参考：<https://www.cnkirito.moe/servlet-explore/?utm_source=tuicool&utm_medium=referral>

# Spring揭秘--寻找遗失的web.xml

原创

[2018-05-04](https://www.cnkirito.moe/servlet-explore/)

[SPRING](https://www.cnkirito.moe/categories/Spring/)

今天我们来放松下心情，不聊分布式，云原生，来聊一聊初学者接触的最多的 java web 基础。几乎所有人都是从 servlet，jsp，filter 开始编写自己的第一个 hello world 工程。那时，还离不开 web.xml 的配置，在 xml 文件中编写繁琐的 servlet 和 filter 的配置。随着 spring 的普及，配置逐渐演变成了两种方式—java configuration 和 xml 配置共存。现如今，springboot 的普及，java configuration 成了主流，xml 配置似乎已经“灭绝”了。不知道你有没有好奇过，这中间都发生了哪些改变，web.xml 中的配置项又是被什么替代项取代了？  
  
[](http://ov0zuistv.bkt.clouddn.com/servlet.png)servlet

### servlet3.0 以前的时代

为了体现出整个演进过程，还是来回顾下 n 年前我们是怎么写 servlet 和 filter 代码的。

项目结构（本文都采用 maven 项目结构）

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | . ├── pom.xml ├── src     ├── main     │   ├── java     │   │   └── moe     │   │   └── cnkirito     │   │   ├── filter     │   │   │   └── HelloWorldFilter.java     │   │   └── servlet     │   │   └── HelloWorldServlet.java     │   └── resources     │   └── WEB-INF     │   └── web.xml     └── test     └── java |

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | **public** **class** **HelloWorldServlet** **extends** **HttpServlet** {   @Override  **protected** **void** **doGet**(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  resp.setContentType("text/plain");  PrintWriter out = resp.getWriter();  out.println("hello world");  }  } |

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | **public** **class** **HelloWorldFilter** **implements** **Filter** {   @Override  **public** **void** **init**(FilterConfig filterConfig) **throws** ServletException {   }   @Override  **public** **void** **doFilter**(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) **throws** IOException, ServletException {  System.out.println("触发 hello world 过滤器...");  filterChain.doFilter(servletRequest,servletResponse);  }   @Override  **public** **void** **destroy**() {   } } |

别忘了在 web.xml 中配置 servlet 和 filter

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 | <?xml version="1.0" encoding="UTF-8"?> <web-app xmlns="http://java.sun.com/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/javaee/web-app\_4\_0.xsd"  version="4.0">   <servlet>  <servlet-name>HelloWorldServlet</servlet-name>  <servlet-class>moe.cnkirito.servlet.HelloWorldServlet</servlet-class>  </servlet>   <servlet-mapping>  <servlet-name>HelloWorldServlet</servlet-name>  <url-pattern>/hello</url-pattern>  </servlet-mapping>   <filter>  <filter-name>HelloWorldFilter</filter-name>  <filter-class>moe.cnkirito.filter.HelloWorldFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>HelloWorldFilter</filter-name>  <url-pattern>/hello</url-pattern>  </filter-mapping>  </web-app> |

这样，一个 java web hello world 就完成了。当然，本文不是 servlet 的入门教程，只是为了对比。

### servlet3.0 新特性

[](http://ov0zuistv.bkt.clouddn.com/servlet_3.0.jpg)servlet\_3.0

Servlet 3.0 作为 Java EE 6 规范体系中一员，随着 Java EE 6 规范一起发布。该版本在前一版本（Servlet 2.5）的基础上提供了若干新特性用于简化 Web 应用的开发和部署。其中一项新特性便是提供了无 xml 配置的特性。

servlet3.0 首先提供了 @WebServlet，@WebFilter 等注解，这样便有了抛弃 web.xml 的第一个途径，凭借注解声明 servlet 和 filter 来做到这一点。

除了这种方式，servlet3.0 规范还提供了更强大的功能，可以在运行时动态注册 servlet ，filter，listener。以 servlet 为例，过滤器与监听器与之类似。ServletContext 为动态配置 Servlet 增加了如下方法：

* ServletRegistration.Dynamic addServlet(String servletName,Class<? extends Servlet> servletClass)
* ServletRegistration.Dynamic addServlet(String servletName, Servlet servlet)
* ServletRegistration.Dynamic addServlet(String servletName, String className)
* T createServlet(Class clazz)
* ServletRegistration getServletRegistration(String servletName)
* Map<String,? extends ServletRegistration> getServletRegistrations()

其中前三个方法的作用是相同的，只是参数类型不同而已；通过 createServlet() 方法创建的 Servlet，通常需要做一些自定义的配置，然后使用 addServlet() 方法来将其动态注册为一个可以用于服务的 Servlet。两个 getServletRegistration() 方法主要用于动态为 Servlet 增加映射信息，这等价于在 web.xml 中使用 标签为存在的 Servlet 增加映射信息。

