第3章

控制器层的常用类和注解


视频讲解

本章主要介绍 Spring MVC 框架中位于控制器层的常用类和注解的用法，第 4 章会介绍如何用 Spring 标签在视图层创建 HTML 表单。

本章提供的范例也位于 helloapp 应用中。在本章范例中，访问控制器类的请求处理方法的 URL 入口都以 helloapp 作为根路径。

本书为了节省篇幅，在展示部分 Java 类的源代码时，没有列出 import 语句。如果要了解一个由 Spring API 提供的类或注解到底来自哪个包，可以参考本书提供的配套源代码，或者查阅 Spring 的 JavaDoc 文档，下载地址为 https://search.maven.org/search?q=g:org.springframework。

3.1 用@Controller 注解标识控制器类

把一个类用@Controller 注解标识，这个类就变成了 Spring MVC 框架中的控制器类。不过，要让这个控制器类服从 Spring MVC 框架的统一调遣，还必须确保 Spring MVC 框架在启动时，会扫描到控制器类中的@Controller 注解，从而把这个控制器类收编到自己的管辖范围内，即把它注册到 Spring MVC 框架中。

在 Spring MVC 的配置文件中，以下代码用于告诉 Spring MVC 框架在哪些 Java 包中扫描 Java 类的 Spring 注解。

```
<context:component-scan base-package = "mypack" />
```

这段代码告诉 Spring MVC 框架需要扫描 mypack 包以及子包中的 Java 类的@Controller 等 Spring 注解。如果希望 Spring MVC 框架忽略扫描某些注解，可以用<context:exclude-filter >元素来设定。例如，以下代码告诉 Spring MVC 框架忽略 Java 类中的@Service 注解。

```
<context:component-scan base-package = "mypack" />
  <context:exclude-filter type = "annotation"
   expression = "org.springframework.stereotype.Service" />
</context:component-scan>
```

3.2 控制器对象的存在范围

一旦 Controller 类按照 3.1 节的方式向 Spring MVC 框架进行了注册，Spring MVC 框架就会管理 Controller 对象的生命周期。

默认情况下，Controller 对象的存在范围为 singleton(单例)，即在整个应用程序的生命周期内，一个 Controller 类只有一个实例。

singleton 范围的优点是节省内存空间，但是也存在以下两个缺点。

(1) 当大量客户请求同时访问一个 Controller 对象的共享数据时，容易造成并发问题。

(2) 如果一个 Controller 对象采用了线程同步机制，那么当大量客户请求同时访问这个 Controller 对象时，会导致部分处理客户请求的线程阻塞，影响 Web 应用的并发性能。

为了克服以上缺点，Spring MVC 框架允许把一个 Controller 对象的存在范围设置为 request 或 session，具体细节如下。

(1) request 范围：对于每一个 HTTP 请求，Spring MVC 框架创建一个 Controller 对象。当完成了对这个 HTTP 请求的响应，Controller 对象就结束生命周期。

(2) session 范围：对于每一个 HTTP 会话，Spring MVC 框架创建一个 Controller 对象。当这个 HTTP 会话结束，Controller 对象就结束生命周期。

在以下代码中，ControllerA 和 ControllerB 分别使用了@RequestScope 和@SessionScope 注解，它们的范围分别为 request 和 session。

```
@Controller
@RequestScope                    //ControllerA 的存在范围为 request
public class ControllerA{}

@Controller
@SessionScope                    //ControllerB 的存在范围为 session
public class ControllerB{}
```

@RequestScope 注解等价于@Scope("request")，@SessionScope 注解等价于@Scope("session")。

除了 request 和 session 范围，还可以把 Controller 对象的存在范围设为 application，这意味着在整个 Web 应用的生命周期内，只有一个 Controller 对象，例如：

```
@Controller
@ApplicationScope                //等价于@Scope("application")
public class ControllerA{}
```

3.3　设置控制器类的请求处理方法的 URL 入口

Controller 类和普通的 Java 类一样，可以包含任意方法。在这些方法中，能够被客户端直接请求访问的方法称为请求处理方法。本章提到的控制器类的方法，如果未作特别说明，都是指请求处理方法。

Spring MVC 框架允许灵活地为请求处理方法指定 URL 入口，具体细节如下。

（1）通过@RequestMapping 注解的 value 属性设定 URL 入口。

（2）通过@RequestMapping 注解的 params、method 和 headers 等属性进一步限制 URL 入口。

3.3.1　设置 URL 入口的普通方式

@RequestMapping 注解既可以标识控制器类，也可以标识方法。以下 ControllerA 的三个方法都使用了@RequestMapping 注解。

```
@Controller
public class ControllerA{
    //设定请求方式以及多个 URL 入口
  @RequestMapping(value = {"/input","/"}, method = RequestMethod.GET)
  public String method1(){…}

  @RequestMapping(value = {"/hello "})      //设定一个 URL 入口
  public String method2(){…}

  @RequestMapping("go")                     //直接设定 URL 入口
  public String method3(){…}
}
```

method1()方法的 URL 入口为 helloapp/input 以及 helloapp/，并且请求方式必须为 GET。method2()方法的 URL 入口为 helloapp/hello，无须考虑是 GET 请求方式还是 POST 等其他请求方式。method3()方法的 URL 入口为 helloapp/go，无须考虑是 GET 请求方式还是 POST 等其他请求方式。

以下 ControllerB 在类和方法前都使用了@RequestMapping 注解。

```
@Controller
@RequestMapping("person")                   //设定相对根路径
public class ControllerB{

  @RequestMapping("/save")
  public String method1(){…}

  @RequestMapping("/")
  public String method2(){…}
}
```

method1()方法的 URL 入口为 helloapp/person/save。method2()方法的 URL 入口为 helloapp/person/。

由此可见,位于控制器类之前的@RequestMapping 注解设定访问该控制器类的所有请求处理方法的相对根路径。

3.3.2 限制 URL 入口的请求参数、请求方式和请求头

在@RequestMapping 注解中不仅可以通过 value 属性设定 URL 入口,还可以通过 params 属性、method 属性和 headers 属性来进一步限制 URL 入口。

1. params 属性

params 属性用于限制客户端访问请求处理方法的请求参数。例如:

```
@RequestMapping(value = "test",
           params = { "username=weiqin", "address","!phone" })
public String testParam() {…}
```

以上代码中@RequestMapping 注解通过 params 属性设置了三个请求参数:username、address 和 phone。客户端访问 testParam()方法的 URL 必须为 helloapp/test。此外,其请求参数还必须满足以下三个条件。

(1) 包含 username 请求参数,并且取值为 weiqin。

(2) 包含 address 请求参数,取值无所谓。

(3) 不能包含 phone 请求参数。

以下 URL 能正常访问 testParam()方法。

```
http://localhost:8080/helloapp/test?
          username=weiqin&address=shanghai

http://localhost:8080/helloapp/test?
          username=weiqin&address=beijing&gender=female
```

以下 URL 不会访问 testParam()方法。

```
//username 取值不是 weiqin
http://localhost:8080/helloapp/test?username=Mary&address=shanghai

//包含 phone 请求参数不符合要求
http://localhost:8080/helloapp/test?
         username=weiqin&address=beijing&phone=56567878

//没有包含 address 请求参数
http://localhost:8080/helloapp/test?username=weiqin

//没有包含取值为 weiqin 的 username 请求参数
http://localhost:8080/helloapp/test?address=shanghai
```

