# SpringMVC源码剖析（三）- DispatcherServlet的初始化流程 - 相见欢

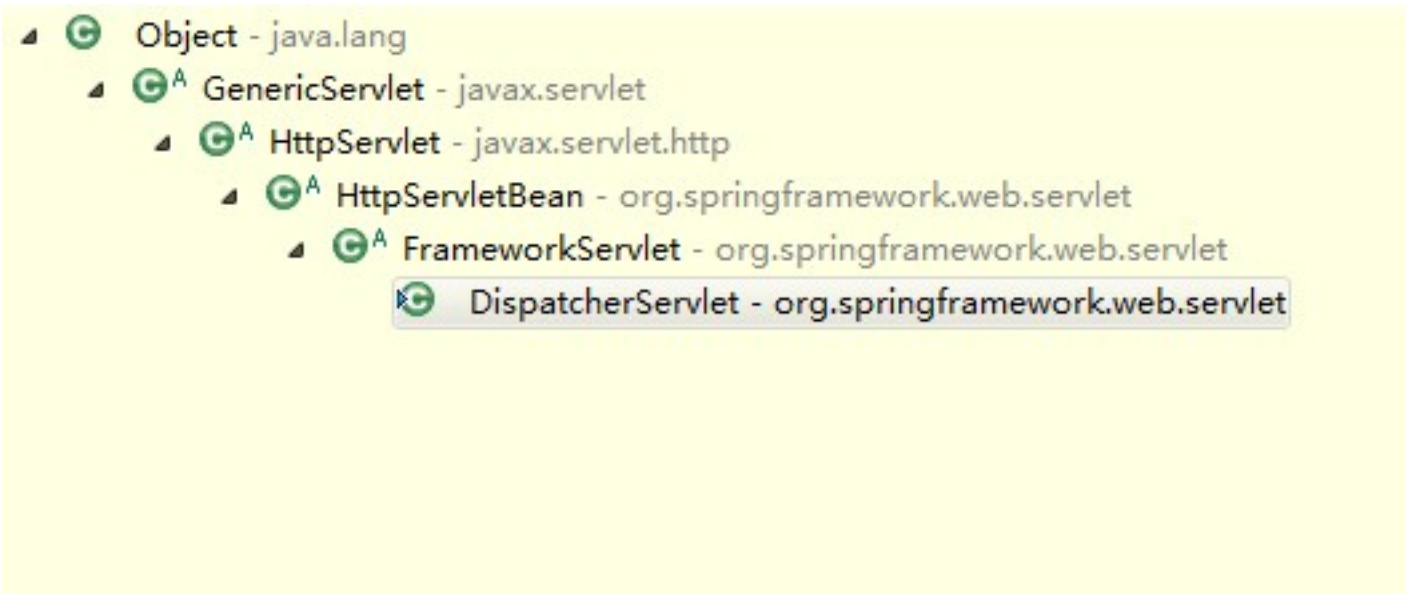
在我们第一次学Servlet编程，学java web的时候，还没有那么多框架。我们开发一个简单的功能要做的事情很简单，就是继承HttpServlet，根据需要重写一下doGet，doPost方法，跳转到我们定义好的jsp页面。Servlet类编写完之后在web.xml里注册这个Servlet类。

除此之外，没有其他了。我们启动web服务器，在浏览器中输入地址，就可以看到浏览器上输出我们写好的页面。为了更好的理解上面这个过程，你需要学习关于Servlet生命周期的三个阶段，就是所谓的“init-service-destroy”。

以上的知识，我觉得对于你理解SpringMVC的设计思想，已经足够了。SpringMVC当然可以称得上是一个复杂的框架，但是同时它又遵循Servlet世界里最简单的法则，那就是“init-service-destroy”。我们要分析SpringMVC的初始化流程，其实就是分析DispatcherServlet类的init()方法，让我们带着这种单纯的观点，打开DispatcherServlet的源码一窥究竟吧。

