\对于任何事物的研究，总是由表及里、由浅入深地进行。在本系列的第二篇文章中，我们将通过不同的观察视角，对 SpringMVC 做一些概要性的分析，帮助大家了解 SpringMVC 的基本构成要素、SpringMVC 的发展历程以及 SpringMVC 的设计原则。

SpringMVC 的构成要素.

了解一个框架的首要任务就是搞清楚这个框架的基本构成要素。当然，这里所说的构成要素实际上还可以被挖掘为两个不同的层次：

- **基于框架所编写的应用程序的构成要素**
- **框架自身的运行主线以及微观构成要素**

我们在这里首先来关注一下第一个层次，因为第一个层次是广大程序员直接能够接触得到的部分。而第二个层次的讨论，我们不得不在第一个层次的讨论基础之上通过不断分析和逻辑论证慢慢给出答案。

在上一篇文章中，我们曾经列举了一段 SpringMVC 的代码示例，用于说明 MVC 框架的构成结构。我们在这里不妨将这个示例细化，总结归纳出构成 SpringMVC 应用程序的基本要素。

1. 指定 SpringMVC 的入口程序（在 web.xml 中）

Xml 代码

```
<!-- Processes application requests -->
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/**</url-pattern>
</servlet-mapping>
```

*以一个 Servlet 作为入口程序是绝大多数 MVC 框架都遵循的基本设计方案。这里的 DispatcherServlet 被我们称之为**核心分发器**，是 SpringMVC 最重要的类之一，之后我们会对其单独展开进行分析。*

2. 编写 SpringMVC 的核心配置文件（在[servlet-name]-servlet.xml 中）

Xml 代码

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:mvc="http://www.springframework.org/schema/mvc"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
                http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
                http://www.springframework.org/schema/context
                http://www.springframework.org/schema/context/spring-context-3.1.xsd
                http://www.springframework.org/schema/mvc
                http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd"
        default-autowire="byName">

    <!-- Enables the Spring MVC @Controller programming model -->
    <mvc:annotation-driven />

    <context:component-scan base-package="com.demo2do" />


    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

SpringMVC 自身由众多不同的组件共同构成，而每一个组件又有众多不同的实现模式。这里的 SpringMVC 核心配置文件是定义 SpringMVC 行为方式的一个窗口，用于指定每一个组件的实现模式。有关 SpringMVC 组件的概念，在之后的讨论中也会涉及。

3. 编写控制(Controller)层的代码

Java 代码

```
@Controller
@RequestMapping
public class UserController {

    @RequestMapping("/login")
    public ModelAndView login(String name, String password) {
        // write your logic here
            return new ModelAndView("success");
    }
```

}

控制（Controller）层的代码编写在一个 Java 文件中。我们可以看到这个 Java 文件是一个普通的 Java 类并不依赖于任何接口。只是在响应类和响应方法上使用了 Annotation 的语法将它与 Http 请求对应起来。

从这个例子中，我们实际上已经归纳了构成基于 SpringMVC 应用程序的最基本要素。它们分别是：

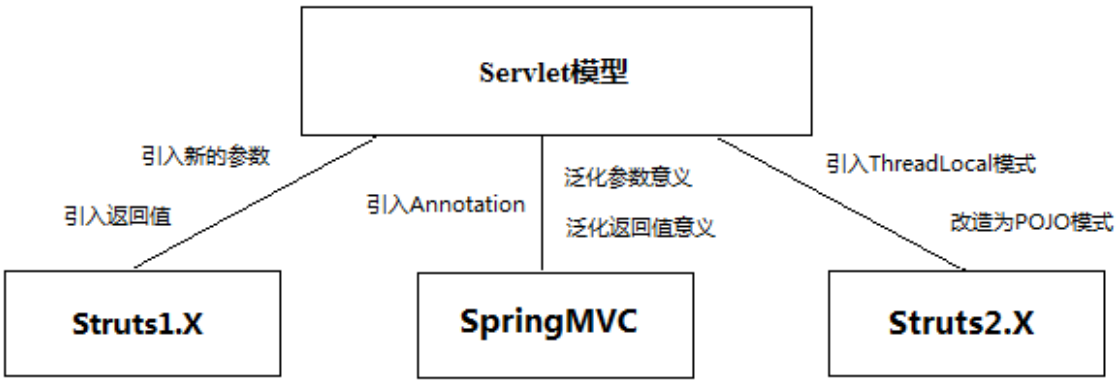
- 入口程序 —— DispatcherServlet
- 核心配置 —— [servlet-name]-servlet.xml
- 控制逻辑 —— UserController

从应用程序自身的角度来看，入口程序和核心配置一旦确定之后将保持固定不变的，而控制逻辑则随着整个应用程序功能模块的扩展而不断增加。所以在这种编程模式下，应用程序的纵向扩展非常简单并且显得游刃有余。

基于 SpringMVC 的应用程序能够表现为现在这个样子，经历了一个不断重构不断改造的过程。接下来的讨论，我们就来试图为大家揭秘这个过程。

SpringMVC 的发展历程

在上一篇文章中，我们曾经讨论过 MVC 的发展轨迹。当时我们总结了一个 MVC 框架的发展轨迹图：



从图中我们可以发现，所有的 MVC 框架都是从基本的 Servlet 模型发展而来。因此，要了解 SpringMVC 的发展历程，我们还是从最基本的 Servlet 模型开始，探究 SpringMVC 对于 Servlet 模型的改造过程中究竟经历了哪些阶段、碰到了哪些问题、并看看 SpringMVC 是如 何解决这些问题的。

【核心 Servlet 的提炼】

在 Servlet 模型中，请求-响应的实现依赖于两大元素的共同配合：

1. 配置 Servlet 及其映射关系（在 web.xml 中）

Xml 代码

```
<servlet>
    <servlet-name>registerServlet</servlet-name>
    <servlet-class>com.demo2do.springmvc.web.RegisterServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>registerServlet</servlet-name>
    <url-pattern>/register</url-pattern>
</servlet-mapping>
```

在这里，<url-pattern>定义了整个请求-响应的映射载体：URL；而<servlet-name>则 将<servlet>节点和<servlet-mapping>节点联系在一起形成请求-响应的映射关 系；<servlet-class>则定义了具体进行响应的 Servlet 实现类。

