在上一篇文章中，我们给出了构成 SpringMVC 应用程序的三要素以及三要素的设计过程。让我们来归纳一下整个设计过程中的一些要点：

- **SpringMVC 将 Http 处理流程抽象为一个又一个处理单元**
- **SpringMVC 定义了一系列组件（接口）与所有的处理单元对应起来**
- **SpringMVC 由 DispatcherServlet 贯穿始终，并将所有的组件串联起来**

在整个过程中，组件和 DispatcherServlet 总是维持着一个相互支撑的关系：

- DispatcherServlet —— **串联起整个逻辑主线，是整个框架的心脏**
- 组件 —— **逻辑处理单元的程序化表示，起到承上启下的作用，是 SpringMVC 行为模式的实际承载者**

在本系列接下来的两篇文章中，我们将分别讨论 DispatcherServlet 和组件的相关内容。本文讨论 DispatcherServlet，而下一篇则重点分析组件。

有关 DispatcherServlet，我们想从构成 DispatcherServlet 的体系结构入手，再根据不同的逻辑主线分别加以分析，希望能够帮助读者整理出学习 SpringMVC 核心类的思路。

DispatcherServlet 的体系结构

通过不同的角度来观察 DispatcherServlet 会得到不同的结论。我们在这里选取了三个不同的角度：运行特性、继承结构和数据结构。

【运行主线】

从 DispatcherServlet 所实现的接口来看，DispatcherServlet 的核心本质：**是一个 Servlet**。这个结论似乎很幼稚，不过这个幼稚的结论却蕴含了一个对整个框架都至关重要的内在原则：**Servlet 可以根据其特性进行运行主线的划分**。

根据 Servlet 规范的定义，Servlet 中的两大核心方法 init 方法和 service 方法，它们的运行时间和触发条件都截然不同：

1. **init 方法**

在整个系统启动时运行，且只运行一次。*因此，在 init 方法中我们往往会对整个应用程序进行初始化操作。这些初始化操作可能包括对容器（WebApplicationContext）的初始化、组件和外部资源的初始化等等。*

2. **service 方法**

在整个系统运行的过程中处于侦听模式，侦听并处理所有的 Web 请求。因此，在 service 及其相关方法中，我们看到的则是对 Http 请求的处理流程。

因而在这里，Servlet 的这一特性就被 SpringMVC 用于对不同的逻辑职责加以划分，从而形成两条互不相关的逻辑运行主线：

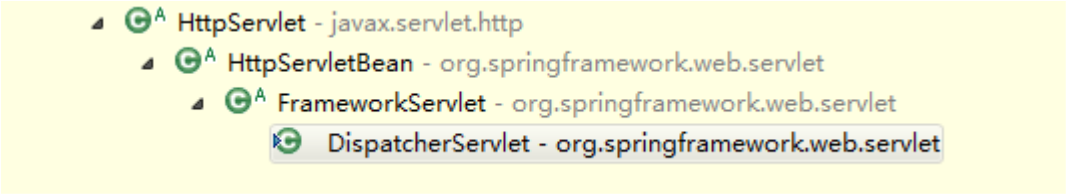
- 初始化主线 —— 负责对 SpringMVC 的运行要素进行初始化
- Http 请求处理主线 —— 负责对 SpringMVC 中的组件进行逻辑调度完成对 Http 请求的处理

对于一个 MVC 框架而言，运行主线的划分非常重要。因为只有弄清楚不同的运行主线，我们才能针对不同的运行主线采取不同的研究策略。而我们在这个系列中的绝大多数分析的切入点，也是围绕着不同的运行主线进行的。

注：SpringMVC 运行 主线的划分依据是 Servlet 对象中不同方法的生命周期。事实上，几乎所有的 MVC 都是以此为依据来进行运行主线的划分。这进一步可以证明所有的 MVC 框架的核心基础还是 Servlet 规范，而设计理念的差异也导致了不同的框架走向了完全不同的发展道路。

【继承结构】

除了运行主线的划分以外，我们再关注一下 DispatcherServlet 的继承结构：



在这个继承结构中，我们可以看到 DispatcherServlet 在其继承树中包含了 2 个 Spring 的支持类：HttpServletBean 和 FrameworkServlet。我们分别来讨论一下这两个 Spring 的支持类在这里所起到的作用。

HttpServletBean 是 Spring 对于 Servlet 最低层次的抽象。在这一层抽象中，Spring 会将这个 Servlet 视作是一个 Spring 的 bean，并将 init-param 中的值作为 bean 的属性注入进来：

Java 代码

```
public final void init() throws ServletException {
    if (logger.isDebugEnabled()) {
        logger.debug("Initializing servlet '" + getServletName() + "'");
    }

    // Set bean properties from init parameters.
    try {
        PropertyValues pvs = new ServletConfigPropertyValues(getServletConfig(),
this.requiredProperties);
```

```
            BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(this);
            ResourceLoader resourceLoader = new ServletContextResourceLoader(getServletContext());
            bw.registerCustomEditor(Resource.class, new ResourceEditor(resourceLoader, this.environment));
            initBeanWrapper(bw);
            bw.setPropertyValues(pvs, true);
        }
        catch (BeansException ex) {
            logger.error("Failed to set bean properties on servlet '" + getServletName() + "'", ex);
            throw ex;
        }

        // Let subclasses do whatever initialization they like.
        initServletBean();

        if (logger.isDebugEnabled()) {
            logger.debug("Servlet '" + getServletName() + "' configured successfully");
        }
    }
```

从源码中，我们可以看到 HttpServletBean 利用了 Servlet 的 init 方法的执行特性，将一个普通的 Servlet 与 Spring 的容器 联系在了一起。在这其中起到核心作用的代码是：BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(this);将当前的这个 Servlet 类转化为一个 BeanWrapper，从而能够以 Spring 的方式来对 init-param 的值进行注入。BeanWrapper 的相关知识属于 Spring Framework 的内容，我们在这里不做详细展开，读者可以具体参考 HttpServletBean 的注释获得更多的信息。

FrameworkServlet 则是在 HttpServletBean 的基础之上的进一步抽象。通过 FrameworkServlet 真正初始化了一个 Spring 的容器（WebApplicationContext），并引入到 Servlet 对象之中：

Java 代码

```
protected final void initServletBean() throws ServletException {
    getServletContext().log("Initializing Spring FrameworkServlet '" + getServletName() + "'");
    if (this.logger.isInfoEnabled()) {
        this.logger.info("FrameworkServlet '" + getServletName() + "': initialization started");
    }
    long startTime = System.currentTimeMillis();

    try {
        this.webApplicationContext = initWebApplicationContext();
        initFrameworkServlet();
    } catch (ServletException ex) {
        this.logger.error("Context initialization failed", ex);
        throw ex;
```

```
        } catch (RuntimeException ex) {
            this.logger.error("Context initialization failed", ex);
            throw ex;
        }

        if (this.logger.isInfoEnabled()) {
            long elapsedTime = System.currentTimeMillis() - startTime;
            this.logger.info("FrameworkServlet '" + getServletName() + "': initialization completed
in " +
                    elapsedTime + " ms");
        }
}
```