## 1.<init-param>配置元素读取

用Eclipse IDE打开DispatcherServlet类的源码，ctrl+T看一下。




DispatcherServlet类的初始化入口方法init()定义在HttpServletBean这个父类中，HttpServletBean类作为一个直接继承于HttpServlet类的类，覆写了HttpServlet类的init()方法，实现了自己的初始化行为。

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

                // Set bean properties from init parameters.
                try {
                        PropertyValues pvs = new ServletConfigPropertyValue
                        BeanWrapper bw = PropertyAccessorFactory.forBeanPro
                        ResourceLoader resourceLoader = new ServletContextR
                        bw.registerCustomEditor(Resource.class, new Resourc
                        initBeanWrapper(bw);
                        bw.setPropertyValues(pvs, true);
                }
                catch (BeansException ex) {
                        logger.error("Failed to set bean properties on serv
                        throw ex;
                }

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

                if (logger.isDebugEnabled()) {
                        logger.debug("Servlet '" + getServletName() + "' co
                }
        }
```

这里的initServletBean()方法在HttpServletBean类中是一个没有任何实现的空方法，它的目的就是留待子类实现自己的初始化逻辑，也就是我们常说的模板方法设计模式。SpringMVC在此生动的运用了这个模式，init()方法就是模版方法模式中的模板方法，SpringMVC真正的初始化过程，由子类FrameworkServlet中覆写的initServletBean()方法触发。

