

10 | MVC架构解析：控制器（Controller）篇

2019-10-02 四火

全栈工程师修炼指南

[进入课程 >](#)



讲述：四火

时长 17:54 大小 14.36M



你好，我是四火。

今天我们继续学习 MVC 架构，主要内容就是 MVC 架构的第三部分——控制器（Controller）。

控制器用于接收请求，校验参数，调用 Model 层获取业务数据，构造和绑定上下文，并转给 View 层去渲染。也就是说，控制器是 MVC 的大脑，它知道接下去该让谁去做什么事。控制器层是大多数 MVC 框架特别愿意做文章的地方，我相信你可能耳闻、了解，甚至熟练使用过一些 MVC 框架了。

那么与其去抽象地学习这一层的重要概念、原理，或是单纯地学习这些框架在这一层略显乏味的具体配置，我想我们今天“不走寻常路”一次，把这两者结合起来——**我们来比较**

Servlet、Struts 和 Sprint MVC 这三种常见的技术和 MVC 框架，在控制器层的工作路数，以及和业务代码整合配置的方式，看看任这些框架形式千变万化，到底有哪些其实是不变的“套路”呢？

随着请求到达控制器，让我们顺着接下去的请求处理流程，看看控制器会通过怎样的步骤，履行完它的职责，并最终转到相应的视图吧。

1. 路径映射和视图指向


我们不妨把 MVC 架构的控制器想象成一个黑盒。当 HTTP 请求从客户端送达的时候，这个黑盒要完成一系列使命，那么它就有一个入口路由和一个出口路由：

入口路由就是路径映射，根据配置的规则，以及请求 URI 的路径，找到具体接收和处理这个请求的控制器逻辑；

出口路由就是视图指向，根据配置的规则，以及控制器处理完毕后返回的信息，找到需要渲染的视图页面。

这两件事情，我们当然可以使用原始的 if-else 来完成，但是一般的 MVC 都提供了更清晰和独立的解决方案。

我们还是从老朋友 Servlet 开始讲起，在 Tomcat 的 web.xml 中，我们可以配置这样的路径映射：

 复制代码

```
1 <servlet>
2     <servlet-name>BookServlet</servlet-name>
3     <servlet-class>com.xxx.xxx.BookServlet</servlet-class>
4 </servlet>
5 <servlet-mapping>
6     <servlet-name>BookServlet</servlet-name>
7     <url-pattern>/books</url-pattern>
8 </servlet-mapping>
```

你看，对于路径映射，一旦请求是 /books 这种形式的，就会被转到 BookServlet 里去处理。而对于视图指向，Servlet 是通过代码完成的，比如：

```
1 request.getRequestDispatcher("/book.jsp").forward(request, response);
```

但是，Servlet 路径映射的表达式匹配不够灵活，而且配置过于冗长；而视图指向更是完全通过代码调用来完成，视图的位置信息完全耦合在控制器主代码逻辑中，而且也并没有体现出配置的集中、清晰的管理优势。于是现今的 MVC 框架都提供了一套自己的映射匹配逻辑，例如 [Struts 2](#)：

```
1 <action name="books" class="xxx.xxx.BookAction">
2     <result name="success" type="dispatcher">/success.jsp</result>
3     <result name="input" ... />
4 </action>
```

其中，name= “books” 这样的配置就会将 /books 的请求转给 BookAction。至于接下来的两个 result 标签，是根据控制器返回的视图名来配对具体的视图页面，也就是说，一旦 BookAction 处理完毕，通过返回的视图名字，请求可以被转发给相应的视图。

这个路径映射的配置是简单一些了，可是都需要放在一个其它位置的、单独的 XML 中配置。不过，Java 5 开始支持注解，因此许多 MVC 框架都开始支持使用注解来让这样的配置变得更加轻量，也就是将路径映射和它所属的控制器代码放在一起。见下面 Struts 的例子：

```
1 public class BookAction extends ActionSupport {
2     @Action(value="/books", results={
3         @Result(name="success",location="/book.jsp")
4     })
5     public String get() {
6         ...
7         return "success";
8     }
9 }
```

代码依然很好理解，当以 /books 为路径的 GET 请求到来时，会被转给 BookAction 的 get 方法。在控制器的活干完之后，根据返回的名称 success，下一步请求就会转到视图 /book.jsp 中去。


你看，对于路径映射和视图指向，为了不把这样的信息和主流程代码耦合在一起，上面讲了两种实现方法，它们各有优劣：

放到配置文件中，好处是所有的映射都在一个文件里，方便管理。但是对于任何一个控制器逻辑，要寻找它对应的配置信息，需要去别的位置（即上文的 XML 中）寻找。**这是一种代码横向分层解耦的方式，即分层方式和业务模块无关，或者说二者是“正交”的**，这种方式我在 [第 11 讲] 讲解 IoC（控制反转）时会继续介绍。

