在上一篇文章中，我们从 DispatcherServlet 谈起，最终为读者详细分析了 SpringMVC 的初始化主线的全部过程。整个初始化主线的研 究，其实始终围绕着 DispatcherServlet、WebApplicationContext 和组件这三大元素之间的关系展开。

在文章写完之后，也陆续收到了一些反馈，其中比较集中的问题，是有关 WebApplicationContext 对组件进行初始化的过程交代的不够清楚。所以，本文作为上一篇文章的续文，就试图来讲清楚这个话题。

SpringMVC 的核心配置文件

SpringMVC 的核心配置文件，我们从整个专栏的第一篇文章就开始接触。所以，我们在这里首先对 SpringMVC 的核心配置文件做一些概括性的回顾。

结论 SpringMVC 的核心配置文件是构成 SpringMVC 应用程序的必要元素之一。

这是我们在讲有关 SpringMVC 的构成要素时就曾经提到过的一个重要结论。当时我们所说的另外两大必要元素就是 DispatcherServlet 和 Controller。因而，SpringMVC 的核心配置文件在整个应用程序中所起到的作用也是举足轻重的。这也就是我们在这里需要补充对这个文件进行 详细分析的原因。

结论 SpringMVC 的核心配置文件与传统的 Spring Framework 的配置文件是一脉相承的。

这个结论很容易理解。作为 Spring Framework 的一部分，我们可以认为 SpringMVC 是整个 Spring Framework 的一个组件。因而两者的配置体系和管理体系完全相同也属情理之中。实际上，SpringMVC 所采取的策略，就是借用 Spring Framework 强大的容器（ApplicationContext）功能，而绝非自行实现。

结论 SpringMVC 的核心配置文件是架起 DispatcherServlet 与 WebApplicationContext 之间的桥梁。

我们在 web.xml 中指定 SpringMVC 的入口程序 DispatcherServlet 时，实际上蕴含了一个对核心配置文件的指定过程（[servlet-name]-servlet.xml）。当然，我们也可以明确指定这个核心配置文件的位置。这些配置选项，我们已经在上一篇文章中详 细介绍过，这里不再重复。

而上面这一结论，除了说明两者之间的配置关系之外，还包含了一层运行关系：**DispatcherServlet 负责对 WebApplicationContext 进行初始化，而初始化的依据，就是这个 SpringMVC 的核心配置文件。**所以，SpringMVC 的核心配置文件的内容解读将揭开整个 SpringMVC 初始化主线的全部秘密。

如果我们把这个结论与上一个结论结合起来来看，也正因为 SpringMVC 的核心配置文件使用了与 Spring Framework 相同的格式，才使其成为 DispatcherServlet 驾驭 Spring 的窗口。

结论 SpringMVC 的核心配置文件是 SpringMVC 中所有组件的定义窗口，通过它我们可以指定整个 SpringMVC 的行为方式。

这个结论告诉了我们 SpringMVC 核心配置文件在整个框架中的作用。组件行为模式的多样化，决定了我们必须借助一个容器（WebApplicationContext）来进行统一的管理。而 SpringMVC 的核心配置文件，就是我们进行组件管理的窗口。

核心配置文件概览

说了那么多有关 SpringMVC 核心配置文件的结论，我们不妨来看一下这个配置文件的概况：

Xml 代码

```
<?xml version="1.0" encoding="UTF-8"?>
<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">

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

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

    <!-- Handles HTTP GET requests for /static/** by efficiently serving up static resources in the ${webappRoot}/static/ directory -->
    <mvc:resources mapping="/static/**" location="/static/" />

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

</beans>
```

这是一个非常典型的 SpringMVC 核心配置文件。虽然我们在这里几乎对每一段重要的配置都做了注释，不过可能对于毫无 SpringMVC 开发经验的读者来说，这段配置基本上还无法阅读。所以接下来，我们就试图对这个文件中的一些细节加以说明。

【头部声明】

配置文件中首先进入我们眼帘的是它的头部的一大段声明：

Xml 代码

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
```

这个部分是整个 SpringMVC 核心配置文件的关键所在。这一段声明，被称之为 **Schema-based XML 的声明**部分。有关 Schema-based XML 的概念，读者可以参考 Spring 官方的 reference：

Appendix C. XML Schema-based configuration

Appendix D. Extensible XML authoring

为了帮助读者快速理解，我们稍后会专门开辟章节针对 Schema-based XML 的来龙去脉进行讲解。

【组件定义】

除了头部声明部分的其他配置部分，就是真正的组件定义部分。在这个部分中，我们可以看到两种不同类型的配置定义模式：

1. 基于 Schema-based XML 的配置定义模式

Xml 代码