2. method 属性

method 属性用于限制客户端访问请求处理方法的请求方式。例如：

```
@RequestMapping(value = "test",
            method = { RequestMethod.GET,RequestMethod.DELETE })
public String testMethod(){…}
```

以上代码表明，当 URL 为 helloapp/test 并且客户端请求方式为 GET 或 DELETE，Spring MVC 框架会调用 testMethod()方法。

3. headers 属性

headers 属性用于限制客户端访问请求处理方法的请求头。例如：

```
@RequestMapping(value = "test",
headers = { "Host=localhost","Accept","!Referer" })
public String testHeaders() {…}
```

headers 属性的赋值语法与 params 属性相似。客户端访问 testHeaders()方法的 URL 必须为 helloapp/test。此外，其请求头还必须满足以下三个条件。

(1) 包含 Host 项，并且取值为 localhost。

(2) 包含 Accept 项，取值无所谓。

(3) 不能包含 Referer 项。

3.3.3　@GetMapping 和@PostMapping 等简化形式的注解

为了简化@RequestMapping 注解中请求方式的设置，Spring MVC 框架还提供了以下 4 种简化形式的映射注解。

(1) @GetMapping：指定请求方式为 GET。

(2) @PostMapping：指定请求方式为 POST。

(3) @PutMapping：指定请求方式为 PUT。

(4) @DeleteMapping：指定请求方式为 DELETE。

例如以下三种映射方式是等价的。

```
//方式一
@RequestMapping(value = "test", method = { RequestMethod.GET })
public String test(){…}

//方式二
@GetMapping(value = "test")
public String test(){…}

//方式三
@GetMapping("test")
public String test(){…}
```

值得注意的是,@GetMapping 等注解只能用来标识请求处理方法,而不能用来标识控制器类,这是和@RequestMapping 注解的一个区别。

3.4 绑定 HTTP 请求数据和控制器类的方法参数

Controller 类负责处理具体的客户请求,需要读取来自客户端的 HTTP 请求数据。在控制器类中读取请求数据的原生态方式是从 HttpServletRequest 对象中读取各种请求数据,例如:

```
@RequestMapping(value = "test")
public String test(HttpServletRequest request){
  //读取请求参数
  String userName = request.getParameter("userName");
  //读取请求头中的 Host 项
  String host = request.getHeader("Host");
  //读取 Cookie
  Cookie[] cookies = request.getCookies();

  return "result";
}
```

此外,Spring MVC 框架对 HttpServletRequest 对象进行了封装,允许在控制器类的请求处理方法中,把特定的请求数据与方法参数绑定。所谓绑定,这里是指由 Spring MVC 框架自动把请求数据赋值给方法参数。这样,控制器类的方法直接从方法参数中就能获取特定请求数据,省去了从 HttpServletRequest 对象中读取特定请求数据的操作。

HTTP 请求数据主要包括以下三部分内容。

(1) 请求参数。HTML 表单数据也属于请求参数。

(2) 请求头。

(3) Cookie。

Spring MVC 框架允许采用以下 5 种方式把请求数据和方法参数绑定。

(1) 直接定义和请求参数同名的方法参数。

(2) 用@RequestParam 注解绑定请求参数。

(3) 用@RequestHeader 注解绑定请求头。

(4) 用@CookieValue 注解绑定 Cookie。

(5) 用@PathVariable 注解绑定 RESTFul 风格的 URL 变量。

3.4.1 直接定义和请求参数同名的方法参数

在控制器类的请求处理方法中,把请求参数与方法参数绑定的最直接方式是定义和请求参数同名的方法参数。例如:

```
@RequestMapping("test")
public String testParam(String name, int age, String address) {
```

```
    System.out.println("name = " + name);
    System.out.println("age = " + age);
    System.out.println("adress = " + address);
    return "result";
}
```

以下 URL 会访问 testParam()方法。

```
http://localhost:8080/helloapp/test?name = Tom&age = 22&address = Shanghai
```

该 URL 中包含 name、age 和 address 三个请求参数。Spring MVC 框架调用 testParam()方法时，会把这三个请求参数分别赋值给 testParam()的 name 参数、age 参数和 address 参数。testParam()方法会在服务器端打印以下内容。

```
name = Tom
age = 22
address = Shanghai
```

3.4.2　用@RequestParam 注解绑定请求参数

@RequestParam 注解能把特定请求参数和请求处理方法中的方法参数绑定，例如：

```
@RequestMapping("test")
public String testParam(
    @RequestParam(required = false,defaultValue = "Guest") String name,
    @RequestParam(name = "age") int age,
    @RequestParam("address") String homeAddress) {

  System.out.println("name = " + name);
  System.out.println("age = " + age);
  System.out.println("homeAdress = " + homeAddress);
  return "result";
}
```

以上 testParam()方法通过@RequestParam 注解绑定了三个请求参数。

(1) 第一个@RequestParam 注解把 name 请求参数绑定到 name 方法参数。默认情况下，请求参数与方法参数同名。required 属性的默认值为 true，这里 required 属性取值为 false，表示这个不是必须提供的请求参数。defaultValue 属性指定 name 请求参数的默认值。

(2) 第二个@RequestParam 注解把 age 请求参数绑定到 age 方法参数。

(3) 第三个@RequestParam 注解把 address 请求参数绑定到 homeAddress 方法参数。

以下 URL 会访问 testParam()方法。

```
http://localhost:8080/helloapp/test?name = Tom&age = 22&address = Shanghai
```

该 URL 包含 name、age 和 address 三个请求参数，testParam()方法会在服务器端打印以下内容。

```
name = Tom
age = 22
homeAddress = Shanghai
```

当客户端请求访问的 URL 为 http://localhost:8080/helloapp/test? age=22&address=Shanghai。该 URL 包含 age 和 address 两个请求参数，name 请求参数取默认值 Guest，testParam()方法会在服务器端打印以下内容。

```
name = Guest
age = 22
homeAddress = Shanghai
```

当客户端请求访问的 URL 为 http://localhost:8080/helloapp/test? name=Tom&age=22。由于该 URL 没有提供 address 请求参数，因此客户端的浏览器会得到以下错误信息。

```
Required String parameter 'address' is not present
```

@RequestParam 注解以及后文将提到的@CookieValue 注解和@RequestHeader 注解具有一些共同的属性，参见表 3-1。

表 3-1 @RequestParam、@CookieValue 和@RequestHeader 的共同属性

属　性	描　述
name 属性	指定请求数据的名字
value 属性	name 属性的别名、作用和 name 属性相同
required 属性	默认值为 true，指明是否为必须提供的请求数据
deafultValue 属性	请求数据的默认值

value 属性是 name 属性的别名，因此两者的作用是相同的。例如以下四个@RequestParam 注解的作用相同。

```
@RequestParam(name = "age") int age
@RequestParam(value = "age") int age
@RequestParam("age") int age
@RequestParam int age
```

3.4.3 用@RequestHeader 注解绑定 HTTP 请求头

在 HTTP 请求头中会包含客户端的主机地址、浏览器类型、请求正文的数据类型以及请求正文的长度等信息。@RequestHeader 注解能够把请求头中的特定项和请求处理方法

