芋道源码 —— 知识星球

我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2020-02-04
Spring MVC

精尽 Spring MVC 源码分析 —— 容器的初始化（二）之 Servlet WebApplicationContext 容器

1. 概述

本文接 《精尽 Spring MVC 源码分析 —— 容器的初始化（一）之 Root WebApplicationContext 容器》 一文，我们来分享下 Servlet WebApplicationContext 容器的初始化的过程。

在开始之前，我们还是回过头看一眼 web.xml 的配置。代码如下：

```
<servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!-- 可以自定义servlet.xml配置文件的位置和名称，默认为WEB-INF目录下，名称为[<servlet-name>]-servlet.xml，如spring
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring-servlet.xml</param-value> // 默认
    </init-param>
    -->
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

即， Servlet WebApplicationContext 容器的初始化，是在 DispatcherServlet 初始化的过程中执行。

DispatcherServlet 的类图如下：

HttpServletBean ，负责将 ServletConfig 设置到当前 Servlet 对象中。类上的简单注释如下：

```
// HttpServletBean.java

/**
 * Simple extension of {@link javax.servlet.http.HttpServlet} which treats
 * its config parameters ({@code init-param} entries within the
 * {@code servlet} tag in {@code web.xml}) as bean properties.
 */
```

FrameworkServlet ，负责初始化 Spring Servlet WebApplicationContext 容器。类上的简单注释如下：

```
// FrameworkServlet.java

/**
 * Base servlet for Spring's web framework. Provides integration with
 * a Spring application context, in a JavaBean-based overall solution.
 */
```

DispatcherServlet ，负责初始化 Spring MVC 的各个组件，以及处理客户端的请求。类上的简单注释如下：

```
// DispatcherServlet.java

/**
 * Central dispatcher for HTTP request handlers/controllers, e.g. for web UI controllers
 * or HTTP-based remote service exporters. Dispatches to registered handlers for processing
 * a web request, providing convenient mapping and exception handling facilities.
 */
```

每一层的 Servlet 实现类，执行对应负责的逻辑。干净~下面，我们逐个类来进行解析。

2. 如何调试

执行 `DispatcherServletTests#configuredDispatcherServlets()` 单元测试方法，即可执行本文涉及的一些逻辑。

3. HttpServletBean

`org.springframework.web.servlet.HttpServletBean` ，实现 EnvironmentCapable、EnvironmentAware 接口，继承 HttpServlet 抽象类，负责将 ServletConfig 集成到 Spring 中。当然，HttpServletBean 自身也是一个抽象类。

3.1 构造方法

```
// HttpServletBean.java

@Nullable
private ConfigurableEnvironment environment;

/**
 * 必须配置的属性的集合
 *
 * 在 {@link ServletConfigPropertyValues} 中，会校验是否有对应的属性
 */
private final Set<String> requiredProperties = new HashSet<>(4);
```

`environment` 属性，相关的方法，代码如下：

```
// HttpServletBean.java

// setting 方法
@Override // 实现自 EnvironmentAware 接口，自动注入
public void setEnvironment(Environment environment) {
    Assert.isInstanceOf(ConfigurableEnvironment.class, environment, "ConfigurableEnvironment required");
 this.environment = (ConfigurableEnvironment) environment;
}

// getting 方法
@Override // 实现自 EnvironmentCapable 接口
public ConfigurableEnvironment getEnvironment() {
    // 如果 environment 为空，主动创建
 if (this.environment == null) {
     this.environment = createEnvironment();
    }
 return this.environment;
}

protected ConfigurableEnvironment createEnvironment() {
 return new StandardServletEnvironment();
}
```

- 为什么 `environment` 属性，能够被自动注入呢？答案是 EnvironmentAware 接口。具体的源码解析，见 《【死磕 Spring】—— IoC 之深入分析 Aware 接口》 。当然，也可以不看。

`requiredProperties` 属性，必须配置的属性的集合。可通过 `#addRequiredProperty(String property)` 方法，添加到其中。代码如下：

```
// HttpServletBean.java

protected final void addRequiredProperty(String property) {
 this.requiredProperties.add(property);
}
```

3.2 init

`#init()` 方法，负责将 ServletConfig 设置到当前 Servlet 对象中。代码如下：