上面的这段代码就是 FrameworkServlet 初始化的核心代码。从中我们可以看到这个 FrameworkServlet 将调用其内部的方法 initWebApplicationContext()对 Spring 的容器（WebApplicationContext）进行初始化。同 时，FrameworkServlet 还暴露了与之通讯的结构可供子类调用：

Java 代码

```
public abstract class FrameworkServlet extends HttpServletBean {

    /** WebApplicationContext for this servlet */
    private WebApplicationContext webApplicationContext;

        // 这里省略了其他所有的代码

    /**
     * Return this servlet's WebApplicationContext.
     */
    public final WebApplicationContext getWebApplicationContext() {
        return this.webApplicationContext;
    }
}
```

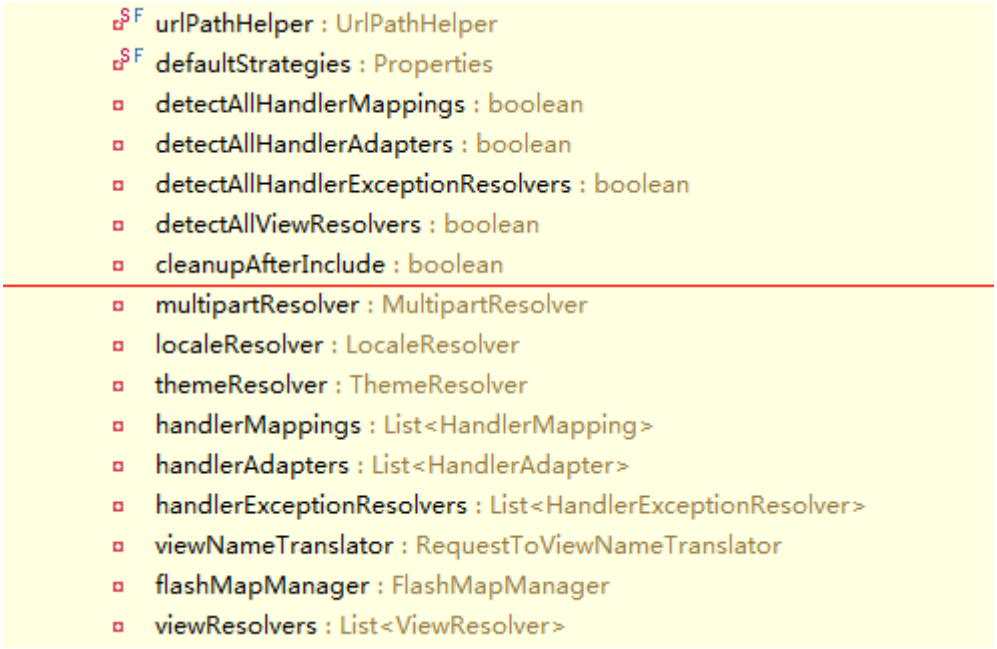
我们在这里暂且不对 Spring 容器（WebApplicationContext）的初始化过程详加探查，稍后我们会讨论一些 WebApplicationContext 初始化过程中的配置选项。不过读者可以在这里体会到：**FrameworkServlet 在其内部初始化了一个 Spring 的容器（WebApplicationContext）并暴露了相关的操作接口，因而继承自 FrameworkServlet 的 DispatcherServlet，也就直接拥有了与 WebApplicationContext 进行通信的能力。**

通过对 DispatcherServlet 继承结构的研究，我们可以明确：

结论 DispatcherServlet 的继承体系架起了 DispatcherServlet 与 Spring 容器进行沟通的桥梁。

【数据结构】

在上一篇文章中，我们曾经提到过 DispatcherServlet 的数据结构：



我们可以把在上面这张图中所构成 DispatcherServlet 的数据结构主要分为两类（我们在这里用一根分割线将其分割开来）：

- 配置参数 —— 控制 SpringMVC 组件的初始化行为方式
- 核心组件 —— SpringMVC 的核心逻辑处理组件

可以看到，这两类数据结构都与 SpringMVC 中的核心要素**组件**有关。因此，我们可以得出这样一个结论：

结论 组件是整个 DispatcherServlet 的灵魂所在：它不仅是初始化主线中的初始化对象，同样也是 Http 请求处理主线中的逻辑调度载体。

***注**：我们可以看到被我们划为配置参数 的那些变量都是 boolean 类型的，它们将在 DispatcherServlet 的初始化主线中起到一定的作用，我们在之后会使用源码进行说明。而这些 boolean 值可以通过 web.xml 中的 init-param 值进行设定覆盖（这是由 HttpServletBean 的特性带来的）。*

SpringMVC 的运行体系

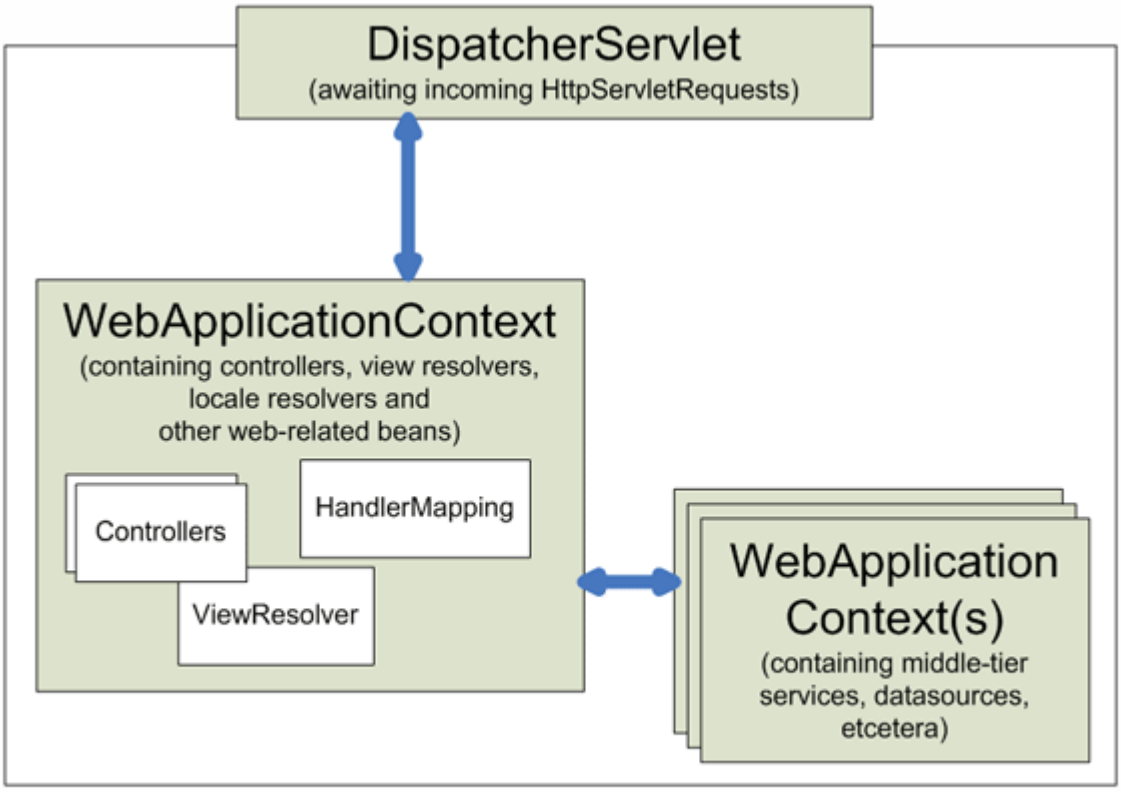
DispatcherServlet 继承结构和数据结构，实际上表述的是 DispatcherServlet 与另外两大要素之间的关系：

