

有关这个资料的整理是来自于 <http://www.iteye.com/topic/1119598> 的整理 仅仅是为了方便新入职的同事对 springmvc 不是很熟悉的人方便学习而整理.

SpringMVC 深度探险（一） —— SpringMVC 前传

在我们熟知的建立在三层结构（表示层、业务逻辑层、持久层）基础之上的 J2EE 应用程序开发之中，表示层的解决方案最多。因为在表示层自身的知识触角很多，需要解决的问题也不少，这也就难免造成与之对应的解决方案层出不穷。

笔者在很多讨论中经常可以看到类似“某某框架已死”，或者“某某框架已经足以打败所有其他的框架”的言论。**事实上，每一种解决方案都有着自身独有的存在价值和历史背景。**如果单单从某一个方面或者某几个方面去看一个框架，那么评论难免有失偏颇。

所以，整个系列的第一篇文章，我们脱开 SpringMVC 框架本身，把 SpringMVC 放到一个更大的知识体系范围之中，讲一讲整个 Web 开发领域、尤其是 MVC 框架的发展历程。正如“认识历史才能看清未来”，当我们能够正确审视整个 MVC 框架的发展历程，也就能够分析它的发展趋势，并且站在一个更高的高度来对所有的解决方案进行评价。

两种模型

从整个 B/S 程序的运行结构来看，J2EE 的表示层解决方案实际上是对“请求-响应”模式的一种实现。既然谓之“请求-响应”也就势必存在着两大沟通角色：

	载体	语言实现基础
请求方	浏览器	HTML
响应方	Web 服务器	Java

由于这两大角色的承载载体和编程语言实现基础都不同，因而也就产生了两种截然不同的针对表示层的解决方案的设计思路：

- 以服务器端应用程序为主导来进行框架设计
- 以浏览器页面组件（及其自身的事件触发模型）为主导来进行框架设计

业界对于上述这两种不同的设计模型也赋予了不同的名词定义：前一种被称之为 **MVC 模型**；后一种则被称之为**组件模型**，也有称之为**事件模型**。

注：笔者个人对于这两种模型的概念定义并不是非常认同。因为在笔者个人的观点认为，MVC 模型的定义角度所针对的是编程元素的划分；而组件模型（事件模型）的定义角度是

动态交互方式的表述。所以我们在这里强调的是解决方案自身所设立的基准和侧重点的不同。

从使用者的社区力量上来看，无疑 MVC 模型获得了更多程序员的青睐。这里面的原因很多，我们在这里也不想过多展开对两种不同编程模型之间的讨论。不过在这里，我们将针对同一个业务场景（用户注册）分别给出基于这两个编程模型的代码示例，帮助读者了解这两种编程模型在设计思想上的不同之处。

【MVC 模型】

在 MVC 模型中，我们选取当前比较热门的两大框架 Struts2 和 SpringMVC 作为代码示例。

首先，我们将用户注册场景中最为核心的“用户类”定义出来：

Java 代码

```
public class User {  
  
    private String email;  
  
    private String password;  
  
    // 省略了 setter 和 getter 方法  
}
```

紧接着是一个简单的 JSP 表单：

Html 代码

```
<form method="post" action="/register">  
<label>Email:</label><input type="text" name="email" />  
<label>Password:</label><input type="password" name="password" />  
<input type="submit" value="submit" />  
</form>
```

上述这两段代码无论是 SpringMVC 还是 Struts2，都可以共用。而在请求响应处理类（也就是 Controller）上的设计差异是两个框架最大的不同。

如果使用 SpringMVC，那么 Controller 的代码看上去就像这样：

Java 代码

```
@Controller
```

```

@RequestMapping
public class UserController {

    @RequestMapping("/register")
    public ModelAndView register(String email, String password) {
        // 在这里调用具体的业务逻辑代码
        return new ModelAndView("register-success");
    }

}

```

如果使用 Struts2，那么 Controller 的代码看上去就稍有不同：

Java 代码

```

public class UserController {

    private String email;

    private String password;

    public String register() {
        // 在这里调用具体的业务逻辑代码
        return "register-success";
    }

    // 这里省略了 setter 和 getter 方法

}

