下来的事情是要进行收集数据啦，接下来我们看看我们能从请求中收集到哪些数据，如图6-11：

  
 图6-11

1、@RequestParam绑定单个请求参数值；

2、@PathVariable绑定URI模板变量值；

3、@CookieValue绑定Cookie数据值

4、@RequestHeader绑定请求头数据；

5、@ModelValue绑定参数到命令对象；

6、@SessionAttributes绑定命令对象到session；

7、@RequestBody绑定请求的内容区数据并能进行自动类型转换等。

8、@RequestPart绑定“multipart/data”数据，除了能绑定@RequestParam能做到的请求参数外，还能绑定上传的文件等。

除了上边提到的注解，我们还可以通过如HttpServletRequest等API得到请求数据，但推荐使用注解方式，因为使用起来更简单。

接下来先看一下功能处理方法支持的参数类型吧。

#### 6.6.1、功能处理方法支持的参数类型

在继续学习之前，我们需要首先看看功能处理方法支持哪些类型的形式参数，以及他们的具体含义。

 一、ServletRequest/HttpServletRequest 和 ServletResponse/HttpServletResponse

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20559771) [copy](http://blog.csdn.net/hongjun1847/article/details/20559771)

[在CODE上查看代码片](https://code.csdn.net/snippets/219191)

1. **public** String requestOrResponse (
2. ServletRequest servletRequest, HttpServletRequest httpServletRequest,
3. ServletResponse servletResponse, HttpServletResponse httpServletResponse
4. )

Spring Web MVC框架会自动帮助我们把相应的Servlet请求/响应（Servlet API）作为参数传递过来。

二、InputStream/OutputStream 和 Reader/Writer

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20559771) [copy](http://blog.csdn.net/hongjun1847/article/details/20559771)

[在CODE上查看代码片](https://code.csdn.net/snippets/219191)

1. **public** **void** inputOrOutBody(InputStream requestBodyIn, OutputStream responseBodyOut)
2. **throws** IOException {
3. responseBodyOut.write("success".getBytes());
4. }

requestBodyIn：获取请求的内容区字节流，等价于request.getInputStream();

responseBodyOut：获取相应的内容区字节流，等价于response.getOutputStream()。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20559771) [copy](http://blog.csdn.net/hongjun1847/article/details/20559771)

[在CODE上查看代码片](https://code.csdn.net/snippets/219191)

1. **public** **void** readerOrWriteBody(Reader reader, Writer writer)
2. **throws** IOException {
3. writer.write("hello");
4. }

reader：获取请求的内容区字符流，等价于request.getReader();

writer：获取相应的内容区字符流，等价于response.getWriter()。

InputStream/OutputStream 和 Reader/Writer两组不能同时使用，只能使用其中的一组。

三、WebRequest/NativeWebRequest

WebRequest是Spring Web MVC提供的统一请求访问接口，不仅仅可以访问请求相关数据（如参数区数据、请求头数据，但访问不到Cookie区数据），还可以访问会话和上下文中的数据；NativeWebRequest继承了WebRequest，并提供访问本地Servlet API的方法。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20559771) [copy](http://blog.csdn.net/hongjun1847/article/details/20559771)

[在CODE上查看代码片](https://code.csdn.net/snippets/219191)

1. **public** String webRequest(WebRequest webRequest, NativeWebRequest nativeWebRequest) {
2. System.out.println(webRequest.getParameter("test"));//①得到请求参数test的值
3. webRequest.setAttribute("name", "value", WebRequest.SCOPE\_REQUEST);//②
4. System.out.println(webRequest.getAttribute("name", WebRequest.SCOPE\_REQUEST));
5. HttpServletRequest request =
6. nativeWebRequest.getNativeRequest(HttpServletRequest.**class**);//③
7. HttpServletResponse response =
8. nativeWebRequest.getNativeResponse(HttpServletResponse.**class**);
9. **return** "success";
10. }