- 继承结构 —— DispatcherServlet 与 Spring 容器（WebApplicationContext）之间的关系
- 数据结构 —— DispatcherServlet 与组件之间的关系

所以，其实我们可以这么说：SpringMVC 的整个运行体系，是由 DispatcherServlet、组件和容器这三者共同构成的。

在这个运行体系中，DispatcherServlet 是逻辑处理的调度中心，组件则是被调度的操作对象。而容器在这里所起到的作用，是协助 DispatcherServlet 更好地对组件进行管理。这就相当于一个工厂招了一大批的工人，并把工人划分到一个统一的工作车间而便于管理。在工厂要进行生产活动时，只需要从工作车间把工人分派到相应的生产流水线上即可。

笔者在这里引用 Spring 官方 reference 中的一幅图，对三者之间的关系进行简单的描述：



Context hierarchy in Spring Web MVC

注：在这幅图中，我们除了看到在图的左半边 DispatcherServlet、组件和容器这三者之间的调用关系以外，还可以看到 SpringMVC 的运行体系与其它运行体系之间存在着关系。有关这一点，我们在之后的讨论中会详细展开。

既然是三个元素之间的关系表述，我们必须以两两关系的形式进行归纳：

- DispatcherServlet - 容器 —— DispatcherServlet 对容器进行初始化
- 容器 - 组件 —— 容器对组件进行全局管理
- DispatcherServlet - 组件 —— DispatcherServlet 对组件进行逻辑调用

值得注意的是，在上面这幅图中，三大元素之间的两两关系其实表现得并不明显，尤其是“容器 - 组件”和“DispatcherServlet - 组件”之间的关系。这主要是由于 Spring 官方 reference 所给出的这幅图是一个静态的关系表述，如果从动态的观点来对整个过程加以审视，我们就 不得不将 SpringMVC 的运行体系与之前所提到的运行主线联系在一起，看看这些元素在不同的逻辑主线中所起到的作用。

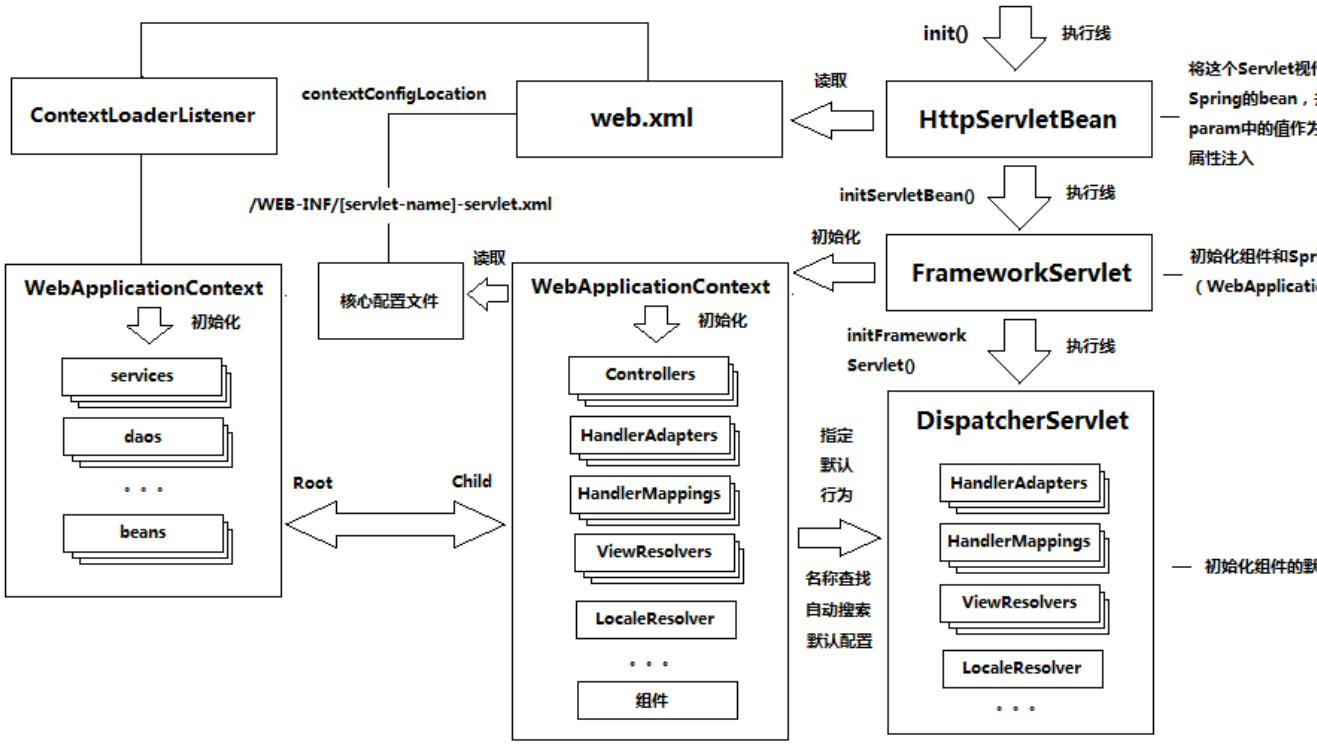
接下来，我们就分别看看 DispatcherServlet 的两条运行主线。

DispatcherServlet 的初始化主线

对于 DispatcherServlet 的初始化主线，我们首先应该明确几个基本观点：

- 初始化主线的驱动要素 —— servlet 中的 init 方法
- 初始化主线的执行次序 —— HttpServletBean -> FrameworkServlet -> DispatcherServlet
- 初始化主线的操作对象 —— Spring 容器（WebApplicationContext）和组件

这三个基本观点，可以说是我们对之前所有讨论的一个小结。明确了这些内容，我们就可以更加深入地看看 DispatcherServlet 初始化主线的过程：



在这幅图中，我们站在一个动态的角度将 DispatcherServlet、容器（WebApplicationContext）和组件这三者之间的关系 表述出来，同时给出了这三者之间的运行顺序和逻辑过程。读者或许对其中的绝大多数细节还很陌生，甚至有一种无从下手的感觉。这没有关系，大家可以首先抓住 图中的执行线，回忆一下之前有关 DispatcherServlet 的继承结构和数据结构的内容。接下来，我们就图中的内容逐一进行解释。

