
回到首页

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
2021-01-16
Spring Boot

精尽 Spring Boot 源码分析 —— ServletWebServerApplicationContext

1. 概述

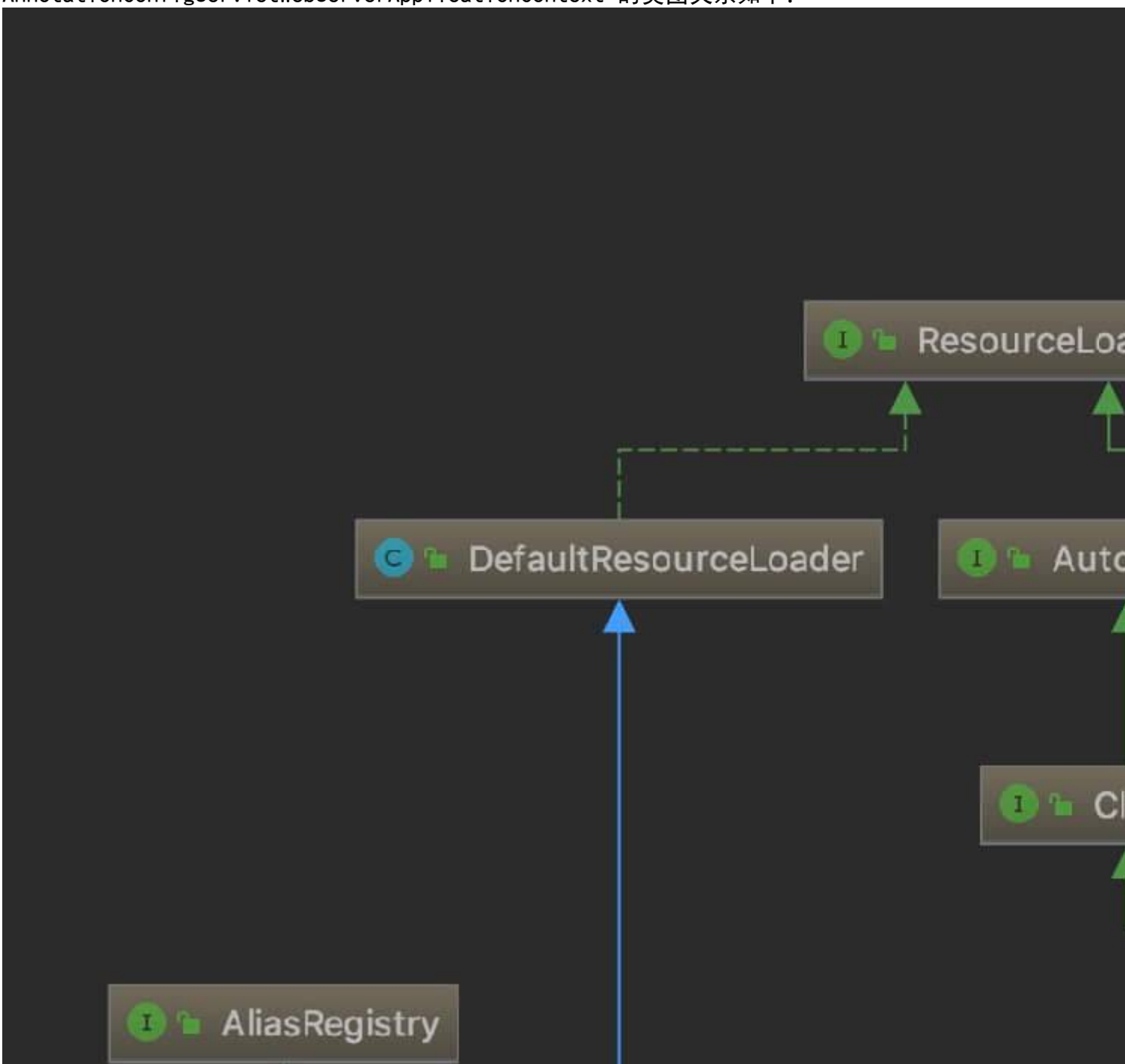
在 《精尽 Spring Boot 源码分析 —— SpringApplication》 一文中，我们看到 SpringApplication#createApplicationContext() 方法，根据不同的 Web 应用类型，创建不同的 Spring 容器。代码如下：

```
// SpringApplication.java

/**
 * The class name of application context that will be used by default for non-web
 * environments.
 */
public static final String DEFAULT_CONTEXT_CLASS = "org.springframework.context."
        + "annotation.AnnotationConfigApplicationContext";

/**
 * The class name of application context that will be used by default for web
 * environments.
 */
public static final String DEFAULT_SERVLET_WEB_CONTEXT_CLASS = "org.springframework.boot."
        + "web.servlet.context.AnnotationConfigServletWebServerApplicationContext";

/**
 * The class name of application context that will be used by default for reactive web
 * environments.
 */
public static final String DEFAULT_REACTIVE_WEB_CONTEXT_CLASS = "org.springframework."
        + "boot.web.reactive.context.AnnotationConfigReactiveWebServerApplicationContext";

protected ConfigurableApplicationContext createApplicationContext() {
    // 根据 webApplicationType 类型，获得 ApplicationContext 类型
    Class<?> contextClass = this.applicationContextClass;
 if (contextClass == null) {
     try {
         switch (this.webApplicationType) {
         case SERVLET:
```

```
                contextClass = Class.forName(DEFAULT_SERVLET_WEB_CONTEXT_CLASS);
            break;
        case REACTIVE:
                contextClass = Class.forName(DEFAULT_REACTIVE_WEB_CONTEXT_CLASS);
            break;
        default:
                contextClass = Class.forName(DEFAULT_CONTEXT_CLASS);
           }
       } catch (ClassNotFoundException ex) {
        throw new IllegalStateException("Unable create a default ApplicationContext, " + "please specify an Applicat
       }
   }
 // 创建 ApplicationContext 对象
 return (ConfigurableApplicationContext) BeanUtils.instantiateClass(contextClass);
}
```