① webRequest.getParameter：访问请求参数区的数据，可以通过getHeader()访问请求头数据；

② webRequest.setAttribute/getAttribute：到指定的作用范围内取/放属性数据，Servlet定义的三个作用范围分别使用如下常量代表：

            SCOPE\_REQUEST ：代表请求作用范围；

           SCOPE\_SESSION ：代表会话作用范围；

           SCOPE\_GLOBAL\_SESSION ：代表全局会话作用范围，即ServletContext上下文作用范围。

③ nativeWebRequest.getNativeRequest/nativeWebRequest.getNativeResponse：得到本地的Servlet API。

四、HttpSession

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20559771) [copy](http://blog.csdn.net/hongjun1847/article/details/20559771)

[在CODE上查看代码片](https://code.csdn.net/snippets/219191)

1. **public** String session(HttpSession session) {
2. System.out.println(session);
3. **return** "success";
4. }

此处的session永远不为null。

注意：session访问不是线程安全的，如果需要线程安全，需要设置AnnotationMethodHandlerAdapter或RequestMappingHandlerAdapter的synchronizeOnSession属性为true，即可线程安全的访问session。

五、命令/表单对象

Spring Web MVC能够自动将请求参数绑定到功能处理方法的命令/表单对象上。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20559771) [copy](http://blog.csdn.net/hongjun1847/article/details/20559771)

[在CODE上查看代码片](https://code.csdn.net/snippets/219191)

1. @RequestMapping(value = "/commandObject", method = RequestMethod.GET)
2. **public** String toCreateUser(HttpServletRequest request, UserModel user) {
3. **return** "customer/create";
4. }
5. @RequestMapping(value = "/commandObject", method = RequestMethod.POST)
6. **public** String createUser(HttpServletRequest request, UserModel user) {
7. System.out.println(user);
8. **return** "success";
9. }

如果提交的表单（包含username和password文本域），将自动将请求参数绑定到命令对象user中去。

六、Model、Map、ModelMap

Spring Web MVC 提供Model、Map或ModelMap让我们能去暴露渲染视图需要的模型数据。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20559771) [copy](http://blog.csdn.net/hongjun1847/article/details/20559771)

[在CODE上查看代码片](https://code.csdn.net/snippets/219191)

1. @RequestMapping(value = "/model")
2. **public** String createUser(Model model, Map model2, ModelMap model3) {
3. model.addAttribute("a", "a");
4. model2.put("b", "b");
5. model3.put("c", "c");
6. System.out.println(model == model2);
7. System.out.println(model2 == model3);
8. **return** "success";                                                                                                               }

虽然此处注入的是三个不同的类型（Model model, Map model2, ModelMap model3），但三者是同一个对象，如图6-12所示：

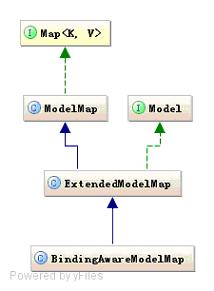


图6-11

AnnotationMethodHandlerAdapter和RequestMappingHandlerAdapter将使用BindingAwareModelMap作为模型对象的实现，即此处我们的形参（Model model, Map model2, ModelMap model3）都是同一个BindingAwareModelMap实例。

此处还有一点需要我们注意：

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20559771) [copy](http://blog.csdn.net/hongjun1847/article/details/20559771)

[在CODE上查看代码片](https://code.csdn.net/snippets/219191)

1. @RequestMapping(value = "/mergeModel")
2. **public** ModelAndView mergeModel(Model model) {
3. model.addAttribute("a", "a");//①添加模型数据
4. ModelAndView mv = **new** ModelAndView("success");
5. mv.addObject("a", "update");//②在视图渲染之前更新③处同名模型数据
6. model.addAttribute("a", "new");//③修改①处同名模型数据
7. //视图页面的a将显示为"update" 而不是"new"
8. **return** mv;
9. }

从代码中我们可以总结出功能处理方法的返回值中的模型数据（如ModelAndView）会 合并 功能处理方法形式参数中的模型数据（如Model），但如果两者之间有同名的，返回值中的模型数据会覆盖形式参数中的模型数据。