的参数绑定，例如：

```
@RequestMapping("test")
public String testRequestHeader(@RequestHeader("Host") String hostAddr,
        @RequestHeader String Host, @RequestHeader String host) {
  System.out.println(hostAddr + "-----" + Host + "-----" + host);
  return "result";
}
```

以上 testRequestHeader()方法使用了三个@RequestHeader 注解：

(1) 第一个注解把请求头中名为 Host 的请求项与方法参数 hostAddr 绑定。

(2) 第二个注解把请求头中名为 Host 或 host 的请求项与方法参数 Host 绑定。默认情况下，请求头中请求项和方法参数同名，但是不区分大小写。

(3) 第三个注解把请求头中名为 Host 或者 host 的请求项与方法参数 host 绑定。

通过浏览器访问 http://localhost:8080/helloapp/test，testRequestHeader()方法会在服务器端打印以下信息。

```
localhost:8080-----localhost:8080-----localhost:8080
```

3.4.4　用@CookieValue 注解绑定 Cookie

先通过一个简单的例子来引入 Cookie 作用的介绍。用户第一次到一个健身房去健身，健身房会为用户办理一张会员卡，这个会员卡由用户保管。以后用户每次去健身房，都会先出示会员卡。Cookie 就类似于会员卡。Cookie 是服务器端事先发送给客户端的用来跟踪客户端状态的一些数据，采用“Cookie 名字＝Cookie 值”的数据格式。客户端得到服务器端发送的 Cookie 后，会把它保存在本地机器上，以后客户端再向服务器端发送请求时，会在请求数据中包含 Cookie 信息。

假定客户端请求访问服务器端的一个控制器类的 testCookie()方法时，在请求数据中包含两个 Cookie：“username＝Tom”和“address＝Shanghai”。以下 testCookie()方法会读取这两个 Cookie。

```
@RequestMapping("test")
public String testCookie(@CookieValue String username,
    @CookieValue("address")String homeAddress) {

  System.out.println("username = " + username);
  System.out.println("homeAddress = " + homeAddress);

  return "result";
}
```

第一个@CookieValue 注解把名字为 username 的 Cookie 和 username 方法参数绑定。默认情况下，Cookie 的名字和方法参数的名字相同。第二个@CookieValue 注解把名字为

address 的 Cookie 和 homeAddress 方法参数绑定。运行 testCookie()方法,在服务器端会打印以下内容。

```
username = Tom
homeAddress = Shanghai
```

如果客户端的请求数据中不包含名字为 username 或者 address 的 Cookie,那么当客户端试图访问 testCookie()方法时,客户端的浏览器会收到以下错误信息。

```
Missing cookie 'username' for method parameter of type String
```

3.4.5 用@PathVariable 注解绑定 RESTFul 风格的 URL 变量

在访问请求处理方法的 URL 中可以加入一些变量,@PathVariable 注解能够把这些 URL 变量和方法参数绑定。第 15 章会进一步介绍 RESTFul 风格的概念和用法。

例程 3-1 的 testPath()方法就使用了@PathVariable 注解。

例程 3-1 TestPathController.java

```
@Controller
@RequestMapping("/main/{variable1}")
public class TestPathController {
  @RequestMapping("/test/{variable2}")
  public String testPath (@PathVariable String variable1,
              @PathVariable("variable2") int variable2) {
    System.out.println("variable1 = " + variable1);
    System.out.println("variable2 = " + variable2);

    return "result";
  }
}
```

第一个@RequestMapping 注解设置的 URL 中包含一个 variable1 变量。第二个@RequestMapping 注解设置的 URL 中包含一个 variable2 变量。

在 testPath()方法中,第一个@PathVariable 注解把 URL 中的 variable1 变量和 variable1 方法参数绑定,默认情况下,URL 变量的名字和方法参数的名字相同。第二个@PathVariable 注解把 URL 中的 variable2 变量和 variable2 方法参数绑定。

当浏览器端请求访问的 URL 为 http://localhost:8080/helloapp/main/hello/test/100,URL 中 variable1 变量的取值为 hello, variable2 变量的取值为 100,因此 testPath()方法在服务器端打印以下内容。

```
variable1 = hello
variable2 = 100
```

3.4.6　把一组请求参数和一个 JavaBean 类型的方法参数绑定

Spring MVC 框架还会自动把一组请求参数(包括表单数据)转换成一个 JavaBean,再把这个 JavaBean 和方法参数绑定。例如 Product 类是一个 JavaBean,它有以下两个属性以及相应的 get 和 set 方法。

```
private String name;
private double price;
```

以下 ProductController 类的 getDetail()方法把 name 请求参数和 price 请求参数转换成一个 Product 对象,再和 product 参数绑定。

```
@Controller
public class ProductController {
  @RequestMapping("/product")
  public String getDetail(Product product) {
    System.out.println("name:" + product.getName());
    System.out.println("price:" + product.getPrice());
    return "result";
  }
}
```

通过浏览器访问 getDetail()方法,URL 为 http://localhost:8080/helloapp/product?name=book&price=25,getDetail()方法会在服务器端打印如下内容:

```
name:book
price:25
```

3.5　请求参数的类型转换

在 Controller 类中,如果通过 HttpServletRequest 的 getParameter()方法来读取请求参数,返回的是 String 类型的请求参数值。如果要获得其他类型的参数,需要在程序中进行类型转换,例如:

```
String param = request.getParameter("age");
int age = Integer.parseInt(param);              //把字符串类型转换为 int 类型
```

而通过 Spring MVC 框架把请求参数绑定到控制器类的方法参数时,Spring MVC 框架会利用内置的数据类型转换器,对一些常见的数据类型自动进行类型转换。例如在以下 testParam()方法中,会把各种类型的请求参数与方法参数绑定。

```
@RequestMapping("test")
public String testParam(String name, int age,
```

```
                    boolean isMarried,double weight) {
    System.out.println("name = " + name);
    System.out.println("age = " + age);
    System.out.println("isMarried = " + isMarried);
    System.out.println("weight = " + weight);

    return "result";
}
```

当客户端请求访问的 URL 为 http://localhost:8080/helloapp/test? name=Tom&age=22&isMarried=false&weight=53.5,该 URL 中包含 name、age、isMarried 和 weight 四个请求参数,testParam()方法会在服务器端打印以下内容。

```
name = Tom
age = 22
isMarried = false
weight = 53.5
```

由此可见,Spring MVC 框架会根据请求处理方法的参数类型,自动把 String 类型转换为 int 类型、boolean 类型或者 double 类型等。

对于复杂的数据类型,还可以自定义类型转换器。例如,假定客户端提供的请求参数为表示用户信息的字符串"Tom,22,false,53.5",如果希望 Spring MVC 框架能把它先转换成一个 Person 对象,再传给控制器类的方法,该如何实现呢? 接下来就结合具体的范例介绍创建和使用自定义类型转换器的 5 个步骤。

(1) 创建 hello.jsp,它接收用户输入的 Person 信息,最后会把 Person 信息显示到网页上。

(2) 创建包含 Person 信息的 Person 类。

(3) 创建类型转换器 PersonConverter 类,它把 String 类型的 Person 信息转换成 Person 对象。

(4) 在 Spring MVC 配置文件中注册 PersonConverter 类型转换器。

(5) 创建控制器类 PersonController,它读取经过数据类型转换的 person 参数,把它保存在 Model 中,再由 hello.jsp 显示 Person 信息。