本文，我们要分享的就是，SERVLET 类型对应的 Spring 容器类型 AnnotationConfigServletWebServerApplicationContext 类。

AnnotationConfigServletWebServerApplicationContext 的类图关系如下：



本文，我们只重点看 ServletWebServerApplicationContext 和 AnnotationConfigServletWebServerApplicationContext 类。

为了阅读的友好性，艿艿希望胖友阅读过 《精尽 Spring MVC 源码分析 —— 容器的初始化（三）之 Servlet 3.0 集成》 和 《精尽 Spring MVC 源码分析 —— 容器的初始化（四）之 Spring Boot 集成》 两文。

> 艿艿：厚着脸皮说，上面两篇文章提到的内容，基本就不再赘述。
> 旁白君：真不要脸！

2. ServletWebServerApplicationContext

org.springframework.boot.web.servlet.context.ServletWebServerApplicationContext ，实现 ConfigurableWebServerApplicationContext 接口，继承 GenericWebApplicationContext 类，Spring Boot 使用 Servlet Web 服务器的 ApplicationContext 实现类。

org.springframework.boot.web.servlet.context.ConfigurableWebServerApplicationContext 接口，实现它后，可以获得管理 WebServer 的能力。代码如下：

```
// ConfigurableWebServerApplicationContext.java

public interface ConfigurableWebServerApplicationContext extends ConfigurableApplicationContext, WebServerAppli

 /**
     * Set the server namespace of the context.
     * @param serverNamespace the server namespace
     * @see #getServerNamespace()
     */
 void setServerNamespace(String serverNamespace);

}
```

- org.springframework.context.ConfigurableApplicationContext ，是 Spring Framework 提供的类，就不细看了。
- org.springframework.boot.web.context.WebServerApplicationContext ，继承 ApplicationContext 接口，WebServer ApplicationContext 接口。代码如下：

```
// WebServerApplicationContext.java

public interface WebServerApplicationContext extends ApplicationContext {

 /**
     * Returns the {@link WebServer} that was created by the context or {@code null} if
     * the server has not yet been created.
     * @return the web server
     */
 WebServer getWebServer();

 /**
     * Returns the namespace of the web server application context or {@code null} if no
     * namespace has been set. Used for disambiguation when multiple web servers are
     * running in the same application (for example a management context running on a
     * different port).
```

```
     * @return the server namespace
     */
    String getServerNamespace();

}
```

- 重点是，可以获得 WebServer 的方法。 因为获得它，可以做各种 WebServer 的管理。

org.springframework.web.context.support.GenericWebApplicationContext ，是 Spring Framework 提供的类，就不细看啦。