```

除此之外，Struts2 还需要在某个配置文件中请求映射的配置：

Xml 代码

```

<action name="register" class="com.demo2do.sandbox.web.UserController" method="register">
    <result name="success">/register-success.jsp</result>
</action>

```

从上面的代码示例中，我们可以为整个 MVC 模型的实现总结归纳出一些特点：

1. 框架本身并不通过某种手段来干预或者控制浏览器发送 Http 请求的行为方式。

从上面的代码中我们就可以看到，无论是 SpringMVC 还是 Struts2，它们在请求页面的实现中都使用了原生 HTML 代码。就算是 Http 请求的发送，也借助于 HTML 之中对 Form 提交请求的支持。

2. 页面（View 层）和请求处理类（Controller）之间的映射关系通过某一种配置形式维系起来。

我们可以看到在浏览器和 Web 服务器之间的映射关系在不同的框架中被赋予了不同的表现形式：在 SpringMVC 中，使用了 Annotation 注解；在 Struts2 中，默认采取 XML 配置文件。不过无论是哪一种配置形式，隐藏在其背后的都是对于请求映射关系的定义。

3. Controller 层的设计差异是不同 MVC 框架之间最主要的差异。

这一点实际上是我们在对于 MVC 模型自身进行定义时就反复强调的一点。在上面的例子中，我们可以看到 SpringMVC 使用方法参数来对请求的数据进行映射；而 Struts2 使用 Controller 类内部的属性来进行数据请求的映射。

在 MVC 模型中，浏览器端和服务端端的交互关系非常明确：无论采取什么样的框架，总是以一个明确的 URL 作为中心，辅之以参数请求。因此，URL 看上去就像是一个明文的契约，当然，真正蕴藏在背后的是 Http 协议。所有的这些东西都被放在了台面上，我们可以非常明确地获取到一次交互中所有的 Http 信息。这也是 MVC 模型中最为突出的一个特点。

【组件模型】

在组件模型中，我们则选取较为成熟的 Tapestry5 作为我们的代码示例。

首先，我们来看看请求页面的情况：

Html 代码

```
<form t:type="form" t:id="form">
<t:label for="email"/><input t:type="TextField" t:id="email" t:validate="required,minlength=3"
size="30"/>
<t:label for="password"/><input t:type="PasswordField" t:id="password"
t:validate="required,minlength=3" size="30"/>
<input type="submit" value="Login"/>
</form>
```

在这里，请求的页面不再是原生的 HTML 代码，而是一个扩展后的 HTML，这一扩展包含了对 HTML 标签的扩展（增加了新的标签，例如 <t:label>），也包含了对 HTML 自身标签中属性的扩展（增加新的支持属性，例如 t:type, t:validate）。

接着我们来看看服务器端响应程序：

Java 代码

```

public class Register {

    private String email;

    private String password;

    @Component(id = "password")
    private PasswordField passwordField;

    @Component
    private Form form;

    String onSuccess() {

        return "PostRegister";
    }

    // 这里省略了 setter 和 getter 方法

```

从上面的代码示例中，我们可以看到一些与 MVC 模型截然不同的特点：

1. 框架通过对 HTML 进行行为扩展来干预和控制浏览器与服务器的交互过程。

我们可以发现，Tapestry5 的请求页面被加入了更多的 HTML 扩展，这些扩展包括对 HTML 标签的扩展以及 HTML 标签中属性的扩展。而这些扩展中，有不少直接干预了浏览器与服务器的交互。例如，上面例子中的 `t:validate="required,minlength=3"` 扩展 实际上就会被自动映射到服务器端程序中带有 `@Component(id="password")` 标注的 `PasswordField` 组件上，并在提交时自动 进行组件化校验。而当页面上的提交按钮被点击触发时，默认在服务器端的 `onSuccess` 方法会形成响应并调用其内部逻辑。

2. 页面组件的实现是整个组件模型的绝对核心

从上述的例子中，我们可以看到组件模型的实现不仅需要服务器端实现，还需要在页面上指定与某个特定组件进行事件绑定。两者缺一不可，必须相互配合，共同完成。因此整个 Web 程序的交互能力完全取决于页面组件的实现好坏。

3. 页面组件与服务器端响应程序之间的映射契约并不基于 Http 协议进行

在上面的例子中，从页面组件到服务器端的响应程序之间的映射关系是通过名称契约而定的。而页面上的每个组件可以指定映射到服务器端程序的具体某一个方法。我们可以看到这种映射方式并不是一种基于 URL 或者 Http 协议的映射方式，而是一种命名指定的方式。