【WebApplicationContext 的初始化】

之前我们讨论了 DispatcherServlet 对于 WebApplicationContext 的初始化是在 FrameworkServlet 中完成的，不过我们并没有细究其中的细节。在默认情况下，这个初始化过程是由 web.xml 中的入口程序配置所驱动的：

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>
```

我们已经不止一次提到过这段配置，不过在这之前都没有对这段配置做过什么很详细的分析。事实上，这段入口程序的配置中隐藏了 SpringMVC 的两大要素 （核心分发器 Dispatcher 和核心配置文件[servlet-name]-servlet.xml）之间的关系表述：
在默认情况下， web.xml 配置节点中<servlet-name>的值就是建立起核心分发器 DispatcherServlet 与核心配 置文件之间联系的桥梁。DispatcherServlet 在初始化时会加载位置在/WEB-INF/[servlet- name]-servlet.xml 的配置文件作为 SpringMVC 的核心配置。

SpringMVC 在这里采用了一个“命名约定”的方法进行关系映射，这种方法很廉价也很管用。以上面的配置为例，我们就必须在/WEB-INF/目录 下，放一个名为 dispatcher-servlet.xml 的 Spring 配置文件作为 SpringMVC 的核心配置用以指定 SpringMVC 的基本 组件声明定义。

这看上去似乎有一点别扭，因为在实际项目中，我们通常喜欢把配置文件放在 classpath 下，并使用不同的 package 进行区分。例如，在基 于 Maven 的项目结构中，所有的配置文件应置于 src/main/resources 目录下，这样才比较符合配置文件统一化管理的最佳实践。

于是，Spring 提供了一个初始化的配置选项，通过指定 contextConfigLocation 选项来自定义 SpringMVC 核心配置文件的位置：

Xml 代码

```
<!-- Processes application requests -->
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:web/applicationContext-dispatcherServlet.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

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

这样一来，DispatcherServlet 在初始化时，就会自动加载在 classpath 下，web 这个 package 下名为 applicationContext-dispatcherServlet.xml 的文件作为其核心配置并用以初始化容器（WebApplicationContext）。

当然，这只是 DispatcherServlet 在进行 WebApplicationContext 初始化过程中的配置选项之一。我们可以在 Spring 的官方 reference 中找到相应的配置选项，有兴趣的读者可以参照 reference 的说明进行尝试：

You can customize individual `DispatcherServlet` instances by adding Servlet initialization parameters (`init-param` elements) to th declaration in the `web.xml` file. See the following table for the list of supported parameters.

Table 16.2. `DispatcherServlet` initialization parameters

Parameter	Explanation
`contextClass`	Class that implements `WebApplicationContext`, which instantiates the context used by this Servlet. By default, the `XmlWebApplicationContext` is used.
`contextConfigLocation`	String that is passed to the context instance (specified by `contextClass`) to indicate where context(s) ca found. The string consists potentially of multiple strings (using a comma as a delimiter) to support multi contexts. In case of multiple context locations with beans that are defined twice, the latest location take precedence.
`namespace`	Namespace of the `WebApplicationContext`. Defaults to `[servlet-name]-servlet`.

所有的这些配置选项，实际上都是为了让 DispatcherServlet 对 WebApplicationContext 的初始化过程显得更加自然。不过 这只是完成了容器（WebApplicationContext）的构建工作，那么容器所管理的那些组件，又是如何进行初始化的呢？

结论 SpringMVC 核心配置文件中所有的 bean 定义，就是 SpringMVC 的组件定义，也是 DispatcherServlet 在初始化容器（WebApplicationContext）时，所要进行初始化的组件。

在上一篇文章我们谈到组件的时候就曾经提到，SpringMVC 自身对于组件并未实现一套完整的管理机制，而是借用了 Spring Framework 核心框架中容器的概念，将所有的组件纳入到容器中进行管理。所以，SpringMVC 的核心配置文件使用与传统 Spring Framework 相同的配置形式，而整个管理的体系也是一脉相承的。

注：Spring3.0 之后，单独为 SpringMVC 建立了专用的 schema，从而使得我们可以使用 Schema Based XML 来对 SpringMVC 的组件进行定义。不过我们可以将其视作是传统 Spring 配置的一个补充，而不要过于纠结不同的配置格式。

我们知道，SpringMVC 的组件是一个个的接口定义，当我们在 SpringMVC 的核心配置文件中定义一个组件时，使用的却是组件的实现类：

Xml 代码

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

这也就是 Spring 管理组件的模式：**用具体的实现类来指定接口的行为方式。**不同的实现类，代表着不同的组件行为模式，它们在 Spring 容器中是可以共存的：

Xml 代码

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

```
<bean class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">
    <property name="prefix" value="/" />
    <property name="suffix" value=".ftl" />
</bean>
```

所以，Spring 的容器就像是一个聚宝盆，它只负责承载对象，管理好对象的生命周期，而并不关心一个组件接口到底有多少种实现类或者行为模式。这也就是我们在上面那幅图中，画了多个 HandlerMappings、HandlerAdapters 和 ViewResolvers 的原因：**一个组件的多种行为模式可以在容器中共存，容器将负责对这些实现类进行管理。**而具体如何使用这些对象，则由应用程序自身来决定。

如此一来，我们可以大致概括一下 WebApplicationContext 初始化的两个逻辑层次：

- DispatcherServlet 负责对容器(WebApplicationContext)进行初始化。
- 容器（WebApplicationContext）将读取 SpringMVC 的核心配置文件进行组件的实例化。

整个过程，我们把应用程序的日志级别调低，可以进行非常详细的观察：

引用

```
14:15:27,037 DEBUG StandardServletEnvironment:100 - Initializing new StandardServletEnvironment
14:15:27,128 DEBUG DispatcherServlet:115 - Initializing servlet 'dispatcher'
14:15:27,438  INFO DispatcherServlet:444 - FrameworkServlet 'dispatcher': initialization started
14:15:27,449 DEBUG DispatcherServlet:572 - Servlet with name 'dispatcher' will try to create custom WebApplicationContext context of class 'org.springframework.web.context.support.XmlWebApplicationContext', using parent context [null]
1571 [main] INFO /sample - Initializing Spring FrameworkServlet 'dispatcher'
14:15:27,505 DEBUG StandardServletEnvironment:100 - Initializing new StandardServletEnvironment
14:15:27,546 INFO XmlWebApplicationContext:495 - Refreshing WebApplicationContext for namespace 'dispatcher-servlet': startup date [Mon Feb 06 14:15:27 CST 2012]; root of context hierarchy
14:15:27,689  INFO XmlBeanDefinitionReader:315 - Loading XML bean definitions from class path resource [web/applicationContext-dispatcherServlet.xml]
14:15:27,872 DEBUG PluggableSchemaResolver:140 - Loading schema mappings from [META-INF/spring.schemas]
14:15:28,442 DEBUG PathMatchingResourcePatternResolver:550 - Looking for matching resources in directory tree [D:\Work\Demo2do\Sample\target\classes\com\demo2do\sample\web\controller]
14:15:28,442 DEBUG PathMatchingResourcePatternResolver:612 - Searching directory
```