2.1 构造方法

```
// ServletWebServerApplicationContext.java

/**
 * Constant value for the DispatcherServlet bean name. A Servlet bean with this name
 * is deemed to be the "main" servlet and is automatically given a mapping of "/" by
 * default. To change the default behavior you can use a
 * {@link ServletRegistrationBean} or a different bean name.
 */
public static final String DISPATCHER_SERVLET_NAME = "dispatcherServlet";

/**
 * Spring  WebServer 对象
 */
private volatile WebServer webServer;
/**
 * Servlet ServletConfig 对象
 */
private ServletConfig servletConfig;
/**
 * 通过 {@link #setServerNamespace(String)} 注入。
 *
 * 不过貌似，一直没被注入过，可以暂时先无视
 */
private String serverNamespace;

public ServletWebServerApplicationContext() {
}

public ServletWebServerApplicationContext(DefaultListableBeanFactory beanFactory) {
    super(beanFactory);
}
```

简单看看即可。

因为后续的逻辑，涉及到 Spring 容器的初始化的生命周期，所以我们来简单看看 `AbstractApplicationContext#refresh()` 的方法。代码如下：

```
// AbstractApplicationContext.java
// `#refresh()` 方法
```

```
// ... 简化版代码

// Allows post-processing of the bean factory in context subclasses.
postProcessBeanFactory(beanFactory); // <1>

// Invoke factory processors registered as beans in the context.
invokeBeanFactoryPostProcessors(beanFactory);

// Register bean processors that intercept bean creation.
registerBeanPostProcessors(beanFactory);

// Initialize message source for this context.
initMessageSource();

// Initialize event multicaster for this context.
initApplicationEventMulticaster();

// Initialize other special beans in specific context subclasses.
onRefresh(); // <2>

// Check for listener beans and register them.
registerListeners();

// Instantiate all remaining (non-lazy-init) singletons.
finishBeanFactoryInitialization(beanFactory);

// Last step: publish corresponding event.
finishRefresh(); // <3>
```

这个方法，会被覆写。具体可以看 「2.2 refresh」 小节。但是，即使覆写了，还是会调用该方法。

<1> 处，调用 `#postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)` 方法，具体可以看 「2.3 postProcessBeanFactory」 小节。

> 对 Spring BeanFactoryPostProcessor 的机制，可以看看 《【死磕 Spring】—— IoC 之深入分析 BeanFactoryPostProcessor》

<2> 处，调用 `#onRefresh()` 方法，具体可以看 「2.4 onRefresh」 小节。

<3> 处，调用 `#finishRefresh()` 方法，具体可以看 「2.5 finishRefresh」 小节。

2.2 refresh

覆写 `#refresh()` 方法，初始化 Spring 容器。代码如下：

```
// ServletWebServerApplicationContext.java

@Override
public final void refresh() throws BeansException, IllegalStateException {
 try {
     super.refresh();
    } catch (RuntimeException ex) {
     // <X> 如果发生异常，停止 WebServer
        stopAndReleaseWebServer();
```

```
            throw ex;
        }
    }
```

主要是 <X> 处，如果发生异常，则调用 #stopAndReleaseWebServer() 方法，停止 WebServer。详细解析，见 「2.2.1 stopAndReleaseWebServer」 。

2.2.1 stopAndReleaseWebServer

#stopAndReleaseWebServer() 方法，停止 WebServer。代码如下：

```
// ServletWebServerApplicationContext.java

private void stopAndReleaseWebServer() {
    // 获得 WebServer 对象，避免被多线程修改了
    WebServer webServer = this.webServer;
    if (webServer != null) {
        try {
            // 停止 WebServer 对象
            webServer.stop();
            // 置空 webServer
            this.webServer = null;
        } catch (Exception ex) {
            throw new IllegalStateException(ex);
        }
    }
}
```

2.3 postProcessBeanFactory

覆写 #postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) 方法，代码如下：

```
// ServletWebServerApplicationContext.java

@Override
protected void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {
 // <1.1> 注册 WebApplicationContextServletContextAwareProcessor
    beanFactory.addBeanPostProcessor(new WebApplicationContextServletContextAwareProcessor(this));
 // <1.2> 忽略 ServletContextAware 接口。
    beanFactory.ignoreDependencyInterface(ServletContextAware.class);

 // <2> 注册 ExistingWebApplicationScopes
    registerWebApplicationScopes();
}
```