七、Errors/BindingResult

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20559771) [copy](http://blog.csdn.net/hongjun1847/article/details/20559771)

[在CODE上查看代码片](https://code.csdn.net/snippets/219191)

1. @RequestMapping(value = "/error1")
2. **public** String error1(UserModel user, BindingResult result)
4. @RequestMapping(value = "/error2")
5. **public** String error2(UserModel user, BindingResult result, Model model) {
7. @RequestMapping(value = "/error3")
8. **public** String error3(UserModel user, Errors errors)

以上代码都能获取错误对象。

Spring3.1之前（使用AnnotationMethodHandlerAdapter）错误对象必须紧跟在命令对象/表单对象之后，如下定义是错误的：

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20559771) [copy](http://blog.csdn.net/hongjun1847/article/details/20559771)

[在CODE上查看代码片](https://code.csdn.net/snippets/219191)

1. @RequestMapping(value = "/error4")
2. **public** String error4(UserModel user, Model model, Errors errors)
3. }

如上代码从Spring3.1开始（使用RequestMappingHandlerAdapter）将能正常工作，但还是推荐“错误对象紧跟在命令对象/表单对象之后”，这样是万无一失的。

Errors及BindingResult的详细使用请参考4.16.2数据验证。

八、其他杂项

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20559771) [copy](http://blog.csdn.net/hongjun1847/article/details/20559771)

1. **public** String other(Locale locale, Principal principal)

 java.util.Locale：得到当前请求的本地化信息，默认等价于ServletRequest.getLocale()，如果配置LocaleResolver解析器则由它决定Locale，后续介绍；

java.security.Principal：该主体对象包含了验证通过的用户信息，等价于HttpServletRequest.getUserPrincipal()。

以上测试在cn.javass.chapter6.web.controller.paramtype.MethodParamTypeController中。

其他功能处理方法的形式参数类型（如HttpEntity、UriComponentsBuilder、SessionStatus、RedirectAttributes）将在后续章节详细讲解。

第二部分会介绍注解方式的数据绑定。

#### 6.6.2、@RequestParam绑定单个请求参数值

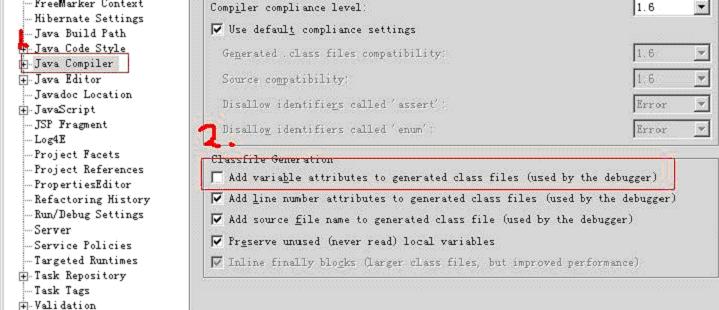
@RequestParam用于将请求参数区数据映射到功能处理方法的参数上。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. **public** String requestparam1(@RequestParam String username)

请求中包含username参数（如/requestparam1?username=zhang），则自动传入。

此处要特别注意：右击项目，选择“属性”，打开“属性对话框”，选择“Java Compiler”然后再打开的选项卡将“Add variable attributes to generated class files”取消勾选，意思是不将局部变量信息添加到类文件中，如图6-12所示：

  
 图6-12

当你在浏览器输入URL，如“requestparam1?username=123”时会报如下错误

Name for argument type [java.lang.String] not available, and parameter name information not found in class file either，表示得不到功能处理方法的参数名，此时我们需要如下方法进行入参：

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. **public** String requestparam2(@RequestParam("username") String username)