2. 在 Servlet 实现类中完成响应逻辑

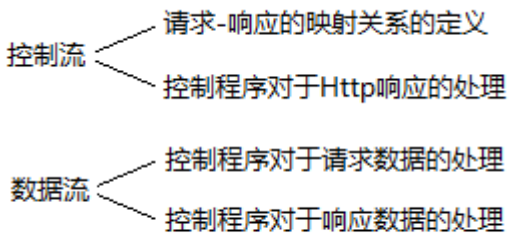
Java 代码

```
public class RegisterServlet extends    HttpServlet {

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

        // 从 request 获取参数
        String name = req.getParameter("name");
        String birthdayString = req.getParameter("birthday");

        // 做必要的类型转化
        Date birthday = null;
        try {
            birthday = new SmpleDateFormat("yyyy-MM-dd").parse(birthdayString);
        } catch (ParseException e) {
        e.printStackTrace();
```

```
            }

            // 初始化 User 类，并设置字段到 user 对象中去
            User user = new User();
            user.setName(name);
            user.setBirthday(birthday);

            // 调用业务逻辑代码完成注册
            UserService userService = new UserService();
            userService.register(user);

            // 设置返回数据
            request.setAttribute("user", user);

            // 返回成功页面
            req.getRequestDispatcher("/success.jsp").forward(req, resp);
        }
}
```

Servlet 实现类本质上是一个 Java 类。通过 Servlet 接口定义中的 HttpServletRequest 对象，我们可以处理整个请求生命周期中的数据；通过 HttpServletResponse 对象，我们可以处理 Http 响应行为。

整个过程并不复杂，因为作为一个底层规范，所规定的编程元素和实现方式应该尽可能直观和简单。在这一点上，Servlet 规范似乎可以满足我们的要求。如果将上述过程中的主要过程加以抽象，我们可以发现有两个非常重要概念蕴含在了 Servlet 的规范之中：



控制流和数据流的问题几乎贯穿了所有 MVC 框架的始末，因而我们不得不在这里率先提出来，希望对读者有一些警示作用。

***注**：对于控制流和数据流的相关概念，请参考另外一篇博客：《Struts2 技术内幕》 新书部分篇章连载（五）—— 请求响应哲学。这一对概念，几乎是所有 MVC 框架背后最为重要的支撑，读者应该尤其重视！*

所有 MVC 框架的核心问题也由控制流和数据流这两大体系延伸开来。比如，在 Servlet 编程模型之下，“请求-响应映射关系的定义”这一问题就会随着项目规模的扩大而显得力不从心：

问题 1
项目规模扩大之后，请求-响应的映射关系全部定义在 web.xml 中，将造成 web.xml 的不断膨胀而变得难以维护。

针对这个问题，SpringMVC 提出的方案就是：**提炼一个核心的 Servlet 覆盖对所有 Http 请求的处理。**。

这一被提炼出来的 Servlet，通常被我们称之为：**核心分发器**。在 SpringMVC 中，核心分发器就是 **org.springframework.web.servlet.DispatcherServlet**。

注：核心分发器的概念并非 SpringMVC 独创。我们可以看到，核心分发器的提炼几乎是所有 MVC 框架设计中的必经之路。在 Struts2 中，也有核心分发器 （Dispatcher）的概念，只是它并不以 Servlet 的形式出现。因此，读者应该把关注点放在核心分发器这个概念的提炼之上，而不是纠结于其形 式。

有了 DispatcherServlet，我们至少从表面上解决了上面的问题。至少在 web.xml 中，我们的配置代码就被固定了下来：

Xml 代码

```
<!-- Processes application requests -->
<servlet>
    <servlet-name>dispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>dispatcherServlet</servlet-name>
    <url-pattern>/**</url-pattern>
</servlet-mapping>
```

有了 DispatcherServlet，我们只相当于迈出了坚实的第一步，因为对核心 Servlet 的提炼不仅仅是将所有的 Servlet 集中在一起那么简单，我们还将面临两大问题：

问题 2
核心 Servlet 应该能够根据一定的规则对不同的 Http 请求分发到不同的 Servlet 对象上去进行处理。
问题 3
核心 Servlet 应该能够建立起一整套完整的对所有 Http 请求进行规范化处理的流程。

而这两大问题的解决，涉及到了 DispatcherServlet 的设计核心。我们也不得不引入另外一个重要的编程元素，那就是：组件。

【组件的引入】

DispatcherServlet 的引入是我们通过加入新的编程元素来对基本的 Servlet 规范进行抽象概括所迈出的第一步。不过接下来，有关 DispatcherServlet 的设计问题又一次摆到了我们的面前。

如果仔细分析一下上一节末尾所提出的两个问题，我们可以发现这两个问题实际上都涉及到了 DispatcherServlet 的处理过程，这一处理 过程首先必须是一剂万能药，能够处理所有的 Http 请求；同时，DispatcherServlet 还需要完成不同协议之间的转化工作（从 Http 协议到 Java 世界的转化）。

对此，SpringMVC 所提出的方案是：**将整个处理流程规范化，并把每一个处理步骤分派到不同的组件中进行处理**。

这个方案实际上涉及到两个方面：

- 处理流程规范化 —— 将处理流程划分为若干个步骤（任务），并使用一条明确的逻辑主线将所有的步骤串联起来
- 处理流程组件化 —— 将处理流程中的每一个步骤（任务）都定义为接口，并为每个接口赋予不同的实现模式

在 SpringMVC 的设计中，这两个方面的内容总是在一个不断交叉、互为补充的过程中逐步完善的。

处理流程规范化是目的，对于处理过程的步骤划分和流程定义则是手段。因而处理流程规范化的首要内容就是考虑一个通用的 Servlet 响应程序大致应该包含的逻辑步骤：

- 步骤 1 —— 对 Http 请求进行初步处理，查找与之对应的 Controller 处理类（方法）
- 步骤 2 —— 调用相应的 Controller 处理类（方法）完成业务逻辑
- 步骤 3 —— 对 Controller 处理类（方法）调用时可能发生的异常进行处理
- 步骤 4 —— 根据 Controller 处理类（方法）的调用结果，进行 Http 响应处理