<1.1> 处，注册 WebApplicationContextServletContextAwareProcessor 。WebApplicationContextServletContextAwareProcessor 的作用，主要是处理实现 ServletContextAware 接口的 Bean 。在这个处理类，初始化这个 Bean 中的 ServletContext 属性，这样在实现 ServletContextAware 接口的 Bean 中就可以拿到 ServletContext 对象了，Spring 中 Aware 接口就是这样实现的。代码如下：

```
// WebApplicationContextServletContextAwareProcessor.java

public class WebApplicationContextServletContextAwareProcessor extends ServletContextAwareProcessor {

 private final ConfigurableWebApplicationContext webApplicationContext;

 public WebApplicationContextServletContextAwareProcessor(ConfigurableWebApplicationContext webApplicationConte
        Assert.notNull(webApplicationContext, "WebApplicationContext must not be null");
     this.webApplicationContext = webApplicationContext;
    }

 @Override
 protected ServletContext getServletContext() {
        ServletContext servletContext = this.webApplicationContext.getServletContext();
     return (servletContext != null) ? servletContext : super.getServletContext();
    }

 @Override
 protected ServletConfig getServletConfig() {
        ServletConfig servletConfig = this.webApplicationContext.getServletConfig();
     return (servletConfig != null) ? servletConfig : super.getServletConfig();
    }

}
```

- 这样，就可以从 `webApplicationContext` 中，获得 ServletContext 和 ServletConfig 属性。

<1.2> 处，忽略 ServletContextAware 接口，因为实现 ServletContextAware 接口的 Bean 在 <1.1> 中的 WebApplicationContextServletContextAwareProcessor 中已经处理了。

关于 <1.1> 和 <1.2> 处的说明，参考 《Spring Boot 源码3 —— refresh ApplicationContext》 文章。

> 芳芳：当读源码碰到困难时，也要善用搜索引擎，去寻找答案。 毕竟，有时候脑子不一定能快速想的明白。哈哈哈哈~

<2> 处，调用 `#registerWebApplicationScopes()` 方法，注册 ExistingWebApplicationScopes 。代码如下：

```
// ServletWebServerApplicationContext.java

private void registerWebApplicationScopes() {
    // 创建 ExistingWebApplicationScopes 对象
    ExistingWebApplicationScopes existingScopes = new ExistingWebApplicationScopes(getBeanFactory());
    // 注册 ExistingWebApplicationScopes 到 WebApplicationContext 中
    WebApplicationContextUtils.registerWebApplicationScopes(getBeanFactory());
    // 恢复
    existingScopes.restore();
}
```

- 可以先不细研究~

2.4 onRefresh

覆写 `#onRefresh()` 方法，在容器初始化时，完成 WebServer 的创建（不包括启动）。代码如下：

```
// ServletWebServerApplicationContext.java

@Override
protected void onRefresh() {
    // <1> 调用父方法
    super.onRefresh();
    try {
        // 创建 WebServer
        createWebServer();
    } catch (Throwable ex) {
        throw new ApplicationContextException("Unable to start web server", ex);
    }
}
```

<1> 处，调用父 `#onRefresh()` 方法，执行父逻辑。这块，暂时不用了解。
<2> 处，调用 `#createWebServer()` 方法，创建 WebServer 对象。详细解析，见 「2.4.1 createWebServer」 。

2.4.1 createWebServer

`#createWebServer()` 方法，创建 WebServer 对象。

```
// ServletWebServerApplicationContext.java

private void createWebServer() {
    WebServer webServer = this.webServer;
    ServletContext servletContext = getServletContext();
    // <1> 如果 webServer 为空，说明未初始化
    if (webServer == null && servletContext == null) {
        // <1.1> 获得 ServletWebServerFactory 对象
        ServletWebServerFactory factory = getWebServerFactory();
        // <1.2> 获得 ServletContextInitializer 对象
        // <1.3> 创建（获得） WebServer 对象
        this.webServer = factory.getWebServer(getSelfInitializer());
    // TODO 1002 芋艿这个情况是？
    } else if (servletContext != null) {
        try {
            getSelfInitializer().onStartup(servletContext);
        } catch (ServletException ex) {
            throw new ApplicationContextException("Cannot initialize servlet context", ex);
        }
    }
    // <3> 初始化 PropertySource
    initPropertySources();
}
```