使用注解，和控制器逻辑放在一起，好处是映射本身是和具体的控制器逻辑放在一起，当然，它们并非代码层面的耦合，而是通过注解的方式分离开。坏处是，如果需要考察所有的映射配置，那么就没有一个统一的文件可供概览。**这是一种代码纵向分层解耦的方式，也就是说，配置是跟着业务模块走的。**

无论使用以上哪一种方法，本质上都逃不过需要显式配置的命运。但无论哪种方法，其实都已经够简单了，可历史总是惊人的相似，总有帮“难伺候”的程序员，还是嫌麻烦！于是就有人想出了一个“终极偷懒”的办法——免掉配置。

这就需要利用 **CoC 原则（Convention over Configuration，即规约优于配置）**。比如，在使用 [Spring MVC](#) 这个 MVC 框架时，声明了 `ControllerClassNameHandlerMapping` 以后，对于这样没有配置任何映射信息的方法，会根据 Controller 类名的规约来完成映射：

 复制代码

```
1 public class BooksController extends AbstractController {
2     @Override
3     protected ModelAndView handleRequestInternal() throws Exception {
4         ...
5     }
6 }
```

在使用 /books 去访问的时候，请求就会被自动转交给定义好的控制器逻辑。

你看，规约优于配置看起来可以省掉很多工作对不对？没错！但是任何技术都有两面性，**CoC 虽然省掉了一部分实际的配置工作，却没有改变映射匹配的流程本身，也不能省掉任何为了理解规约背后的“隐性知识”的学习成本。**而且，规约往往只方便于解决最常见的配置，也就意味着，**当需要更灵活的配置时，我们还是会被迫退化回显式配置。**

2. 请求参数绑定

请求被送到了指定的控制器方法，接下去，需要从 HTTP 请求中把参数取出来，绑定到控制器这一层，以便使用。**整个控制器的流程中，有两次重要的数据绑定，这是第一次，是为了控制器而绑定请求数据**，后面在视图上下文构造这一步中还有一次绑定，那是为了视图而进行的。


和路径映射的配置一样，最先被考虑的方式，一定是用编程的方法实现的。比如在 Servlet 中，可以这样做：

 复制代码

```
1 request.getParameter("name")
```

这并没有什么稀奇的对不对，想想我们前面学习的处理方法，参数应该能通过某种配置方式自动注入到控制器的对象属性或者方法参数中吧？一点都没错，并且，Struts 和 Spring MVC 各有各的做法，二者加起来，就恰巧印证了这句话。

还记得前面 Struts 的那个例子吗？给 BookAction 设置一个和参数同名的属性，并辅以规则的 get/set 方法，就能将请求中的参数自动注入。更强大的地方在于，如果这个属性是个复杂对象，只要参数按照规约命名了，那么它也能够被正确处理：

 复制代码

```
1 public class BookAction extends ActionSupport {  
2     private Page page;  
3     public void setPage { ... }  
4     public Page getPage { ... }  
5 }
```

在这种设定下，如果 URI 是：


```
1 /books?page.pageSize=1&page.pageNo=2&page.orderBy=desc
```

那么，pageSize、pageNo 和 orderBy 这三个值就会被设置到一个 Page 对象中，而这个 Page 对象则会被自动注入到 BookAction 的实例中去。

再来看看 Spring MVC 使用注解的方式来处理，和 URL 的结构放在一起观察，这种方式显然更为形象直观：

```
1 @RequestMapping("/{category}/books")
2 public ModelAndView get(@PathVariable("category") String category, @RequestParam("author")
```

在这种配置下，如果 URI 是：

```
1 /comic/books?author=Jim
```

那么，分类 comic 就会作为方法参数 category 的值传入，而作者 Jim 就会作为方法参数 author 的值传入。

3. 参数验证


参数验证的操作因为和请求对象密切相关，因此通常都是在控制器层完成的。在参数验证没有通过的情况下，往往会执行异常流程，转到错误页面，返回失败请求。Struts 提供了一个将参数验证解耦到配置文件的办法，请看下面的例子：

```
1 <validators>
2   <field name="name">
3     <field-validator type="requiredstring">
4       <param name="trim">true</param>
5       <message> 书名不得为空 </message>
```

```
6     </field-validator>
7     <field-validator type="stringlength">
8         <param name="maxLength">100</param>
9         <param name="minLength">1</param>
10        <message> 书名的长度必须在 1~100 之间 </message>
11    </field-validator>
12 </field>
13 </validators>
```