这些逻辑步骤虽然还在我们的脑海中，不过这些过程恰恰正是我们对整个处理过程的流程化概括，稍后我们就会把它们进行程序化处理。

所谓的程序化，实际上也就是使用编程语言将这些逻辑语义表达出来。在 Java

语言中，最适合表达逻辑处理语义的语法结构是**接口**，因此上述的四个流程也就被定义为了四个不同接口，它们分别是：

- 步骤 1 —— HandlerMapping
- 步骤 2 —— HandlerAdapter
- 步骤 3 —— HandlerExceptionResolver
- 步骤 4 —— ViewResolver

结合之前我们对流程组件化的解释，这些接口的定义不正是处理流程组件化的步骤嘛？**这些接口，就是组件**。

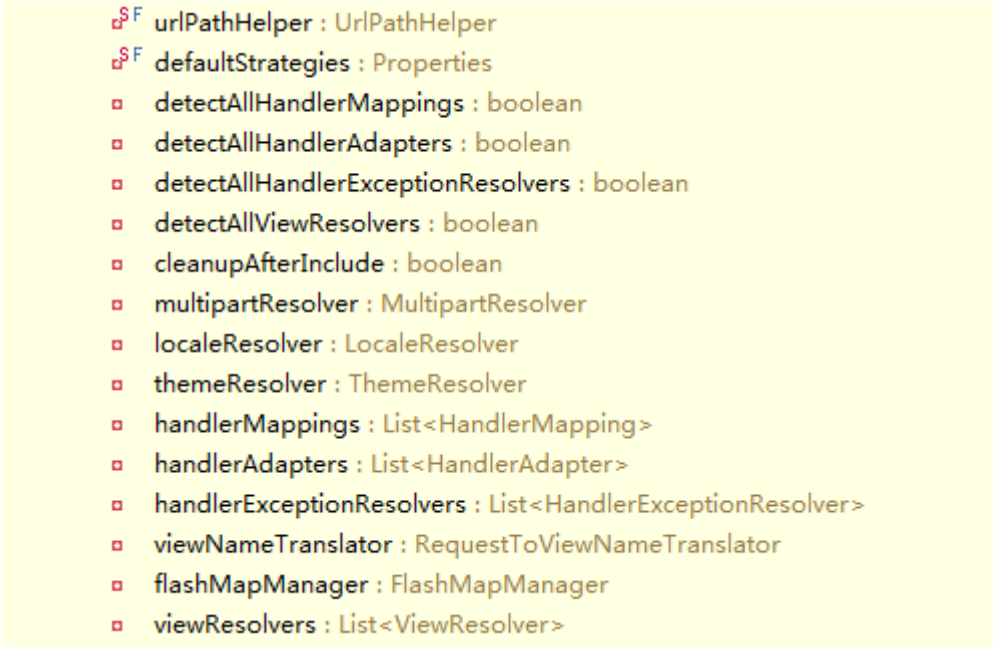
除了上述组件之外，SpringMVC 所定义的组件几乎涵盖了每一个处理过程中的重要节点。我们在这里引用 Spring 官方 reference 中对于最基本的组件的一些说明：

Bean type	Explanation
HandlerMapping	Maps incoming requests to handlers and a list of pre- and post-processors (handler interceptors) on some criteria the details of which vary by `HandlerMapping` implementation. The most popular implementation supports annotated controllers but other implementations exists as well.
HandlerAdapter	Helps the `DispatcherServlet` to invoke a handler mapped to a request regardless of the handler actually invoked. For example, invoking an annotated controller requires resolving various annot Thus the main purpose of a `HandlerAdapter` is to shield the `DispatcherServlet` from such details
HandlerExceptionResolver	Maps exceptions to views also allowing for more complex exception handling code.
ViewResolver	Resolves logical String-based view names to actual View types.
LocaleResolver	Resolves the locale a client is using, in order to be able to offer internationalized views
ThemeResolver	Resolves themes your web application can use, for example, to offer personalized layouts
MultipartResolver	Parses multi-part requests for example to support processing file uploads from HTML forms.
FlashMapManager	Stores and retrieves the "input" and the "output" `FlashMap` that can be used to pass attributes fro request to another, usually across a redirect.

我们在下一篇文章中将重点对这里所提到的所有组件做深入的分析。大家在这里需要理解的是 SpringMVC 定义这些组件的目的和初衷。

这些组件一旦被定义，自然而然也就引出了下一个问题：这些组件是如何串联在一起的？这个过程，是在 DispatcherServlet 中完成的。有关这一点，我们可以从两个不同的角度加以证明。

1. 从 DispatcherServlet 自身数据结构的角度



如图中所示，DispatcherServlet 中包含了众多 SpringMVC 的组件，这些组件是实现 DispatcherServlet 核心逻辑的基础。

2. 从 DispatcherServlet 的核心源码的角度

Java 代码

```
try {
    // 这里省略了部分代码

    // 获取 HandlerMapping 组件返回的执行链
    mappedHandler = getHandler(processedRequest, false);
    if (mappedHandler == null || mappedHandler.getHandler() == null) {
        noHandlerFound(processedRequest, response);
        return;
    }

    // 获取 HandlerAdapter 组件
    HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

    // 这里省略了部分源码

    // 调用 HandlerAdapter 组件
    mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

    // 这里省略了部分源码

}catch (ModelAndViewDefiningException ex) {
    logger.debug("ModelAndViewDefiningException encountered", ex);
```

```
        mv = ex.getModelAndView();
    }catch (Exception ex) {
        Object handler = (mappedHandler != null ? mappedHandler.getHandler() : null);
        // 调用 HandlerExceptionResolver 进行异常处理
        mv = processHandlerException(processedRequest, response, handler, ex);
        errorView = (mv != null);
    }
```