 即通过@RequestParam("username")明确告诉Spring Web MVC使用username进行入参。

接下来我们看一下@RequestParam注解主要有哪些参数：

value：参数名字，即入参的请求参数名字，如username表示请求的参数区中的名字为username的参数的值将传入；

required：是否必须，默认是true，表示请求中一定要有相应的参数，否则将报404错误码；

defaultValue：默认值，表示如果请求中没有同名参数时的默认值，默认值可以是SpEL表达式，如“#{systemProperties['java.vm.version']}”。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. **public** String requestparam4(@RequestParam(value="username",required=**false**) String username)

 表示请求中可以没有名字为username的参数，如果没有默认为null，此处需要注意如下几点：

     原子类型：必须有值，否则抛出异常，如果允许空值请使用包装类代替。

     Boolean包装类型类型：默认Boolean.FALSE，其他引用类型默认为null。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. **public** String requestparam5(
2. @RequestParam(value="username", required=**true**, defaultValue="zhang") String username)

表示如果请求中没有名字为username的参数，默认值为“zhang”。

如果请求中有多个同名的应该如何接收呢？如给用户授权时，可能授予多个权限，首先看下如下代码：

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. **public** String requestparam7(@RequestParam(value="role") String roleList)

如果请求参数类似于url?role=admin&rule=user，则实际roleList参数入参的数据为“admin,user”，即多个数据之间使用“，”分割；我们应该使用如下方式来接收多个请求参数：

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. **public** String requestparam7(@RequestParam(value="role") String[] roleList)

或

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. **public** String requestparam8(@RequestParam(value="list") List<String> list)

 到此@RequestParam我们就介绍完了，以上测试代码在cn.javass.chapter6.web.controller. paramtype.RequestParamTypeController中。

#### 6.6.3、@PathVariable绑定URI模板变量值

@PathVariable用于将请求URL中的模板变量映射到功能处理方法的参数上。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. @RequestMapping(value="/users/{userId}/topics/{topicId}")
2. **public** String test(
3. @PathVariable(value="userId") **int** userId,
4. @PathVariable(value="topicId") **int** topicId)

 如请求的URL为“控制器URL/users/123/topics/456”，则自动将URL中模板变量{userId}和{topicId}绑定到通过@PathVariable注解的同名参数上，即入参后userId=123、topicId=456。代码在PathVariableTypeController中。

#### 6.6.4、@CookieValue绑定Cookie数据值

@CookieValue用于将请求的Cookie数据映射到功能处理方法的参数上。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. **public** String test(@CookieValue(value="JSESSIONID", defaultValue="") String sessionId)

如上配置将自动将JSESSIONID值入参到sessionId参数上，defaultValue表示Cookie中没有JSESSIONID时默认为空。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. **public** String test2(@CookieValue(value="JSESSIONID", defaultValue="") Cookie sessionId)

传入参数类型也可以是javax.servlet.http.Cookie类型。

测试代码在CookieValueTypeController中。@CookieValue也拥有和@RequestParam相同的三个参数，含义一样。

#### 6.6.5、@RequestHeader绑定请求头数据

@RequestHeader用于将请求的头信息区数据映射到功能处理方法的参数上。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. @RequestMapping(value="/header")
2. **public** String test(
3. @RequestHeader("User-Agent") String userAgent,
4. @RequestHeader(value="Accept") String[] accepts)

如上配置将自动将请求头“User-Agent”值入参到userAgent参数上，并将“Accept”请求头值入参到accepts参数上。测试代码在HeaderValueTypeController中。

@RequestHeader也拥有和@RequestParam相同的三个参数，含义一样。

#### 6.6.6、@ModelAttribute绑定请求参数到命令对象

@ModelAttribute一个具有如下三个作用：

①绑定请求参数到命令对象：放在功能处理方法的入参上时，用于将多个请求参数绑定到一个命令对象，从而简化绑定流程，而且自动暴露为模型数据用于视图页面展示时使用；

②暴露表单引用对象为模型数据：放在处理器的一般方法（非功能处理方法）上时，是为表单准备要展示的表单引用对象，如注册时需要选择的所在城市等，而且在执行功能处理方法（@RequestMapping注解的方法）之前，自动添加到模型对象中，用于视图页面展示时使用；