[D:\Work\Demo2do\Sample\target\classes\com\demo2do\sample\web\controller] for files matching pattern [D:/Work/Demo2do/Sample/target/classes/com/demo2do/sample/web/controller/**/*.class]
14:15:28,450 DEBUG PathMatchingResourcePatternResolver:351 - Resolved location pattern [classpath*:com/demo2do/sample/web/controller/**/*.class] to resources [file [D:\Work\Demo2do\Sample\target\classes\com\demo2do\sample\web\controller\BlogController.class], file [D:\Work\Demo2do\Sample\target\classes\com\demo2do\sample\web\controller\UserController.class]]
14:15:28,569 DEBUG ClassPathBeanDefinitionScanner:243 - Identified candidate component class: file [D:\Work\Demo2do\Sample\target\classes\com\demo2do\sample\web\controller\BlogController.class]
14:15:28,571 DEBUG ClassPathBeanDefinitionScanner:243 - Identified candidate component class: file [D:\Work\Demo2do\Sample\target\classes\com\demo2do\sample\web\controller\UserController.class]
14:15:28,634 DEBUG BeanDefinitionParserDelegate:497 - Neither XML 'id' nor 'name' specified - using generated bean name [org.springframework.web.servlet.view.InternalResourceViewResolver#0]
14:15:28,635 DEBUG XmlBeanDefinitionReader:216 - Loaded 19 bean definitions from location pattern [classpath:web/applicationContext-dispatcherServlet.xml]
14:15:28,635 DEBUG XmlWebApplicationContext:525 - Bean factory for WebApplicationContext for namespace 'dispatcher-servlet': org.springframework.beans.factory.support.DefaultListableBeanFactory@2321b59a: defining beans [org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping#0,org.springframework.format.support.FormattingConversionServiceFactoryBean#0,org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter#0,org.springframework.web.servlet.handler.MappedInterceptor#0,org.springframework.web.servlet.mvc.method.annotation.ExceptionHandlerExceptionResolver#0,org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver#0,org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver#0,org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,blogController,userController,org.springframework.context.annotation.internalConfigurationAnnotationProcessor,org.springframework.context.annotation.internalAutowiredAnnotationProcessor,org.springframework.context.annotation.internalRequiredAnnotationProcessor,org.springframework.context.annotation.internalCommonAnnotationProcessor,org.springframework.web.servlet.resource.ResourceHttpRequestHandler#0,org.springframework.web.servlet.handler.SimpleUrlHandlerMapping#0,org.springframework.web.servlet.view.InternalResourceViewResolver#0]; root of factory hierarchy
14:15:29,015 DEBUG RequestMappingHandlerMapping:98 - Looking for request mappings in application context: WebApplicationContext for namespace 'dispatcher-servlet': startup date [Mon Feb 06 14:15:27 CST 2012]; root of context hierarchy

14:15:29,037 INFO RequestMappingHandlerMapping:188 - Mapped "{[/blog],methods=[],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public org.springframework.web.servlet.ModelAndView com.demo2do.station.web.controller.BlogController.index()
14:15:29,039 INFO RequestMappingHandlerMapping:188 - Mapped "{[/login],methods=[],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public org.springframework.web.servlet.ModelAndView com.demo2do.station.web.controller.UserController.login(java.lang.String,java.lang.String)
14:15:29,040 DEBUG DefaultListableBeanFactory:458 - Finished creating instance of bean 'org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping#0'
14:15:29,460 DEBUG BeanNameUrlHandlerMapping:71 - Looking for URL mappings in application context: WebApplicationContext for namespace 'dispatcher-servlet': startup date [Mon Feb 06 14:15:27 CST 2012]; root of context hierarchy
14:15:29,539 DEBUG DefaultListableBeanFactory:458 - Finished creating instance of bean 'org.springframework.web.servlet.resource.ResourceHttpRequestHandler#0'
14:15:29,540 DEBUG DefaultListableBeanFactory:217 - Creating shared instance of singleton bean 'org.springframework.web.servlet.handler.SimpleUrlHandlerMapping#0'
14:15:29,555 INFO SimpleUrlHandlerMapping:314 - Mapped URL path [/static/**] onto handler 'org.springframework.web.servlet.resource.ResourceHttpRequestHandler#0'
14:15:29,556 DEBUG DefaultListableBeanFactory:458 - Finished creating instance of bean 'org.springframework.web.servlet.handler.SimpleUrlHandlerMapping#0'
14:15:29,827 DEBUG DispatcherServlet:523 - Published WebApplicationContext of servlet 'dispatcher' as ServletContext attribute with name [org.springframework.web.servlet.FrameworkServlet.CONTEXT.dispatcher]
14:15:29,827 INFO DispatcherServlet:463 - FrameworkServlet 'dispatcher': initialization completed in 2389 ms
14:15:29,827 DEBUG DispatcherServlet:136 - Servlet 'dispatcher' configured successfully
4047 [main] INFO org.mortbay.log - Started SelectChannelConnector@0.0.0.0:8080
Jetty Server started, use 4267 ms

在这段启动日志（笔者进行了一定删减）中，笔者刻意将 WebApplicationContext 的初始化的标志日志使用红色的标进行区分，而将核心配置 文件的读取位置和组件定义初始化的标志日志使用蓝色标记加以区分。相信有了这段日志的帮助，读者应该可以对 WebApplicationContext 的 初始化过程有了更加直观的认识。

注：启动日志是我们研究 SpringMVC 的主要途径之一，之后我们还将反复使用这种方法对 SpringMVC 的运行过程进行研究。读者应该仔细品味每一条日志的作用，从而能够与之后的分析讲解呼应起来。

或许读者对 WebApplicationContext 对组件进行初始化的过程还有点困惑，大家不妨先将这个过程省略，把握住整个 DispatcherServlet 的大方向。我们在之后的文章中，还将对 SpringMVC 的组件、这些组件的定义以及组件的初始化方式做进一步的分析 和探讨。

到此为止，图中顺着 FrameworkServlet 的那些箭头，我们已经交代清楚，读者可以回味一下

整个过程。

【独立的 WebApplicationContext 体系】

独立的 WebApplicationContext 体系，是 SpringMVC 初始化主线中的一个非常重要的概念。回顾一下刚才曾经提到过的 DispatcherServlet、容器和组件三者之间的关系，我们在引用的那副官方 reference 的示意图中，实际上已经包含了这一层意思：