```
<mvc:annotation-driven />
```

2. 基于 Traditional XML 的配置定义模式

Xml 代码

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

两种不同的组件定义模式，其目的是统一的：**对 SpringMVC 中的组件进行声明，指定组件的行为方式。**

虽然两种不同的组件定义模式的外在表现看上去有所不同，但是 SpringMVC 在对其进行解析时最终都会将其转化为组件的定义而加载到 WebApplicationContext 之中进行管理。所以我们需要理解的是蕴藏在配置背后的目的而非配置本身的形式。

至于这两种不同的配置形式之间的关系，我们稍后会在 Schema-based XML 的讲解中详细展开。

Schema-based XML

【基本概念】

Schema-based XML 本身并不是 SpringMVC 或者 Spring Framework 独创的一种配置模式。我们可以看看 W3C 对于其用途的一个大概解释：

W3C 写道

The purpose of an XSD schema is to define and describe a class of XML documents by using schema components to constrain and document the meaning, usage and relationships of their constituent parts: datatypes, elements and their content and attributes and their values. Schemas can also provide for the specification of additional document information, such as normalization and defaulting of attribute and element values. Schemas have facilities for self-documentation. Thus, XML Schema Definition Language: Structures can be used to define, describe and catalogue XML vocabularies for classes of XML documents.

这个解释稍微有点抽象。所以我们可以来看看 Spring 官方 reference 对于引入 Schema-based XML 的说法：

Spring Reference 写道

The central motivation for moving to XML Schema based configuration files was to make Spring XML configuration easier. The 'classic' <bean/>-based approach is good, but its generic-nature comes with a price in terms of configuration overhead.

也就是说，我们引入 Schema-based XML 是为了对 Traditional 的 XML 配置形式进行简化。**通过 Schema 的定义，把一些原本需要通过几个 bean 的定义或者复杂的 bean 的组合定义的配置形式，用另外一种简单而可读的配置形式呈现出来。**

所以，我们也可以由此得出一些有用的推论：

downpour 写道

Schema-based XML 可以代替 Traditional 的 XML 配置形式，在 Spring 容器中进行组件的定义。

这里的代替一词非常重要，这就意味着传统的 XML 配置形式在这里会被颠覆，我们在对 Schema-based XML 进行解读时，需要使用一种全新的语义规范来理解。

Schema-based XML 可以极大改善配置文件的可读性并且缩小配置文件的规模。

这是从引入 Schema-based XML 的目的反过来得出的推论。因为如果引入 Schema-based XML 之后，整个配置变得更加复杂，那么 Schema-based XML 的引入也就失去了意义。

同时，笔者在这里需要特别强调的是 Schema-based XML 的引入，实际上是把原本静态的配置动态化、过程化。有关这一点，我们稍后会有说明。

【引入目的】

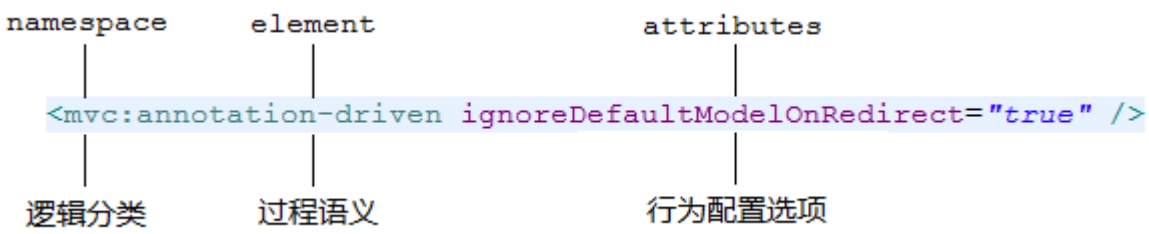
在早期的 Spring 版本中，只有 Traditional XML 一种组件定义模式。当时，XML 作为 Java 最好的朋友，自然而然在整个框架中起到了举足轻重的作用。根据 Spring 的设计原则，所有纳入 WebApplicationContext 中管理的对象，都被映射为 XML 中的一个<bean>节点，通过对于<bean>节 点的一个完整描述，我们可以有效地将整个应用程序中所有的对象都纳入到一个统一的容器中进行管理。

这种统一化的描述，带来的是管理上的便利，不过同时也带来了逻辑上的困扰。因为统一的节点，降低了配置的难度，我们几乎只需要 将<bean>节点与 Java 的对象模型对应起来即可。（有一定经验的 Spring 程序员可以回忆一下，我们在编写 Spring 配置文件时， 是否也是一个将配置选项与 Java 对象中属性或者方法对应起来的过程）但是这样的配置形式本身并不具备逻辑语义，也就是说我们无法非常直观地看出某一个特 定的<bean>定义，从逻辑上它到底想说明什么问题？

这也就是后来 Schema-based XML 开始被引入并流行开来的重要原因。从形式上看，Schema-based XML 相比较 Traditional XML 至少有三个方面的优势：

- namespace —— 拥有很明确的逻辑分类
- element —— 拥有很明确的过程语义
- attributes —— 拥有很简明的配置选项

这三方面的优势，我们可以用一幅图来进行说明：



在图中，我们分别用上下两层来说明某一个配置节点的结构名称以及它们的具体作用。由此可见，**Schema-based XML 中的配置节点拥有比较鲜明的功能特性，通过 namespace、element 和 attributes 这三大元素之间的配合，共同完成对一个动态过程的描述。**

例如，<mvc:annotation-driven />这段配置想要表达的意思，就是在 mvc 的空间内实现 Annotation 驱动的配置方式。其中，mvc 表示配置的有效范 围，annotation-driven 则表达了一个动态的过程，实际的逻辑含义是：整个 SpringMVC 的实现是基于 Annotation 模式，请为 我注册相关的行为模式。

这种配置方式下，可读性大大提高：我们无需再去理解其中的实现细节。同时，配置的简易性也大大提高：我们甚至不用去关心哪些 bean 被定义了。

所以总体来说，Schema-based XML 的引入，对于配置的简化是一个极大的进步。

【构成要素】

在 Spring 中，一个 Schema-based XML 有两大构成要素：**过程实现**和**配置定义**。

先谈谈过程实现。所谓过程实现，其实就是我们刚才所举的那个例子中，实现实际背后逻辑的过程。这个过程由两个 Java 接口来进行表述：

- NamespaceHandler —— 对 Schema 定义中 namespace 的逻辑处理接口
- BeanDefinitionParser —— 对 Schema 定义中 element 的逻辑处理接口

很显然，NamespaceHandler 是入口程序，它包含了所有的属于该 namespace 定义下所有 element 的处理调 用，所以 BeanDefinitionParser 的实现就成为了 NamespaceHandler 的调用对象了。这一点，我们可以通过 NamesapceHandler 的 MVC 实现类来加以证明：

Java 代码