3.5.1 创建包含表单的 hello.jsp

在 hello.jsp 中定义了一个表单,它包含一个名字为 personInfo 的文本框。此外,它还通过 EL 表达式显示用户输入的表单数据。例程 3-2 是 hello.jsp 的主要源代码。

例程 3-2 hello.jsp 的主要源代码

```
<spring:message code = "hello.jsp.prompt.person.userName"/>
${person.userName}<br>
<spring:message code = "hello.jsp.prompt.person.age"/>
${person.age}<br>
```

```
<spring:message code="hello.jsp.prompt.person.isMarried"/>
${person.isMarried}<br>
<spring:message code="hello.jsp.prompt.person.weight"/>
${person.weight}<br><br>

<form action="${pageContext.request.contextPath}/sayHello"
            method="POST">
  <spring:message code="hello.jsp.prompt.person"/>
  <input type="text" name="personInfo"
            value="${person.personInfo}" /><br>
  <input type="submit" value="Submit"/>
</form>
```

3.5.2　创建包含 Person 信息的 Person 类

在 Person 类中定义了 userName、age、isMarried 和 weight 属性，以及相应的 get 和 set 方法。例程 3-3 是 Person 类的源代码。

例程 3-3　Person.java

```
public class Person {
  private String userName = null;
  private int age;
  private boolean isMarried;
  private double weight;

  public String getUserName() {
      return this.userName;
  }
  public void setUserName(String userName) {
    this.userName = userName;
  }
  …
  public String getPersonInfo( ) {
    return(userName + "," + age + "," + isMarried + "," + weight);
  }
}
```

Person 类有一个 getPersonInfo()方法，返回包含 Person 信息的字符串。但是，Person 类中并没有定义 personInfo 属性。在 hello.jsp 中，仍然可以通过 ${person.personInfo}的形式访问 Person 信息，例如：

```
<input type="text" name="personInfo" value="${person.personInfo}" />
```

由此可见，EL 表达式 ${person.personInfo}会自动调用 Person 对象的 getPersonInfo()方法。

3.5.3 创建类型转换器 PersonConverter 类

PersonConverter 类实现了 org. springframework. core. convert. converter. Converter 接口,它把 String 类型的 Person 信息转换成 Person 对象。例程 3-4 是 PersonConverter 类的源代码。

例程 3-4 PersonConverter. java

```
package mypack;
import org.springframework.core.convert.converter.Converter;

public class PersonConverter implements Converter<String, Person> {
  public Person convert(String source) {
    // 创建一个 Person 对象
    Person person = new Person();

    // 以","分隔
    String stringValues[] = source.split(",");
    if (stringValues != null && stringValues.length == 4) {
      // 为 Person 实例赋值
      person.setUserName(stringValues[0]);
      person.setAge(Integer.parseInt(stringValues[1]));
      person.setIsMarried(Boolean.parseBoolean(stringValues[2]));
      person.setWeight(Double.parseDouble(stringValues[3]));
      return person;
    } else {
      throw new IllegalArgumentException(
          String.format(
                "类型转换失败,"
                 + "需要格式'userName,age,isMarried,weight ',但格式是[ % s ]"
                , source));
    }
  }
}
```

PersonConverter 类的 convert(String source)方法会解析 String 类型的 source 参数,把它转换成一个 Person 对象。

3.5.4 在 Spring MVC 配置文件中注册类型转换器

只有在 Spring MVC 的配置文件中注册了 PersonConverter 类型转换器,Spring MVC 框架才会在需要的场合,自动把 String 类型的 Person 信息转换为 Person 对象。以下是在 Spring MVC 配置文件中的配置代码。

```
<bean id = "conversionService"
        class = "org.springframework.context
                    .support.ConversionServiceFactoryBean">
```

```
    <property name = "converters">
      <list>
        <bean class = "mypack.PersonConverter"/>
      </list>
    </property>
  </bean>

  <!-- 指定使用 PersonConverter 类型转换器 -->
  <mvc:annotation-driven conversion-service = "conversionService" />
```

<bean>元素定义了一个 id 为 conversionService 的 Bean 组件，它是创建 PersonConverter 类型转换器的工厂 Bean。<mvc:annotation-driven>元素的 conversion-service 属性的取值为 conversionService。因此，Spring MVC 框架会利用 PersonConverter 类型转换器进行数据类型转换。

3.5.5　创建处理请求参数的控制器类 PersonController

在 PersonController 类的 greet()方法中，把名字为 personInfo 的请求参数绑定到 Person 类型的 person 参数，代码如下：

```
@RequestMapping(value = "/sayHello", method = RequestMethod.POST)
public String greet(
          @RequestParam("personInfo") Person person, Model model) {
  model.addAttribute("person", person);
  return "hello";
}
```

由于 3.5.4 节已经在 Spring MVC 框架中注册了 PersonConverter 类型转换器，Spring MVC 框架就会利用该转换器，自动把 String 类型的 personInfo 请求参数转换成 Person 类型的对象。

如图 3-1 所示，在 hello.jsp 页面的文本框中输入字符串“Tom,22,false,52.3”，这个字符串经过 PersonConverter 的数据类型转换，再由 PersonController 把它保存到 Model 中，最后在 hello.jsp 网页上显示出来。

图 3-1　hello.jsp 网页显示 Person 信息

如果在 hello.jsp 网页上输入的字符串为“Tom,22,not married,null”,PersonConverter 试图对它进行类型转换时,会抛出 IllegalArgumentException 异常。本章没有介绍如何处理程序运行中产生的异常。关于异常的处理,请参见第 7 章。

3.6 请求参数的格式转换

对于日期和数字等类型的请求参数,Spring MVC 框架提供了内置的类型转换器,会进行简单的类型转换。例如在例程 3-5 中,showDate()方法有一个 Date 类型的 date 参数。Spring MVC 框架的内置类型转换器会自动把 String 类型的 date 请求参数转换为 Date 类型的 date 方法参数。

例程 3-5 TestFormatController.java

```
@Controller
public class TestFormatController {
  @RequestMapping(value = "/showDate")
  public String showDate(Date date,Model model) {
    model.addAttribute("date", date);
    System.out.println(date);
    return "showDate";
  }

  @RequestMapping(value = "/useFormat")
  public String useFormat(
      @DateTimeFormat(pattern = "yyyy-MM-dd")Date date,
      @NumberFormat(pattern = "#,###")int salary,
      Model model) {

    model.addAttribute("date", date);
    model.addAttribute("salary", salary);
    System.out.println("date = " + date);
    System.out.println("salary = " + salary);
    return "showFormatData";
  }
}
```

通过浏览器访问 showDate()方法,URL 为 http://localhost:8080/helloapp/showDate?date=2020/08/20,该 URL 中 date 请求参数的值为 2020/08/20,Spring MVC 框架会把它转换成一个 Date 对象,赋值给 showDate()方法的 date 参数。

如果通过 http://localhost:8080/helloapp/showDate?date=2020-08-20 访问 showDate()方法,Spring MVC 框架的内置类型转换器无法把 2020-08-20 转换为 Date 对象,会向客户端返回错误信息。

在这种情况下,可以使用 Spring MVC 框架的内置格式转换器。下面介绍两个常用的内置格式转换器。

(1) @DateTimeFormat 注解:日期和时间的格式转换器。

(2) @NumberFormat 注解：数字的格式转换器。

例程 3-5 的 useFormat()方法使用了@DateTimeFormat 注解和@NumberFormat 注解，并且设定了日期和数字的格式，分别为 yyyy-MM-dd 和＃，＃＃＃。