结论 在 DispatcherServlet 初始化的过程中所构建的 WebApplicationContext 独立于 Spring 自身的所构建的其他 WebApplicationContext 体系而存在。

稍有一些 Spring 编程经验的程序员，对于下面的配置应该非常熟悉：

Xml 代码

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:context/applicationContext-*.xml</param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

在上面的代码中，我们定义了一个 Listener，它会在整个 Web 应用程序启动的时候运行一次，并初始化**传统意义上的 Spring 的容器**。这也是一般情况下，当并不使用 SpringMVC 作为我们的表示层解决方案，却希望在我们的 Web 应用程序中使用 Spring 相关功能时所采取的一种配置方式。

如果我们要在这里引入 SpringMVC，整个配置看上去就像这样：

Xml 代码

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:context/applicationContext-*.xml</param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<!-- Processes application requests -->
<servlet>
    <servlet-name>dispatcher</servlet-name>
```

```
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:web/applicationContext-dispatcherServlet.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

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

在这种情况下，DispatcherServlet 和 ContextLoaderListener 会分别构建不同作用范围的容器（WebApplicationContext）。我们可以引入两个不同的概念来对其进行表述：ContextLoaderListener 所初始化的容器，我们称之为 **Root WebApplicationContext**；而 DispatcherServlet 所初始化的容器，是 **SpringMVC WebApplicationContext**。

同样采取日志分析的方法，加入了 ContextLoaderListener 之后，整个启动日志变成了这样：

引用

```
[main] INFO /sample - Initializing Spring root WebApplicationContext
14:56:42,261  INFO ContextLoader:272 - Root WebApplicationContext: initialization started
14:56:42,343  DEBUG  StandardServletEnvironment:100  -  Initializing  new
StandardServletEnvironment
14:56:42,365  INFO XmlWebApplicationContext:495 - Refreshing Root WebApplicationContext:
startup date [Mon Feb 06 14:56:42 CST 2012]; root of context hierarchy
14:56:42,441  DEBUG  PathMatchingResourcePatternResolver:550  -  Looking  for  matching
resources in directory tree [D:\Work\Demo2do\Sample\target\classes\context]
14:56:42,442  DEBUG  PathMatchingResourcePatternResolver:612  -  Searching  directory
[D:\Work\Demo2do\Sample\target\classes\context]  for  files  matching  pattern
[D:/Work/Demo2do/Sample/target/classes/context/applicationContext-*.xml]
14:56:42,446 DEBUG PathMatchingResourcePatternResolver:351 - Resolved location pattern
[classpath:context/applicationContext-*.xml]  to  resources  [file
[D:\Work\Demo2do\Sample\target\classes\context\applicationContext-configuration.xml]]
14:56:42,447  INFO XmlBeanDefinitionReader:315 - Loading XML bean definitions from file
[D:\Work\Demo2do\Sample\target\classes\context\applicationContext-configuration.xml]
14:56:42,486  DEBUG  PluggableSchemaResolver:140  -  Loading  schema  mappings  from
[META-INF/spring.schemas]
14:56:42,597 DEBUG DefaultBeanDefinitionDocumentReader:108 - Loading bean definitions
14:56:42,658  DEBUG  PathMatchingResourcePatternResolver:550  -  Looking  for  matching
resources in directory tree [D:\Work\Demo2do\Sample\target\classes\com\demo2do\sample]
14:56:42,699  DEBUG  ClassPathBeanDefinitionScanner:243  -  Identified  candidate  component
```

class: file [D:\Work\Demo2do\Sample\target\classes\com\demo2do\sample\service\impl\BlogServiceImpl.class]
14:56:42,750 DEBUG XmlBeanDefinitionReader:216 - Loaded 5 bean definitions from location pattern [classpath:context/applicationContext-*.xml]
14:56:42,750 DEBUG XmlWebApplicationContext:525 - Bean factory for Root WebApplicationContext: org.springframework.beans.factory.support.DefaultListableBeanFactory@478e4327: defining beans [blogService,org.springframework.context.annotation.internalConfigurationAnnotationProcessor,org.springframework.context.annotation.internalAutowiredAnnotationProcessor,org.springframework.context.annotation.internalRequiredAnnotationProcessor,org.springframework.context.annotation.internalCommonAnnotationProcessor]; root of factory hierarchy
14:56:42,860 DEBUG ContextLoader:296 - Published root WebApplicationContext as ServletContext attribute with name [org.springframework.web.context.WebApplicationContext.ROOT]
14:56:42,860 INFO ContextLoader:301 - Root WebApplicationContext: initialization completed in 596 ms

14:56:42,935 DEBUG DispatcherServlet:115 - Initializing servlet 'dispatcher'
14:56:42,974 INFO DispatcherServlet:444 - FrameworkServlet 'dispatcher': initialization started
14:56:42,974 DEBUG DispatcherServlet:572 - Servlet with name 'dispatcher' will try to create custom WebApplicationContext context of class 'org.springframework.web.context.support.XmlWebApplicationContext', using parent context [Root WebApplicationContext: startup date [Mon Feb 06 14:56:42 CST 2012]; root of context hierarchy]
14:56:42,979 INFO XmlWebApplicationContext:495 - Refreshing WebApplicationContext for namespace 'dispatcher-servlet': startup date [Mon Feb 06 14:56:42 CST 2012]; parent: Root WebApplicationContext
14:56:42,983 INFO XmlBeanDefinitionReader:315 - Loading XML bean definitions from class path resource [web/applicationContext-dispatcherServlet.xml]
14:56:42,987 DEBUG PluggableSchemaResolver:140 - Loading schema mappings from [META-INF/spring.schemas]
14:56:43,035 DEBUG DefaultBeanDefinitionDocumentReader:108 - Loading bean definitions
14:56:43,075 DEBUG PathMatchingResourcePatternResolver:550 - Looking for matching resources in directory tree [D:\Work\Demo2do\Sample\target\classes\com\demo2do\sample\web\controller]
14:56:43,075 DEBUG PathMatchingResourcePatternResolver:612 - Searching directory [D:\Work\Demo2do\Sample\target\classes\com\demo2do\sample\web\controller] for files matching pattern [D:/Work/Demo2do/Sample/target/classes/com/demo2do/sample/web/controller/**/*.class]
14:56:43,077 DEBUG PathMatchingResourcePatternResolver:351 - Resolved location pattern [classpath*:com/demo2do/sample/web/controller/**/*.class] to resources [file