以上 ServletContext 新增的方法要么是在 ServletContextListener 的 contexInitialized 方法中调用，要么是在 ServletContainerInitializer 的 onStartup() 方法中调用。

ServletContainerInitializer 也是 Servlet 3.0 新增的一个接口，容器在启动时使用 JAR 服务 API(JAR Service API) 来发现 ServletContainerInitializer 的实现类，并且容器将 WEB-INF/lib 目录下 JAR 包中的类都交给该类的 onStartup() 方法处理，我们通常需要在该实现类上使用 @HandlesTypes 注解来指定希望被处理的类，过滤掉不希望给 onStartup() 处理的类。

一个典型的 servlet3.0+ 的 web 项目结构如下：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | . ├── pom.xml └── src  ├── main  │   ├── java  │   │   └── moe  │   │   └── cnkirito  │   │   ├── CustomServletContainerInitializer.java  │   │   ├── filter  │   │   │   └── HelloWorldFilter.java  │   │   └── servlet  │   │   └── HelloWorldServlet.java  │   └── resources  │   └── META-INF  │   └── services  │   └── javax.servlet.ServletContainerInitializer  └── test  └── java |

我并未对 HelloWorldServlet 和 HelloWorldFilter 做任何改动，而是新增了一个 CustomServletContainerInitializer ,它实现了 javax.servlet.ServletContainerInitializer 接口，用来在 web 容器启动时加载指定的 servlet 和 filter，代码如下：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 | **public** **class** **CustomServletContainerInitializer** **implements** **ServletContainerInitializer** {   **private** **final** **static** String JAR\_HELLO\_URL = "/hello";   @Override  **public** **void** **onStartup**(Set<Class<?>> c, ServletContext servletContext) {   System.out.println("创建 helloWorldServlet...");   ServletRegistration.Dynamic servlet = servletContext.addServlet(  HelloWorldServlet.class.getSimpleName(),  HelloWorldServlet.class);  servlet.addMapping(JAR\_HELLO\_URL);   System.out.println("创建 helloWorldFilter...");   FilterRegistration.Dynamic filter = servletContext.addFilter(  HelloWorldFilter.class.getSimpleName(), HelloWorldFilter.class);   EnumSet<DispatcherType> dispatcherTypes = EnumSet.allOf(DispatcherType.class);  dispatcherTypes.add(DispatcherType.REQUEST);   dispatcherTypes.add(DispatcherType.FORWARD);    filter.addMappingForUrlPatterns(dispatcherTypes, **true**, JAR\_HELLO\_URL);   } } |

对上述代码进行一些解读。ServletContext 我们称之为 servlet 上下文，它维护了整个 web 容器中注册的 servlet，filter，listener，以 servlet 为例，可以使用 servletContext.addServlet 等方法来添加 servlet。而方法入参中 Set<Class<?>> c 和 @HandlesTypes 注解在 demo 中我并未使用，感兴趣的朋友可以 debug 看看到底获取了哪些 class ，一般正常的流程是使用 @HandlesTypes 指定需要处理的 class，而后对 Set<Class<?>> 进行判断是否属于该 class，正如前文所言，onStartup 会加载不需要被处理的一些 class。

这么声明一个 ServletContainerInitializer 的实现类，web 容器并不会识别它，所以，需要借助 SPI 机制来指定该初始化类，这一步骤是通过在项目路径下创建 META-INF/services/javax.servlet.ServletContainerInitializer 来做到的，它只包含一行内容：

|  |  |
| --- | --- |
| 1 | moe.cnkirito.CustomServletContainerInitializer |

使用 ServletContainerInitializer 和 SPI 机制，我们的 web 应用便可以彻底摆脱 web.xml 了。

### Spring 是如何支持 servlet3.0 的？

回到我们的 spring 全家桶，可能已经忘了具体是什么时候开始不写 web.xml 了，我只知道现在的项目已经再也看不到它了，spring 又是如何支持 servlet3.0 规范的呢？

寻找 spring 中 ServletContainerInitializer 的实现类并不困难，可以迅速定位到 SpringServletContainerInitializer 该实现类。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 | @HandlesTypes(WebApplicationInitializer.class) **public** **class** **SpringServletContainerInitializer** **implements** **ServletContainerInitializer** {  @Override  **public** **void** **onStartup**(Set<Class<?>> webAppInitializerClasses, ServletContext servletContext)  **throws** ServletException {   List<WebApplicationInitializer> initializers = **new** LinkedList<WebApplicationInitializer>();   **if** (webAppInitializerClasses != **null**) {  **for** (Class<?> waiClass : webAppInitializerClasses) {  // Be defensive: Some servlet containers provide us with invalid classes,  // no matter what @HandlesTypes says...  // <1>  **if** (!waiClass.isInterface() && !Modifier.isAbstract(waiClass.getModifiers()) &&  WebApplicationInitializer.class.isAssignableFrom(waiClass)) {  **try** {  initializers.add((WebApplicationInitializer) waiClass.newInstance());  }  **catch** (Throwable ex) {  **throw** **new** ServletException("Failed to instantiate WebApplicationInitializer class", ex);  }  }  }  }   **if** (initializers.isEmpty()) {  servletContext.log("No Spring WebApplicationInitializer types detected on classpath");  **return**;  }   servletContext.log(initializers.size() + " Spring WebApplicationInitializers detected on classpath");  AnnotationAwareOrderComparator.sort(initializers);  // <2>  **for** (WebApplicationInitializer initializer : initializers) {  initializer.onStartup(servletContext);  }  } } |