从上面的代码片段中，我们可以看到 DispatcherServlet 的核心逻辑不过是对组件的获取和调用。

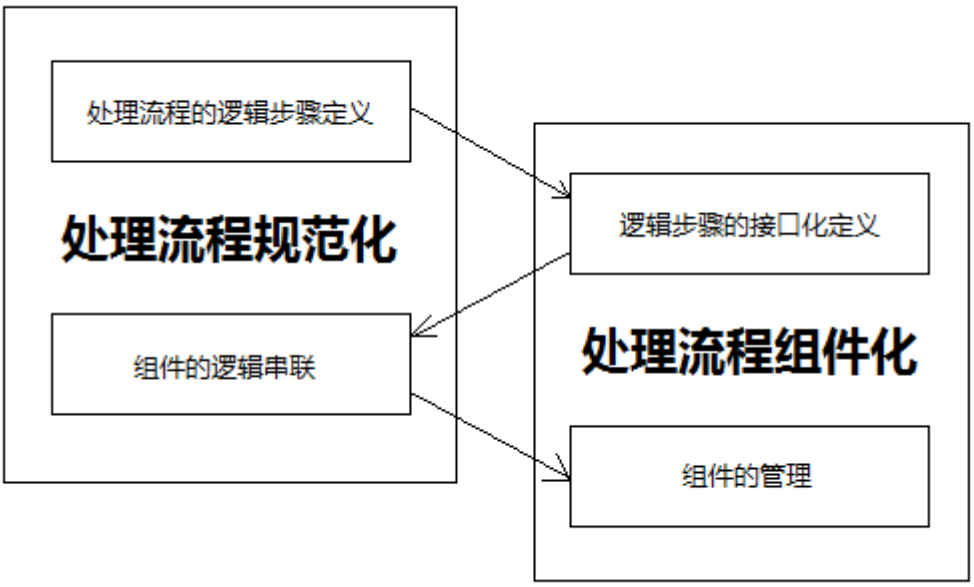
除此之外，SpringMVC 对处理流程的规范化和组件化所引出的另外一个问题就是如何针对所有的组件进行**管理**。

先说说管理。其实管理这些组件对于 SpringMVC 来说完全不是问题，因为 SpringMVC 作为 Spring Framework 的一部分，其自身的运行环境就是 Spring 所定义的容器之中。我们知道，Spring Framework 的核心作用之一就是对整个应用程序的组件进行管理。所以 SpringMVC 对于这些已定义组件的管理，只不过是借用了 Spring 自身 已经提供的容器功能而已。

注：SpringMVC 在进行组件管理时，会单独为 SpringMVC 相关的组件构建一个容器环境，这一容器环境可以独立于应用程序自身所创建的 Spring 容器。有关这一点，我们在之后的讨论中将详细给出分析。

而 SpringMVC 对这些组件的管理载体，就是我们在上一节中所提到的**核心配置文件**。我们可以看到，核心配置文件在整个 SpringMVC 的构成要素中占有一席之地的重要原因就是在于：**我们必须借助一个有效的手段对整个 SpringMVC 的组件进行定义，而这一点正是通过核心配置文件来完成的。**

如果我们把上面的整个过程重新梳理一下，整个逻辑看起来就像这样：



这四个方面的内容，我们是顺着设计思路的不断推进而总结归纳出来的。这也恰好证明之前所提到的一个重要观点，我们在这里强调一下：

处理流程的规范化和组件化，是在一个不断交叉、互为补充的过程中逐步完善的。

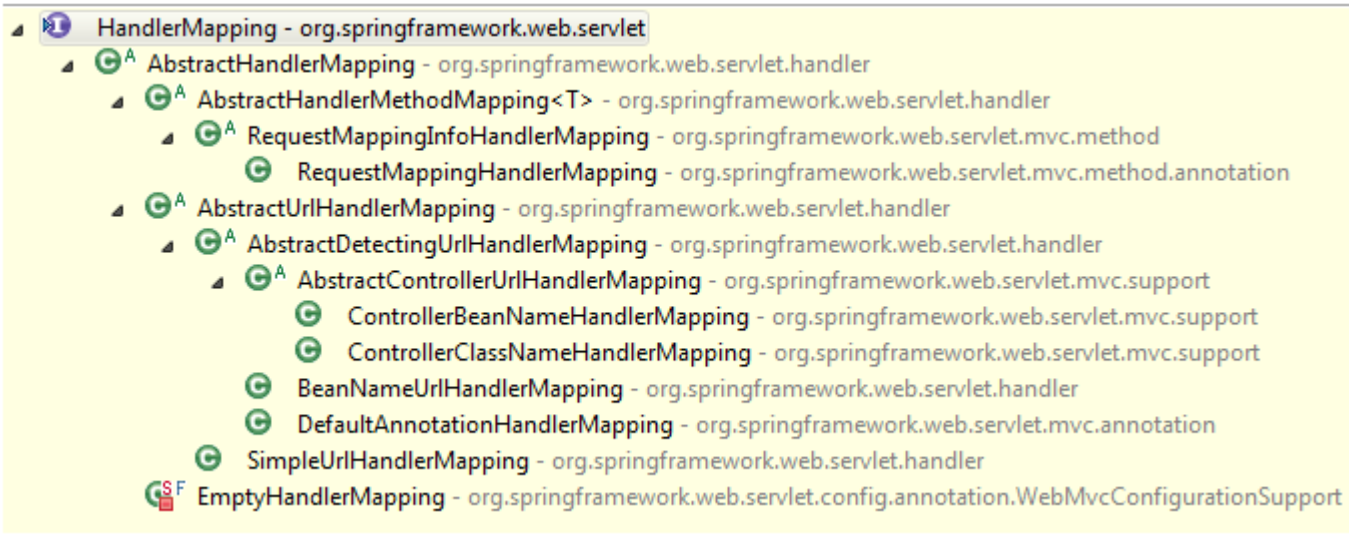
【行为模式的扩展】

有了组件，也有了 DispatcherServlet 对所有组件的串联，我们之前所提出的两个问题似乎已经可以迎刃而解。所以，我们可以说：
SpringMVC 就是通过 DispatcherServlet 将一堆组件串联起来的 Web 框架。

在引入组件这个概念的时候，我们所强调的是处理流程的抽象化，因而所有组件的外在表现形式是接口。接口最重要意义是定义操作规范，所以接口用来表达每一个处理单元的逻辑语义是最合适不过的。但光有接口，并不能完整地构成一个框架的行为模式。**从操作规范到行为模式的变化，是由接口所对应的实现类来完成的。**