[D:\Work\Demo2do\Sample\target\classes\com\demo2do\sample\web\controller\BlogController.class], file [D:\Work\Demo2do\Sample\target\classes\com\demo2do\sample\web\controller\UserController.class]]
14:56:43,079 DEBUG ClassPathBeanDefinitionScanner:243 - Identified candidate component class: file [D:\Work\Demo2do\Sample\target\classes\com\demo2do\sample\web\controller\BlogController.class]
14:56:43,080 DEBUG ClassPathBeanDefinitionScanner:243 - Identified candidate component class: file [D:\Work\Demo2do\Sample\target\classes\com\demo2do\sample\web\controller\UserController.class]
14:56:43,089 DEBUG XmlBeanDefinitionReader:216 - Loaded 19 bean definitions from location pattern [classpath:web/applicationContext-dispatcherServlet.xml]
14:56:43,089 DEBUG XmlWebApplicationContext:525 - Bean factory for WebApplicationContext for namespace 'dispatcher-servlet': org.springframework.beans.factory.support.DefaultListableBeanFactory@5e6458a6: defining beans [org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping#0,org.springframework.format.support.FormattingConversionServiceFactoryBean#0,org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter#0,org.springframework.web.servlet.handler.MappedInterceptor#0,org.springframework.web.servlet.mvc.method.annotation.ExceptionHandlerExceptionResolver#0,org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver#0,org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver#0,org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,blogController,userController,org.springframework.context.annotation.internalConfigurationAnnotationProcessor,org.springframework.context.annotation.internalAutowiredAnnotationProcessor,org.springframework.context.annotation.internalRequiredAnnotationProcessor,org.springframework.context.annotation.internalCommonAnnotationProcessor,org.springframework.web.servlet.resource.ResourceHttpRequestHandler#0,org.springframework.web.servlet.handler.SimpleUrlHandlerMapping#0,org.springframework.web.servlet.view.InternalResourceViewResolver#0]; parent: org.springframework.beans.factory.support.DefaultListableBeanFactory@478e4327
14:56:43,323 DEBUG RequestMappingHandlerMapping:98 - Looking for request mappings in application context: WebApplicationContext for namespace 'dispatcher-servlet': startup date [Mon Feb 06 14:56:42 CST 2012]; parent: Root WebApplicationContext
14:56:43,345 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()
14:56:43,346 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)
14:56:43,707 DEBUG BeanNameUrlHandlerMapping:71 - Looking for URL mappings in application context: WebApplicationContext for namespace 'dispatcher-servlet': startup date [Mon Feb 06 14:56:42 CST 2012]; parent: Root WebApplicationContext
14:56:43,828 INFO SimpleUrlHandlerMapping:314 - Mapped URL path [/static/**] onto handler 'org.springframework.web.servlet.resource.ResourceHttpRequestHandler#0'
14:56:43,883 DEBUG DispatcherServlet:523 - Published WebApplicationContext of servlet 'dispatcher' as ServletContext attribute with name [org.springframework.web.servlet.FrameworkServlet.CONTEXT.dispatcher]
14:56:43,883 INFO DispatcherServlet:463 - FrameworkServlet 'dispatcher': initialization completed in 909 ms
14:56:43,883 DEBUG DispatcherServlet:136 - Servlet 'dispatcher' configured successfully
2687 [main] INFO org.mortbay.log - Started SelectChannelConnector@0.0.0.0:8080
Jetty Server started, use 2901 ms

整个启动日志被我们分为了 2 段。第一段的过程初始化的是 Root WebApplicationContext；而第二段的过程初始化的是 SpringMVC 的 WebApplicationContext。我们还是使用 了红色的标记和蓝色标记指出了在整个初始化过程中的一些重要事件。其中，有这样一段内容值得我们注意：

引用

14:56:42,979 INFO XmlWebApplicationContext:495 - Refreshing WebApplicationContext for namespace 'dispatcher-servlet': startup date [Mon Feb 06 14:56:42 CST 2012]; parent: Root WebApplicationContext

在这段日志中，非常明确地指出了 SpringMVC WebApplicationContext 与 Root WebApplicationContext 之间的关系：**从属关系**。因为根据这段日志的表述，**SpringMVC WebApplicationContext 能够感知到 Root WebApplicationContext 的存在，并且将其作为 parent 容器。**

Spring 正是使用这种 Parent-Child 的容器关系来对不同的编程层次进行划分。这种我们俗称的父子关系实际上不仅仅是一种从属关系，更是一种引用关系。从刚才的日志分析中，我们可以看出：**SpringMVC 中所定义的一切组件能够无缝地与 Root WebApplicationContext 中的组件整合。**

到此为止，我们针对图中以 web.xml 为核心的箭头分支进行了讲解，读者可以将图中的内容与上面的文字说明对照再次加以理解。

【组件默认行为的指定】

DispatcherServlet 的初始化主线的执行体系是顺着其继承结构依次进行的，我们在之前曾经讨论过它的执行次序。所以，只有在 FrameworkServlet 完成了对于 WebApplicationContext

和组件的初始化之后，执行权才被正式转移到 DispatcherServlet 中。我们可以来看看 DispatcherServlet 此时究竟干了哪些事：

Java 代码

```
/**
 * This implementation calls {@link #initStrategies}.
 */
@Override
protected void onRefresh(ApplicationContext context) {
	initStrategies(context);
}

/**
 * Initialize the strategy objects that this servlet uses.
 * <p>May be overridden in subclasses in order to initialize further strategy objects.
 */
protected void initStrategies(ApplicationContext context) {
	initMultipartResolver(context);
	initLocaleResolver(context);
	initThemeResolver(context);
	initHandlerMappings(context);
	initHandlerAdapters(context);
	initHandlerExceptionResolvers(context);
	initRequestToViewNameTranslator(context);
	initViewResolvers(context);
	initFlashMapManager(context);
}
```

onRefresh 是 FrameworkServlet 中预留的扩展方法，在 DispatcherServlet 中做了一个基本实现：initStrategies。我们粗略一看，很容易就能明白 DispatcherServlet 到底在这里干些什么了：**初始化组件**。

读者或许会问，组件不是已经在 WebApplicationContext 初始化的时候已经被初始化过了嘛？这里所谓的组件初始化，指的又是什么呢？让我们来看看其中的一个方法的源码：

Java 代码

```
/**
 * Initialize the MultipartResolver used by this class.
 * <p>If no bean is defined with the given name in the BeanFactory for this namespace,
 * no multipart handling is provided.
 */
private void initMultipartResolver(ApplicationContext context) {
	try {
		this.multipartResolver = context.getBean(MULTIPART_RESOLVER_BEAN_NAME,
```

```
MultipartResolver.class);
            if (logger.isDebugEnabled()) {
                logger.debug("Using MultipartResolver [" + this.multipartResolver + "]");
            }
        } catch (NoSuchBeanDefinitionException ex) {
            // Default is no multipart resolver.
            this.multipartResolver = null;
            if (logger.isDebugEnabled()) {
                logger.debug("Unable to locate MultipartResolver with name '" +
MULTIPART_RESOLVER_BEAN_NAME +
                        "': no multipart request handling provided");
            }
        }
}
```

原来，这里的初始化，指的是 **DispatcherServlet 从容器（WebApplicationContext）中读取组件的实现类，并缓存于 DispatcherServlet 内部的过程**。还记得我们之前给出的 DispatcherServlet 的数据结构吗？这些位于 DispatcherServlet 内部的组件实际上只是一些来源于容器缓存实例，不过它们同样也是 DispatcherServlet 进行后续操作的基础。