再看一下init()方法内被try,catch块包裹的代码，里面涉及到BeanWrapper，PropertyValues，ResourceEditor这些Spring内部非常底层的类。要深究具体代码实现上面的细节，需要对Spring框架源码具有相当深入的了解。我们这里先避繁就简，从代码效果和设计思想上面来分析这段try,catch块内的代码所做的事情：

- 注册一个字符串到资源文件的编辑器，让Servlet下面的<init-param>配置元素可以使用形如“classpath:”这种方式指定SpringMVC框架bean配置

文件的来源。

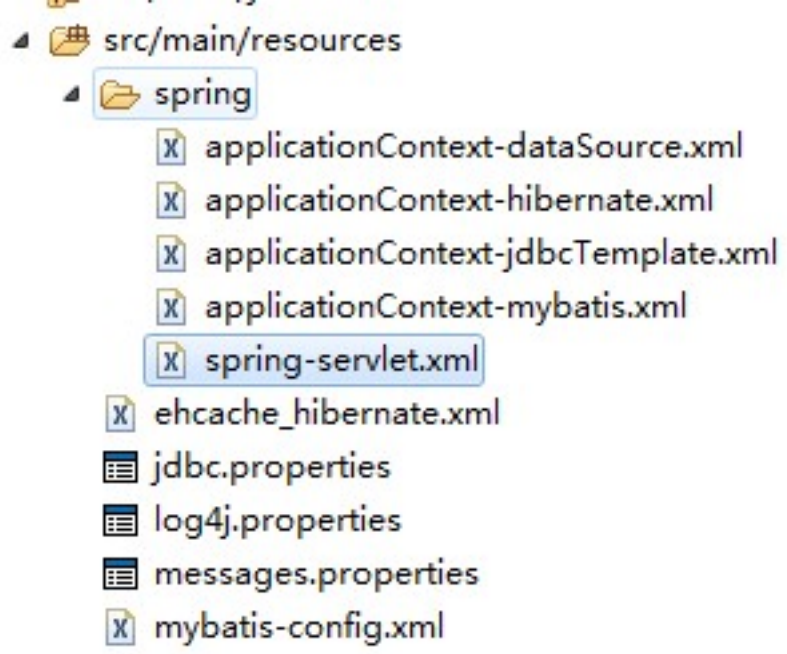
- 将web.xml中在DispatcherServlet这个Servlet下面的<init-param>配置元素利用JavaBean的方式（即通过setter方法）读取到DispatcherServlet中来。

这两点，我想通过下面一个例子来说明一下。

我在web.xml中注册的DispatcherServlet配置如下：

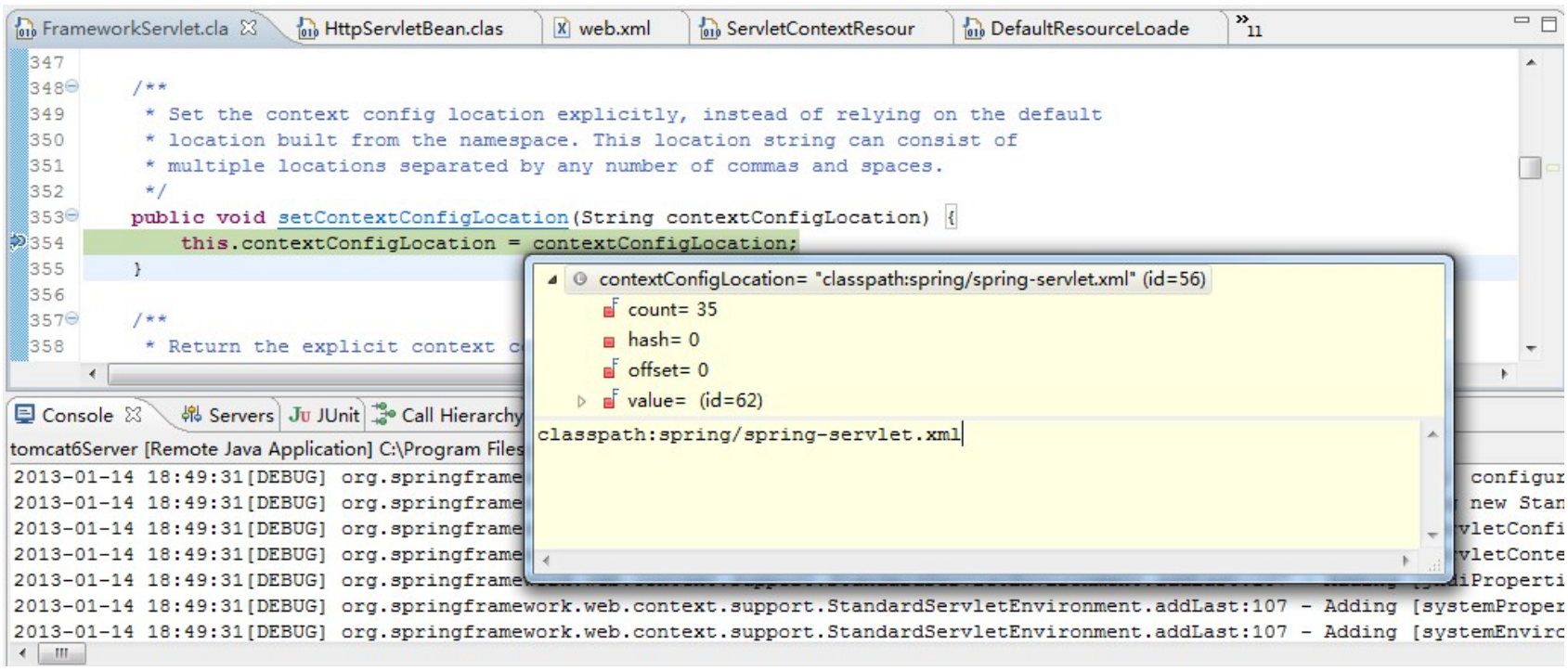
```
<!-- springMVC配置开始 -->
        <servlet>
                <servlet-name>appServlet</servlet-name>
                <servlet-class>org.springframework.web.servlet.DispatcherSe
                <init-param>
                        <param-name>contextConfigLocation</param-name>
                        <param-value>classpath:spring/spring-servlet.xml</p
                </init-param>
                <load-on-startup>1</load-on-startup>
        </servlet>
        <servlet-mapping>
                <servlet-name>appServlet</servlet-name>
                <url-pattern>/</url-pattern>
        </servlet-mapping>
        <!-- springMVC配置结束 -->
```