```
// HttpServletBean.java

@Override
public final void init() throws ServletException {
 // Set bean properties from init parameters.
    // <1> 解析 <init-param /> 标签，封装到 PropertyValues pvs 中
    PropertyValues pvs = new ServletConfigPropertyValues(getServletConfig(), this.requiredProperties);
 if (!pvs.isEmpty()) {
     try {
            // <2.1> 将当前的这个 Servlet 对象，转化成一个 BeanWrapper 对象。从而能够以 Spring 的方式来将 pvs 注入到该
            BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(this);
            ResourceLoader resourceLoader = new ServletContextResourceLoader(getServletContext());
         // <2.2> 注册自定义属性编辑器，一旦碰到 Resource 类型的属性，将会使用 ResourceEditor 进行解析
            bw.registerCustomEditor(Resource.class, new ResourceEditor(resourceLoader, getEnvironment()));
            // <2.3> 空实现，留给子类覆盖
            initBeanWrapper(bw);
         // <2.4> 以 Spring 的方式来将 pvs 注入到该 BeanWrapper 对象中
            bw.setPropertyValues(pvs, true);
        } catch (BeansException ex) {
         if (logger.isErrorEnabled()) {
                logger.error("Failed to set bean properties on servlet '" + getServletName() + "'", ex);
            }
         throw ex;
        }
    }

 // Let subclasses do whatever initialization they like.
    // <3> 子类来实现，实现自定义的初始化逻辑。目前，有具体的代码实现。
    initServletBean();
}
```

<1> 处，解析 Servlet 配置的 <init-param /> 标签，封装到 PropertyValues pvs 中。其中，ServletConfigPropertyValues 是 HttpServletBean 的私有静态类，继承 MutablePropertyValues 类，ServletConfig 的 PropertyValues 封装实现类。代码如下：

```
       // HttpServletBean.java

       private static class ServletConfigPropertyValues extends MutablePropertyValues {

        /**
            * Create new ServletConfigPropertyValues.
            * @param config the ServletConfig we'll use to take PropertyValues from
            * @param requiredProperties set of property names we need, where
            * we can't accept default values
            * @throws ServletException if any required properties are missing
            */
        public ServletConfigPropertyValues(ServletConfig config, Set<String> requiredProperties)
                throws ServletException {
            // 获得缺失的属性的集合
            Set<String> missingProps = (!CollectionUtils.isEmpty(requiredProperties) ?
                 new HashSet<>(requiredProperties) : null);

            // <1> 遍历 ServletConfig 的初始化参数集合，添加到 ServletConfigPropertyValues 中，并从 missingProps 移除
            Enumeration<String> paramNames = config.getInitParameterNames();
            while (paramNames.hasMoreElements()) {
                String property = paramNames.nextElement();
                Object value = config.getInitParameter(property);
             // 添加到 ServletConfigPropertyValues 中
```

```
                addPropertyValue(new PropertyValue(property, value));
            // 从 missingProps 中移除
            if (missingProps != null) {
                missingProps.remove(property);
            }
        }

        // Fail if we are still missing properties.
        // <2> 如果存在缺失的属性，抛出 ServletException 异常
        if (!CollectionUtils.isEmpty(missingProps)) {
            throw new ServletException(
                    "Initialization from ServletConfig for servlet '" + config.getServletName() +
                    "' failed; the following required properties were missing: " +
                    StringUtils.collectionToDelimitedString(missingProps, ", "));
        }
    }

}
```

- 代码简单，实现两方面的逻辑：<1> 处，遍历 ServletConfig 的初始化参数集合，添加到 ServletConfigPropertyValues 中；<2> 处，判断要求的属性是否齐全。如果不齐全，则抛出 ServletException 异常。

<2.1> 处，将当前的这个 Servlet 对象，转化成一个 BeanWrapper 对象。从而能够以 Spring 的方式来将 pvs 注入到该 BeanWrapper 对象中。简单来说，BeanWrapper 是 Spring 提供的一个用来操作 Java Bean 属性的工具，使用它可以直接修改一个对象的属性。

<2.2> 处，注册自定义属性编辑器，一旦碰到 Resource 类型的属性，将会使用 ResourceEditor 进行解析。

<2.3> 处，空实现，留给子类覆盖。代码如下：