在组件模型中，浏览器端和服务端交互关系并不以一个具体的 URL 为核心，我们在上述的例子中甚至完全没有看到任何 URL 的影子。不过这种事件 响应式的方式，也提供给我

们另外一个编程的思路，而这种基于契约式的请求-响应映射也得到了一部分程序员的喜爱。因而组件模型的粉丝数量也是很多的。

MVC 模型的各种形态

之前我们已经谈到，MVC 模型是一种以服务器响应程序（也就是 Controller）为核心进行程序设计的，因而**所有的 MVC 框架的历史发展进程实际上是一个围绕着 Controller 不断进行重构和改造的过程**。而在这个过程中，不同的 MVC 框架也就表现出了不同的表现形态。接下来，我们就给出一些具有代表意义的 MVC 框架表现形态。

注：笔者在这里将提到三种不同的 MVC 框架的表现形态，实际上与请求-响应的实现模式有着密切的联系，有关这一方面的内容，请参阅另外一篇博文的内容：[《Struts2 技术内幕》新书部分篇章连载（五）—— 请求响应哲学](#)

【Servlet】

Servlet 规范是最基本的 J2EE 规范，也是我们进行 Web 开发的核心依赖。它虽然自身并不构成开发框架，但是我们不得不承认所有的 MVC 框架都是从最基本的 Servlet 规范发展而来。因此，我们可以得出一个基本结论：

Servlet 是 MVC 模型最为基本的表现形态。

在 Servlet 规范中所定义的处理请求响应接口是这样的：

```
◆ doGet(HttpServletRequest, HttpServletResponse) : void
◆ getLastModified(HttpServletRequest) : long
◆ doHead(HttpServletRequest, HttpServletResponse) : void
◆ doPost(HttpServletRequest, HttpServletResponse) : void
◆ doPut(HttpServletRequest, HttpServletResponse) : void
◆ doDelete(HttpServletRequest, HttpServletResponse) : void
■ getAllDeclaredMethods(Class) : Method[]
◆ doOptions(HttpServletRequest, HttpServletResponse) : void
◆ doTrace(HttpServletRequest, HttpServletResponse) : void
◆ service(HttpServletRequest, HttpServletResponse) : void
■ maybeSetLastModified(HttpServletResponse, long) : void
● ▲ service(ServletRequest, ServletResponse) : void
```

我们可以看到，Servlet 的基本接口定义中：

参数列表 —— Http 请求被封装为一个 `HttpServletRequest` 对象（或者 `ServletRequest` 对象），而 `Http` 响应封装为一个 `HttpServletResponse` 对象（或者 `ServletResponse` 对象）

返回值 —— 方法不存在返回值（返回值为 `void`）

在这个设计中，`HttpServletRequest` 和 `HttpServletResponse` 承担了完整的处理 `Http` 请求的任务。而这两个 `Servlet` 对象的职责也有所分工：

`HttpServletRequest` 对象 —— 主要用于处理整个 `Http` 生命周期中的数据。

`HttpServletResponse` 对象 —— 主要用于处理 `Http` 的响应结果。

这里实际上有一点“数据与行为分离”的意味。也就是说，在 `Servlet` 处理请求的过程中，其实也是 `Servlet` 中响应方法内部的逻辑执行过程中，如果需要处理请求数据或者返回数据，那么我们需要和 `HttpServletRequest` 打交道；如果需要处理执行完毕之后的响应结果，那么我们需要和 `HttpServletResponse` 打交道。

这样的设计方式，是一种**骨架式**的设计方式。因为 `Servlet` 是我们进行 `Web` 开发中最底层的标准，所以我们可以看到接口设计中的返回值对于一个最底层标准而言毫无意义。因为不存在一个更底层的处理程序会对返回值进行进一步的处理，我们不得不在 `Servlet` 的过程中自行处理浏览器的行为控制。

`MVC` 模型的这一种形态，被笔者冠以一个名称：**参数-参数（Param-Param）实现模式**。因为在响应方法中，数据与行为的操作载体都以参数的形式出现。