在 Java 语言中，一个接口可以有多个不同的实现类，从而构成一个树形的实现体系。而每一个不同的实现分支，实际上代表的是对于相同的逻辑语义的不同解读方式。结合上面我们的描述，也可以说：**一个接口的每一个不同的实现分支，代表了相同操作规范的不同行为模式。**

我们可以通过之前曾经提到过的一个 SpringMVC 组件 HandlerMapping 为例进行说明。



上图就是 HandlerMapping 接口的树形实现体系。在这个实现体系结构中，每一个树形结构的末端实现都是 SpringMVC 中比较具有典型意义的行为模式。我们可以截取其中的几个实现来加以说明：

- BeanNameUrlHandlerMapping —— 根据 Spring 容器中的 bean 的定义来指定请求映射关系
- SimpleUrlHandlerMapping —— 直接指定 URL 与 Controller 的映射关系，其中的 URL 支持 Ant 风格
- DefaultAnnotationHandlerMapping —— 支持通过直接扫描 Controller 类中的 Annotation 来确定请求映射关系
- RequestMappingHandlerMapping —— 通过扫描 Controller 类中的 Annotation 来确定请求映射关系的另外一个实现类

有关这几个实现类的具体示例和使用说明，读者可以参考不同版本的 Spring 官方文档来获取具体的细节。

注：我们在这里之所以要强调不同版本的 Spring 官方文档的原因在于这些不同的实现类，正代表了不同版本 SpringMVC 在默认行为模式上选择的不同。在下图中，我们列出了不同重大版本的 SpringMVC 的实现体系结构，并用红色框圈出了每个版本默认的实现类。

我们可以看到，上述这些不同的 HandlerMapping 的实现类，其运行机制和行为模式完全不同。这也就意味着对于 HandlerMapping 这个 组件而言，可以进行选择的余地就很大。我们既可以选择其中的一种实现模式作为默认的行为模式，也可以将这些实现类依次串联起来成为一个执行链。不过这已经 是实现层面和设计模式上的小技巧了。

单就 HandlerMapping 一个组件，我们就能看到各种不同的行为模式。如果我们将逻辑主线中所有的组件全部考虑进来，那么整个实现机制就会随着这些组件实现体系的不同而表现出截然不同的行为方式了。因此，我们的结论是：

SpringMVC 各种不同的组件实现体系成为了 SpringMVC 行为模式扩展的有效途径。

有关 SpringMVC 的各种组件和实现体系，我们将在之后的讨论中详细展开。

SpringMVC 的设计原则

最后我们来讨论一下 SpringMVC 的设计原则。任何框架在设计的时候都必须遵循一些基本的原则，而这些原则也成为整个框架的理论基础。对于那 些有一定 SpringMVC 使用经验的程序员来说，这些基本的设计原则本身也一定是给大家留下深刻印象的那些闪光点，所以我们非常有必要在这里加以总结。

【Open for extension / closed for modification】

这条重要的设计原则被写在了 Spring 官方的 reference 中 SpringMVC 章节的起始段

Spring Reference 写道

A key design principle in Spring Web MVC and in Spring in general is the "**Open for extension, closed for modification**" principle.

SpringMVC 在整个官方 reference 的起始就强调这一原则，可见其对于整个框架的重要性。那么我们又如何来理解这段话的含义呢？笔者在这里从源码的角度归纳了四个方面：

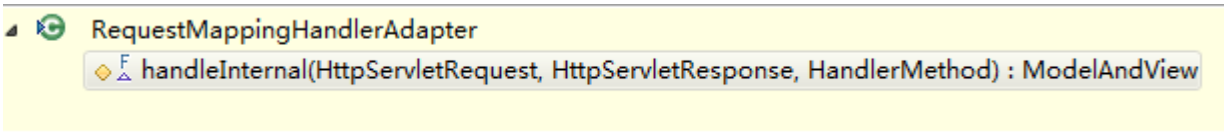
1. 使用 final 关键字来限定核心组件中的核心方法

有关这一点，我们还可以在 Spring 官方的 reference 中找到非常明确的说明：

Spring Reference 写道

Some methods in the core classes of Spring Web MVC are marked final. As a developer you cannot override these methods to supply your own behavior. This has not been done arbitrarily, but specifically with this principle in mind.

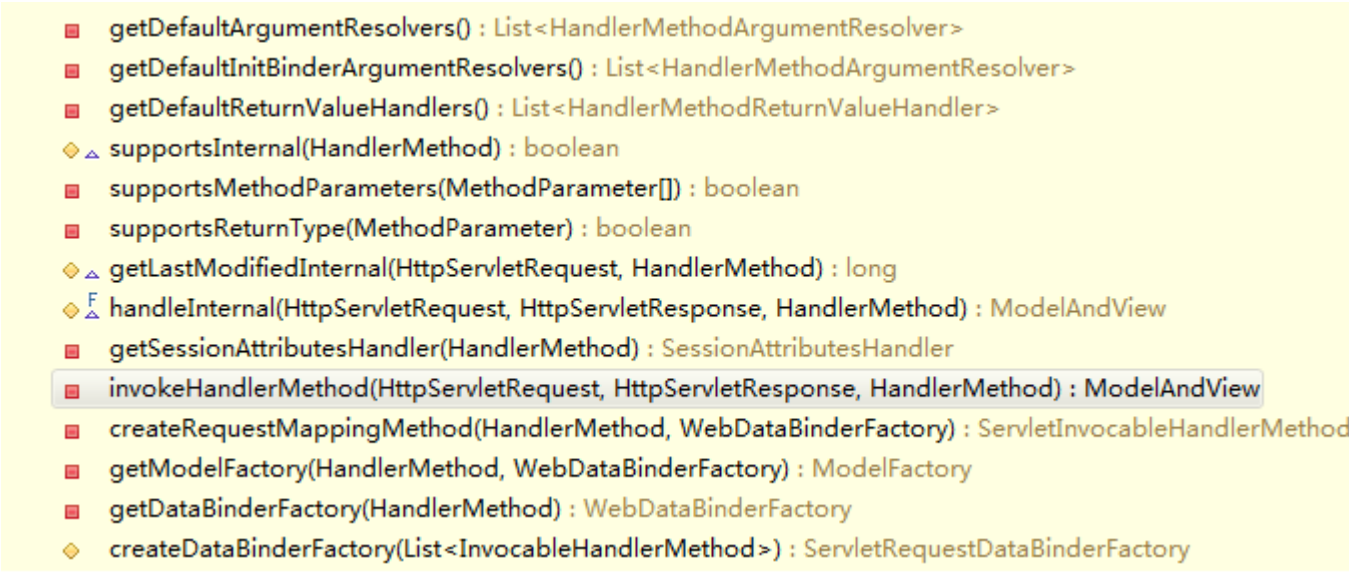
在 SpringMVC 的源码中，HandlerAdapter 实现类 RequestMappingHandlerAdapter 中，核心方法 handleInternal 就被定义为 final：