③暴露@RequestMapping方法返回值为模型数据：放在功能处理方法的返回值上时，是暴露功能处理方法的返回值为模型数据，用于视图页面展示时使用。

一、绑定请求参数到命令对象

如用户登录，我们需要捕获用户登录的请求参数（用户名、密码）并封装为用户对象，此时我们可以使用@ModelAttribute绑定多个请求参数到我们的命令对象。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. **public** String test1(@ModelAttribute("user") UserModel user)

和6.6.1一节中的五、命令/表单对象功能一样。只是此处多了一个注解@ModelAttribute("user")，它的作用是将该绑定的命令对象以“user”为名称添加到模型对象中供视图页面展示使用。我们此时可以在视图页面使用${user.username}来获取绑定的命令对象的属性。

绑定请求参数到命令对象支持对象图导航式的绑定，如请求参数包含“?username=zhang&password=123&workInfo.city=bj”自动绑定到user中的workInfo属性的city属性中。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. @RequestMapping(value="/model2/{username}")
2. **public** String test2(@ModelAttribute("model") DataBinderTestModel model) {

DataBinderTestModel相关模型请从第三章拷贝过来，请求参数到命令对象的绑定规则详见【4.16.1、数据绑定】一节，URI模板变量也能自动绑定到命令对象中，当你请求的URL中包含“bool=yes&schooInfo.specialty=computer&hobbyList[0]=program&hobbyList[1]=music&map[key1]=value1&map[key2]=value2&state=blocked”会自动绑定到命令对象上。

当URI模板变量和请求参数同名时，URI模板变量具有高优先权。

二、暴露表单引用对象为模型数据

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. @ModelAttribute("cityList")
2. **public** List<String> cityList() {
3. **return** Arrays.asList("北京", "山东");
4. }

如上代码会在执行功能处理方法之前执行，并将其自动添加到模型对象中，在功能处理方法中调用Model 入参的containsAttribute("cityList")将会返回true。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. @ModelAttribute("user")  //①
2. **public** UserModel getUser(@RequestParam(value="username", defaultValue="") String username) {
3. //TODO 去数据库根据用户名查找用户对象
4. UserModel user = **new** UserModel();
5. user.setRealname("zhang");
6. **return** user;
7. }

如你要修改用户资料时一般需要根据用户的编号/用户名查找用户来进行编辑，此时可以通过如上代码查找要编辑的用户。

也可以进行一些默认值的处理。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. @RequestMapping(value="/model1") //②
2. **public** String test1(@ModelAttribute("user") UserModel user, Model model)

此处我们看到①和②有同名的命令对象，那Spring Web MVC内部如何处理的呢：

(1、首先执行@ModelAttribute注解的方法，准备视图展示时所需要的模型数据；@ModelAttribute注解方法形式参数规则和@RequestMapping规则一样，如可以有@RequestParam等；

（2、执行@RequestMapping注解方法，进行模型绑定时首先查找模型数据中是否含有同名对象，如果有直接使用，如果没有通过反射创建一个，因此②处的user将使用①处返回的命令对象。即②处的user等于①处的user。

三、暴露@RequestMapping方法返回值为模型数据

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. **public** @ModelAttribute("user2") UserModel test3(@ModelAttribute("user2") UserModel user)

大家可以看到返回值类型是命令对象类型，而且通过@ModelAttribute("user2")注解，此时会暴露返回值到模型数据（名字为user2）中供视图展示使用。那哪个视图应该展示呢？此时Spring Web MVC会根据RequestToViewNameTranslator进行逻辑视图名的翻译，详见【4.15.5、RequestToViewNameTranslator】一节。

此时又有问题了，@RequestMapping注解方法的入参user暴露到模型数据中的名字也是user2，其实我们能猜到：

（3、@ModelAttribute注解的返回值会覆盖@RequestMapping注解方法中的@ModelAttribute注解的同名命令对象。

四、匿名绑定命令参数

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. **public** String test4(@ModelAttribute UserModel user, Model model)
2. 或
3. **public** String test5(UserModel user, Model model)