`Servlet` 的设计模型是所有 `MVC` 模型表现形态中最为基础也是最为底层的一种模型，所有其他模型都是建立在这一模型的基础之上扩展而来。

【Struts1.X】

Struts1.X 是一个较为早期的 MVC 框架实现，它的历史最早可以追溯到 2000 年，作为 Apache 开源组织的一个重要项目，取名为“Struts”，有“基础构建”的含义。在那个程序框架尚处于朦胧阶段的年代，“基础构建”无疑是每个程序员梦寐以求的东西。

对于 Struts1.X，我们还是把关注的重点放在 Struts 中的 Controller 层的定义上：

Java 代码

```
public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response);
```

如果和之前的 Servlet 模型加以比较我们就可以发现，Struts1.X 对于基本的 Servlet 模型做了一定的扩展和重构：

- 保留了 HttpServletRequest 和 HttpServletResponse 这两大接口作为参数
- 将返回值改为 ActionForward，并由 Struts1.X 框架通过处理 ActionForward 完成对响应结果的处理
- 增加了 ActionMapping 和 ActionForm 两大参数，前者表示 Http 请求的一个简要概括，后者表示一个数据模型，用以承载整个请求生命周期中的数据

经过一番扩展和重构，我们可以发现 Struts1.X 相比较于原始的 Servlet 模型已经有了一定的进步。比如，我们可以不再直接操作 HttpServletResponse 这样的原生 Servlet 对象来进行 Http 返回的处理；再比如，对于一些简单的请求数据读取，我们可以不必直接操作生硬的 HttpServletRequest 接口，而通过 ActionForm 来完成。

MVC 模型发展到了这里，我们可以看到响应方法中的“返回值”已经能够被调动起来用在整个 Http 请求的处理过程中。因此，这种在响应方法中参数和返回值同时参与到 Http 请求的处理过程中的表现形态，被笔者冠以另外一个名称：**参数-返回值（Param-Return）实现模式**。

由于 Struts1.X 已经不再是一个底层的实现规范，于是响应方法“返回值”被框架引入，加入到了整个处理过程之中。我们可以看到，在这里最大的进步之处就在于：**引入了新的编程元素，从而优化整个逻辑处理过程**。编程元素的引入非常重要，因为对于一个任何一个程序员而言，充分调用所有可以利用的编程要素是衡量一个程序写得好坏的重要标准。之后，我们还可以看到其他的框架在引入编程元素这方面所做的努力。

【Webwork2 / Struts2】

随着时间的推移，越来越多的程序员在使用 Struts1.X 进行开发的过程中发现 Struts1.X 在设计上存在的一些不足。而与此同时，各种各样的 Web 层的解决方案也如雨后天春笋般涌现出来。不仅仅是以 MVC 模型为基础的开发框架，还有包括 JSF 和 Tapestry 之类的基于组件模型的开发框架 也在这个时期诞生并不断发展壮大。因此，这个时期应该是整个 Web 层解决方案的大力发展时期。

而在这些框架中，有一个来自于 Opensymphony 开源社区的优秀框架 Webwork2 探索了一条与传统 Servlet 模型不同的解决方案，逐渐被大家熟识和理解，不断发展并得到了广大程序员的认可。2004 年，Webwork2.1.7 版本发布，成为 Webwork2 的一个重要里程碑，它以优秀的设计思想和灵活的实现，吸引了大批的 Web 层开发人员投入它的怀抱。

或许是看到了 Struts1.X 发展上的局限性，Apache 社区与 Opensymphony 开源组织在 2005 年底宣布未来的 Struts 项目 将与 Webwork2 项目合并，并联合推出 Struts2，通过 Apache 社区的人气优势与 OpenSymphony 的技术优势，共同打造下一代的 Web 层开发框架。这也就是 Struts2 的由来。

从整个过程中，我们可以发现，**Webwork2 和 Struts2 是一脉相承的 Web 层解决方案**。而两者能够在一个相当长的时间段内占据开发市场主导地位的重要原因在于其技术上的领先优势。而这一技术上的领先优势，突出表现为对 Controller 的彻底改造：

Java 代码

```
public class UserController {

    private User user

    public String execute() {
        // 这里加入业务逻辑代码
        return "success";
    }

    // 这里省略了 setter 和 getter 方法
}
```

