



下载APP



## 14 | Spring Web 过滤器使用常见错误（下）

2021-05-24 傅健

《Spring编程常见错误50例》

[课程介绍 >](#)**讲述：傅健**

时长 16:32 大小 15.15M




你好，我是傅健。

通过上节课的两个案例，我们了解了容器运行时过滤器的工作原理，那么这节课我们还是通过两个错误案例，来学习下容器启动时过滤器初始化以及排序注册等相关逻辑。了解了它们，你会对如何使用好过滤器更有信心。下面，我们具体来看一下。

### 案例 1：@WebFilter 过滤器使用 @Order 无效


假设我们还是基于 Spring Boot 去开发上节课的学籍管理系统，这里我们简单复习下 课用到的代码。

首先，创建启动程序的代码如下：

 复制代码

```
1 @SpringBootApplication
2 @ServletComponentScan
3 @Slf4j
4 public class Application {
5     public static void main(String[] args) {
6         SpringApplication.run(Application.class, args);
7         log.info("启动成功");
8     }
9 }
```

实现的 Controller 代码如下：

 复制代码

```
1 @Controller
2 @Slf4j
3 public class StudentController {
4     @PostMapping("/regStudent/{name}")
5     @ResponseBody
6     public String saveUser(String name) throws Exception {
7         System.out.println(".....用户注册成功");
8         return "success";
9     }
10 }
```

上述代码提供了一个 Restful 接口 "/regStudent"。该接口只有一个参数 name，注册成功会返回"success"。

现在，我们来实现两个新的过滤器，代码如下：

AuthFilter：例如，限制特定 IP 地址段（例如校园网内）的用户方可注册为新用户，当然这里我们仅仅 Sleep 1 秒来模拟这个过程。

 复制代码

```
1 @WebFilter
2 @Slf4j
3 @Order(2)
4 public class AuthFilter implements Filter {
5     @SneakyThrows
6     @Override
7     public void doFilter(ServletRequest request, ServletResponse response, Fil
8         if(isPassAuth()){
```

```
9         System.out.println("通过授权");
10        chain.doFilter(request, response);
11    }else{
12        System.out.println("未通过授权");
13        ((HttpServletResponse)response).sendError(401);
14    }
15 }
16 private boolean isPassAuth() throws InterruptedException {
17     System.out.println("执行检查权限");
18     Thread.sleep(1000);
19     return true;
20 }
21 }
```

TimeCostFilter：计算注册学生的执行耗时，需要包括授权过程。

[复制代码](#)

```
1 @WebFilter
2 @Slf4j
3 @Order(1)
4 public class TimeCostFilter implements Filter {
5     @Override
6     public void doFilter(ServletRequest request, ServletResponse response, Fil
7         System.out.println("#开始计算接口耗时");
8         long start = System.currentTimeMillis();
9         chain.doFilter(request, response);
10        long end = System.currentTimeMillis();
11        long time = end - start;
12        System.out.println("#执行时间(ms)：" + time);
13    }
14 }
```

在上述代码中，我们使用了 @Order，期望 TimeCostFilter 先被执行，因为 TimeCostFilter 设计的初衷是统计这个接口的性能，所以是需要统计 AuthFilter 执行的授权过程的。

全部代码实现完毕，执行结果如下：

[复制代码](#)

```
1 执行检查权限
2 通过授权
3 #开始计算接口耗时
4 .....用户注册成功
```

```
5 #执行时间(ms) : 33
```

从结果来看，执行时间并不包含授权过程，所以这并不符合我们的预期，毕竟我们是加了 `@Order` 的。但是如果我们交换 `Order` 指定的值，你会发现也不见效果，为什么会如此？难道 `Order` 不能用来排序 `WebFilter` 么？下面我们来具体解析下这个问题及其背后的原理。

## 案例解析

通过上节课的学习，我们得知：当一个请求来临时，会执行到 `StandardWrapperValve` 的 `invoke()`，这个方法会创建 `ApplicationFilterChain`，并通过 `ApplicationFilterChain#doFilter()` 触发过滤器执行，并最终执行到内部私有方法 `internalDoFilter()`，我们可以尝试在 `internalDoFilter()` 中寻找一些启示：

[复制代码](#)

```
1 private void internalDoFilter(ServletRequest request,
2                               ServletResponse response)
3     throws IOException, ServletException {
4
5     // Call the next filter if there is one
6     if (pos < n) {
7         ApplicationFilterConfig filterConfig = filters[pos++];
8         try {
9             Filter filter = filterConfig.getFilter();
```

从上述代码我们得知：过滤器的执行顺序是由类成员变量 `Filters` 决定的，而 `Filters` 变量则是 `createFilterChain()` 在容器启动时顺序遍历 `StandardContext` 中的成员变量 `FilterMaps` 获得的：