```
// HttpServletBean.java

/**
 * Initialize the BeanWrapper for this HttpServletBean,
 * possibly with custom editors.
 * <p>This default implementation is empty.
 * @param bw the BeanWrapper to initialize
 * @throws BeansException if thrown by BeanWrapper methods
 * @see org.springframework.beans.BeanWrapper#registerCustomEditor
 */
protected void initBeanWrapper(BeanWrapper bw) throws BeansException {
}
```

- 然而实际上，子类暂时木有任何实现。

<2.4> 处，以 Spring 的方式来将 pvs 注入到该 BeanWrapper 对象中，技设置到当前 Servlet 对象中。可能比较费解，我们还是举个例子。假设如下：

```
// web.xml

<servlet>
    <servlet-name>spring</servlet-name>
```

```
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring-servlet.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
```

- 此处有配置了 contextConfigLocation 属性，那么通过 <2.4> 处的逻辑，会反射设置到 FrameworkServlet.contextConfigLocation 属性。代码如下：

```
// FrameworkServlet.java

/** Explicit context config location. */
@Nullable
private String contextConfigLocation;

public void setContextConfigLocation(@Nullable String contextConfigLocation) {
 this.contextConfigLocation = contextConfigLocation;
}
```

- 看懂了这波骚操作了么？

<3> 处，调用 #initServletBean() 方法，子类来实现，实现自定义的初始化逻辑。目前，FrameworkServlet 实现类该方法。代码如下：

```
// HttpServletBean.java

protected void initServletBean() throws ServletException {
}
```

- 详细解析，见 「4. FrameworkServlet」 。

4. FrameworkServlet

org.springframework.web.servlet.FrameworkServlet ，实现 ApplicationContextAware 接口，继承 HttpServletBean 抽象类，负责初始化 Spring Servlet WebApplicationContext 容器。同时，FrameworkServlet 自身也是一个抽象类。

4.1 构造方法

FrameworkServlet 的属性还是非常多，我们还是只看部分的关键属性。代码如下：

```
// FrameworkServlet.java

/**
```

```
 * WebApplicationContext implementation class to create.
 *
 * 创建的 WebApplicationContext 类型
 */
private Class<?> contextClass = DEFAULT_CONTEXT_CLASS;

/**
 * Explicit context config location.
 *
 * 配置文件的地址
 */
@Nullable
private String contextConfigLocation;

/**
 * WebApplicationContext for this servlet.
 *
 * WebApplicationContext 对象
 */
@Nullable
private WebApplicationContext webApplicationContext;
```

contextClass 属性，创建的 WebApplicationContext 类型，默认为 DEFAULT_CONTEXT_CLASS 。代码如下：

```
/**
 * Default context class for FrameworkServlet.
 * @see org.springframework.web.context.support.XmlWebApplicationContext
 */
public static final Class<?> DEFAULT_CONTEXT_CLASS = XmlWebApplicationContext.class;
```

- 又是我们熟悉的 XmlWebApplicationContext 类。在上一篇文章的 ContextLoader.properties 配置文件中，我们已经看到咯。

contextConfigLocation 属性，配置文件的地址。例如：/WEB-INF/spring-servlet.xml 。

webApplicationContext 属性，WebApplicationContext 对象，即本文的关键，Servlet WebApplicationContext 容器。它有四种方式进行“创建”。

- 方式一：通过构造方法，代码如下：

```
// FrameworkServlet.java

public FrameworkServlet(WebApplicationContext webApplicationContext) {
 this.webApplicationContext = webApplicationContext;
}
```

  - 通过方法参数 webApplicationContext 。
- 方式二：因为实现 ApplicationContextAware 接口，也可以 Spring 注入。代码如下：

```
// FrameworkServlet.java
```

```
/**
 * If the WebApplicationContext was injected via {@link #setApplicationContext}.
 *
 * 标记 {@link #webApplicationContext} 属性，是否通过 {@link #setApplicationContext(ApplicationContext)}
 */
private boolean webApplicationContextInjected = false;

@Override
public void setApplicationContext(ApplicationContext applicationContext) {
 if (this.webApplicationContext == null && applicationContext instanceof WebApplicationContext) {
     this.webApplicationContext = (WebApplicationContext) applicationContext;
     this.webApplicationContextInjected = true;
    }
}
```

    - 和方式一，是有几分类似的。
  - 方式三：见 #findWebApplicationContext() 方法。
  - 方式四：见 #createWebApplicationContext(WebApplicationContext parent) 方法。

4.2 initServletBean

#initServletBean() 方法，进一步初始化当前 Servlet 对象。实际上，重心在初始化 Servlet WebApplicationContext 容器。代码如下：