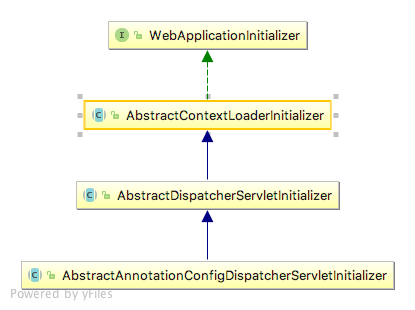
查看其 java doc，描述如下：

Servlet 3.0 {@link ServletContainerInitializer} designed to support code-based configuration of the servlet container using Spring’s {@link WebApplicationInitializer} SPI as opposed to (or possibly in combination with) the traditional {@code web.xml}-based approach.

注意我在源码中标注两个序号，这对于我们理解 spring 装配 servlet 的流程来说非常重要。

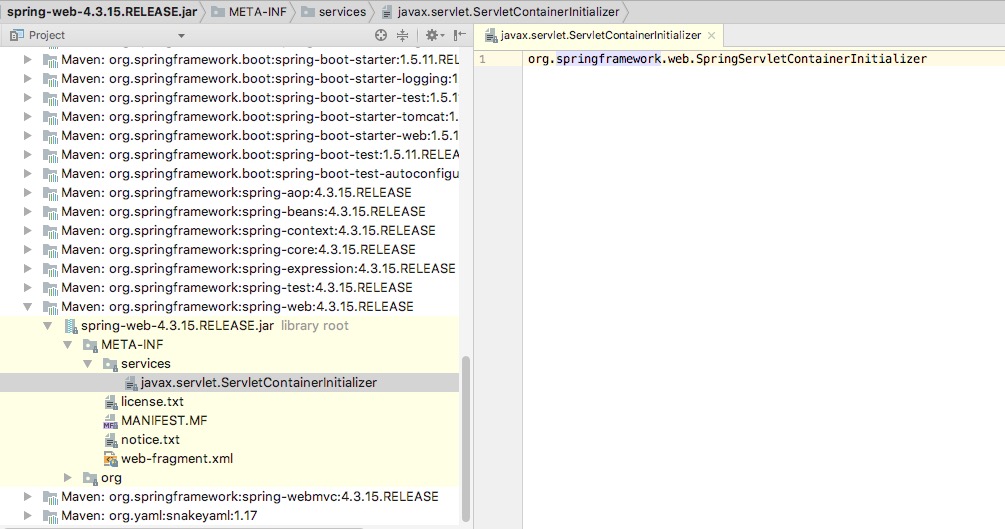
<1> 英文注释是 spring 源码中自带的，它提示我们由于 servlet 厂商实现的差异，onStartup 方法会加载我们本不想处理的 class，所以进行了特判。

<2> spring 与我们之前的 demo 不同，并没有在 SpringServletContainerInitializer 中直接对 servlet 和 filter 进行注册，而是委托给了一个陌生的类 WebApplicationInitializer ，WebApplicationInitializer 类便是 spring 用来初始化 web 环境的委托者类，它通常有三个实现类：

[](http://ov0zuistv.bkt.clouddn.com/WebApplicationInitializer.png)WebApplicationInitializer

你一定不会对 dispatcherServlet 感到陌生，AbstractDispatcherServletInitializer#registerDispatcherServlet 便是无 web.xml 前提下创建 dispatcherServlet 的关键代码。

可以去项目中寻找一下 org.springframework:spring-web:version 的依赖，它下面就存在一个 servletContainerInitializer 的扩展，指向了 SpringServletContainerInitializer，这样只要在 servlet3.0 环境下部署，spring 便可以自动加载进行初始化：

[](http://ov0zuistv.bkt.clouddn.com/F835D518-A725-40D9-84BA-6AC014DAE5A7.png)SpringServletContainerInitializer

注意，上述这一切特性从 spring 3 就已经存在了，而如今 spring 5 已经伴随 springboot 2.0 一起发行了。

### SpringBoot 如何加载 Servlet？

读到这儿，你已经阅读了全文的 1/2。springboot 对于 servlet 的处理才是重头戏，其一，是因为 springboot 使用范围很广，很少有人用 spring 而不用 springboot 了；其二，是因为它没有完全遵守 servlet3.0 的规范！