从上面的代码中，我们可以看到 Webwork2 / Struts2 对于 Controller 最大的改造有两点：

- 在 Controller 中彻底杜绝引入 HttpServletRequest 或者 HttpServletResponse 这样的原生 Servlet 对象。
- 将请求参数和响应数据都从响应方法中剥离到了 Controller 中的属性变量。

这两大改造被看作是框架的神来之笔。因为通过这一改造，整个 Controller 类彻底与 Web 容器解耦，可以方便地进行单元测试。而摆脱了 Servlet 束缚的 Controller，也为整个编程模型赋予了全新的定义。

当然，这种改造的前提条件在于 Webwork2 / Struts2 引入了另外一个重要的编程概念：ThreadLocal 模式。使得 Controller 成为一个线程安全的对象被 Servlet 模型所调用，这也就突破了传统 Servlet 体系下，Servlet 对象并非一个线程安全的对象的限制条件。

注：有关 ThreadLocal 模式相关的话题，请参考另外一篇博文：[《Struts2 技术内幕》新书部分篇章连载（七）—— ThreadLocal 模式](#)

从引入新的编程元素的角度来说，Webwork2 / Struts2 无疑也是成功的。因为在传统 Servlet 模式中的禁地 Controller 中的属性变量被合理利用了起来作为请求处理过程中的数据部分。这样的改造不仅使得表达式引擎能够得到最大限度的发挥，同时使得整个 Controller 看起来更像一个 POJO。因而，这种表现形态被笔者冠以的名称是：**POJO 实现模式**。

POJO 实现模式是一种具有革命性意义的模式，因为它能够把解耦合这样一个观点发挥到极致。从面向对象的角度来看，POJO 模式无疑也是所有程序员所追求的一个目标。这也就是 Webwork2 / Struts2 那么多年来经久不衰的一个重要原因。

【SpringMVC】

相比较 Webwork2 / Struts2，SpringMVC 走了一条比较温和的改良路线。因为 SpringMVC 自始至终都没有突破传统 Servlet 编程模型的限制，而是在这过程中不断改良，不断重构，反而在发展中开拓了一条崭新的道路。

我们可以看看目前最新版本的 SpringMVC 中对于 Controller 的定义：

Java 代码

@Controller

@RequestMapping

public class UserController {

 @RequestMapping("/register")

 public ModelAndView register(String email, String password) {

 // 在这里调用具体的业务逻辑代码

 return new ModelAndView("register-success");

 }

}

我们在这里引用了在之前的讲解中曾经使用过的代码片段。不过这一代码片段刚刚好可以说明 SpringMVC 在整个 Controller 改造中所涉及到的一些要点：

1. 使用参数-返回值（Param-Return）实现模式来打造 Controller

方法的参数（*email* 和 *password*）被视作是 *Http* 请求参数的概括。而在这里，它们已经被 *SpringMVC* 的框架有效处理并屏蔽了内在的处理细节，呈现出来的是与请求参数名称一一对应的参数列表。而返回值 *ModelAndView* 则表示 *Http* 的响应是一个数据与视图的结合体，表示 *Http* 的处理结果。

2. 引入 Annotation 来完成请求-响应的映射关系

引入 *Annotation* 来完成请求-响应的映射关系，是 *SpringMVC* 的一个重大改造。在早期的 *SpringMVC* 以及其他的 *MVC* 框架中，通常都是使用 *XML* 作为基础配置的。而 *Annotation* 的引入将原本分散的关注点合并到了一起，为实现配置简化打下了坚实的基础。