```
// FrameworkServlet.java

@Override
protected final void initServletBean() throws ServletException {
 // 打日志
    getServletContext().log("Initializing Spring " + getClass().getSimpleName() + " '" + getServletName() + "'");
 if (logger.isInfoEnabled()) {
        logger.info("Initializing Servlet '" + getServletName() + "'");
    }

 // 记录开始时间
 long startTime = System.currentTimeMillis();

 try {
        // 初始化 WebApplicationContext 对象
     this.webApplicationContext = initWebApplicationContext();
     // 空实现。子类有需要，可以实现该方法，实现自定义逻辑
        initFrameworkServlet();
    } catch (ServletException | RuntimeException ex) {
        logger.error("Context initialization failed", ex);
     throw ex;
    }

 // 打日志
 if (logger.isDebugEnabled()) {
        String value = this.enableLoggingRequestDetails ?
             "shown which may lead to unsafe logging of potentially sensitive data" :
             "masked to prevent unsafe logging of potentially sensitive data";
        logger.debug("enableLoggingRequestDetails='" + this.enableLoggingRequestDetails +
             "': request parameters and headers will be " + value);
    }
```

```
        // 打日志
        if (logger.isInfoEnabled()) {
            logger.info("Completed initialization in " + (System.currentTimeMillis() - startTime) + " ms");
        }
    }
```

<1> 处，调用 #initWebApplicationContext() 方法，初始化 Servlet WebApplicationContext 对象。详细解析，见 「4.3 initWebApplicationContext」 。

<2> 处，调用 #initFrameworkServlet() 方法，空实现。子类有需要，可以实现该方法，实现自定义逻辑。代码如下：

```
// FrameworkServlet.java

/**
 * This method will be invoked after any bean properties have been set and
 * the WebApplicationContext has been loaded. The default implementation is empty;
 * subclasses may override this method to perform any initialization they require.
 * @throws ServletException in case of an initialization exception
 */
protected void initFrameworkServlet() throws ServletException {
}
```

- 然而实际上，并没有子类，对该方法重新实现。

4.3 initWebApplicationContext

#initWebApplicationContext() 方法，初始化 Servlet WebApplicationContext 对象。代码如下：

芮芮提示：这个方法的逻辑并不复杂，但是涉及调用的方法的逻辑比较多。同时，也是本文最最最核心的方法了。

```
// FrameworkServlet.java

protected WebApplicationContext initWebApplicationContext() {
    // <1> 获得根 WebApplicationContext 对象
    WebApplicationContext rootContext = WebApplicationContextUtils.getWebApplicationContext(getServletContext());

    // <2> 获得 WebApplicationContext wac 变量
    WebApplicationContext wac = null;
    // 第一种情况，如果构造方法已经传入 webApplicationContext 属性，则直接使用
    if (this.webApplicationContext != null) {
        // A context instance was injected at construction time -> use it
        // 赋值给 wac 变量
        wac = this.webApplicationContext;
        // 如果是 ConfigurableWebApplicationContext 类型，并且未激活，则进行初始化
        if (wac instanceof ConfigurableWebApplicationContext) {
            ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) wac;
            if (!cwac.isActive()) { // 未激活
                // The context has not yet been refreshed -> provide services such as
                // setting the parent context, setting the application context id, etc
                // 设置 wac 的父 context 为 rootContext 对象
                if (cwac.getParent() == null) {
                    // The context instance was injected without an explicit parent -> set
```

```
                // the root application context (if any; may be null) as the parent
                    cwac.setParent(rootContext);
                }
                // 配置和初始化 wac
                configureAndRefreshWebApplicationContext(cwac);
            }
        }
    }
    // 第二种情况，从 ServletContext 获取对应的 WebApplicationContext 对象
    if (wac == null) {
        // No context instance was injected at construction time -> see if one
        // has been registered in the servlet context. If one exists, it is assumed
        // that the parent context (if any) has already been set and that the
        // user has performed any initialization such as setting the context id
        wac = findWebApplicationContext();
    }
    // 第三种，创建一个 WebApplicationContext 对象
    if (wac == null) {
        // No context instance is defined for this servlet -> create a local one
        wac = createWebApplicationContext(rootContext);
    }

    // <3> 如果未触发刷新事件，则主动触发刷新事件
    if (!this.refreshEventReceived) {
        // Either the context is not a ConfigurableApplicationContext with refresh
        // support or the context injected at construction time had already been
        // refreshed -> trigger initial onRefresh manually here.
        onRefresh(wac);
    }