是的，前面所讲述的 servlet 的规范，无论是 web.xml 中的配置，还是 servlet3.0 中的 ServletContainerInitializer 和 springboot 的加载流程都没有太大的关联。按照惯例，先卖个关子，先看看如何在 springboot 中注册 servlet 和 filter，再来解释下 springboot 的独特之处。

#### 注册方式一：servlet3.0注解+@ServletComponentScan

springboot 依旧兼容 servlet3.0 一系列以 @Web\* 开头的注解：@WebServlet，@WebFilter，@WebListener

|  |  |
| --- | --- |
| 1 2 | @WebServlet("/hello") **public** **class** **HelloWorldServlet** **extends** **HttpServlet**{} |

|  |  |
| --- | --- |
| 1 2 | @WebFilter("/hello/\*") **public** **class** **HelloWorldFilter** **implements** **Filter** {} |

不要忘记让启动类去扫描到这些注解

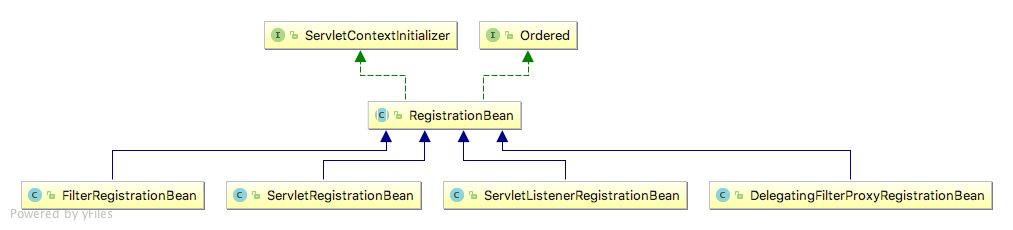
|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | @SpringBootApplication @ServletComponentScan **public** **class** **SpringBootServletApplication** {   **public** **static** **void** **main**(String[] args) {  SpringApplication.run(SpringBootServletApplication.class, args);  } } |

我认为这是几种方式中最为简洁的方式，如果真的有特殊需求，需要在 springboot 下注册 servlet，filter，可以采用这样的方式，比较直观。

#### 注册方式二：RegistrationBean

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | @Bean **public** ServletRegistrationBean **helloWorldServlet**() {  ServletRegistrationBean helloWorldServlet = **new** ServletRegistrationBean();  myServlet.addUrlMappings("/hello");  myServlet.setServlet(**new** HelloWorldServlet());  **return** helloWorldServlet; }  @Bean **public** FilterRegistrationBean **helloWorldFilter**() {  FilterRegistrationBean helloWorldFilter = **new** FilterRegistrationBean();  myFilter.addUrlPatterns("/hello/\*");  myFilter.setFilter(**new** HelloWorldFilter());  **return** helloWorldFilter; } |

ServletRegistrationBean 和 FilterRegistrationBean 都集成自 RegistrationBean ，RegistrationBean 是 springboot 中广泛应用的一个注册类，负责把 servlet，filter，listener 给容器化，使他们被 spring 托管，并且完成自身对 web 容器的注册。这种注册方式也值得推崇。

[](http://ov0zuistv.bkt.clouddn.com/RegistrationBean.png)RegistrationBean

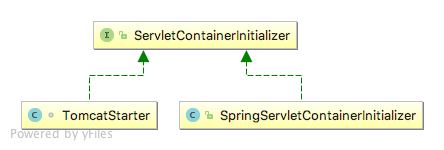
从图中可以看出 RegistrationBean 的地位，它的几个实现类作用分别是：帮助容器注册 filter，servlet，listener，最后的 DelegatingFilterProxyRegistrationBean 使用的不多，但熟悉 SpringSecurity 的朋友不会感到陌生，SpringSecurityFilterChain 就是通过这个代理类来调用的。另外 RegistrationBean 实现了 ServletContextInitializer 接口，这个接口将会是下面分析的核心接口，大家先混个眼熟，了解下它有一个抽象实现 RegistrationBean 即可。

### SpringBoot中servlet加载流程的源码分析

暂时只介绍这两种方式，下面解释下之前卖的关子，为什么说 springboot 没有完全遵守 servlet3.0 规范。讨论的前提是 springboot 环境下使用内嵌的容器，比如最典型的 tomcat。高能预警，以下内容比较烧脑，觉得看起来吃力的朋友可以跳过本节直接看下一节的总结！

#### Initializer被替换为TomcatStarter

当使用内嵌的 tomcat 时，你会发现 springboot 完全走了另一套初始化流程，完全没有使用前面提到的 SpringServletContainerInitializer，实际上一开始我在各种 ServletContainerInitializer 的实现类中打了断点，最终定位到，根本没有运行到 SpringServletContainerInitializer 内部，而是进入了 TomcatStarter 这个类中。

[](http://ov0zuistv.bkt.clouddn.com/TomcatStarter.png)TomcatStarter

并且，仔细扫了一眼源码的包，并没有发现有 SPI 文件对应到 TomcatStarter。于是我猜想，内嵌 tomcat 的加载可能不依赖于 servlet3.0 规范和 SPI！它完全走了一套独立的逻辑。为了验证这一点，我翻阅了 spring github 中的 issue，得到了 spring 作者肯定的答复：<https://github.com/spring-projects/spring-boot/issues/321>

This was actually an intentional design decision. The search algorithm used by the containers was problematic. It also causes problems when you want to develop an executable WAR as you often want a javax.servlet.ServletContainerInitializer for the WAR that is not executed when you run java -jar.