<1> 处，如果 webServer 为空，说明未初始化。

- <1.1> 处，调用 `#getWebServerFactory()` 方法，获得 ServletWebServerFactory 对象。代码如下：

```
// ServletWebServerApplicationContext.java

protected ServletWebServerFactory getWebServerFactory() {
```

```
        // Use bean names so that we don't consider the hierarchy
        // 获得 ServletWebServerFactory 类型对应的 Bean 的名字们
        String[] beanNames = getBeanFactory().getBeanNamesForType(ServletWebServerFactory.class);
        // 如果是 0 个，抛出 ApplicationContextException 异常，因为至少要一个
        if (beanNames.length == 0) {
            throw new ApplicationContextException("Unable to start ServletWebServerApplicationContext due to
        }
        // 如果是 > 1 个，抛出 ApplicationContextException 异常，因为不知道初始化哪个
        if (beanNames.length > 1) {
            throw new ApplicationContextException("Unable to start ServletWebServerApplicationContext due to
        }
        // 获得 ServletWebServerFactory 类型对应的 Bean 对象
        return getBeanFactory().getBean(beanNames[0], ServletWebServerFactory.class);
    }
```

- 默认情况下，此处返回的会是 `org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory` 对象。
- 在我们引入 `spring-boot-starter-web` 依赖时，默认会引入 `spring-boot-starter-tomcat` 依赖。此时，`org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryConfiguration` 在自动配置时，会配置出 TomcatServletWebServerFactory Bean 对象。因此，此时会获得 TomcatServletWebServerFactory 对象。

- <1.2> 处，调用 `#getSelfInitializer()` 方法，获得 ServletContextInitializer 对象。代码如下：

```
// ServletWebServerApplicationContext.java

private org.springframework.boot.web.servlet.ServletContextInitializer getSelfInitializer() {
 return this::selfInitialize; // 和下面等价
//        return new ServletContextInitializer() {
//
//            @Override
//            public void onStartup(ServletContext servletContext) throws ServletException {
//                selfInitialize(servletContext);
//            }
//
//        };
}
```

- 嘻嘻，返回的是 ServletContextInitializer 匿名对象，内部会调用 `#selfInitialize(servletContext)` 方法。该方法会在 WebServer 创建后，进行初始化。详细解析，见 「2.4.2 finishRefresh」 小节。

- <1.3> 处，调用 `ServletWebServerFactory#getWebServer(ServletContextInitializer)` 方法，创建（获得） WebServer 对象。在这个过程中，会调用 「2.4.2 selfInitialize」 方法。

至此，和 《精尽 Spring MVC 源码分析 —— 容器的初始化（四）之 Spring Boot 集成》 文章，基本是能穿起来了。

<2> 处，TODO 1002 不知道原因。有知道的胖友，星球里告知下哟。

<3> 处，调用父 `#initPropertySources()` 方法，初始化 PropertySource 。

2.4.2 selfInitialize

`#selfInitialize()` 方法，初始化 WebServer 。代码如下：

```
// ServletWebServerApplicationContext.java

private void selfInitialize(ServletContext servletContext) throws ServletException {
    // <1> 添加 Spring 容器到 servletContext 属性中。
    prepareWebApplicationContext(servletContext);
    // <2> 注册 ServletContextScope
    registerApplicationScope(servletContext);
    // <3> 注册 web-specific environment beans ("contextParameters", "contextAttributes")
    WebApplicationContextUtils.registerEnvironmentBeans(getBeanFactory(), servletContext);
    // <4> 获得所有 ServletContextInitializer ，并逐个进行启动
    for (ServletContextInitializer beans : getServletContextInitializerBeans()) {
        beans.onStartup(servletContext);
    }
}
```