    // <4> 将 context 设置到 ServletContext 中
    if (this.publishContext) {
        // Publish the context as a servlet context attribute.
        String attrName = getServletContextAttributeName();
        getServletContext().setAttribute(attrName, wac);
    }

    return wac;
}
```

<1> 处，调用 WebApplicationContextUtils#getWebApplicationContext((ServletContext sc) 方法，获得 Root WebApplicationContext 对象，这就是在 《精尽 Spring MVC 源码分析 —— 容器的初始化（一）之 Root WebApplicationContext 容器》 中初始化的呀。代码如下：

```
// WebApplicationContextUtils.java

@Nullable
public static WebApplicationContext getWebApplicationContext(ServletContext sc) {
    return getWebApplicationContext(sc, WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE);
}

@Nullable
public static WebApplicationContext getWebApplicationContext(ServletContext sc, String attrName) {
    Assert.notNull(sc, "ServletContext must not be null");
    Object attr = sc.getAttribute(attrName);
    // ... 省略各种校验的代码
    return (WebApplicationContext) attr;
}
```

- 熟不熟悉，惊喜不惊喜。

<2> 处，获得 WebApplicationContext wac 变量。下面，会分成三种情况。

========== 第一种情况 ==========
如果构造方法已经传入 webApplicationContext 属性，则直接使用。实际上，就是我们在 「4.1 构造方法」 提到的 Servlet WebApplicationContext 容器的第一、二种方式。
实际上，这块代码和 ContextLoader#initWebApplicationContext(ServletContext servletContext) 的中间段是一样的。除了 #configureAndRefreshWebApplicationContext(ConfigurableWebApplicationContext wac) 的具体实现代码不同。详细解析，见 「4.4 configureAndRefreshWebApplicationContext」 。
========== 第二种情况 ==========
这种情况，就是我们在 「4.1 构造方法」 提到的 Servlet WebApplicationContext 容器的第三种方式。

如果此处 wac 还是为空，则调用 #findWebApplicationContext() 方法，从 ServletContext 获取对应的 WebApplicationContext 对象。代码如下：

```
// FrameworkServlet.java

/** ServletContext attribute to find the WebApplicationContext in. */
@Nullable
private String contextAttribute;
@Nullable
public String getContextAttribute() {
 return this.contextAttribute;
}

@Nullable
protected WebApplicationContext findWebApplicationContext() {
    String attrName = getContextAttribute();
 // 需要配置了 contextAttribute 属性下，才会去查找
 if (attrName == null) {
     return null;
    }
 // 从 ServletContext 中，获得属性名对应的 WebApplicationContext 对象
    WebApplicationContext wac = WebApplicationContextUtils.getWebApplicationContext(getServletContext(), attrNa
 // 如果不存在，则抛出 IllegalStateException 异常
 if (wac == null) {
     throw new IllegalStateException("No WebApplicationContext found: initializer not registered?");
    }
 return wac;
}
```

- 一般情况下，我们不会配置 contextAttribute 属性。所以，这段逻辑暂时无视。

========== 第三种情况 ==========
这种情况，就是我们在 「4.1 构造方法」 提到的 Servlet WebApplicationContext 容器的第四种方式。

如果此处 wac 还是为空，则调用 #createWebApplicationContext(WebApplicationContext parent) 方法，创建一个 WebApplicationContext 对象。代码如下：

```
// FrameworkServlet.java

/**
 * WebApplicationContext implementation class to create.
```

```
     *
     * 创建的 WebApplicationContext 类型
     */
    private Class<?> contextClass = DEFAULT_CONTEXT_CLASS;
    public Class<?> getContextClass() {
     return this.contextClass;
    }

    protected WebApplicationContext createWebApplicationContext(@Nullable ApplicationContext parent) {
        // <a> 获得 context 的类
        Class<?> contextClass = getContextClass();
     // 如果非 ConfigurableWebApplicationContext 类型，抛出 ApplicationContextException 异常
     if (!ConfigurableWebApplicationContext.class.isAssignableFrom(contextClass)) {
         throw new ApplicationContextException(
                 "Fatal initialization error in servlet with name '" + getServletName() +
                 "': custom WebApplicationContext class [" + contextClass.getName() +
                 "] is not of type ConfigurableWebApplicationContext");
        }
     // <b> 创建 context 类的对象
        ConfigurableWebApplicationContext wac =
                (ConfigurableWebApplicationContext) BeanUtils.instantiateClass(contextClass);