See the org.springframework.boot.context.embedded.ServletContextInitializer for an option that works with Spring Beans.

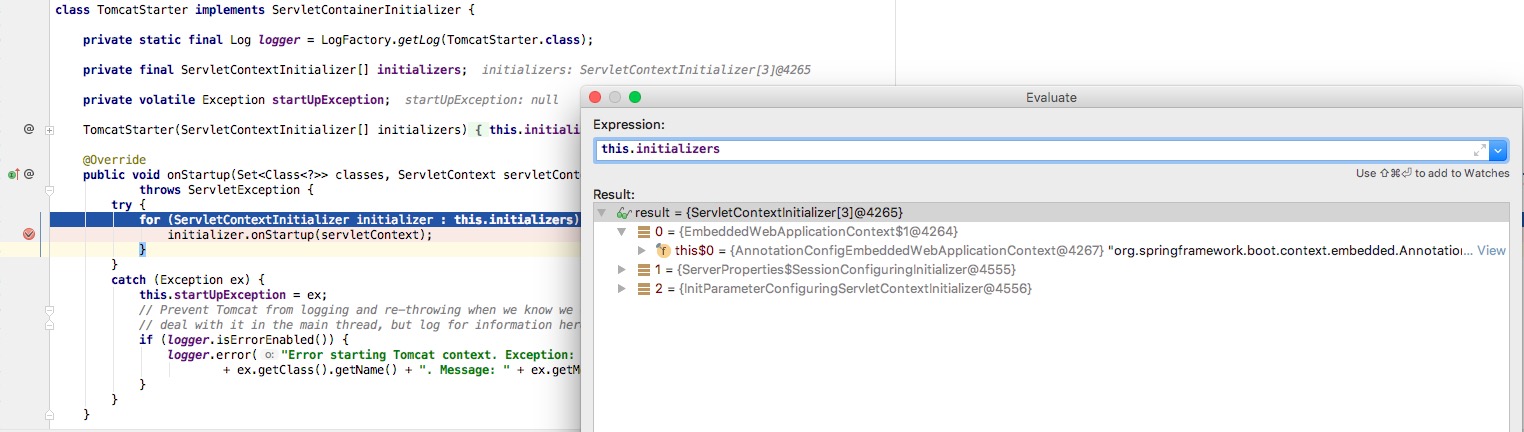
springboot 这么做是有意而为之。springboot 考虑到了如下的问题，我们在使用 springboot 时，开发阶段一般都是使用内嵌 tomcat 容器，但部署时却存在两种选择：一种是打成 jar 包，使用 java -jar 的方式运行；另一种是打成 war 包，交给外置容器去运行。前者就会导致容器搜索算法出现问题，因为这是 jar 包的运行策略，不会按照 servlet3.0 的策略去加载 ServletContainerInitializer！最后作者还提供了一个替代选项：ServletContextInitializer，注意是 ServletContextInitializer！它和 ServletContainerInitializer 长得特别像，别搞混淆了，前者 ServletContextInitializer 是 org.springframework.boot.web.servlet.ServletContextInitializer，后者 ServletContainerInitializer 是 javax.servlet.ServletContainerInitializer，前文还提到 RegistrationBean 实现了 ServletContextInitializer 接口。

#### TomcatStarter中的ServletContextInitializer是关键

TomcatStarter 中的 org.springframework.boot.context.embedded.ServletContextInitializer 是 springboot 初始化 servlet，filter，listener 的关键。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | **class** **TomcatStarter** **implements** **ServletContainerInitializer** {   **private** **final** ServletContextInitializer[] initializers;   TomcatStarter(ServletContextInitializer[] initializers) {  **this**.initializers = initializers;  }   @Override  **public** **void** **onStartup**(Set<Class<?>> classes, ServletContext servletContext)  **throws** ServletException {  **for** (ServletContextInitializer initializer : **this**.initializers) {  initializer.onStartup(servletContext);  }  } } |

经过删减源码后，可以看出 TomcatStarter 的主要逻辑，它其实就是负责调用一系列 ServletContextInitializer 的 onStartup 方法，那么在 debug 中，ServletContextInitializer[] initializers 到底包含了哪些类呢？会不会有我们前面介绍的 RegisterBean 呢？

[](http://ov0zuistv.bkt.clouddn.com/35560726-DD9D-478A-BFCA-12ACF4DB497D.png)initializers