<1> 处，调用 #prepareWebApplicationContext(ServletContext servletContext) 方法，添加 Spring 容器到 servletContext 属性中。代码如下：

```
// ServletWebServerApplicationContext.java

protected void prepareWebApplicationContext(ServletContext servletContext) {
    // 如果已经在 ServletContext 中，则根据情况进行判断。
    Object rootContext = servletContext.getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBU
    if (rootContext != null) {
        // 如果是相同容器，抛出 IllegalStateException 异常。说明可能有重复的 ServletContextInitializers 。
        if (rootContext == this) {
            throw new IllegalStateException("Cannot initialize context because there is already a root applicat
        }
        // 如果不同容器，则直接返回。
        return;
    }
    Log logger = LogFactory.getLog(ContextLoader.class);
    servletContext.log("Initializing Spring embedded WebApplicationContext");
    try {
        // <X> 设置当前 Spring 容器到 ServletContext 中
        servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, this);
        // 打印日志
        if (logger.isDebugEnabled()) {
            logger.debug("Published root WebApplicationContext as ServletContext attribute with name [" + WebAp
        }
        // <Y> 设置到 `servletContext` 属性中。
        setServletContext(servletContext);
        // 打印日志
        if (logger.isInfoEnabled()) {
            long elapsedTime = System.currentTimeMillis() - getStartupDate();
            logger.info("Root WebApplicationContext: initialization completed in " + elapsedTime + " ms");
        }
    } catch (RuntimeException | Error ex) {
        logger.error("Context initialization failed", ex);
        servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, ex);
        throw ex;
    }
}
```

- 虽然代码非常长，但是核心在 <X> 和 <Y> 处。
- 通过 <X> 处，从 servletContext 的属性种，可以拿到其拥有的 Spring 容器。
- 通过 <Y> 处，Spring 容器的 servletContext 属性，可以拿到 ServletContext 对象。

<2> 处，调用 #registerApplicationScope(ServletContext servletContext) 方法，注册 ServletContextScope 。代码如下：

```
// ServletWebServerApplicationContext.java

private void registerApplicationScope(ServletContext servletContext) {
    ServletContextScope appScope = new ServletContextScope(servletContext);
    getBeanFactory().registerScope(WebApplicationContext.SCOPE_APPLICATION, appScope);
 // Register as ServletContext attribute, for ContextCleanupListener to detect it.
    servletContext.setAttribute(ServletContextScope.class.getName(), appScope);
}
```

- 不用细了解。

<3> 处，调用 WebApplicationContextUtils#registerEnvironmentBeans(ConfigurableListableBeanFactory bf, ServletContext sc) 方法，注册 web-specific environment beans ("contextParameters", "contextAttributes") 。这样，从 BeanFactory 中，也可以获得到 servletContext 。当然，也可以暂时不用细了解。

<4> 处，获得所有 ServletContextInitializer ，并逐个进行启动。关于这块的解析，我们在 《精尽 Spring MVC 源码分析 —— 容器的初始化（四）之 Spring Boot 集成》 中，已经详细写到。
至此，内嵌的 Servlet Web 服务器，已经能够被请求了。

2.5 finishRefresh

覆写 #finishRefresh() 方法，在容器初始化完成时，启动 WebServer 。代码如下：

```
// ServletWebServerApplicationContext.java

@Override
protected void finishRefresh() {
    // <1> 调用父方法
    super.finishRefresh();
    // <2> 启动 WebServer
    WebServer webServer = startWebServer();
    // <3> 如果创建 WebServer 成功，发布 ServletWebServerInitializedEvent 事件
    if (webServer != null) {
        publishEvent(new ServletWebServerInitializedEvent(webServer, this));
    }
}
```

<1> 处，调用 #finishRefresh() 方法，执行父逻辑。这块，暂时不用了解。
<2> 处，调用 #startWebServer() 方法，启动 WebServer 。详细解析，见 「2.5.1 startWebServer」 。
<3> 处，如果创建 WebServer 成功，发布 ServletWebServerInitializedEvent 事件。

2.5.1 startWebServer

#startWebServer() 方法，启动 WebServer 。代码如下：

```
// ServletWebServerApplicationContext.java
```

```
private WebServer startWebServer() {
    WebServer webServer = this.webServer;
 if (webServer != null) {
        webServer.start();
    }
 return webServer;
}
```

2.6 onClose

覆写 `#onClose()` 方法，在 Spring 容器被关闭时，关闭 WebServer 。代码如下：

```
// ServletWebServerApplicationContext.java

@Override
protected void onClose() {
    // 调用父方法
    super.onClose();
    // 停止 WebServer
    stopAndReleaseWebServer();
}
```

3. AnnotationConfigServletWebServerApplicationCont

`org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext` ，继承 ServletWebServerApplicationContext 类，实现 AnnotationConfigRegistry 接口，进一步提供了两个功能：