```
public void init() {
    registerBeanDefinitionParser("annotation-driven", new AnnotationDrivenBeanDefinitionParser());
    registerBeanDefinitionParser("default-servlet-handler", new DefaultServletHandlerBeanDefinitionParser());
    registerBeanDefinitionParser("interceptors", new InterceptorsBeanDefinitionParser());
    registerBeanDefinitionParser("resources", new ResourcesBeanDefinitionParser());
    registerBeanDefinitionParser("view-controller", new ViewControllerBeanDefinitionParser());
}
```

我们可以看到，MvcNamespaceHandler 的执行，只不过依次调用了不同的 BeanDefinitionParser 的实现类而已，而每一个 BeanDefinitionParser 的实现，则对应于 Schema 定义中的 element 逻辑处理。例 如，AnnotationDrivenBeanDefinitionParser 对应于：<mvc:annotation-driven />这个 element 实现；ResourcesBeanDefinitionParser 则对应于<mvc:resources />的实现等等。

所以，要具体了解每个 element 的行为过程，只要研究每一个 BeanDefinitionParser 的实现类即可。我们以整个 MVC 空间中 最重要的一个节点<mvc:annotation-driven />为例，对 AnnotationDrivenBeanDefinitionParser 进行说明，其源码如下：

Java 代码

```
public BeanDefinition parse(Element element, ParserContext parserContext) {
    Object source = parserContext.extractSource(element);

    CompositeComponentDefinition compDefinition = new CompositeComponentDefinition(element.getTagName(), source);
    parserContext.pushContainingComponent(compDefinition);