结论 As a developer you cannot override these methods to supply your own behavior

2. 大量地在核心组件中使用 private 方法

我们依然以 SpringMVC 默认的 HandlerAdapter 实现 RequestMappingHandlerAdapter 为例进行说明：



可以看到，几乎所有的核心处理方法全部被定义成了带有红色标记的 private 方法，这就充分表明了 SpringMVC 对于“子类扩展”这种方式的态度：

结论 子类不允许通过继承的方式改变父类的默认行为。

3. 限定某些类对外部程序不可见

有关这一点，有好几个类可以加以证明，我们不妨来看看它们的源码定义：
Java 代码

```
class AnnotationDrivenBeanDefinitionParser implements BeanDefinitionParser {
    // 这里省略了所有的代码
}

class DefaultServletHandlerBeanDefinitionParser implements BeanDefinitionParser {
    // 这里省略了所有的代码
}

class InterceptorsBeanDefinitionParser implements BeanDefinitionParser {
    // 这里省略了所有的代码
}

class ResourcesBeanDefinitionParser implements BeanDefinitionParser {
    // 这里省略了所有的代码
}
```

结论 不允许外部程序对这些系统配置类进行访问，从而杜绝外部程序对 SpringMVC 默认行为的任何修改。

4. 提供自定义扩展接口，却不提供完整覆盖默认行为的方式

这一点，需要深入到SpringMVC的请求处理内部才能够体会得到，我们在这里截取了其中的一段源码加以说明：

Java 代码

```
    private List<HandlerMethodArgumentResolver> getDefaultArgumentResolvers() {
        List<HandlerMethodArgumentResolver> resolvers = new ArrayList<HandlerMethodArgumentResolver>();

        // Annotation-based argument resolution
        resolvers.add(new RequestParamMethodArgumentResolver(getBeanFactory(), false));
        resolvers.add(new RequestParamMapMethodArgumentResolver());
        resolvers.add(new PathVariableMethodArgumentResolver());
        resolvers.add(new ServletModelAttributeMethodProcessor(false));
        resolvers.add(new RequestResponseBodyMethodProcessor(getMessageConverters()));
        resolvers.add(new RequestPartMethodArgumentResolver(getMessageConverters()));
        resolvers.add(new RequestHeaderMethodArgumentResolver(getBeanFactory()));
        resolvers.add(new RequestHeaderMapMethodArgumentResolver());
        resolvers.add(new ServletCookieValueMethodArgumentResolver(getBeanFactory()));
        resolvers.add(new ExpressionValueMethodArgumentResolver(getBeanFactory()));

        // Type-based argument resolution
        resolvers.add(new ServletRequestMethodArgumentResolver());
        resolvers.add(new ServletResponseMethodArgumentResolver());
        resolvers.add(new HttpEntityMethodProcessor(getMessageConverters()));
        resolvers.add(new RedirectAttributesMethodArgumentResolver());
        resolvers.add(new ModelMethodProcessor());
        resolvers.add(new MapMethodProcessor());
        resolvers.add(new ErrorsMethodArgumentResolver());
        resolvers.add(new SessionStatusMethodArgumentResolver());
        resolvers.add(new UriComponentsBuilderMethodArgumentResolver());

        // Custom arguments
        if (getCustomArgumentResolvers() != null) {
            resolvers.addAll(getCustomArgumentResolvers());
        }

        // Catch-all
        resolvers.add(new RequestParamMethodArgumentResolver(getBeanFactory(), true));
        resolvers.add(new ServletModelAttributeMethodProcessor(true));

        return resolvers;
    }
```

这是RequestMappingHandlerAdapter内部的一个重要方法，用以获取所有的参数处理实现类

*（HandlerMethodArgumentResolver）。从源码中，我们可以看到虽然这个方法是一个 private 的方法，但是它在源码中却提 供了 getCustomArgumentResolvers()方法作为切入口，允许用户自行进行扩展。不过我们同样可以发现，**用户自定义的扩展类，只是被插入到整个寻址过程中，并不能通过用户自定义的扩展类来实现对其他 HandlerMethodArgumentResolver 行为的覆盖；也不能改变 HandlerMethodArgumentResolver 的处理顺序。**也就是说：*

结论 SpringMVC 提供的扩展切入点无法改变框架默认的行为方式。

上述这四个方面，都是这一条设计原则在源码级别的佐证。或许有的读者会产生这样的疑虑：这个不能改，那个也不能改，我们对于 SpringMVC 的使用岂不是丧失了很多灵活性？这个疑虑的确存在，但是只说对了一半。因为 SpringMVC 的这一条设计原则说的是：**不能动其根本，只能在一定范围内进行扩展。**

至于说到 SpringMVC 为什么会基于这样一条设计原则，这里面的原因很多。除了之前所提到的编程模型和组件模型的影响，其中更加牵涉到一个编程哲学的取向问题。有关这一点，我们在之后的文章中将陆续展开。