太天真了，RegisterBean 并没有出现在 TomcatStarter 的 debug 信息中，initializers 只包含了三个类，其中只有第一个类看上去比较核心，注意第一个类不是 EmbeddedWebApplicationContext！而是这个类中的 $1 匿名类，为了搞清楚 springboot 如何加载 filter servlet listener ，看来还得研究下 EmbeddedWebApplicationContext 的结构。

#### EmbeddedWebApplicationContext中的6层迭代加载

ApplicationContext 大家应该是比较熟悉的，这是 spring 一个比较核心的类，一般我们可以从中获取到那些注册在容器中的托管 Bean，而这篇文章，主要分析的便是它在内嵌容器中的实现类：EmbeddedWebApplicationContext，重点分析它加载 filter servlet listener 这部分的代码。这里是整个代码中迭代层次最深的部分，做好心理准备起航，来看看 EmbeddedWebApplicationContext 是怎么获取到所有的 servlet filter listener 的！以下方法均出自于 EmbeddedWebApplicationContext。

**第一层：onRefresh()**

onRefresh 是 ApplicationContext 的生命周期方法，EmbeddedWebApplicationContext 的实现非常简单，只干了一件事：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | @Override **protected** **void** **onRefresh**() {  **super**.onRefresh();  **try** {  createEmbeddedServletContainer();//第二层的入口  }  **catch** (Throwable ex) {  **throw** **new** ApplicationContextException("Unable to start embedded container",  ex);  } } |

createEmbeddedServletContainer 连接到了第二层

**第二层：createEmbeddedServletContainer()**

看名字 spring 是想创建一个内嵌的 servlet 容器，ServletContainer 其实就是 servlet filter listener 的总称。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | **private** **void** **createEmbeddedServletContainer**() {  EmbeddedServletContainer localContainer = **this**.embeddedServletContainer;  ServletContext localServletContext = getServletContext();  **if** (localContainer == **null** && localServletContext == **null**) {  EmbeddedServletContainerFactory containerFactory = getEmbeddedServletContainerFactory();  **this**.embeddedServletContainer = containerFactory  .getEmbeddedServletContainer(getSelfInitializer());//第三层的入口  }  **else** **if** (localServletContext != **null**) {  **try** {  getSelfInitializer().onStartup(localServletContext);  }  **catch** (ServletException ex) {  **throw** **new** ApplicationContextException("Cannot initialize servlet context",  ex);  }  }  initPropertySources(); } |

凡是带有 servlet，initializer 字样的方法都是我们需要留意的，getSelfInitializer() 便涉及到了我们最为关心的初始化流程。

**第三层：getSelfInitializer()**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 | **private** org.springframework.boot.web.servlet.ServletContextInitializer **getSelfInitializer**() {  **return** **new** ServletContextInitializer() {  @Override  **public** **void** **onStartup**(ServletContext servletContext) **throws** ServletException {  selfInitialize(servletContext);  }  }; }  **private** **void** **selfInitialize**(ServletContext servletContext) **throws** ServletException {  prepareEmbeddedWebApplicationContext(servletContext);  ConfigurableListableBeanFactory beanFactory = getBeanFactory();  ExistingWebApplicationScopes existingScopes = **new** ExistingWebApplicationScopes(  beanFactory);  WebApplicationContextUtils.registerWebApplicationScopes(beanFactory,  getServletContext());  existingScopes.restore();  WebApplicationContextUtils.registerEnvironmentBeans(beanFactory,  getServletContext());  //第四层的入口  **for** (ServletContextInitializer beans : getServletContextInitializerBeans()) {  beans.onStartup(servletContext);  } } |

还记得前面 TomcatStarter 的 debug 信息中，第一个 ServletContextInitializer 就是出现在 EmbeddedWebApplicationContext 中的一个匿名类，没错了，就是这里的 getSelfInitializer() 方法创建的！解释下这里的 getSelfInitializer() 和 selfInitialize(ServletContext servletContext) 为什么要这么设计：这是典型的回调式方式，当匿名 ServletContextInitializer 类被 TomcatStarter 的 onStartup 方法调用，设计上是触发了 selfInitialize(ServletContext servletContext) 的调用。所以这下就清晰了，为什么 TomcatStarter 中没有出现 RegisterBean ，其实是隐式触发了 EmbeddedWebApplicationContext 中的 selfInitialize 方法。selfInitialize 方法中的 getServletContextInitializerBeans() 成了关键。

**第四层：getServletContextInitializerBeans()**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | /\*\*  \* Returns {**@link** ServletContextInitializer}s that should be used with the embedded  \* Servlet context. By default this method will first attempt to find  \* {**@link** ServletContextInitializer}, {**@link** Servlet}, {**@link** Filter} and certain  \* {**@link** EventListener} beans.  \* **@return** the servlet initializer beans  \*/ **protected** Collection<ServletContextInitializer> **getServletContextInitializerBeans**() {  **return** **new** ServletContextInitializerBeans(getBeanFactory());//第五层的入口 } |