    RootBeanDefinition methodMappingDef = new RootBeanDefinition(RequestMappingHandlerMapping.class);
    methodMappingDef.setSource(source);
    methodMappingDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
    methodMappingDef.getPropertyValues().add("order", 0);
    String methodMappingName = parserContext.getReaderContext().registerWithGeneratedName(methodMappingDef);

    RuntimeBeanReference conversionService = getConversionService(element, source, parserContext);
    RuntimeBeanReference validator = getValidator(element, source, parserContext);
    RuntimeBeanReference messageCodesResolver = getMessageCodesResolver(element, source, parserContext);

    RootBeanDefinition bindingDef = new RootBeanDefinition(ConfigurableWebBindingInitializer.class);
    bindingDef.setSource(source);
    bindingDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
    bindingDef.getPropertyValues().add("conversionService", conversionService);
    bindingDef.getPropertyValues().add("validator", validator);
    bindingDef.getPropertyValues().add("messageCodesResolver", messageCodesResolver);

    ManagedList<?> messageConverters = getMessageConverters(element, source, parserContext);
    ManagedList<?> argumentResolvers = getArgumentResolvers(element, source, parserContext);
    ManagedList<?> returnValueHandlers = getReturnValueHandlers(element, source, parserContext);
```

```
        RootBeanDefinition methodAdapterDef = new RootBeanDefinition(RequestMappingHandlerAdapter.class);
        methodAdapterDef.setSource(source);
        methodAdapterDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
        methodAdapterDef.getPropertyValues().add("webBindingInitializer", bindingDef);
        methodAdapterDef.getPropertyValues().add("messageConverters", messageConverters);
        if (element.hasAttribute("ignoreDefaultModelOnRedirect")) {
            Boolean ignoreDefaultModel = Boolean.valueOf(element.getAttribute("ignoreDefaultModelOnRedirect"));

methodAdapterDef.getPropertyValues().add("ignoreDefaultModelOnRedirect", ignoreDefaultModel);
        }
        if (argumentResolvers != null) {
            methodAdapterDef.getPropertyValues().add("customArgumentResolvers", argumentResolvers);
        }
        if (returnValueHandlers != null) {
                methodAdapterDef.getPropertyValues().add("customReturnValueHandlers", returnValueHandlers);
        }
        String methodAdapterName = parserContext.getReaderContext().registerWithGeneratedName(methodAdapterDef);

        RootBeanDefinition csInterceptorDef = new RootBeanDefinition(ConversionServiceExposingInterceptor.class);
        csInterceptorDef.setSource(source);
        csInterceptorDef.getConstructorArgumentValues().addIndexedArgumentValue(0, conversionService);
        RootBeanDefinition mappedCsInterceptorDef = new RootBeanDefinition(MappedInterceptor.class);
        mappedCsInterceptorDef.setSource(source);
            mappedCsInterceptorDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
            mappedCsInterceptorDef.getConstructorArgumentValues().addIndexedArgumentValue(0, (Object) null);
            mappedCsInterceptorDef.getConstructorArgumentValues().addIndexedArgumentValue(1, csInterceptorDef);
        String                                                mappedInterceptorName                                                =
parserContext.getReaderContext().registerWithGeneratedName(mappedCsInterceptorDef);

        RootBeanDefinition methodExceptionResolver = new RootBeanDefinition(ExceptionHandlerExceptionResolver.class);
        methodExceptionResolver.setSource(source);
            methodExceptionResolver.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
            methodExceptionResolver.getPropertyValues().add("messageConverters", messageConverters);
            methodExceptionResolver.getPropertyValues().add("order", 0);
        String methodExceptionResolverName =
parserContext.getReaderContext().registerWithGeneratedName(methodExceptionResolver);

        RootBeanDefinition responseStatusExceptionResolver = new RootBeanDefinition(ResponseStatusExceptionResolver.class);
        responseStatusExceptionResolver.setSource(source);
            responseStatusExceptionResolver.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
            responseStatusExceptionResolver.getPropertyValues().add("order", 1);
        String responseStatusExceptionResolverName =
                        parserContext.getReaderContext().registerWithGeneratedName(responseStatusExceptionResolver);