[复制代码](#)

```
1 public static ApplicationFilterChain createFilterChain(ServletRequest request,
2                                                         Wrapper wrapper, Servlet servlet) {
3
4     // 省略非关键代码
5     // Acquire the filter mappings for this Context
6     StandardContext context = (StandardContext) wrapper.getParent();
7     FilterMap filterMaps[] = context.findFilterMaps();
8     // 省略非关键代码
9     // Add the relevant path-mapped filters to this filter chain
10    for (int i = 0; i < filterMaps.length; i++) {
11
```

```
12         if (!matchDispatcher(filterMaps[i], dispatcher)) {
13             continue;
14         }
15         if (!matchFiltersURL(filterMaps[i], requestPath))
16             continue;
17         ApplicationFilterConfig filterConfig = (ApplicationFilterConfig)
18             context.findFilterConfig(filterMaps[i].getFilterName());
19         if (filterConfig == null) {
20             continue;
21         }
22         filterChain.addFilter(filterConfig);
23     }
24     // 省略非关键代码
25     // Return the completed filter chain
26     return filterChain;
    }
```

下面继续查找对 StandardContext 成员变量 FilterMaps 的写入引用，我们找到了 addFilterMapBefore()：

[复制代码](#)

```
1 public void addFilterMapBefore(FilterMap filterMap) {
2     validateFilterMap(filterMap);
3     // Add this filter mapping to our registered set
4     filterMaps.addBefore(filterMap);
5     fireContainerEvent("addFilterMap", filterMap);
6 }
```

到这，我们已经知道过滤器的执行顺序是由 StandardContext 类成员变量 FilterMaps 的顺序决定，而 FilterMaps 则是一个包装过的数组，所以我们只要进一步弄清楚 **FilterMaps 中各元素的排列顺序**即可。

我们继续在 addFilterMapBefore() 中加入断点，尝试从调用栈中找到一些线索：

[复制代码](#)

```
1 addFilterMapBefore:2992, StandardContext
2 addMappingForUrlPatterns:107, ApplicationFilterRegistration
3 configure:229, AbstractFilterRegistrationBean
4 configure:44, AbstractFilterRegistrationBean
5 register:113, DynamicRegistrationBean
6 onStartup:53, RegistrationBean
7 selfInitialize:228, ServletWebServerApplicationContext
8 // 省略非关键代码
```

可知，Spring 从 `selfInitialize()` 一直依次调用到 `addFilterMapBefore()`，稍微分析下 `selfInitialize()`，我们可以了解到，这里是通过调用 `getServletContextInitializerBeans()`，获取所有的 `ServletContextInitializer` 类型的 Bean，并调用该 Bean 的 `onStartup()`，从而一步步以调用栈显示的顺序，最终调用到 `addFilterMapBefore()`。

[复制代码](#)

```
1 private void selfInitialize(ServletContext servletContext) throws ServletException {
2     prepareWebApplicationContext(servletContext);
3     registerApplicationScope(servletContext);
4     WebApplicationContextUtils.registerEnvironmentBeans(getBeanFactory(), servletContext);
5     for (ServletContextInitializer beans : getServletContextInitializerBeans()) {
6         beans.onStartup(servletContext);
7     }
8 }
```

那么上述的 `selfInitialize()` 又从何处调用过来呢？这里你可以先想想，我会在思考题中给你做进一步解释。

现在我们继续查看 `selfInitialize()` 的细节。

首先，查看上述代码中的 `getServletContextInitializerBeans()`，因为此方法返回的 `ServletContextInitializer` 类型的 Bean 集合顺序决定了 `addFilterMapBefore()` 调用的顺序，从而决定了 `FilterMaps` 内元素的顺序，最终决定了过滤器的执行顺序。

`getServletContextInitializerBeans()` 的实现非常简单，只是返回了 `ServletContextInitializerBeans` 类的一个实例，参考代码如下：

[复制代码](#)

```
1 protected Collection<ServletContextInitializer> getServletContextInitializerBeans() {
2     return new ServletContextInitializerBeans(getBeanFactory());
3 }
```

上述方法的返回值是个 `Collection`，可见 `ServletContextInitializerBeans` 类是一个集合类，它继承了 `AbstractCollection` 抽象类。也因为如此，上述 `selfInitialize()` 才可以遍历



## ServletContextInitializerBeans 的实例对象。

既然 ServletContextInitializerBeans 是集合类，那么我们就可以先查看其 iterator()，看看它遍历的是什么。

[复制代码](#)

```
1 @Override
2 public Iterator<ServletContextInitializer> iterator() {
3     return this.sortedList.iterator();
4 }
```

此集合类对外暴露的集合遍历元素为 sortedList 成员变量，也就是说，上述 selfInitialize() 最终遍历的即为 sortedList 成员变量。