***注**：我们在第一篇文章中就曾经提到过 Servlet 实例内部的属性的访问有线程安全问题。而在这里，我们可以看到所有的组件都以 Servlet 内部属性的形式被调用，充分证实了这些组件本身也都是无状态的单例对象，所以我们在这里不必考虑线程安全的问题。*

如果对上面的代码加以详细分析，我们会发现 initMultipartResolver 的过程是查找特定 MultipartResolver 实现 类的过程。因为在容器中查找组件的时候，采取的是根据特定名称（MULTIPART_RESOLVER_BEAN_NAME）进行查找的策略。由此，我们 可以看到 DispatcherServlet 进行组件初始化的特点：

结论 DispatcherServlet 中对于组件的初始化过程实际上是应用程序在 WebApplicationContext 中选择和查找组件实现类的过程，也是指定组件在 SpringMVC 中的默认行为方式的过程。

除了根据特定名称进行查找的策略以外，我们还对 DispatcherServlet 中指定 SpringMVC 默认行为方式的其他的策略进行的总结：

- **名称查找 —— 根据 bean 的名字在容器中查找相应的实现类**
- **自动搜索 —— 自动搜索容器中所有某个特定组件（接口）的所有实现类**
- **默认配置 —— 根据一个默认的配置文件指定进行实现类加载**

这三条策略恰巧在 initHandlerMappings 的过程中都有体现，读者可以从其源码中找到相应的线索：

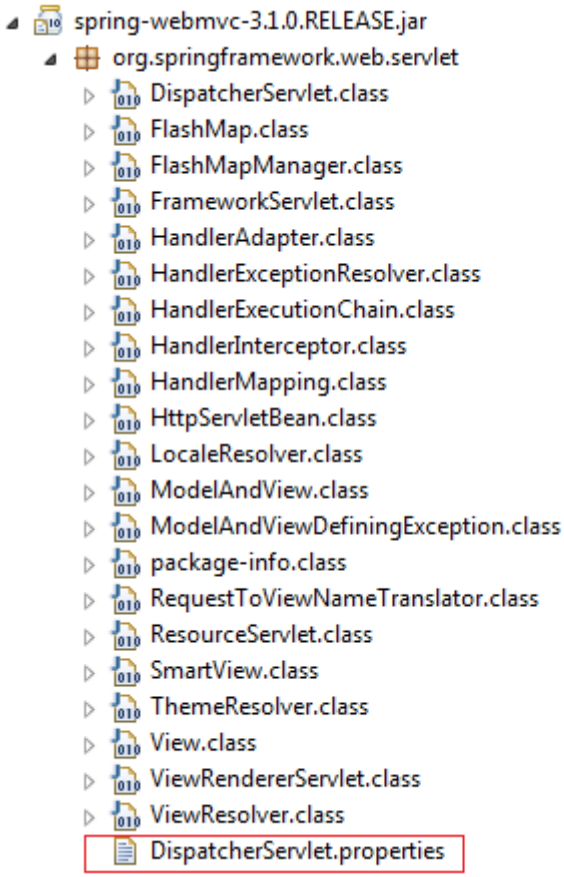
Java 代码

```
private void initHandlerAdapters(ApplicationContext context) {
	this.handlerAdapters = null;

	if (this.detectAllHandlerAdapters) {
		// Find all HandlerAdapters in the ApplicationContext, including ancestor contexts.
		Map<String, HandlerAdapter> matchingBeans =
BeanFactoryUtils.beansOfTypeIncludingAncestors(context, HandlerAdapter.class, true, false);
		if (!matchingBeans.isEmpty()) {
			this.handlerAdapters = new ArrayList<HandlerAdapter>(matchingBeans.values());
			// We keep HandlerAdapters in sorted order.
			OrderComparator.sort(this.handlerAdapters);
		}
	}
	else {
		try {
			HandlerAdapter ha = context.getBean(HANDLER_ADAPTER_BEAN_NAME,
HandlerAdapter.class);
			this.handlerAdapters = Collections.singletonList(ha);
		}
		catch (NoSuchBeanDefinitionException ex) {
			// Ignore, we'll add a default HandlerAdapter later.
		}
	}

	// Ensure we have at least some HandlerAdapters, by registering
	// default HandlerAdapters if no other adapters are found.
	if (this.handlerAdapters == null) {
		this.handlerAdapters = getDefaultStrategies(context, HandlerAdapter.class);
		if (logger.isDebugEnabled()) {
			logger.debug("No HandlerAdapters found in servlet '" + getServletName() + "':
using default");
		}
	}
}
```

这里有必要对“默认策略”做一个简要的说明。SpringMVC 为一些核心组件设置了默认行为方式的说明，这个说明以一个 properties 文件的形式位于 SpringMVC 分发包（例如 spring-webmvc-3.1.0.RELEASE.jar）的内部：



我们可以观察一下 DispatcherServlet.properties 的内容：

引用

```
# Default implementation classes for DispatcherServlet's strategy interfaces.
# Used as fallback when no matching beans are found in the DispatcherServlet context.
# Not meant to be customized by application developers.

org.springframework.web.servlet.LocaleResolver=org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver

org.springframework.web.servlet.ThemeResolver=org.springframework.web.servlet.theme.FixedThemeResolver

org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\
org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping

org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\
org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\
```

org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter

org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver,\
org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\
org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver

org.springframework.web.servlet.RequestToViewNameTranslator=org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator

org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResourceViewResolver

org.springframework.web.servlet.FlashMapManager=org.springframework.web.servlet.support.DefaultFlashMapManager

结合刚才 initHandlerMappings 的源码，我们可以发现如果没有开启 detectAllHandlerAdapters 选项或者根据 HANDLER_ADAPTER_BEAN_NAME 的名称没有找到相应的组件实现类，就会使用 DispatcherServlet.properties 文件中对于 HandlerMapping 接口的实现来进行组件默认行为的初始化。

由此可见，DispatcherServlet.properties 中所指定的所有接口的实现方式在 Spring 的容器 WebApplicationContext 中总有相应的定义。这一点，我们在组件的讨论中还会详谈。

这个部分我们的侧重点是图中 DispatcherServlet 与容器之间的关系。读者需要理解的是图中为什么会有两份组件定义，它们之间的区别在哪里，以及 DispatcherServlet 在容器中查找组件的三种策略。

小结

在本文中，我们对 SpringMVC 的核心类：DispatcherServlet 进行了一番梳理。也对整个 SpringMVC 的两条主线之一的初始化主线做了详细的分析。

对于 DispatcherServlet 而言，重要的其实并不是这个类中的代码和逻辑，而是应该掌握这个类在整个框架中的作用以及与 SpringMVC 中其他要素的关系。

对于初始化主线而言，核心其实仅仅在于那张笔者为大家精心打造的图。读者只要掌握了这张图，相信对整个 SpringMVC 的初始化过程会有一个全新的认识。