如果通过 http://localhost:8080/helloapp/useFormat? date＝2020-08-20&salary＝5,300 访问 useFormat()方法，@DateTimeFormat 注解会把这个 URL 中的 2020-08-20 转换为 Date 对象，@NumberFormat 注解把 5,300 转换为 int 类型的 5300。

提示：在 Spring 5 版本中，如果在 Spring MVC 的配置文件中为< mvc:annotation-driven >元素设定了 conversion-service 属性，那么@NumberFormat 注解就不起作用，这是需要注意的地方。

除了使用 Spring 的内置格式转换器，还可以灵活地自定义格式转换器。例程 3-6 实现了 Formatter 接口，能够把基于 yyyy * MM * dd 格式的字符串转换成 Date 对象。

例程 3-6　DateFormatter.java

```
import org.springframework.format.Formatter;

public class DateFormatter implements Formatter<Date> {
  SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy*MM*dd");

  public String print(Date object, Locale arg) {
    return dateFormat.format(object);
  }

  public Date parse(String source, Locale arg) throws ParseException {
    return dateFormat.parse(source);
  }
}
```

接下来，在 Spring MVC 的配置文件中需要注册自定义的格式转换器，代码如下：

```
<!-- 注册格式转换器 DateFormatter -->
<bean id="conversionService"
          class="org.springframework.format.support
                  .FormattingConversionServiceFactoryBean">

  <property name="formatters">
    <list>
      <bean class="mypack.DateFormatter"/>
    </list>
  </property>
</bean>

<mvc:annotation-driven conversion-service="conversionService" />
```

这段代码使得 Spring MVC 框架会在需要的场合使用自定义的 DateFormatter。

通过浏览器再次访问例程 3-5 的 showDate()方法，URL 为 http://localhost:8080/helloapp/showDate? date=2020 * 08 * 20，Spring MVC 框架会利用自定义的 DateFormatter 把以上 URL 中的 2020 * 08 * 20 转换成一个 Date 对象，再把它传给 showDate()方法的 date 参数。

自定义的格式转换器和 3.5 节的自定义类型转换器可以完成相似的数据类型转换功能。两者的区别如下。

(1) 自定义类型转换器能够对各种数据类型进行转换，而自定义格式转换器只能把 String 类型转换成其他类型。

(2) 自定义类型转换器会忽略 Locale 信息，而自定义格式转换器可以根据 Locale 信息，进行本地化的数据格式转换，这有助于实现 Web 应用的国际化。第 8 章将详细介绍了 Web 应用的国际化。

3.7 控制器类的方法的参数类型

Controller 类在 Spring MVC 框架中享有优越地位，Spring MVC 框架为控制器类提供了各种资源。从 3.4 节就可以看出，控制器类如果需要访问客户端的某种请求数据，只需要声明一个与特定请求数据绑定的方法参数。

在 Spring MVC 框架的全力支持下，控制器类的请求处理方法可以把方法参数定义为以下 20 种类型。

(1) javax.servlet.ServletRequest 或 javax.servlet.http.HttpServletRequest。

(2) javax.servlet.ServletResponse 或 javax.servlet.http.HttpServletResponse。

(3) javax.servlet.http.HttpSession。

(4) org.springframework.web.context.request.WebRequest。

(5) org.springframework.web.context.request.NativeWebRequest。

(6) java.util.Locale。

(7) 用于读取请求数据的 java.io.InputStream 或 java.io.Reader。

(8) 用于生成响应结果的 java.io.OutputStream 或 java.io.Writer。

(9) java.security.Principal。

(10) org.springframework.http.HttpEntity<?>。

(11) java.util.Map 或 org.springframework.ui.Model。

(12) org.springframework.ui.ModelMap。

(13) org.springframework.web.servlet.ModelAndView。

(14) org.springframework.web.servlet.mvc.support.RedirectAttributes。

(15) org.springframework.validation.Errors。

(16) org.springframework.validation.BindingResult。

(17) org.springframework.web.bind.support.SessionStatus。

(18) org.springframework.web.util.UriComponentsBuilder。

(19) 用 @ModelAttribute、@PathVariable、@CookieValue、@RequestParam、@RequestHeader、@RequestBody 和@RequestPart 注解标识的参数。

(20) 和请求参数对应的数据类型。

第 2 章以及本章已经介绍了 HttpServletRequest 类型、BindingResult 类型和 Model 类型的参数，以及用@ModelAttribute、@RequestParam、@CookieValue 和@RequestHeader 等注解标识的参数，后文还将陆续介绍其他类型参数的用法。

org. springframework. web. context. request. WebRequest 接口的用法和 HttpServletRequest 接口相似，它也表示客户请求。WebRequest 接口的 getParameter(String paramName)方法用于读取请求参数，getHeader(String headerName)方法用于读取请求头中的特定项，getContextPath()方法返回当前 Web 应用的根路径。

3.8　控制器类的方法的返回类型

以下是控制器类的请求处理方法的 4 种常用返回类型。

(1) ModelAndView 类型：包含了 Model 数据以及视图组件。3.9.4 节将介绍把 ModelAndView 作为返回类型的范例。

(2) String 类型：Web 组件的逻辑名字。

(3) void：没有返回值。在这种情况下，在请求处理方法中可以直接通过 Writer 输出响应结果。

(4) 如果请求处理方法用@ModelAttribute 注解来标识，那么方法的返回值无论是什么类型，都会添加到 Model 中，参见 3.9.1 节。

3.8.1　String 返回类型

Controller 类的请求处理方法返回 String 类型字符串，有以下三种用途。

(1) 把请求转发给视图组件。

(2) 把请求转发给其他控制器类组件，返回值以 forward:开头。

(3) 把请求重定向到其他控制器类组件，返回值以 redirect:开头。

例如，在以下请求处理方法 dispatch()中，根据请求参数 action 的取值返回特定的字符串。

```
@RequestMapping("test")
public String dispatch(@RequestParam("action")String action) {
  switch(action){
    case "forward":
      //把请求转发给 URL 入口为 input 的控制器类的特定方法
      return "forward:input";

    case "redirect":
      //重定向到 URL 入口为 output 的控制器类的特定方法
      return "redirect:output";

    case "jsp":
    default:
      return "result";          //把请求转发给 result.jsp
  }
}
```

dispatch()方法有以下三种返回值。

(1) forward:input：把请求转发给 URL 入口为 input 的控制器类的特定方法。

(2) redirect:output：重定向到 URL 入口为 output 的控制器类的特定方法。

(3) result：把请求转发给 result.jsp 文件。

3.8.2 void 返回类型

当请求处理方法的返回类型为 void,可以通过 Writer 直接在网页上输出数据,例如：

```
@RequestMapping("/testvoid")
public void testvoid(Writer writer)throws IOException{
  writer.write("hello");              //在网页上输出字符串 hello
}
```

3.9 控制器与视图的数据共享

视图会把请求数据传给控制器进行处理,控制器也会把响应结果传给视图进行展示。前文已经介绍了视图向控制器传递请求数据的方法。本节将继续介绍由 Spring MVC 框架提供的用于数据共享的注解、接口和类的用法,主要有以下 4 种：