到这，我们可以进一步确定下结论：selfInitialize() 中是通过 getServletContextInitializerBeans() 获取到的 ServletContextInitializer 类型的 Beans 集合，即为 ServletContextInitializerBeans 的类型成员变量 sortedList。反过来说，**sortedList 中的过滤器 Bean 元素顺序，决定了最终过滤器的执行顺序。**

现在我们继续查看 ServletContextInitializerBeans 的构造方法如下：

[复制代码](#)

```
1 public ServletContextInitializerBeans(ListableBeanFactory beanFactory,
2     Class<? extends ServletContextInitializer>... initializerTypes) {
3     this.initializers = new LinkedMultiValueMap<>();
4     this.initializerTypes = (initializerTypes.length != 0) ? Arrays.asList(init
5         : Collections.singletonList(ServletContextInitializer.class);
6     addServletContextInitializerBeans(beanFactory);
7     addAdaptableBeans(beanFactory);
8     List<ServletContextInitializer> sortedInitializers = this.initializers.valu
9         .flatMap((value) -> value.stream().sorted(AnnotationAwareOrderCompara
10         .collect(Collectors.toList()));
11     this.sortedList = Collections.unmodifiableList(sortedInitializers);
12     logMappings(this.initializers);
13 }
```

通过第 8 行，可以得知：我们关心的类成员变量 this.sortedList，其元素顺序是由类成员变量 this.initializers 的 values 通过比较器 AnnotationAwareOrderComparator 进行排

序的。

继续查看 AnnotationAwareOrderComparator 比较器，忽略比较器调用的细节过程，其最终是通过两种方式获取比较器需要的 order 值，来决定 sortedInitializers 的排列顺序：


待排序的对象元素自身实现了 Order 接口，则直接通过 getOrder() 获取 order 值；

否则执行 OrderUtils.findOrder() 获取该对象类 @Order 的属性。

这里多解释一句，因为 this.initializers 的 values 类型为 ServletContextInitializer，其实现了 Ordered 接口，所以这里的比较器显然是使用了 getOrder() 获取比较器所需的 order 值，对应的类成员变量即为 order。

继续查看 this.initializers 中的元素在何处被添加，我们最终得知，addServletContextInitializerBeans() 以及 addAdaptableBeans() 这两个方法均构建了 ServletContextInitializer 子类的实例，并添加到了 this.initializers 成员变量中。在这里，我们只研究 addServletContextInitializerBeans，毕竟我们使用的添加过滤器方式（使用 @WebFilter 标记）最终只会通过这个方法生效。

在这个方法中，Spring 通过 getOrderedBeansOfType() 实例化了所有 ServletContextInitializer 的子类：

 复制代码

```
1 private void addServletContextInitializerBeans(ListableBeanFactory beanFactory
2     for (Class<? extends ServletContextInitializer> initializerType : this.init
3         for (Entry<String, ? extends ServletContextInitializer> initializerBean
4             initializerType)) {
5             addServletContextInitializerBean(initializerBean.getKey(), initialize
6         }
7     }
8 }
```

根据其不同类型，调用 addServletContextInitializerBean()，我们可以看出 ServletContextInitializer 的子类包括了 ServletRegistrationBean、FilterRegistrationBean 以及 ServletListenerRegistrationBean，正好对应了 Servlet 的三大要素。



而这里我们只需要关心对应于 Filter 的 FilterRegistrationBean，显然，FilterRegistrationBean 是 ServletContextInitializer 的子类（实现了 Ordered 接口），同样由**成员变量 order 的值决定其执行的优先级**。

[复制代码](#)

```
1 private void addServletContextInitializerBean(String beanName, ServletContextI
2     ListableBeanFactory beanFactory) {
3     if (initializer instanceof ServletRegistrationBean) {
4         Servlet source = ((ServletRegistrationBean<?>) initializer).getServlet()
5         addServletContextInitializerBean(Servlet.class, beanName, initializer, b
6     }
7     else if (initializer instanceof FilterRegistrationBean) {
8         Filter source = ((FilterRegistrationBean<?>) initializer).getFilter();
9         addServletContextInitializerBean(Filter.class, beanName, initializer, be
10    }
11    else if (initializer instanceof DelegatingFilterProxyRegistrationBean) {
12        String source = ((DelegatingFilterProxyRegistrationBean) initializer).ge
13        addServletContextInitializerBean(Filter.class, beanName, initializer, be
14    }
15    else if (initializer instanceof ServletListenerRegistrationBean) {
16        EventListener source = ((ServletListenerRegistrationBean<?>) initializer
17        addServletContextInitializerBean(EventListener.class, beanName, initiali
18    }
19    else {
20        addServletContextInitializerBean(ServletContextInitializer.class, beanNa
21            initializer);
22    }
23 }
```