        RootBeanDefinition defaultExceptionResolver = new RootBeanDefinition(DefaultHandlerExceptionResolver.class);
        defaultExceptionResolver.setSource(source);
            defaultExceptionResolver.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
            defaultExceptionResolver.getPropertyValues().add("order", 2);
        String defaultExceptionResolverName =
                        parserContext.getReaderContext().registerWithGeneratedName(defaultExceptionResolver);

        parserContext.registerComponent(new BeanComponentDefinition(methodMappingDef, methodMappingName));
        parserContext.registerComponent(new BeanComponentDefinition(methodAdapterDef, methodAdapterName));
        parserContext.registerComponent(new                                BeanComponentDefinition(methodExceptionResolver,
methodExceptionResolverName));
        parserContext.registerComponent(new                              BeanComponentDefinition(responseStatusExceptionResolver,
responseStatusExceptionResolverName));
```

```
        parserContext.registerComponent(new                    BeanComponentDefinition(defaultExceptionResolver,
defaultExceptionResolverName));
        parserContext.registerComponent(new BeanComponentDefinition(mappedCsInterceptorDef, mappedInterceptorName));

        // Ensure BeanNameUrlHandlerMapping (SPR-8289) and default HandlerAdapters are not "turned off"
            MvcNamespaceUtils.registerDefaultComponents(parserContext, source);

        parserContext.popAndRegisterContainingComponent();

        return null;
}
```

整个过程看上去稍显凌乱，不过我们发现其中围绕的一条主线就是：**使用编程的方式来对 bean 进行注册。**也就是说，<mvc:annotation-driven />这样一句配置，顶上了我们如此多的 bean 定义。难怪 Schema-based XML 被誉为是简化 XML 配置的绝佳帮手了。

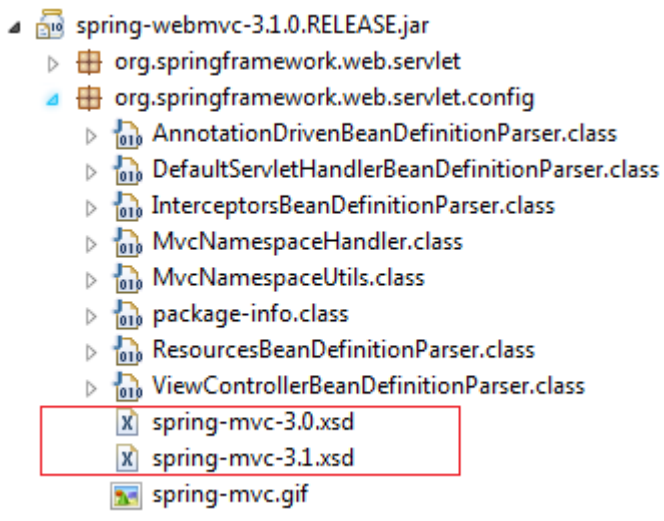
有了过程实现，我们再来谈谈配置定义。配置定义的目的非常简单，就是通过一些配置文件，将上述的过程实现类串联起来，从而完成整个 Schema-based XML 的定义。

整个配置定义，也分为两个部分：

- Schema 定义 —— 一个 xsd 文件，描述整个 Schema 空间中 element 和 attribute 的定义
- 注册配置文件 —— 由 META-INF/spring.handlers 和 META-INF/spring.schemas 构成，用以注册 Schema 和 Handler

Schema 定义是由一个 xsd 文件完成的。这个文件在 Spring 发布的时候同时发布在网络上。例如 SpringMVC 的 Schema 定义，就发布在这个地址：
http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd

同时，这个地址在 Spring 的发布包中也存有一个备份。这个备份位于 SpringMVC 的分发包 spring-webmvc 的 JAR 包之中。



这样做的好处在于，我们在对 Schema 进行引用时，可以通过本地寻址来加快加载速度。

注：如果我们回顾一下之前的 核心配置文件中的头部声明部分。其中的 xsi:schemaLocation 声明就是用于指定映射于本地的 XSD 文件。所以 xsi:schemaLocation 的定义不是必须的，不过声明它能够使 Spring 自动查找本地的缓存来进行 schema 的寻址。

我们在这里不对 XSD 文件做过多的内容分析，因为其中不外乎是对 element 的定义、attributes 的定义等等。这些内容是我们进行 Schema-based XML 配置的核心基础。

配置定义的另外一个元素构成是 META-INF/spring.handlers 和 META-INF/spring.schemas 这两个文件。 它们同样位于 SpringMVC 的分发包下。当我们在 XML 的头部声明中引用了相关的 Schema 定义之后，Spring 会自动查找 spring.schemas 和 spring.handlers 的定义，根据其中指定的 NamespaceHandler 实现类加载执行。

有关这个过程，我们在之后的日志分析中还会涉及。

初始化日志的再分析

有了 Schema Based XML 的相关知识，就可以对 DispatcherServlet 的初始化启动日志做进一步的详细分析。而这次的分析，我们试图弄清楚以下问题：

- Where —— 组件的声明在哪里？
- How —— 组件是如何被注册的？
- What —— 究竟哪些组件被注册了？

对于这三个问题的研究，我们需要结合日志和 Schema based XML 的运行机理来共同进行分析。

引用