这就是一个非常简单的参数验证的规则，对于属性 name 定义了两条规则，一条是不得为空，另一条是长度必须在 1~100 之间，否则将返回错误信息。

类似的，Struts 也提供了基于注解的参数验证方式，上面的例子，如果使用注解来实现，就需要将注解加在自动注入参数的 set 方法处。代码见下：


 复制代码

```
1 @RequiredFieldValidator(trim = true, message = " 书名不得为空.")
2 @StringLengthFieldValidator(minLength = "1", maxLength = "100", message = " 书名的长度必:
3 void setName(String name) { ... }
```

4. 视图上下文绑定

在控制器中，我们经常需要将数据传入视图层，它可能会携带用户传入的参数，也可能会携带在控制器中查询模型得到的数据，而这个传入方式，就是**将数据绑定到视图的上下文中**。**这就是我刚刚提到过的控制器层两大绑定中的第二个。**

如果是使用 Servlet，那么我们一般可以用 setAttribute 的方法将参数设置到 request 对象中，这样在视图层就可以相应地使用 getAttribute 方法把该参数的值取出来。


 复制代码

```
1 request.setAttribute("page", xxx);
```

对于 Struts 来说，它的方法和前面说的请求参数绑定统一了，即可以将想传递的值放到 Action 的对象属性中，这种方式绑定的属性，和请求参数自动绑定的属性没有什么区别，

在视图层都可以直接从上下文中取出来。

接着前面 BookAction 的例子，绑定了一个 Page 对象，那么在视图层中就可以使用 OGNL（Object-Graph Navigation Language，对象导航图语言）表达式直接取得：

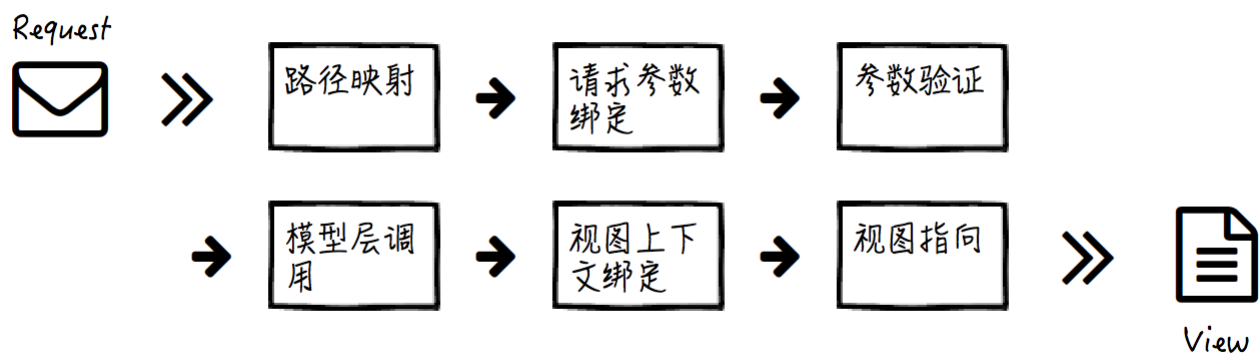
 复制代码

```
1 <p> 第 ${page.pageNo} 页 </p>
```

对于 Spring MVC，则是需要在控制器方法中传入一个类型为 Model 的对象，同时将需要绑定的对象通过调用 addAttribute 来完成绑定，这个过程和 Servlet 是类似的。

总结思考

今天我们学习了 MVC 架构中的控制器层，整个控制器的逻辑比较密集，从请求抵达，到转出到视图层去渲染，控制器的逻辑通常包括下面这几步，但是，严格说起来，下面这些步骤的任何一步，根据实际情况，都是可以省略的。



我们对比了在原生 Servlet、Struts 框架和 Spring MVC 框架下，上面各个步骤的实现，希望你能够感悟到其中的“套路”。

是的，具体某一个框架的配置使用，是很容易学习和掌握的，这当然很好，但那只是死的知识，而这也只是机械记忆。而当我们去思考同一个框架中实现同一个特性的不同方法，或者是不同框架实现同一个特性的不同方法时，我们就会慢慢体会到技术的有趣之处。

因为我们会去思考，这些不同的“玩法”比较起来，各有什么优缺点，在实际应用中应该怎么去权衡和选择，甚至去想，如果让我去设计一个类似的特性，都有哪些办法可以实现。

好，下面我们就来检验一下今天所学的知识，请思考下面这样两个问题：

我们提到了 MVC 框架中，两种常见的配置方式，一种是将配置放在横向解耦的单独一层，另一种是将配置和业务模块放在一起。你更喜欢哪一种，为什么？


在上面的图中，我列出了控制器层常见的六大步骤。那么，回想你经历过的项目，是将怎样的代码逻辑放在了控制器层呢？

对于今天学习的内容，对于思考题，以及通过比较学习“套路”的方式，如果你有想法，不妨和我在留言区一起讨论吧。

选修课堂：动手实现一个简单的 MVC 系统


这一章我们一直在学习 MVC，不动手实践是不行的。我们要使用 Servlet + JSP + JavaBean 这种相对原始的方法来实现一个最简单的 MVC 系统。