此时我们没有为命令对象提供暴露到模型数据中的名字，此时的名字是什么呢？Spring Web MVC自动将简单类名（首字母小写）作为名字暴露，如“cn.javass.chapter6.model.UserModel”暴露的名字为“userModel”。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. **public** @ModelAttribute List<String> test6()
2. 或
3. **public** @ModelAttribute List<UserModel> test7()

对于集合类型（Collection接口的实现者们，包括数组），生成的模型对象属性名为“简单类名（首字母小写）”+“List”，如List<String>生成的模型对象属性名为“stringList”，List<UserModel>生成的模型对象属性名为“userModelList”。

其他情况一律都是使用简单类名（首字母小写）作为模型对象属性名，如Map<String, UserModel>类型的模型对象属性名为“map”。

#### 6.6.7、@SessionAttributes绑定命令对象到session

有时候我们需要在多次请求之间保持数据，一般情况需要我们明确的调用HttpSession的API来存取会话数据，如多步骤提交的表单。Spring Web MVC提供了@SessionAttributes进行请求间透明的存取会话数据。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. //1、在控制器类头上添加@SessionAttributes注解
2. @SessionAttributes(value = {"user"})    //①
3. **public** **class** SessionAttributeController
5. //2、@ModelAttribute注解的方法进行表单引用对象的创建
6. @ModelAttribute("user")    //②
7. **public** UserModel initUser()
9. //3、@RequestMapping注解方法的@ModelAttribute注解的参数进行命令对象的绑定
10. @RequestMapping("/session1")   //③
11. **public** String session1(@ModelAttribute("user") UserModel user)
13. //4、通过SessionStatus的setComplete()方法清除@SessionAttributes指定的会话数据
14. @RequestMapping("/session2")   //③
15. **public** String session(@ModelAttribute("user") UserModel user, SessionStatus status) {
16. **if**(**true**) { //④
17. status.setComplete();
18. }
19. **return** "success";
20. }

@SessionAttributes(value = {"user"})含义：

@SessionAttributes(value = {"user"}) 标识将模型数据中的名字为“user” 的对象存储到会话中（默认HttpSession），此处value指定将模型数据中的哪些数据（名字进行匹配）存储到会话中，此外还有一个types属性表示模型数据中的哪些类型的对象存储到会话范围内，如果同时指定value和types属性则那些名字和类型都匹配的对象才能存储到会话范围内。

包含@SessionAttributes的执行流程如下所示：

① 首先根据@SessionAttributes注解信息查找会话内的对象放入到模型数据中；

② 执行@ModelAttribute注解的方法：如果模型数据中包含同名的数据，则不执行@ModelAttribute注解方法进行准备表单引用数据，而是使用①步骤中的会话数据；如果模型数据中不包含同名的数据，执行@ModelAttribute注解的方法并将返回值添加到模型数据中；

③ 执行@RequestMapping方法，绑定@ModelAttribute注解的参数：查找模型数据中是否有@ModelAttribute注解的同名对象，如果有直接使用，否则通过反射创建一个；并将请求参数绑定到该命令对象；

此处需要注意：如果使用@SessionAttributes注解控制器类之后，③步骤一定是从模型对象中取得同名的命令对象，如果模型数据中不存在将抛出HttpSessionRequiredException Expected session attribute ‘user’(Spring3.1)

或HttpSessionRequiredException Session attribute ‘user’ required - not found in session(Spring3.0)异常。

④ 如果会话可以销毁了，如多步骤提交表单的最后一步，此时可以调用SessionStatus对象的setComplete()标识当前会话的@SessionAttributes指定的数据可以清理了，此时当@RequestMapping功能处理方法执行完毕会进行清理会话数据。

我们通过Spring Web MVC的源代码验证一下吧，此处我们分析的是Spring3.1的RequestMappingHandlerAdapter，读者可以自行验证Spring3.0的AnnotationMethodHandlerAdapter，流程一样：