     // <c> 设置 environment、parent、configLocation 属性
        wac.setEnvironment(getEnvironment());
        wac.setParent(parent);
        String configLocation = getContextConfigLocation();
     if (configLocation != null) {
            wac.setConfigLocation(configLocation);
        }

     // <d> 配置和初始化 wac
        configureAndRefreshWebApplicationContext(wac);

     return wac;
    }
```

- <a> 处，获得 context 的类，即 contextClass 属性。并且，如果非 ConfigurableWebApplicationContext 类型，抛出 ApplicationContextException 异常。
- <b> 处，创建 context 类的对象。
- <c> 处，设置 environment、parent、configLocation 属性。其中，configLocation 是个重要属性。
- <d> 处，调用 #configureAndRefreshWebApplicationContext(ConfigurableWebApplicationContext wac) 方法，配置和初始化 wac 。详细解析，见 「4.4 configureAndRefreshWebApplicationContext」 。

========== END ==========

<3> 处，如果未触发刷新事件，则调用 #onRefresh(ApplicationContext context) 主动触发刷新事件。详细解析，见 「4.5 onRefresh」 中。另外，refreshEventReceived 属性，定义如下：

```
    // FrameworkServlet.java

    /**
     * Flag used to detect whether onRefresh has already been called.
     *
     * 标记是否接收到 ContextRefreshedEvent 事件。即 {@link #onApplicationEvent(ContextRefreshedEvent)}
```

```
     */
    private boolean refreshEventReceived = false;
```

<4> 处，如果 publishContext 为 true 时，则将 context 设置到 ServletContext 中。涉及到的变量和方法如下：

```
    // FrameworkServlet.java

    /**
     * Should we publish the context as a ServletContext attribute?.
     *
     * 是否将 {@link #webApplicationContext} 设置到 {@link ServletContext} 的属性种
     */
    private boolean publishContext = true;

    /**
     * Prefix for the ServletContext attribute for the WebApplicationContext.
     * The completion is the servlet name.
     */
    public static final String SERVLET_CONTEXT_PREFIX = FrameworkServlet.class.getName() + ".CONTEXT.";

    public String getServletContextAttributeName() {
     return SERVLET_CONTEXT_PREFIX + getServletName();
    }

    // HttpServletBean.java

    @Override
    @Nullable
    public String getServletName() {
     return (getServletConfig() != null ? getServletConfig().getServletName() : null);
    }
```

4.4 configureAndRefreshWebApplicationContext

#configureAndRefreshWebApplicationContext(ConfigurableWebApplicationContext wac) 方法，配置和初始化 wac 。代码如下：

```
// FrameworkServlet.java

protected void configureAndRefreshWebApplicationContext(ConfigurableWebApplicationContext wac) {
 // <1> 如果 wac 使用了默认编号，则重新设置 id 属性
    if (ObjectUtils.identityToString(wac).equals(wac.getId())) {
     // The application context id is still set to its original default value
     // -> assign a more useful id based on available information
        // 情况一，使用 contextId 属性
     if (this.contextId != null) {
            wac.setId(this.contextId);
        // 情况二，自动生成
        } else {
         // Generate default id...
            wac.setId(ConfigurableWebApplicationContext.APPLICATION_CONTEXT_ID_PREFIX +
                    ObjectUtils.getDisplayString(getServletContext().getContextPath()) + '/' + getServletName());
        }
```

```
        }

    // <2> 设置 wac 的 servletContext、servletConfig、namespace 属性
        wac.setServletContext(getServletContext());
        wac.setServletConfig(getServletConfig());
        wac.setNamespace(getNamespace());

    // <3> 添加监听器 SourceFilteringListener 到 wac 中
        wac.addApplicationListener(new SourceFilteringListener(wac, new ContextRefreshListener()));

    // <4> TODO 芋艿，暂时忽略
    // The wac environment's #initPropertySources will be called in any case when the context
    // is refreshed; do it eagerly here to ensure servlet property sources are in place for
    // use in any post-processing or initialization that occurs below prior to #refresh
        ConfigurableEnvironment env = wac.getEnvironment();
    if (env instanceof ConfigurableWebEnvironment) {
            ((ConfigurableWebEnvironment) env).initPropertySources(getServletContext(), getServletConfig());
        }