没错了，注释都告诉我们，这个 ServletContextInitializerBeans 是用来加载 Servlet 和 Filter 的。

**第五层：ServletContextInitializerBeans的构造方法**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | **public** **ServletContextInitializerBeans**(ListableBeanFactory beanFactory) {  **this**.initializers = **new** LinkedMultiValueMap<Class<?>, ServletContextInitializer>();  addServletContextInitializerBeans(beanFactory);// 第六层的入口  addAdaptableBeans(beanFactory);  List<ServletContextInitializer> sortedInitializers = **new** ArrayList<ServletContextInitializer>();  **for** (Map.Entry<?, List<ServletContextInitializer>> entry : **this**.initializers  .entrySet()) {  AnnotationAwareOrderComparator.sort(entry.getValue());  sortedInitializers.addAll(entry.getValue());  }  **this**.sortedList = Collections.unmodifiableList(sortedInitializers); } |

**第六层：addServletContextInitializerBeans(beanFactory)**

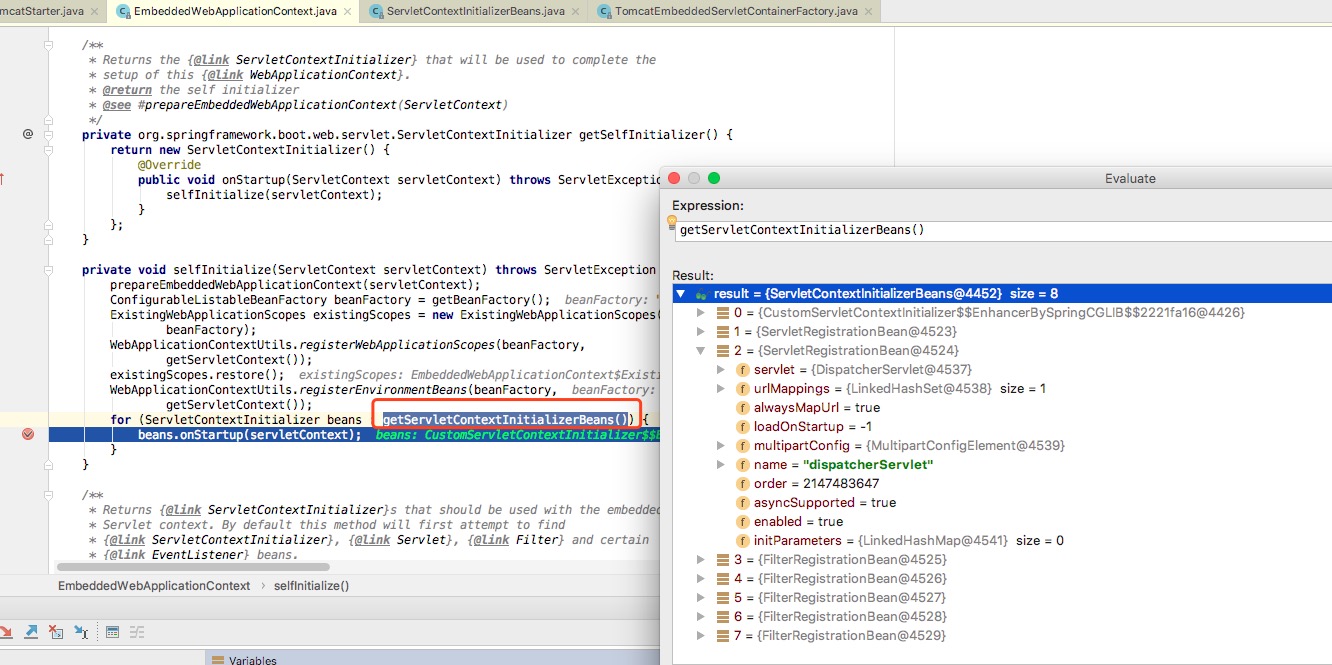
|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 | **private** **void** **addServletContextInitializerBeans**(ListableBeanFactory beanFactory) {  **for** (Entry<String, ServletContextInitializer> initializerBean : getOrderedBeansOfType(  beanFactory, ServletContextInitializer.class)) {  addServletContextInitializerBean(initializerBean.getKey(),  initializerBean.getValue(), beanFactory);  } } |

getOrderedBeansOfType 方法便是去容器中寻找注册过得 ServletContextInitializer ，这时候就可以把之前那些 RegisterBean 全部加载出来了，并且 RegisterBean 还实现了 Ordered 接口，在这儿用于排序。不再往下迭代了。

### EmbeddedWebApplicationContext加载流程总结

如果你对具体的代码流程不感兴趣，可以跳过上述的6层分析，直接看本节的结论。总结如下：

* EmbeddedWebApplicationContext 的 onRefresh 方法触发配置了一个匿名的 ServletContextInitializer。
* 这个匿名的 ServletContextInitializer 的 onStartup 方法会去容器中搜索到了所有的 RegisterBean 并按照顺序加载到 ServletContext 中。
* 这个匿名的 ServletContextInitializer 最终传递给 TomcatStarter，由 TomcatStarter 的 onStartup 方法去触发 ServletContextInitializer 的 onStartup 方法，最终完成装配！