可以看到，我注册了一个名为contextConfigLocation的<init-param>元素，其值为“classpath:spring/spring-servlet.xml”，这也是大家常常用来指定SpringMVC配置文件路径的方法。上面那段try,catch块包裹的代码发挥的作用，一个是将“classpath:spring/spring-servlet.xml”这段字符串转换成classpath路径下的一个资源文件，供框架初始化读取配置元素。在我的工程中是在spring文件夹下面的配置文件spring-servlet.xml。




另外一个作用，就是将contextConfigLocation的值读取出来，然后通过setContextConfigLocation()方法设置到DispatcherServlet中，这个setContextConfigLocation()方法是在FrameworkServlet类中定义的，也就是上面继承类图中DispatcherServlet的直接父类。

我们在setContextConfigLocation()方法上面打上一个断点，启动web工程，可以看到下面的调试结果。




HttpServletBean类的作者是大名鼎鼎的Spring之父Rod Johnson。作为POJO编程哲学的大师，他在HttpServletBean这个类的设计中，运用了依赖注入思想完成了<init-param>配置元素的读取。他抽离出HttpServletBean这个类的目的也在于此，就是“以依赖注入的方式来读取Servlet类的<init-param>配置信息”，而且这里很明显是一种setter注入。

明白了HttpServletBean类的设计思想，我们也就知道可以如何从中获益。具体来说，我们继承HttpServletBean类（就像DispatcherServlet做的那样），在

类中定义一个属性，为这个属性加上setter方法后，我们就可以在<init-param>元素中为其定义值。在类被初始化后，值就会被注入进来，我们可以直接使用它，避免了样板式的getInitParameter()方法的使用，而且还免费享有Spring中资源编辑器的功能，可以在web.xml中，通过“classpath:”直接指定类路径下的资源文件。

注意，虽然SpringMVC本身为了后面初始化上下文的方便，使用了字符串来声明和设置contextConfigLocation参数，但是将其声明为Resource类型，同样能够成功获取。鼓励读者们自己继承HttpServletBean写一个测试用的Servlet类，并设置一个参数来调试一下，这样能够帮助你更好的理解获取配置参数的过程。

2.容器上下文的建立

上一篇文章中提到过，SpringMVC使用了Spring容器来容纳自己的配置元素，拥有自己的bean容器上下文。在SpringMVC初始化的过程中，非常关键的一步就是要建立起这个容器上下文，而这个建立上下文的过程，发生在FrameworkServlet类中，由上面init()方法中的initServletBean()方法触发。