```
[main] INFO /sample - Initializing Spring FrameworkServlet 'dispatcher'
19:49:48,670    INFO XmlWebApplicationContext:495 - Refreshing WebApplicationContext for namespace 'dispatcher-servlet': startup date [Thu Feb 16 19:49:48 CST 2012]; parent: Root WebApplicationContext
19:49:48,674    INFO XmlBeanDefinitionReader:315 - Loading XML bean definitions from class path resource [web/applicationContext-dispatcher.xml]

## Schema 定位和加载 (开始) ##

19:49:48,676 DEBUG DefaultDocumentLoader:72 - Using JAXP provider [com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderFactoryImpl]
19:49:48,678 DEBUG PluggableSchemaResolver:140 - Loading schema mappings from [META-INF/spring.schemas]
19:49:48,690 DEBUG PluggableSchemaResolver:118 - Found XML schema [http://www.springframework.org/schema/beans/spring-beans-3.1.xsd] in classpath: org/springframework/beans/factory/xml/spring-beans-3.1.xsd
19:49:48,710 DEBUG PluggableSchemaResolver:118 - Found XML schema [http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd] in classpath: org/springframework/web/servlet/config/spring-mvc-3.1.xsd
19:49:48,715 DEBUG PluggableSchemaResolver:118 - Found XML schema [http://www.springframework.org/schema/tool/spring-tool-3.1.xsd] in classpath: org/springframework/beans/factory/xml/spring-tool-3.1.xsd
19:49:48,722 DEBUG PluggableSchemaResolver:118 - Found XML schema [http://www.springframework.org/schema/context/spring-context-3.1.xsd] in classpath: org/springframework/context/config/spring-context-3.1.xsd

## Schema 定位和加载 (结束) ##

## NamespaceHandler 执行阶段 (开始) ##

19:49:48,731 DEBUG DefaultBeanDefinitionDocumentReader:108 - Loading bean definitions
19:49:48,742 DEBUG DefaultNamespaceHandlerResolver:156 - Loaded NamespaceHandler mappings: {...}

19:49:48,886 DEBUG PathMatchingResourcePatternResolver:550 - Looking for matching resources in directory tree [D:\Work\Demo2do\Sample\target\classes\com\demo2do\sample\web\controller]