【形散神不散】

这一条编程原则实际上与上一条原则只是在表达方式上有所不同，其表达的核心意思是比较类似的。那么我们如何来定义这里的“形”和“神”呢？

- 神 —— SpringMVC 总是沿着一条固定的逻辑主线运行
- 形 —— SpringMVC 却拥有多种不同的行为模式

SpringMVC 是一个基于组件的开发框架，组件的不同实现体系构成了“形”；组件的逻辑串联构成了“神”。因此，“形散神不散”，实际上是说：

结论 SpringMVC 的逻辑主线始终不变，而行为模式却可以多种多样。

我们在之前有关组件的讨论中，已经见识到了组件的实现体系，也领略了在不同的 SpringMVC 版本中，组件的行为模式的不同。这些已经能够充分证明“形散”的事实。接下来，我们再通过源码来证明一下“神不散”：

```
        try {
            ModelAndView mv;
            boolean errorView = false;

            try {
                processedRequest = checkMultipart(request);

                // Determine handler for the current request.
                mappedHandler = getHandler(processedRequest, false)
                if (mappedHandler == null || mappedHandler.getHandl
                    noHandlerFound(processedRequest, response);
                    return;
                }

                // Process last-modified header, if supported by th
                String method = request.getMethod();
                boolean isGet = "GET".equals(method);
                if (isGet || "HEAD".equals(method)) {
                    long lastModified = ha.getLastModified(request,
                    if (logger.isDebugEnabled()) {
                        String requestUri = urlPathHelper.getReques
                        logger.debug("Last-Modified value for [" +
                    }
                    if (new ServletWebRequest(request, response).ch
                        return;
                    }
                }

                // Apply preHandle methods of registered intercepto
                HandlerInterceptor[] interceptors = mappedHandler.g
                if (interceptors != null) {
                    for (int i = 0; i < interceptors.length; i++) {
                        HandlerInterceptor interceptor = intercepto
                        if (!interceptor.preHandle(processedRequest
                            triggerAfterCompletion(mappedHandler, i
                            return;
                        }
                        interceptorIndex = i;
                    }
                }

                // Actually invoke the handler.
                HandlerAdapter ha = getHandlerAdapter(mappedHandler
                mv = ha.handle(processedRequest, response, mappedHa

                // Do we need view name translation?
                if (mv != null && !mv.hasView()) {
                    mv.setViewName(getDefaultViewName(request));
                }

                // Apply postHandle methods of registered intercept
                if (interceptors != null) {
                    for (int i = interceptors.length - 1; i >= 0; i
                        HandlerInterceptor interceptor = intercepto
                        interceptor.postHandle(processedRequest, re
                    }
                }
            }
            catch (ModelAndViewDefiningException ex) {
                logger.debug("ModelAndViewDefiningException encount
                mv = ex.getModelAndView();
            }
            catch (Exception ex) {
                Object handler = (mappedHandler != null ? mappedHan
                mv = processHandlerException(processedRequest, resp
                errorView = (mv != null);
            }

            // Did the handler return a view to render?
            if (mv != null && !mv.wasCleared()) {
                render(mv, processedRequest, response);
                if (errorView) {
                    WebUtils.clearErrorRequestAttributes(request);
                }
            }
            else {
                if (logger.isDebugEnabled()) {
                    logger.debug("Null ModelAndView returned to Dis
                            "': assuming HandlerAdapter completed r
                }
            }

            // Trigger after-completion for successful outcome.
            triggerAfterCompletion(mappedHandler, interceptorIndex,
        }
        catch (Error err) {
            ServletException ex = new NestedServletException("Handl
            // Trigger after-completion for thrown exception.
            triggerAfterCompletion(mappedHandler, interceptorIndex,
            throw ex;
        }
```

```
        try {
            ModelAndView mv = null;

            try {
                processedRequest = checkMultipart(reque

                // Determine handler for the current re
                mappedHandler = getHandler(processedReq
                if (mappedHandler == null || mappedHand
                    noHandlerFound(processedRequest, re
                    return;
                }

                // Apply preHandle methods of registere
                HandlerInterceptor[] interceptors = map
                if (interceptors != null) {
                    for (int i = 0; i < interceptors.le
                        HandlerInterceptor interceptor =
                        if (!interceptor.preHandle(proc
                            triggerAfterCompletion(mapp
                            return;
                        }
                        interceptorIndex = i;
                    }
                }

                // Actually invoke the handler.
                HandlerAdapter ha = getHandlerAdapter(m
                mv = ha.handle(processedRequest, respon

                // Apply postHandle methods of register
                if (interceptors != null) {
                    for (int i = interceptors.length -
                        HandlerInterceptor interceptor =
                        interceptor.postHandle(processe
                    }
                }
            }
            catch (ModelAndViewDefiningException ex) {
                logger.debug("ModelAndViewDefiningExcep
                mv = ex.getModelAndView();
            }
            catch (Exception ex) {
                Object handler = (mappedHandler != null
                mv = processHandlerException(processedR

            }

            // Did the handler return a view to render?
            if (mv != null && !mv.wasCleared()) {
                render(mv, processedRequest, response);

            }
            else {
                if (logger.isDebugEnabled()) {
                    logger.debug("Null ModelAndView ret
                            "': assuming HandlerAdapter
                }
            }

            // Trigger after-completion for successful
            triggerAfterCompletion(mappedHandler, inter
        }
        catch (Error err) {
            ServletException ex = new NestedServletExce
            // Trigger after-completion for thrown exce
            triggerAfterCompletion(mappedHandler, inter
            throw ex;
        }
```

图中的代码是 DispatcherServlet 中的核心方法 doDispatch，我们这里使用了比较工具将 Spring3.1 中的实现代码和 Spring2.0.8 中的实现代码做了比较，其中的区别之处比较工具使用了不同的颜色标注了出来。

我们可以很明显地看到，虽然 Spring2.0 到 Spring3.1 之间，SpringMVC 的行为方式已经有了翻天覆地的变化，然而整个 DispatcherServlet 的核心处理主线却并没有很大的变化。这种稳定性，恰巧证明了整个 SpringMVC 的体系结构设计的精妙之处。

【简化、简化、还是简化】

在 Spring2.5 之前的 SpringMVC 版本并没有很强的生命力，因为它只是通过组件将整个 MVC 的概念加以诠释，从开发流程的简易度来看 并没有很明显的提升。有关 SpringMVC 发展的里程碑，我们将在下一篇文章中重点讲述。我们在这里想要谈到的 SpringMVC 的另外一大设计原则， 实际上主要是从 Spring2.5 这个版本之后才不断显现出来的。这条设计原则可以用 2 个字来概括：**简化**。