（1、RequestMappingHandlerAdapter.invokeHandlerMethod

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. //1、RequestMappingHandlerAdapter首先调用ModelFactory的initModel方法准备模型数据：
2. modelFactory.initModel(webRequest, mavContainer, requestMappingMethod);
3. //2、调用@RequestMapping注解的功能处理方法
4. requestMappingMethod.invokeAndHandle(webRequest, mavContainer);
5. //3、更新/合并模型数据
6. modelFactory.updateModel(webRequest, mavContainer);

（2、ModelFactory.initModel

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. Map<String, ?> attributesInSession = **this**.sessionAttributesHandler.retrieveAttributes(request);
2. //1.1、将与@SessionAttributes注解相关的会话对象放入模型数据中
3. mavContainer.mergeAttributes(attributesInSession);
4. //1.2、调用@ModelAttribute方法添加表单引用对象
5. invokeModelAttributeMethods(request, mavContainer);
6. //1.3、验证模型数据中是否包含@SessionAttributes注解相关的会话对象，不包含抛出异常
7. **for** (String name : findSessionAttributeArguments(handlerMethod)) {
8. **if** (!mavContainer.containsAttribute(name)) {
9. //1.4、此处防止在@ModelAttribute注解方法又添加了会话对象
10. //如在@ModelAttribute注解方法调用session.setAttribute("user", new UserModel());
11. Object value = **this**.sessionAttributesHandler.retrieveAttribute(request, name);
12. **if** (value == **null**) {
13. **throw** **new** HttpSessionRequiredException("Expected session attribute '" + name + "'");
14. }
15. mavContainer.addAttribute(name, value);
16. }

（3、ModelFactory.invokeModelAttributeMethods

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. **for** (InvocableHandlerMethod attrMethod : **this**.attributeMethods) {
2. String modelName = attrMethod.getMethodAnnotation(ModelAttribute.**class**).value();
3. //1.2.1、如果模型数据中包含同名数据则不再添加
4. **if** (mavContainer.containsAttribute(modelName)) {
5. **continue**;
6. }
7. //1.2.2、调用@ModelAttribute注解方法并将返回值添加到模型数据中，此处省略实现代码
8. }

（4、requestMappingMethod.invokeAndHandle 调用功能处理方法，此处省略

（5、ModelFactory.updateMode 更新模型数据

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. //3.1、如果会话被标识为完成，此时从会话中清除@SessionAttributes注解相关的会话对象
2. **if** (mavContainer.getSessionStatus().isComplete()){
3. **this**.sessionAttributesHandler.cleanupAttributes(request);
4. }
5. //3.2、如果会话没有完成，将模型数据中的@SessionAttributes注解相关的对象添加到会话中
6. **else** {
7. **this**.sessionAttributesHandler.storeAttributes(request, mavContainer.getModel());
8. }
9. //省略部分代码

到此@SessionAtrribute介绍完毕，测试代码在cn.javass.chapter6.web.controller.paramtype.SessionAttributeController中。

另外cn.javass.chapter6.web.controller.paramtype.WizardFormController是一个类似于【4.11、AbstractWizardFormController】中介绍的多步骤表单实现，此处不再贴代码，多步骤提交表单需要考虑会话超时问题，这种方式可能对用户不太友好，我们可以采取隐藏表单（即当前步骤将其他步骤的表单隐藏）或表单数据存数据库（每步骤更新下数据库数据）等方案解决。

#### 6.6.8、@Value绑定SpEL表示式

@Value用于将一个SpEL表达式结果映射到到功能处理方法的参数上。

**[java]** [view plain](http://blog.csdn.net/hongjun1847/article/details/20560515) [copy](http://blog.csdn.net/hongjun1847/article/details/20560515)

1. **public** String test(@Value("#{systemProperties['java.vm.version']}") String jvmVersion)

到此数据绑定我们就介绍完了，对于没有介绍的方法参数和注解（包括自定义注解）在后续章节进行介绍。接下来我们学习下数据类型转换吧。