还记得我们在 [\[第 07 讲\]](#) 中动手跑起来的 Tomcat 吗？现在请打开 Tomcat 的安装目录，设置好环境变量 CATALINA_HOME，以便于我们后面使用正确的 Tomcat 路径。以我的电脑为例：

 复制代码

```
1 export CATALINA_HOME=/usr/local/Cellar/tomcat/9.0.22/libexec
```


我们打开 `${CATALINA_HOME}/webapps/ROOT/WEB-INF/web.xml`，在这个结束标签前，添加如下子标签：

 复制代码

```
1 <servlet>
2   <servlet-name>BookServlet</servlet-name>
3   <servlet-class>BookServlet</servlet-class>
4 </servlet>
5 <servlet-mapping>
6   <servlet-name>BookServlet</servlet-name>
7   <url-pattern>/books</url-pattern>
8   <url-pattern>/books/*</url-pattern>
9 </servlet-mapping>
```

注意这里配置了两个 URL 映射，/books 和 /books/{bookId} 两种类型的请求会全部映射到我们将建立的 Servlet 中。


在配置好 Servlet 的映射之后，进入 \${CATALINA_HOME}/webapps/ROOT/WEB-INF，并创建一个名为 classes 的文件夹，接着在这个文件夹下建立一个名为 BookServlet.java 的文件，并编辑它：

 复制代码

```
1 import java.io.IOException;
2 import javax.servlet.ServletException;
3 import javax.servlet.http.HttpServlet;
4 import javax.servlet.http.HttpServletRequest;
5 import javax.servlet.http.HttpServletResponse;
6
7 public class BookServlet extends HttpServlet {
8     protected void doGet(HttpServletRequest request, HttpServletResponse response) thro
9         String category = request.getParameter("category");
10        request.setAttribute("categoryName", category);
11        request.getRequestDispatcher("/book.jsp").forward(request, response);
12    }
13 }
```

嗯，其实代码逻辑很简单，把 URL 中的 category 参数的值取出来，给一个新名字 categoryName 并传给 book.jsp。

好，接下来我们就要把上面的 Java 源文件编译成 class 文件了，执行：

 复制代码

```
1 javac BookServlet.java -classpath ${CATALINA_HOME}/lib/servlet-api.jar
```

其中 servlet-api.jar 是 Tomcat 中存放的编译运行 Servlet 所必须的类库。这样，你应该能看到在 classes 目录下生成了 BookServlet.class 文件。


接着，在 \${CATALINA_HOME}/webapps/ROOT 下建立 book.jsp，并写入：

 复制代码

```
1 <jsp:useBean id="date" class="java.util.Date" />
2 Category name: <%=request.getAttribute("categoryName") %>, date: ${date.getYear()+1900}
```

第一行表示创建并使用一个 Date 类型的 JavaBean，第二行在显示结果的时候，category 使用了 JSP 特有的 scriptlet 的表达式，而日期则使用了 OGNL 表达式。注意 Date 对象返回的年份是以 1900 年为基准的偏移量，因此需要加上 1900；而返回的月份是从 0 开始往后排的，因此需要加上修正值 1。

好了，大功告成，我们快来执行 Tomcat 看看结果吧！启动 Tomcat：

 复制代码

```
1 catalina run
```

打开浏览器，访问：

 复制代码

```
1 http://localhost:8080/books?category=art
```

如果你看到类似如下字样，那么，恭喜你，成功了！现在，你可以回想一下刚才的实现，这些代码该怎样对应到 MVC 各个部分呢？

 复制代码

```
1 Category name: art, date: 2019-8-5
```

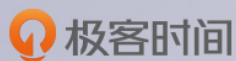
扩展阅读

对于 [Struts](#) 和 [Spring MVC](#)，文中已经给出了官方链接，如果你想阅读简洁的中文版教程，可以看看这个 [Struts 2 教程](#)和这个 [Spring MVC 教程](#)。

文中提到了使用 ControllerClassNameHandlerMapping 来贯彻“规约优于配置”的思想，达到对具体的映射免配置的目的，如果你感兴趣的话，[Spring MVC - Controller](#)

[Class Name Handler Mapping Example](#) 这篇文章有很好的介绍。

[OGNL 语言介绍与实践](#)，文中提到了 OGNL 表达式，感兴趣的话这篇文章是很好的入门。



全栈工程师修炼指南

从全栈入门到技能实战

熊燚

Oracle 首席软件工程师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 09 | MVC架构解析：视图（View）篇

下一篇 11 | 剑走偏锋：面向切面编程

精选留言 (1)

写留言



Kada

2019-10-03

Django使用配置文件，Flask使用装饰器。

展开 ▾