芮芮：不过一般情况下，我们用不到这两个功能。简单看了下，更多的是单元测试，需要使用到这两个功能。

从指定的 `basePackages` 包中，扫描 BeanDefinition 们。
从指定的 `annotatedClasses` 注解的配置类（Configuration）中，读取 BeanDefinition 们。

所以啊，这类，简单看看就成啦。

3.1 构造方法

```
// AnnotationConfigServletWebServerApplicationContext.java

private final AnnotatedBeanDefinitionReader reader;

private final ClassPathBeanDefinitionScanner scanner;

/**
 * 需要被 {@link #reader} 读取的注册类们
 */
```

```
private final Set<Class<?>> annotatedClasses = new LinkedHashSet<>();

/**
 * 需要被 {@link #scanner} 扫描的包
 */
private String[] basePackages;

public AnnotationConfigServletWebServerApplicationContext() {
    this.reader = new AnnotatedBeanDefinitionReader(this);
    this.scanner = new ClassPathBeanDefinitionScanner(this);
}

public AnnotationConfigServletWebServerApplicationContext(DefaultListableBeanFactory beanFactory) {
    super(beanFactory);
    this.reader = new AnnotatedBeanDefinitionReader(this);
    this.scanner = new ClassPathBeanDefinitionScanner(this);
}

public AnnotationConfigServletWebServerApplicationContext(Class<?>... annotatedClasses) {
    this();
    // <1> 注册指定的注解的类们
    register(annotatedClasses);
    // 初始化 Spring 容器
    refresh();
}

public AnnotationConfigServletWebServerApplicationContext(String... basePackages) {
    this();
    // <2> 扫描指定包
    scan(basePackages);
    // 初始化 Spring 容器
    refresh();
}
```

<1> 处，如果已经传入 annotatedClasses 参数，则调用 #register(Class<?>... annotatedClasses) 方法，设置到 annotatedClasses 中。然后，调用 #refresh() 方法，初始化 Spring 容器。代码如下：

```
// AnnotationConfigServletWebServerApplicationContext.java

@Override // 实现自 AnnotationConfigRegistry 接口
public final void register(Class<?>... annotatedClasses) {
    Assert.notEmpty(annotatedClasses, "At least one annotated class must be specified");
 this.annotatedClasses.addAll(Arrays.asList(annotatedClasses));
}
```

<2> 处，如果已经传入 basePackages 参数，则调用 #scan(String... basePackages) 方法，设置到 annotatedClasses 中。然后，调用 #refresh() 方法，初始化 Spring 容器。代码如下：

```
// AnnotationConfigServletWebServerApplicationContext.java

@Override
public final void scan(String... basePackages) {
    Assert.notEmpty(basePackages, "At least one base package must be specified");
 this.basePackages = basePackages;
}
```

3.2 prepareRefresh

覆写 `#prepareRefresh()` 方法，代码如下：

```
// AnnotationConfigServletWebServerApplicationContext.java

@Override // 实现自 AbstractApplicationContext 抽象类
protected void prepareRefresh() {
    // 清空 scanner 的缓存
    this.scanner.clearCache();
    // 调用父类
    super.prepareRefresh();
}
```

在 Spring 容器初始化前，需要清空 `scanner` 的缓存。

3.3 postProcessBeanFactory

覆写 `#postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)` 方法，执行 BeanDefinition 的读取。代码如下：

```
// AnnotationConfigServletWebServerApplicationContext.java

@Override
protected void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {
    // 调用父类
    super.postProcessBeanFactory(beanFactory);
    // 扫描指定的包
    if (this.basePackages != null && this.basePackages.length > 0) {
        this.scanner.scan(this.basePackages);
    }
    // 注册指定的注解的类们定的
    if (!this.annotatedClasses.isEmpty()) {
        this.reader.register(ClassUtils.toClassArray(this.annotatedClasses));
    }
}
```

实际场景下，`this.basePackages` 和 `annotatedClasses` 都是空的。所以呢，哈哈哈哈，AnnotationConfigServletWebServerApplicationContext 基本没啥子用~

666. 彩蛋

简单小文一篇~很妥~

参考和推荐如下文章：

oldflame-Jm

- 《Spring boot 源码分析-AnnotationConfigApplicationContext 非 web 环境下的启动容器（2）》
- 《Spring boot 源码分析-AnnotationConfigEmbeddedWebApplicationContext 默认 web 环境下的启动容器（3）》

文章目录

回到首页