[](http://ov0zuistv.bkt.clouddn.com/8FFCA673-DB72-4C0A-BDE9-58CB4B80C484.png)getServletContextInitializerBeans

### 第三种注册 Servlet 的方式

研究完了上述 springboot 启动的内部原理，可以发现 ServletContextInitializer 其实是 spring 中 ServletContainerInitializer 的代理，虽然 springboot 中 Servlet3.0 不起作用了，但它的代理还是会被加载的，于是我们有了第三种方式注册 servlet。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 | @Configuration **public** **class** **CustomServletContextInitializer** **implements** **ServletContextInitializer** {   **private** **final** **static** String JAR\_HELLO\_URL = "/hello";   @Override  **public** **void** **onStartup**(ServletContext servletContext) **throws** ServletException {  System.out.println("创建 helloWorldServlet...");   ServletRegistration.Dynamic servlet = servletContext.addServlet(  HelloWorldServlet.class.getSimpleName(),  HelloWorldServlet.class);  servlet.addMapping(JAR\_HELLO\_URL);   System.out.println("创建 helloWorldFilter...");   FilterRegistration.Dynamic filter = servletContext.addFilter(  HelloWorldFilter.class.getSimpleName(), HelloWorldFilter.class);   EnumSet<DispatcherType> dispatcherTypes = EnumSet.allOf(DispatcherType.class);  dispatcherTypes.add(DispatcherType.REQUEST);  dispatcherTypes.add(DispatcherType.FORWARD);   filter.addMappingForUrlPatterns(dispatcherTypes, **true**, JAR\_HELLO\_URL);  } } |

虽然 ServletCantainerInitializer 不能被内嵌容器加载，ServletContextInitializer 却能被 springboot 的 EmbeddedWebApplicationContext 加载到，从而装配其中的 servlet 和 filter。实际开发中，还是以一，二两种方法来注册为主，这里只是提供一个可能性，来让我们理解 springboot 的加载流程。

### 加载流程拾遗

1. TomcatStarter 既然不是通过 SPI 机制装配的，那是怎么被 spring 使用的？

自然是被 new 出来的，在 TomcatEmbeddedServletContainerFactory#configureContext 中可以看到，TomcatStarter 是被主动实例化出来的，并且还传入了 ServletContextInitializer 的数组，和上面分析的一样，一共有三个 ServletContextInitializer，包含了 EmbeddedWebApplicationContext 中的匿名实现。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | **protected** **void** **configureContext**(Context context,  ServletContextInitializer[] initializers) {  TomcatStarter starter = **new** TomcatStarter(initializers);  **if** (context **instanceof** TomcatEmbeddedContext) {  // Should be true  ((TomcatEmbeddedContext) context).setStarter(starter);  }  context.addServletContainerInitializer(starter, NO\_CLASSES);  ...  } } |

1. TomcatEmbeddedServletContainerFactory 又是如何被声明的？

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 | @AutoConfigureOrder(Ordered.HIGHEST\_PRECEDENCE) @Configuration @ConditionalOnWebApplication @Import(BeanPostProcessorsRegistrar.class) **public** **class** **EmbeddedServletContainerAutoConfiguration** {   /\*\*  \* Nested configuration if Tomcat is being used.  \*/  @Configuration  @ConditionalOnClass({ Servlet.class, Tomcat.class })  @ConditionalOnMissingBean(value = EmbeddedServletContainerFactory.class, search = SearchStrategy.CURRENT)  **public** **static** **class** **EmbeddedTomcat** {   @Bean  **public** TomcatEmbeddedServletContainerFactory **tomcatEmbeddedServletContainerFactory**() {  **return** **new** TomcatEmbeddedServletContainerFactory();  }   } } |

只要类路径下存在 Tomcat 类，以及在 web 环境下，就会触发 springboot 的自动配置。

### 总结

存在 web.xml 配置的 java web 项目，servlet3.0 的 java web 项目，springboot 内嵌容器的 java web 项目加载 servlet，filter，listener 的流程都是有所差异的，理解清楚这其中的原来，其实并不容易，至少得搞懂 servlet3.0 的规范，springboot 内嵌容器的加载流程等等前置逻辑。

最后感谢下小马哥的点拨，在此之前误以为： TomcatStarter 既然继承了 ServletContainerInitializer，应该也是符合 servlet3.0 规范的，但实际上并没有被 SPI 加载。

### 推荐阅读

JAVA拾遗–关于SPI机制 <https://www.cnkirito.moe/spi/>

**欢迎关注我的微信公众号：「Kirito的技术分享」，关于文章的任何疑问都会得到回复，带来更多 Java 相关的技术分享。**

[](http://ov0zuistv.bkt.clouddn.com/qrcode_for_gh_c06057be7960_258%20%281%29.jpg)关注微信公众号