(1) @ModelAttribute 注解：表示 Model 的特定数据。

(2) org.springframework.ui.Model 接口：表示 Model 数据。控制器和视图都能访问 Model。

(3) org.springframework.ui.ModelMap 类：表示 Model 数据的映射。控制器和视图都能访问 ModelMap。

(4) org.springframework.web.servlet.ModelAndView 类：表示 Model 数据和视图。控制器和视图都能访问 ModelAndView。

图 3-2 显示了这 4 种注解、接口和类的作用。

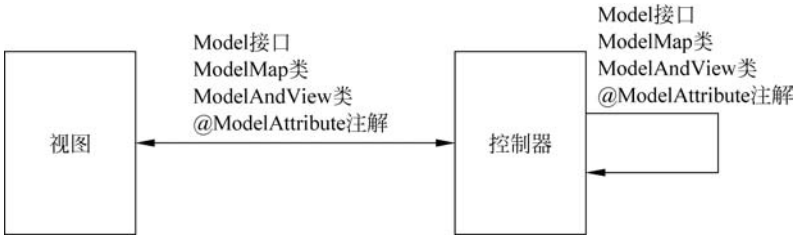

图 3-2 视图与控制器之间共享数据

Model 接口、ModelMap 类、ModelAndView 类、@ ModelAttribute 注解中都包含 Model。这里的 Model 和 MVC 框架中的模型层的概念有区别。MVC 框架中的模型层包含业务数据和业务逻辑,而这里的 Model 是指控制器层的一种用于存放共享数据的容器,它仅包含业务数据,不包含业务逻辑。

Model 接口、ModelMap 类和 ModelAndView 类都包含了 Model 数据,因此它们都具

有存取共享数据的功能。它们的区别在于，ModelMap 类实现了 java.util.Map 映射接口，可以直接通过 get(String attributeName)方法来读取特定的共享数据；ModelAndView 类不仅包含 Model 数据，还和特定的视图组件绑定，由这个视图组件来展示 Model 数据。

共享数据在 Model 中以“属性名＝属性值”的形式存放。@ModelAttribute 注解表示 Model 的一个属性，也就是 Model 中的特定共享数据。

3.9.1　@ModelAttribute 注解

@ModelAttribute 注解有以下三个作用。

(1) 用在控制器类的请求处理方法的参数前面，把方法参数保存到 Model 中。

(2) 用在控制器类的请求处理方法的参数前面，把 Model 的特定属性赋值给方法参数。

(3) 用在控制器类的方法前面，表明该方法会向 Model 中添加特定属性。

1. 把方法参数保存到 Model 中

在第 2 章中，PersonController 类的 greet()方法的 person 参数用@ModelAttribute 注解来标识，例如：

```
@RequestMapping(value = "/sayHello", method = RequestMethod.POST)
public String greet(
        @Valid @ModelAttribute("personbean")Person person,
        BindingResult bindingResult,Model model) {…}
```

Spring MVC 框架先把客户端提供的表单数据和 person 参数绑定，接着@ModelAttribute 注解把 person 参数保存到 Model 中，属性名为 personbean。

2. 把 Model 的特定属性赋值给方法参数

假定请求访问以下 output()方法的 URL 中不存在 name 请求参数，此时，@ModelAttribute 注解会把 Model 中的 userName 属性赋值给 name 参数。

```
@RequestMapping(value = "/output")
public String output(@ModelAttribute("userName") String name){
  System.out.println(name);
  return "result";
}
```

例程 3-8 将提供完整的范例。

3. 标识用于向 Model 中添加属性的方法

如果控制器类的一个方法 A 用@ModelAttribute 注解来标识，那么意味着该方法会设置 Model 数据。当 Spring MVC 框架调用请求处理方法 B 之前，会先调用方法 A。

提示：如果控制器类有多个方法(按照定义的先后顺序分别为方法 A、方法 B 和方法 C)都用@ModelAttribute 注解来标识，那么当 Spring MVC 框架调用请求处理方法 D 之前，会依次先调用方法 A、方法 B 和方法 C，然后再调用方法 D。

例程 3-7 的 setModel()方法用@ModelAttribute 注解来标识。

例程 3-7　TestModelController.java

```
@Controller
public class TestModelController {
  @ModelAttribute
  public void setModel(String userName,Model model){
    //把 userName 方法参数作为 userName 属性加入到 Model 中
    model.addAttribute("userName", userName);
  }

  @RequestMapping(value = "/testmodel")
  public String login(Model model){
    // 从 Model 中读取 userName
    String userName = (String)model.getAttribute("userName");
    System.out.println(userName);
    return "showUser";
  }
}
```

在 setModel()方法中,userName 请求参数和 userName 方法参数绑定。setModel()方法把 userName 方法参数作为 userName 属性添加到 Model 中。在 login()方法中,从 Model 读取 userName 属性,并且把请求转发给 showUser.jsp。showUser.jsp 通过 EL 表达式 ${userName}输出 Model 的 userName 属性。

通过浏览器访问 http://localhost:8080/helloapp/testmodel? userName = Mary,Spring MVC 框架先调用 setModel()方法,再调用 login()方法,最后把请求转发给 showUser.jsp。

例程 3-7 中用@ModelAttribute 注解标识的 setModel()方法的返回类型为 void。以下代码重新实现了 setModel()方法,它具有 String 类的返回值。它能完成同样的功能,把 userName 属性添加到 Model 中。

```
@ModelAttribute("userName")
public String setModel(String userName) {
  return userName;
}
```

@ModelAttribute 注解为 Model 添加了一个 userName 属性,setModel()方法的返回值就是 Model 中 userName 属性的值。

@ModelAttribute 注解标识的 setModel()方法返回 Person 对象,这个 Person 对象会作为 person 属性添加到 Model 中。

```
  @ModelAttribute("person")
  public Person setModel(String userName) {
    Person p = new Person();
    p.setUserName(userName);
    return p;              //返回的 Person 对象保存到 Model 中,属性名为 person
```

```
    }

    @RequestMapping(value = "/getperson")
    public String login(Model model){
      // 从 Model 中读取 person
      Person person = (Person)model.getAttribute("person");
      System.out.println(person.getUserName());
      return "result";
    }
```

例程 3-8 在 setdata()方法前和 output()方法的 name 参数前都使用了@ModelAttribute 注解。

例程 3-8　TestAttributeController.java

```
@Controller
public class TestAttributeController {

  @ModelAttribute("userName")
  public String setdata(String name){
    return name.toUpperCase();
  }

  @RequestMapping(value = "/output")
  public String output(@ModelAttribute("userName") String name){
    System.out.println(name);
    return "result";
  }
}
```

setdata()方法前的@ModelAttribute 注解向 Model 添加了一个 userName 属性。output()方法的 name 参数前的@ModelAttribute 注解把 Model 中的 userName 属性和 name 参数绑定。

当通过浏览器访问 http://localhost:8080/helloapp/output?name=Tom，TestAttributeController 类的 output()方法会在服务器端打印 TOM。

3.9.2　Model 接口

Model 接口表示 Model 数据，存放在 Model 中的数据采用“属性名/属性值”的形式。Model 中的数据能够被控制器和视图共享，例程 3-7 也演示了 Model 接口的用法。

```
//向 Model 中存放共享数据
model.addAttribute("userName", userName);

//从 Model 中读取共享数据
String userName = (String)model.getAttribute("userName");
```