19:49:48,896 DEBUG XmlBeanDefinitionReader:216 - Loaded 18 bean definitions from location pattern [classpath:web/applicationContext-dispatcher.xml]
19:49:48,897 DEBUG XmlWebApplicationContext:525 - Bean factory for WebApplicationContext for
```

namespace 'dispatcher-servlet':
org.springframework.beans.factory.support.DefaultListableBeanFactory@495c998a: defining beans [[
1. org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping#0,
2. org.springframework.format.support.FormattingConversionServiceFactoryBean#0,
3. org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter#0,
4. org.springframework.web.servlet.handler.MappedInterceptor#0,
5. org.springframework.web.servlet.mvc.method.annotation.ExceptionHandlerExceptionResolver#0,
6. org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver#0,
7. org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver#0,
8. org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,
9. org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,
10. org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,
11. blogController,
12. userController,
13. org.springframework.context.annotation.internalConfigurationAnnotationProcessor,
14. org.springframework.context.annotation.internalAutowiredAnnotationProcessor,
15. org.springframework.context.annotation.internalRequiredAnnotationProcessor,
16. org.springframework.context.annotation.internalCommonAnnotationProcessor,
17. org.springframework.web.servlet.resource.ResourceHttpRequestHandler#0,
18. org.springframework.web.servlet.handler.SimpleUrlHandlerMapping#0
]; parent: org.springframework.beans.factory.support.DefaultListableBeanFactory@6602e323
19:49:48,949 DEBUG XmlWebApplicationContext:794 - Unable to locate MessageSource with name 'messageSource': using default
[org.springframework.context.support.DelegatingMessageSource@4b2922f6]
19:49:48,949 DEBUG XmlWebApplicationContext:818 - Unable to locate ApplicationEventMulticaster with name 'applicationEventMulticaster': using default
[org.springframework.context.event.SimpleApplicationEventMulticaster@79b66b06]
19:49:48,949 DEBUG UiApplicationContextUtils:85 - Unable to locate ThemeSource with name 'themeSource': using default
[org.springframework.ui.context.support.DelegatingThemeSource@372c9557]
19:49:49,154 DEBUG RequestMappingHandlerMapping:98 - Looking for request mappings in application context: WebApplicationContext for namespace 'dispatcher-servlet': startup date [Thu Feb 16 19:49:48 CST 2012]; parent: Root WebApplicationContext
19:49:49,175 INFO RequestMappingHandlerMapping:188 - Mapped "{[/blog],methods=[],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public org.springframework.web.servlet.ModelAndView com.demo2do.sample.web.controller.BlogController.index()
19:49:49,177 INFO RequestMappingHandlerMapping:188 - Mapped "{[/register],methods=[],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public org.springframework.web.servlet.ModelAndView com.demo2do.sample.web.controller.UserController.register(com.demo2do.sample.entity.User)
19:49:49,180 INFO RequestMappingHandlerMapping:188 - Mapped "{[/login],methods=[],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public org.springframework.web.servlet.ModelAndView com.demo2do.sample.web.controller.UserController.login(java.lang.String,java.lang.String)
19:49:49,632 DEBUG BeanNameUrlHandlerMapping:71 - Looking for URL mappings in application context: WebApplicationContext for namespace 'dispatcher-servlet': startup date [Thu Feb 16 19:49:48 CST 2012]; parent: Root WebApplicationContext
19:49:49,924 INFO SimpleUrlHandlerMapping:314 - Mapped URL path [/static/**] onto handler 'org.springframework.web.servlet.resource.ResourceHttpRequestHandler#0'

NamespaceHandler 执行阶段（结束）##

19:49:49,956 DEBUG DispatcherServlet:627 - Unable to locate RequestToViewNameTranslator with

```
name 'viewNameTranslator': using default
[org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator@4d16318b]
19:49:49,980 DEBUG DispatcherServlet:667 - No ViewResolvers found in servlet 'dispatcher': using
default
19:49:49,986 DEBUG DispatcherServlet:689 - Unable to locate FlashMapManager with name
'flashMapManager': using default
[org.springframework.web.servlet.support.DefaultFlashMapManager@1816daa9]
19:49:49,986 DEBUG DispatcherServlet:523 - Published WebApplicationContext of servlet
'dispatcher' as ServletContext attribute with name
[org.springframework.web.servlet.FrameworkServlet.CONTEXT.dispatcher]
19:49:49,986   INFO DispatcherServlet:463 - FrameworkServlet 'dispatcher': initialization
completed in 1320 ms
19:49:49,987 DEBUG DispatcherServlet:136 - Servlet 'dispatcher' configured successfully
```

在上面的启动日志中，笔者还是把不同的日志功能使用不同的颜色进行了区分。这里进行逐一分析：

1. 黑色加粗标记区域 —— 容器的启动和结束标志

这个部分的日志比较明显，位于容器的启动阶段和结束阶段，在之前的讨论中我们已经分析过，这里不再重复。

2. 黄色注释段落 —— Schema 定位和加载

这个部分的日志反应出刚才我们所分析的 Schema-based XML 的工作原理。这是其中的第一步：读取 META-INF/spring.schemas 的内容，加载 schema 定义。然后找到相应的 NamespaceHandler，执行其实现类。

3. 蓝色注释部分 —— NamespaceHandler 执行阶段

*这个部分的日志，可以帮助我们回答本节一开始所提出的两个问题。**绝大多数的组件，都是在 BeanDefinitionParser 的实现类中使用编程的方式注册的。***

4. 红色标记区域 —— 组件注册细节

这个部分的日志区域彻底回答了本节一开始所提出的最后一个问题：一共有 18 个组件被注册，就是红色标记的那 18 个 bean。

小结

本文所涉及到的话题，主要围绕着 SpringMVC 的核心配置问题展开。读者可以将本文作为上一篇文章的续篇，将两者结合起来阅读。因为从宏观上说，本文的话题实际上也属于初始化主线的一个部分。