```
@Override
        protected final void initServletBean() throws ServletException {
                getServletContext().log("Initializing Spring FrameworkServl
                if (this.logger.isInfoEnabled()) {
                        this.logger.info("FrameworkServlet '" + getServletN
                }
                long startTime = System.currentTimeMillis();

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

                if (this.logger.isInfoEnabled()) {
                        long elapsedTime = System.currentTimeMillis() - sta
```

```
                    this.logger.info("FrameworkServlet '" + getServletN
                                    elapsedTime + " ms");
            }
        }
```

initFrameworkServlet()方法是一个没有任何实现的空方法，除去一些样板式的代码，那么这个initServletBean()方法所做的事情已经非常明白：

```
this.webApplicationContext = initWebApplicationContext();
```

这一句简单直白的代码，道破了FrameworkServlet这个类，在SpringMVC类体系中的设计目的，它是 用来抽离出建立 WebApplicationContext 上下文这个过程的。

initWebApplicationContext()方法，封装了建立Spring容器上下文的整个过程，方法内的逻辑如下：

1. 获取由ContextLoaderListener初始化并注册在ServletContext中的根上下文，记为rootContext
2. 如果webApplicationContext已经不为空，表示这个Servlet类是通过编程式注册到容器中的（Servlet 3.0+中的ServletContext.addServlet() ），上下文也由编程式传入。若这个传入的上下文还没被初始化，将rootContext上下文设置为它的父上下文，然后将其初始化，否则直接使用。
3. 通过wac变量的引用是否为null，判断第2步中是否已经完成上下文的设置（即上下文是否已经用编程式方式传入），如果wac==null成立，说明该Servlet不是由编程式注册到容器中的。此时以contextAttribute属性的值为键，在ServletContext中查找上下文，查找得到，说明上下文已经以别的方式初始化并注册在contextAttribute下，直接使用。
4. 检查wac变量的引用是否为null，如果wac==null成立，说明2、3两步中的上下文初始化策略都没成功，此时调用createWebApplicationContext(rootContext)，建立一个全新的以rootContext为父上下文的上下文，作为SpringMVC配置元素的容器上下文。大多数情况下我们所使用的上下文，就是这个新建的上下文。
5. 以上三种初始化上下文的策略，都会回调onRefresh(ApplicationContext context)方法（回调的方式根据不同策略有不同），onRefresh方法在DispatcherServlet类中被覆写，以上面得到的上下文为依托，完成

SpringMVC中默认实现类的初始化。

6. 最后，将这个上下文发布到ServletContext中，也就是将上下文以一个和Servlet类在web.xml中注册名字有关的值为键，设置为ServletContext的一个属性。你可以通过改变publishContext的值来决定是否发布到ServletContext中，默认为true。

以上面6点跟踪FrameworkServlet类中的代码，可以比较清晰的了解到整个容器上下文的建立过程，也就能够领会到FrameworkServlet类的设计目的，它是用来建立一个和Servlet关联的Spring容器上下文，并将其注册到ServletContext中的。跳脱开SpringMVC体系，我们也能通过继承FrameworkServlet类，得到与Spring容器整合的好处，FrameworkServlet和HttpServletBean一样，是一个可以独立使用的类。整个SpringMVC设计中，处处体现开闭原则，这里显然也是其中一点。

### 3.初始化SpringMVC默认实现类

初始化流程在FrameworkServlet类中流转，建立了上下文后，通过onRefresh(ApplicationContext context)方法的回调，进入到DispatcherServlet类中。

```
@Override
        protected void onRefresh(ApplicationContext context) {
                initStrategies(context);
        }
```

DispatcherServlet类覆写了父类FrameworkServlet中的onRefresh(ApplicationContext context)方法，提供了SpringMVC各种编程元素的初始化。当然这些编程元素，都是作为容器上下文中一个个bean而存在的。具体的初始化策略，在initStrategies()方法中封装。

```
protected void initStrategies(ApplicationContext context) {
                initMultipartResolver(context);
                initLocaleResolver(context);
                initThemeResolver(context);
                initHandlerMappings(context);
                initHandlerAdapters(context);
                initHandlerExceptionResolvers(context);
                initRequestToViewNameTranslator(context);
```

```
                initViewResolvers(context);
                initFlashMapManager(context);
        }
```

我们以其中initHandlerMappings(context)方法为例，分析一下这些SpringMVC编程元素的初始化策略，其他的方法，都是以类似的策略初始化的。

```
private void initHandlerMappings(ApplicationContext context) {
                this.handlerMappings = null;

                if (this.detectAllHandlerMappings) {
                        // Find all HandlerMappings in the ApplicationConte
                        Map<String, HandlerMapping> matchingBeans =
                                        BeanFactoryUtils.beansOfTypeIncludi
                        if (!matchingBeans.isEmpty()) {
                                this.handlerMappings = new ArrayList<Handle
                                // We keep HandlerMappings in sorted order.
                                OrderComparator.sort(this.handlerMappings);
                        }
                }
                else {
                        try {
                                HandlerMapping hm = context.getBean(HANDLER
                                this.handlerMappings = Collections.singleto
                        }
                        catch (NoSuchBeanDefinitionException ex) {
                                // Ignore, we'll add a default HandlerMappi
                        }
                }

                // Ensure we have at least one HandlerMapping, by registeri
                // a default HandlerMapping if no other mappings are found.
                if (this.handlerMappings == null) {
                        this.handlerMappings = getDefaultStrategies(context
                        if (logger.isDebugEnabled()) {
                                logger.debug("No HandlerMappings found in s
                        }
                }
        }
```

detectAllHandlerMappings变量默认为true，所以在初始化HandlerMapping接口默认实现类的时候，会把上下文中所有HandlerMapping类型的Bean都注册在handlerMappings这个List变量中。如果你手工将其设置为false，那么将尝试获取名为handlerMapping的Bean，新建一个只有一个元素的List，将其赋

给handlerMappings。如果经过上面的过程，handlerMappings变量仍为空，那么说明你没有在上下文中提供自己HandlerMapping类型的Bean定义。此时，SpringMVC将采用默认初始化策略来初始化handlerMappings。

点进去getDefaultStrategies看一下。