    // <5> 执行处理完 WebApplicationContext 后的逻辑。目前是个空方法，暂无任何实现
        postProcessWebApplicationContext(wac);

    // <6> 执行自定义初始化 context TODO 芋艿，暂时忽略
        applyInitializers(wac);

    // <7> 刷新 wac ，从而初始化 wac
        wac.refresh();
    }
```

实际上，大体逻辑上，和 《精尽 Spring MVC 源码分析 —— 容器的初始化（一）之 Root WebApplicationContext 容器》 的 「4.3 configureAndRefreshWebApplicationContext」 小节是一致的。

【相同】<1> 处，如果 wac 使用了默认编号，则重新设置 id 属性。

【类似】<2> 处，设置 wac 的 servletContext、servletConfig、namespace 属性。

【独有】<3> 处，添加监听器 SourceFilteringListener 到 wac 中。这块的详细解析，见 「4.5 onRefresh」 中。

【相同】<4> 处，TODO 芋艿，暂时忽略。

【独有】<5> 处，执行处理完 WebApplicationContext 后的逻辑。目前是个空方法，暂无任何实现。代码如下：

```
    // FrameworkServlet.java

    protected void postProcessWebApplicationContext(ConfigurableWebApplicationContext wac) {
    }
```

【相同】<6> 处，执行自定义初始化 context TODO 芋艿，暂时忽略。

【相同】<7> 处，刷新 wac ，从而初始化 wac 。

4.5 onRefresh

#onRefresh(ApplicationContext context) 方法，当 Servlet WebApplicationContext 刷新完成后，触发 Spring MVC 组件的初始化。代码如下：

```
// FrameworkServlet.java

/**
 * Template method which can be overridden to add servlet-specific refresh work.
 * Called after successful context refresh.
 * <p>This implementation is empty.
 * @param context the current WebApplicationContext
 * @see #refresh()
 */
protected void onRefresh(ApplicationContext context) {
 // For subclasses: do nothing by default.
}
```

这是一个空方法，具体的实现，在子类 DispatcherServlet 中。代码如下：

```
// DispatcherServlet.java

/**
 * 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) {
    // 初始化 MultipartResolver
    initMultipartResolver(context);
 // 初始化 LocaleResolver
    initLocaleResolver(context);
 // 初始化 ThemeResolver
    initThemeResolver(context);
 // 初始化 HandlerMappings
    initHandlerMappings(context);
 // 初始化 HandlerAdapters
    initHandlerAdapters(context);
 // 初始化 HandlerExceptionResolvers
    initHandlerExceptionResolvers(context);
 // 初始化 RequestToViewNameTranslator
    initRequestToViewNameTranslator(context);
 // 初始化 ViewResolvers
    initViewResolvers(context);
 // 初始化 FlashMapManager
    initFlashMapManager(context);
}
```

- 这里，我们先不深究，在 DispatcherServlet 的初始化的文章中，详细解析。

`#onRefresh()` 方法，有两种方式被触发：

方式一，在 「4.3 initWebApplicationContext」 中，有两种情形，会触发。

- 情形一：情况一 + `wac` 已激活。

- 情形二：情况二。
- 这两种情形，此时 refreshEventReceived 为 false ，所以会顺着 #initWebApplicationContext() 方法的 <3> 的逻辑，调用 #onRefresh() 方法。 貌似说的有点绕，大家自己顺顺。

方式二，在 「4.3 initWebApplicationContext」 中，也有两种情况，会触发。不过相比方式一来说，过程会“曲折”一点。

- 情形一：情况一 + wac 未激活。
- 情形二：情况三。
- 这两种情形，都会调用 #configureAndRefreshWebApplicationContext(ConfigurableWebApplicationContext wac) 方法，在 wac 执行刷新完成后，会回调在该方法中，注册的 SourceFilteringListener 监听器。详细解析，见 「5. SourceFilteringListener」 。

5. SourceFilteringListener

org.springframework.context.event.SourceFilteringListener ，实现 GenericApplicationListener、SmartApplicationListener 监听器，实现将原始对象触发的事件，转发给指定监听器。代码如下：

```
// SourceFilteringListener.java

public class SourceFilteringListener implements GenericApplicationListener, SmartApplicationListener {

    /**
     * 原始类
     */
 private final Object source;
    /**
     * 代理的监听器
     */
 @Nullable
 private GenericApplicationListener delegate;