3.9.3 ModelMap 类

ModelMap 类和 Model 接口的功能相似,区别在于两者的语义不同。要从 Model 对象中读取共享数据,可以调用 getAttribute(String attributeName)方法;而 ModelMap 类表示 Model 的映射类型,可以调用 ModelMap 类的 get(String attributeName)方法获得特定的属性值。

此外,Model 接口的 asMap()方法会返回一个 Map 对象,通过这个 Map 对象也能读取特定的属性,例如以下三种方式分别从 Model 或 ModelMap 中读取特定属性。

```
Model model = …
ModelMap modelMap = …
String userName1 = (String)model.getAttribute("userName");
String userName2 = (String)(model.asMap().get("userName"));
String userName3 = (String)modelMap.get("userName");
```

例程 3-9 的功能与例程 3-7 相同,区别在于本节范例用 ModelMap 类来存放共享数据。

例程 3-9 TestModelMapController.java

```
@Controller
public class TestModelMapController {
  @ModelAttribute
  public void setModel(String userName,
              ModelMap modelMap){
    modelMap.addAttribute("userName", userName);
  }

  @RequestMapping(value = "/testmap")
  public String login(ModelMap modelMap){
    // 从 ModelMap 中读取 userName 属性
    String userName = (String)modelMap.get("userName");
    System.out.println(userName);
    return "showUser";
  }
}
```

3.9.4 ModelAndView 类

ModelAndView 类和 Model 接口一样,也能存放共享数据。但它们有以下两点区别。

(1) ModelAndView 类添加共享数据的方法是 addObject(String attributeName, Object attributeValue),而 Model 接口添加共享数据的方法是 addAddtribute(String attributeName,Object attributeValue)。

(2) ModelAndView 类的 setViewName(String viewName)方法指定用于展示 Model 数据的视图组件,参数 viewName 指定视图组件的逻辑名字,而 Model 接口不具有这样的方法。

ModelAndView 类的以下方法返回包含 Model 数据的 Map 对象或 ModelMap 对象。

```
Map < String,Object > getModel()
ModelMap getModelMap()
```

例程 3-10 的功能与例程 3-7 相同，区别在于本节范例用 ModelAndView 类来存放共享数据。

例程 3-10　TestViewController.java

```
@Controller
public class TestViewController {

  @ModelAttribute
  public void setModel(String userName,
              ModelAndView modelAndView){
    modelAndView.addObject("userName", userName);
  }

  @RequestMapping(value = "/testview")
  public ModelAndView login(ModelAndView modelAndView){
    // 从 ModelAndView 中读取 userName 属性
    String userName =
            (String)(modelAndView.getModel().get("userName"));
    System.out.println(userName);

    //showUser 是 showUser.jsp 的逻辑名字
    modelAndView.setViewName("showUser");
    return modelAndView;          //把请求转发给 showUser.jsp
  }
}
```

login()方法的返回类型为 ModelAndView，login()方法会把请求转发给返回值 modelAndView 对象指定的视图组件 showUser.jsp。

3.9.5　把 Model 中的数据存放在 session 范围内

2.4.3 节已经介绍过，默认情况下，添加到 Model 中的数据存放在 request 范围内。如果要把数据存放到 session 范围内，可以使用@SessionAttributes 注解。例程 3-11 使用了@SessionAttributes 注解。

例程 3-11　TestSessionController.java

```
@Controller
@SessionAttributes(value = {"person","age"})
public class TestSessionController {

  @ModelAttribute("person")
  public Person setModel(){
```

```
        Person p = new Person();
        p.setUserName("Tom");
        return p;
    }

    @RequestMapping(value = "/testsession")
    public String testSession(Model model,int age,String address){
        model.addAttribute("age",age);
        model.addAttribute("address",address);
        return "sessiontest";
    }

    @RequestMapping(value = "/repeat")
    public String repeat(SessionStatus sessionStatus){
        return "sessiontest";
    }

    @RequestMapping(value = "/sessionclear")
    public String testSessionClear(SessionStatus sessionStatus){
        sessionStatus.setComplete();          //清除 session 范围内的 Model 数据
        return "sessiontest";
    }
}
```

以上 TestSessionController 类前的@SessionAttributes(value={"person","age"})注解的作用是声明 Model 中的 person 属性和 age 属性存放在 request 范围内。

setModel()方法向 Model 添加了 person 属性,testSession()方法向 Model 添加了 age 属性,person 属性和 age 属性都会保存在 session 范围内。testSession()方法还向 Model 添加了 address 属性,该属性会保存在默认的 request 范围内。

在 testSessionClear()方法中,通过 SessionStatus 类的 setComplete()方法清除 session 范围内的 Model 数据。

testSession()、repeat()和 testSessionClear()方法都会把请求转发给 sessiontest.jsp。sessiontest.jsp 输出 session 范围和 request 范围内的共享数据,例如:

```
UserName: ${sessionScope.person.userName} <br>
Age: ${sessionScope.age}<br>
Address: ${requestScope.address}
```

下面按以下三个步骤来运行本范例。

(1) 通过浏览器访问 testSession()方法,URL 为 localhost:8080/helloapp/testsession?age=22&address=shanghai,Spring MVC 框架先调用 setModel()方法,再调用 testSession()方法,这两个方法向 session 范围存放 person 属性和 age 属性,还向 request 范围存放 address 属性。sessiontest.jsp 会在网页上输出如下信息:

```
UserName:Tom
Age:22
Address:shanghai
```

(2) 通过浏览器访问 repeat()方法，URL 为 localhost:8080/helloapp/repeat。此时，在 session 范围内存在 person 和 age 属性，但是在 request 范围内没有 address 属性，sessiontest.jsp 会在网页上输出如下信息：

```
UserName:Tom
Age:22
Address:
```

(3) 通过浏览器访问 testSessionClear()方法，URL 为 localhost:8080/helloapp/sessionclear，testSessionClear()方法清除 session 范围内的 Model 数据，sessiontest.jsp 会在网页上输出如下信息。

```
UserName:
Age:
Address:
```

3.9.6　通过@SessionAttribute 注解读取 session 范围内的 Model 数据

@SessionAttributes 注解把 Model 数据存放在 session 范围内，而@SessionAttribute 注解读取 session 范围内的 Model 数据。@SessionAttribute 注解用来标识控制器类的请求处理方法的参数。

在例程 3-12 中，testShare()方法的 person 参数用@SessionAttribute 注解标识。

例程 3-12　TestShareController.java

```
@Controller
@SessionAttributes("person")
public class TestShareController {

  @RequestMapping(value = "/setdata")
  public String testData(Model model){
    Person p = new Person();
    p.setUserName("Tom");
    model.addAttribute("person",p);
    return "redirect:testshare";
  }

  @RequestMapping(value = "/testshare")
  public String testShare(@SessionAttribute("person") Person person){
    System.out.println(person.getUserName()); //打印 Tom
    return "result";
  }
}
```

testData()方法向 Model 加入了一个 Person 对象。TestShareController 类的@SessionAttributes("person")注解使得该 Person 对象实际上存放在 session 范围内。接

下来,testData()方法把请求重定向到 testShare()方法。

testShare()方法的@SessionAttribute("person")注解从 session 范围内获得 Person 对象,把它赋值给 testShare()方法的 person 方法参数。由此可见,@SessionAttribute 注解能把 session 范围内的特定数据与请求处理方法的参数绑定。