```
@SuppressWarnings("unchecked")
        protected <T> List<T> getDefaultStrategies(ApplicationContext conte
                String key = strategyInterface.getName();
                String value = defaultStrategies.getProperty(key);
                if (value != null) {
                        String[] classNames = StringUtils.commaDelimitedLis
                        List<T> strategies = new ArrayList<T>(classNames.le
                        for (String className : classNames) {
                                try {
                                        Class<?> clazz = ClassUtils.forName
                                        Object strategy = createDefaultStra
                                        strategies.add((T) strategy);
                                }
                                catch (ClassNotFoundException ex) {
                                        throw new BeanInitializationExcepti
                                                        "Could not find Dis
                                                                        "]
                                }
                                catch (LinkageError err) {
                                        throw new BeanInitializationExcepti
                                                        "Error loading Disp
                                                                        "]
                                }
                        }
                        return strategies;
                }
                else {
                        return new LinkedList<T>();
                }
        }
```

它是一个范型的方法，承担所有SpringMVC编程元素的默认初始化策略。方法的内容比较直白，就是以传递类的名称为键，从defaultStrategies这个Properties变量中获取实现类，然后反射初始化。

需要说明一下的是defaultStrategies变量的初始化，它是在DispatcherServlet的静态初始化代码块中加载的。

```
private static final Properties defaultStrategies;

        static {
                // Load default strategy implementations from properties fi
                // This is currently strictly internal and not meant to be
                // by application developers.
                try {
                        ClassPathResource resource = new ClassPathResource(
                        defaultStrategies = PropertiesLoaderUtils.loadPrope
                }
                catch (IOException ex) {
                        throw new IllegalStateException("Could not load 'Di
                }
        }

private static final String DEFAULT_STRATEGIES_PATH = "DispatcherServlet.pr
```

这个DispatcherServlet.properties里面，以键值对的方式，记录了SpringMVC默认实现类，它在spring-webmvc-3.1.3.RELEASE.jar这个jar包内，在org.springframework.web.servlet包里面。

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

org.springframework.web.servlet.LocaleResolver=org.springframework.web.serv

org.springframework.web.servlet.ThemeResolver=org.springframework.web.servl

org.springframework.web.servlet.HandlerMapping=org.springframework.web.serv
        org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHan

org.springframework.web.servlet.HandlerAdapter=org.springframework.web.serv
        org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,
        org.springframework.web.servlet.mvc.annotation.AnnotationMethodHand

org.springframework.web.servlet.HandlerExceptionResolver=org.springframewor
        org.springframework.web.servlet.mvc.annotation.ResponseStatusExcept
        org.springframework.web.servlet.mvc.support.DefaultHandlerException

org.springframework.web.servlet.RequestToViewNameTranslator=org.springframe

org.springframework.web.servlet.ViewResolver=org.springframework.web.servle

org.springframework.web.servlet.FlashMapManager=org.springframework.web.ser
```

至此，我们分析完了initHandlerMappings(context)方法的执行过程，其他的初始化过程与这个方法非常类似。所有初始化方法执行完后，SpringMVC正式完成初始化，静静等待Web请求的到来。

## 4.总结

回顾整个SpringMVC的初始化流程，我们看到，通过HttpServletBean、FrameworkServlet、DispatcherServlet三个不同的类层次，SpringMVC的设计者将三种不同的职责分别抽象，运用模版方法设计模式分别固定在三个类层次中。其中HttpServletBean完成的是<init-param>配置元素的依赖注入，FrameworkServlet完成的是容器上下文的建立，DispatcherServlet完成的是SpringMVC具体编程元素的初始化策略。

标签： SpringMVC