 /**
     * Create a SourceFilteringListener for the given event source.
     * @param source the event source that this listener filters for,
     * only processing events from this source
     * @param delegate the delegate listener to invoke with event
     * from the specified source
     */
 public SourceFilteringListener(Object source, ApplicationListener<?> delegate) {
     this.source = source;
     this.delegate = (delegate instanceof GenericApplicationListener ?
                 (GenericApplicationListener) delegate : new GenericApplicationListenerAdapter(delegate));
    }

 /**
     * Create a SourceFilteringListener for the given event source,
     * expecting subclasses to override the {@link #onApplicationEventInternal}
     * method (instead of specifying a delegate listener).
     * @param source the event source that this listener filters for,
     * only processing events from this source
     */
 protected SourceFilteringListener(Object source) {
     this.source = source;
    }
```

```
    @Override
    public void onApplicationEvent(ApplicationEvent event) {
        if (event.getSource() == this.source) { // 判断来源
                onApplicationEventInternal(event);
            }
       }

    @Override
    public boolean supportsEventType(ResolvableType eventType) {
        return (this.delegate == null || this.delegate.supportsEventType(eventType));
       }

    @Override
    public boolean supportsEventType(Class<? extends ApplicationEvent> eventType) {
        return supportsEventType(ResolvableType.forType(eventType));
       }

    @Override
    public boolean supportsSourceType(@Nullable Class<?> sourceType) {
        return (sourceType != null && sourceType.isInstance(this.source));
       }

    @Override
    public int getOrder() {
        return (this.delegate != null ? this.delegate.getOrder() : Ordered.LOWEST_PRECEDENCE);
       }


    /**
        * Actually process the event, after having filtered according to the
        * desired event source already.
        * <p>The default implementation invokes the specified delegate, if any.
        * @param event the event to process (matching the specified source)
        */
    protected void onApplicationEventInternal(ApplicationEvent event) {
        if (this.delegate == null) {
            throw new IllegalStateException(
                    "Must specify a delegate object or override the onApplicationEventInternal method");
            }
        this.delegate.onApplicationEvent(event);
       }

}
```

这个类的核心代码，就是 `#onApplicationEvent(ApplicationEvent event)` 方法中，判断事件的来源，就是原始类 `source` 。如果是，则调用 `#onApplicationEventInternal(ApplicationEvent event)` 方法，将事件转发给 `delegate` 监听器。

我们在回看下 `#configureAndRefreshWebApplicationContext(ConfigurableWebApplicationContext wac)` 方法，创建 SourceFilteringListener 对象时，传入的两个参数：

`source` 属性，就是 `wac` 对象。
`delegate` 属性，就是 ContextRefreshListener 对象。

下面，让我们来看看 ContextRefreshListener 具体的代码实现，代码如下：

```
// FrameworkServlet.java

private class ContextRefreshListener implements ApplicationListener<ContextRefreshedEvent> {

 @Override
 public void onApplicationEvent(ContextRefreshedEvent event) {
        FrameworkServlet.this.onApplicationEvent(event);
    }

}
```

ContextRefreshListener 是 FrameworkServlet 的内部类。

在 #onApplicationEvent(ContextRefreshedEvent event) 方法中，会回调 FrameworkServlet#onApplicationEvent(event) 方法，代码如下：

```
// FrameworkServlet.java

public void onApplicationEvent(ContextRefreshedEvent event) {
    // <1> 标记 refreshEventReceived 为 true
 this.refreshEventReceived = true;
 // <2> 处理事件中的 ApplicationContext 对象。这个方法，目前是空实现，由子类 DispatcherServlet 来实现。
    onRefresh(event.getApplicationContext());
}
```

- <1> 处，标记 refreshEventReceived 为 true 。这样，在 #initWebApplicationContext() 方法的 <3> 的逻辑，就不会调用 #onRefresh() 方法。
- <2> 处，调用 #onRefresh(ApplicationContext context) 方法，也就回到了 「4.5 onRefresh」 的逻辑了。

666. 彩蛋

舒服，真舒服。哈哈哈哈~写舒服了。

参考和推荐如下文章：

田小波 《Spring MVC 原理探秘 - 容器的创建过程》
郝佳 《Spring 源码深度解析》 的 「11.3 DispatcherServlet」 小节
韩路彪 《看透 Spring MVC：源代码分析与实践》 的 「第9章 创建 Spring MVC 之器」 小节

文章目录

回到首页