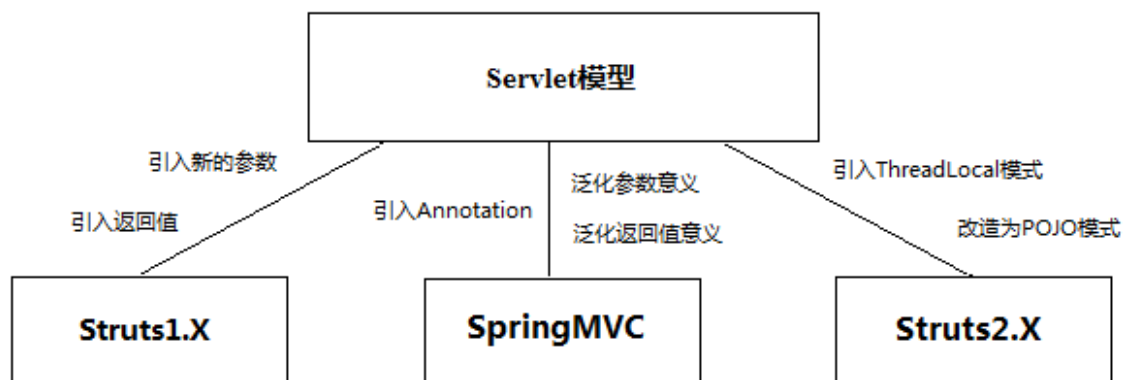
3. 泛化参数和返回值的含义

这是一个蕴含的特点。事实上，*SpringMVC* 在响应方法上，可以支持多种多样不同的参数类型和返回值类型。例如，当参数类型为 *Model* 时，*SpringMVC* 将会自动将请求参数封装于 *Model* 内部而传入请求方法；当返回值类型是 *String* 时，直接表示 *SpringMVC* 需要返回的视图类型和视图内容。当然，这些泛化的参数和返回值的内容全部都由 *SpringMVC* 在框架内部处理了。

如果我们来评述一下这些特点就会发现，*SpringMVC* 虽然是一个温和的改良派，却是在改良这个领域做得最为出色的。以引入 *Annotation* 为例，引入 *Annotation* 来完成请求-响应映射，不正是我们反复强调的引入并合理使用新的编程元素来完成处理任务嘛？而泛化后的参数和返回值，则可以让程序员在写 *Controller* 的代码时可以随心所欲，不再受到任何契约的束缚，这样一来接口的逻辑语义也就能够更加清晰。

MVC 模型的发展轨迹

之前讲了那么多 *MVC* 模型的实现形态，我们是否能从中总结出一条发展轨迹呢？答案是肯定的，笔者在这里作了一副图：



从图中，我们可以看到三类完全不同的发展方向。目前，**Struts1.X** 这一条路被证明已经穷途末路；另外的两条发展轨迹总体来说实力相当，**SpringMVC** 大有赶超之势。

那么，为什么曾经一度占领了大部分市场的 **Struts2** 会在近一段时间内被 **SpringMVC** 大幅赶超呢？这里面的原因多种多样，有自身架构上的原因，有设计理念上的原因，但是笔者认为，其本质原因还是在于 **Struts2** 对于技术革新的力度远不及 **SpringMVC**。

如果我们回顾一下 **Struts2** 过去能够独占鳌头的原因就可以发现，**Struts2** 的领先在于编程模型上的领先。其引入的 **POJO** 模型几乎是一个杀手级的武器。而基于这一模型上的拦截器、**OGNL** 等技术的支持使得其他的编程模型在短时间很难超越它。

但是随着时代的发展，**Struts2** 在技术革新上的作为似乎步子就迈得比较小。我们可以看到，在 **JDK1.5** 普及之后，**Annotation** 作为一种新兴的 **Java** 语法，逐渐被大家熟知和应用。这一点上 **SpringMVC** 紧跟了时代的潮流，直接用于请求-响应的映射。而 **Struts2** 却迟迟无法在单一配置源的问题上形成突破。当然，这只是技术革新上的一个简单的例子，其他的例子还有很多。

有关 **Struts2** 和 **SpringMVC** 的比较话题，我们在之后的讨论中还会有所涉及，不过笔者并不希望在这里引起框架之间的争斗。大家应该客观看待每个框架自身设计上的优秀之处和不足之处，从而形成个人自己的观点。

从整个 **MVC** 框架的发展轨迹来看，我们可以得出一个很重要的结论：

MVC 框架的发展轨迹，始终是伴随着技术的革新（无论是编程模型的改变还是引入新的编程元素）共同向前发展。而每一次的技术革新，都会成为 MVC 框架发展过程中的里程碑。

小结

在本文中所讲的一些话题触角涉及到了 **Web** 开发的各个方面。作为 **SpringMVC** 的前传，笔

者个人认为将整个 MVC 框架的发展历程讲清楚，大家 才能更好地去了解 SpringMVC 本身。而我们在这一章所谈到的一些概念性的话题，也算是对过去十年以来 MVC 框架的一个小结，希望对读者有所启示。