这里说的简化，其实包含的内容非常广泛。笔者在这里挑选了两个比较重要的方面来进行说明：

- Annotation —— 简化各类配置定义
- Schema Based XML —— 简化组件定义

先谈谈 Annotation。Annotation 是 JDK5.0 带来的一种全新的 Java 语法。这种语法的设计初衷众说纷纭，并没有一个标准的答案。笔者在这里给出一个个人观点以供参考：

结论 Annotation 的原型是注释。作为一种对注释的扩展而被引入成为一个语法要素，其本身就是为了对所标注的编程元素进行补充说明，从而进一步完善编程元素的逻辑语义。

从这个结论中，我们可以看到一层潜在的意思：**在 Annotation 出现之前，Java 自身语法所定义的编程元素已经不足以表达足够多的信息或者逻辑语义**。 在这种情况下，过去经常使用的方法是引入新的编程元素（例如使用最多的就是 XML 形式的结构化配置文件）来对 Java 程序进行补充说明。而在 Annotation 出现之后，可以在一定程度上有效解决这一问题。因此 Annotation 在很长一段时间都被当作是 XML 配置文件的替代品。

这也就是 Annotation 经常被用来和 XML 进行比较的原因。孰优孰劣其实还是要视具体情况而定，并没有什么标准答案。不过我们在这里想强调 的是 Annotation 在整个 SpringMVC 中所起到的作用，并非仅仅是代替 XML 那么简单。我们归纳了有三个不同的方面：

1. 简化请求映射的定义

在 Spring2.5 之前，所有的 Http 请求与 Controller 核心处理器之间的映射关系都是在 XML 文件中定义的。作为 XML 配 置文件的有效替代品，Annotation 接过了定义映射关系的重任。

我们可以将@RequestMapping 加在 Controller 的 class- level 和 method-level 进行 Http 请求的抽象。

2. 消除 Controller 对接口的依赖

在 Spring2.5 之前，SpringMVC 规定所有的 Controller 都必须实现 Controller 接口：

Java 代码

```
public interface Controller {

    /**
     * Process the request and return a ModelAndView object which the DispatcherServlet
     * will render. A <code>null</code> return value is not an error: It indicates that
     * this object completed request processing itself, thus there is no ModelAndView
     * to render.
     * @param request current HTTP request
     * @param response current HTTP response
     * @return a ModelAndView to render, or <code>null</code> if handled directly
     * @throws Exception in case of errors
     */
    ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)
throws Exception;

}
```

也就是说，应用程序不得不严重依赖于接口所规定的处理模式。而我们看到 Controller 接口除了对处理接口的返回值做了一次封装以外，我们依然需要面对原生的 HttpServletRequest 和 HttpServletResponse 对象进行操作。

而在 Spring2.5 之后，我们可以通过@Controller 来指定 SpringMVC 可识别的 Controller，彻底消除了对接口的依赖：

Java 代码

```
@Controller
public class UserController {
        // 这里省略了许多代码
}
```

3. 成为框架进行逻辑处理的标识

之前已经谈到，Annotation 主要被用于对编程元素进行补充说明。因而 Spring 就利用这一特性，使得那些被加入了特殊 Annotation 的编程元素可以得到特殊的处理。例如，SpringMVC 引入的@SessionAttribute、@RequestBody、 @ModelAttribute 等等，可以说既是对 Controller 的一种逻辑声明，也成为了框架本身对相关元素进行处理的一个标识符。

再谈谈 Schema Based XML。Schema Based XML 并不是一个陌生的概念，早在 Spring2.0 时代

就被用于进行 XML 配置的简化。SpringMVC 在进入到 3.0 版本之后，正式将其引入并作为 SpringMVC 组件定义的一个重要手段。

在 XML 中引入 Schema，只需要在 XML 文件的开头加入相关的定义。例如：
Xml 代码

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:mvc="http://www.springframework.org/schema/mvc"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context-3.1.xsd
            http://www.springframework.org/schema/mvc
            http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd">


</beans>
```

而 Schema 的具体处理，则位于 Spring 的 JAR 中的/META-INF/spring.handlers 文件中进行定义：

Xml 代码

```
http\://www.springframework.org/schema/mvc=org.springframework.web.servlet.config.MvcNamespaceHandler
```

我们会在之后的讨论中详细分析 MvcNamespaceHandler 的源码。不过我们可以明确的是，在我们使用 Schema Based XML 的同时，有许多 SpringMVC 的内置对象会被预先定义成为组件，我们的配置将是对这些预先定义好的组件的一个二次配置的过程。可以想象，二次配置一定会比较省力，因为它至少省去了很多内置对象的定义过程。这也就是 Schema Based XML 带来的简化效果了。

小结

本文从逻辑上讲，可以分成三个部分：

- **SpringMVC 的构成要素 —— 是什么 —— 阐述框架的主体结构**
- **SpringMVC 的发展历程 —— 为什么 —— 阐述框架各要素产生的内因**
- **SpringMVC 的设计原则 —— 怎么样 —— 阐述框架的共性思想**

“是什么”是框架最根本的问题。我们从 SpringMVC 的三要素入手，帮助大家分析构成 SpringMVC 的基本元素主要是为了让读者对整个 SpringMVC 的架构有一个宏观的认识。在之

后的分析中，我们研究的主体内容也将始终围绕着这些 SpringMVC 的构成要素，并进行逐一分析。

“为什么”是框架的存在基础。我们可以看到，整个 SpringMVC 的发展历程是一个对于开发模式不断进行优化的过程，也是不断解决 Web 开发中所面临的一个又一个问题的过程。之前我们也曾经提到过一个重要观点：**任何框架无所谓好与坏、优与劣，它们只是在不同的领域解决问题的方式不同。**所以，我们分析这些 SpringMVC 基本构成要素产生的原因实际上也是对整个 Web 开发进行重新思考的过程。

“怎么样”是一种深层次的需求。对于 SpringMVC 而言，了解其基本构成和用法并不是一件难事，但是要从中提炼并总结出一些共性的东西就需要我们能够站在一个更高的高度来进行分析。也只有了解了这些共性的东西，我们才能进一步总结出使用框架的最佳实践。

读到这里，希望读者能够回味一下本文的写作思路，并且能够举一反三将这种思考问题的方式运用到其他一些框架的学习中去。这样，本文的目的也就达到了。