通过浏览器访问 http://localhost:8080/helloapp/setdata,testShare()方法会在服务器端打印 person.getUserName()方法的返回值 Tom。

3.10 @ControllerAdvice 注解的用法

当一个 Web 应用中的多个控制器类要完成一些共同的操作,传统的做法是定义一个控制器父类(例如 BaseController),它包含了执行共同操作的方法,其他的控制器类(例如 ControllerA 和 ControllerB)继承这个控制器父类。图 3-3 显示了控制器父类和控制器子类的关系。

继承是提高控制器类的代码可重用性的有效手段,但是它有一个缺点,那就是由于 Java 语言不支持多继承,当控制器类继承了一个控制器父类后,就不能再继承其他的类。

Spring MVC 框架提供了另一种方式来为多个控制器类提供共同的方法,那就是利用 @ControllerAdvice 注解来定义一个控制器增强类。

控制器增强类并不是控制器类的父类。在程序运行时,Spring MVC 框架会把控制器增强类的方法代码块动态注入其他控制器类中,通过这种方式来增强控制器类的功能。图 3-4 显示了控制器增强类(例如 MyControllerAdvice)和控制器类的关系。

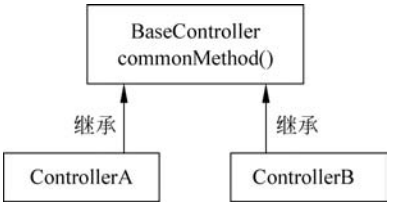

图 3-3 控制器父类和控制器子类的关系

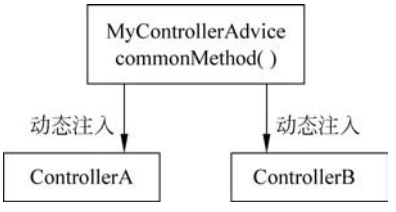

图 3-4 控制器增强类和控制器类的关系

例程 3-13 的 setColors()方法向 Model 中加入一个 colors 属性。

例程 3-13 MyControllerAdvice.java

```
package mypack;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ModelAttribute;
import java.util.*;

@ControllerAdvice
public class MyControllerAdvice {

  @ModelAttribute(name = "colors")
  public Map<String,String> setColors() {
    HashMap<String, String> colors = new HashMap<String,String>();
    colors.put("RED", "红色");
```

```
        colors.put("BLUE", "蓝色");
        colors.put("GREEN", "绿色");
        return colors;
    }
}
```

当程序运行时，Spring MVC 框架会把 MyControllerAdvice 类的 setColors()方法动态注入其他控制器类中，因此其他控制器类就自动拥有了该方法。例如，在 TestAttributeController 类中可以直接访问 Model 中的 colors 属性，代码如下：

```
@RequestMapping(value = "/testColor")
public String testColor(
        @ModelAttribute("colors") Map < String,String > colors,
        @ModelAttribute("userName") String name){

    System.out.println(name + "'s favourite color:"
                                              + colors.get("RED"));
    return "result";
}
```

通过浏览器访问 http://localhost:8080/helloapp/testColor? name=Tom，testColor()方法会在服务器端打印 TOM's favourite color:红色。

默认情况下，@ControllerAdvice 注解用来增强当前 Web 应用中所有控制器类的功能。此外，它的 assignableTypes 属性和 basePackages 属性用来指定需要增强功能的控制器类，例如：

```
//增强 PersonController 和 TestAttributeController 的功能
@ControllerAdvice(assignableTypes = {PersonController.class,
                                          TestAttributeController.class})
public class MyControllerAdvice1{ … }

//增强 mypack 包和 net.javathinker 包中的控制器类的功能
@ControllerAdvice(basePackages = {"mypack","net.javathinker"})
public class MyControllerAdvice2{ … }
```

3.11　小结

当视图和控制器分离后，视图与控制器之间需要进行数据的传递和共享。为了方便存取共享数据，Spring MVC 框架提供了实用的注解和类，包括：

（1）把请求数据和控制器类的方法参数绑定的注解：@RequestParam 注解、@RequestHeader 注解、@CookieValue 注解和@PathVariable 注解。

（2）对请求参数进行类型转换的接口：org.springframework.core.convert.converter.Converter 接口。

（3）对请求参数进行格式转换的接口：org.springframework.format.Formatter 接口。

(4) 存取 Model 数据的接口和类：Model 接口、ModelMap 类和 ModelAndView 类。

(5) 把 Model 数据和控制器类的方法参数绑定的注解：@ModelAttribute 注解。

(6) 把 Model 数据存放在 session 范围内的注解：@SessionAttributes 注解。

(7) 把 session 范围内的 Model 数据和控制器类的方法参数绑定的注解：@SessionAttribute 注解。

Controller 类是 Spring MVC 框架中的主力军，Spring MVC 框架会掌管 Controller 对象的生命周期，同时，Spring MVC 框架也给控制器类提供了自由发挥的空间：

(1) 控制器类的请求处理方法的参数类型可以是来自 Servlet API 的 ServletRequest、ServletResponse 和 HttpSession 等，也可以是来自 Spring API 的 Model、ModelMap 和 ModelAndView 等，还可以是和请求参数对应的任意的数据类型等。

(2) 控制器类的请求处理方法的返回类型可以是 void、String 以及 ModelAndView 等。

3.12 思考题

1. (　　)注解用来设定控制器对象的存在范围。(多选)

   A. @RequestMapping

   B. @SessionScope

   C. @Scope

   D. @RequestScope

2. 对于一个控制器类的以下方法：

```
@RequestMapping(value = "test",
            params = { "username = Tom", "age","!address" })
public String sayHello() {…}
```

   (　　)是能正常访问 sayHello()方法的 URL。(多选)

   A. /test? username=Tom&age=20

   B. /sayHello? username=Tom&age=20

   C. /test? username=Mike

   D. /test? username=Tom&age=20&gender=male

3. 在一个控制器类的请求处理方法中，应该用(　　)定义 double 类型的 salary 参数，从而把 salary 请求参数和 salary 方法参数绑定。(多选)

   A. @RequestParam(name="salary") double salary

   B. @RequestParam(value="salary") double salary

   C. @RequestParam double salary

   D. double salary

4. 对于控制类的请求处理方法，它的方法参数可以定义为(　　)类型。(多选)

   A. javax.servlet.ServletRequest

   B. java.io.Writer

   C. org.springframework.ui.ModelMap

D. 用@SessionAttributes 注解标识的参数

5. (　　)注解既能标识控制器类,又能标识控制器类的方法。(单选)

A. @SessionAttribute

B. @RequestParam

C. @PathVariable

D. @RequestMapping

6. 以下具有 asMap()方法的是(　　)。(单选)

A. ModelMap 类

B. ModelAndView 类

C. Model 接口

D. 用@Controller 注解标识的控制器类

7. 一个控制器类的请求处理方法的返回值是 forward:hello,以下说法正确的是(　　)。(单选)

A. Spring MVC 框架会把请求转发给 URL 入口为/hello 的控制器类的请求处理方法

B. Spring MVC 框架会把请求转发给 hello.jsp

C. Spring MVC 框架会在服务器端打印字符串 forward:hello

D. Spring MVC 框架会把请求重定向到 URL 入口为/hello 的控制器类的请求处理方法