最终添加到 this.initializers 成员变量中：

[复制代码](#)

```
1 private void addServletContextInitializerBean(Class<?> type, String beanName,
2     ListableBeanFactory beanFactory, Object source) {
3     this.initializers.add(type, initializer);
4     // 省略非关键代码
5 }
```

通过上述代码，我们再次看到了 FilterRegistrationBean。但问题来了，我们没有定义 FilterRegistrationBean，那么这里的 FilterRegistrationBean 是在哪里被定义的呢？其 order 类成员变量是否有特定的取值逻辑？

不妨回想下上节课的案例 1，它是在 `WebFilterHandler` 类的 `doHandle()` 动态构建了 `FilterRegistrationBean` 的 `BeanDefinition`：

[复制代码](#)

```
1 class WebFilterHandler extends ServletComponentHandler {
2
3     WebFilterHandler() {
4         super(WebFilter.class);
5     }
6
7     @Override
8     public void doHandle(Map<String, Object> attributes, AnnotatedBeanDefinitio
9         BeanDefinitionRegistry registry) {
10         BeanDefinitionBuilder builder = BeanDefinitionBuilder.rootBeanDefinition
11         builder.addPropertyValue("asyncSupported", attributes.get("asyncSupporte
12         builder.addPropertyValue("dispatcherTypes", extractDispatcherTypes(attri
13         builder.addPropertyValue("filter", beanDefinition);
14         builder.addPropertyValue("initParameters", extractInitParameters(attribu
15         String name = determineName(attributes, beanDefinition);
16         builder.addPropertyValue("name", name);
17         builder.addPropertyValue("servletNames", attributes.get("servletNames"))
18         builder.addPropertyValue("urlPatterns", extractUrlPatterns(attributes));
19         registry.registerBeanDefinition(name, builder.getBeanDefinition());
20     }
21     // 省略非关键代码
```

这里我再次贴出了 `WebFilterHandler` 中 `doHandle()` 的逻辑（即通过 `BeanDefinitionBuilder` 动态构建了 `FilterRegistrationBean` 类型的 `BeanDefinition`）。然而遗憾的是，**此处并没有设置 `order` 的值，更没有根据 `@Order` 指定的值去设置。**

到这里我们终于看清楚问题的本质，所有被 `@WebFilter` 注解的类，最终都会在此处被包装为 `FilterRegistrationBean` 类的 `BeanDefinition`。虽然 `FilterRegistrationBean` 也拥有 `Ordered` 接口，但此处却并没有填充值，因为这里所有的属性都是从 `@WebFilter` 对应的属性获取的，而 `@WebFilter` 本身没有指定可以辅助排序的属性。

现在我们来总结下，过滤器的执行顺序是由下面这个串联决定的：

RegistrationBean 中 `order` 属性的值 ->

`ServletContextInitializerBeans` 类成员变量 `sortedList` 中元素的顺序 ->


ServletWebServerApplicationContext 中 selfInitialize() 遍历  
FilterRegistrationBean 的顺序 ->  
addFilterMapBefore() 调用的顺序 ->  
filterMaps 内元素的顺序 ->  
过滤器的执行顺序

可见，RegistrationBean 中 order 属性的值最终可以决定过滤器的执行顺序。但是可惜的是：当使用 @WebFilter 时，构建的 FilterRegistrationBean 并没有依据 @Order 的值去设置 order 属性，所以 @Order 失效了。

## 问题修正

现在，我们理清了 Spring 启动 Web 服务之前的一些必要类的初始化流程，同时也弄清楚了 @Order 和 @WebFilter 同时使用失效的原因，但这个问题想要解决却并非那么简单。

这里我先提供给你一个常见的做法，即实现自己的 FilterRegistrationBean 来配置添加过滤器，不再使用 @WebFilter。具体代码如下：

 复制代码

```
1 @Configuration
2 public class FilterConfiguration {
3     @Bean
4     public FilterRegistrationBean authFilter() {
5         FilterRegistrationBean registration = new FilterRegistrationBean();
6         registration.setFilter(new AuthFilter());
7         registration.addUrlPatterns("/");
8         registration.setOrder(2);
9         return registration;
10    }
11
12    @Bean
13    public FilterRegistrationBean timeCostFilter() {
14        FilterRegistrationBean registration = new FilterRegistrationBean();
15        registration.setFilter(new TimeCostFilter());
16        registration.addUrlPatterns("/");
17        registration.setOrder(1);
18        return registration;
19    }
20 }
```


按照我们查看的源码中的逻辑，虽然 `WebFilterHandler` 中 `doHandle()` 构建了 `FilterRegistrationBean` 类型的 `BeanDefinition`，但没有设置 **order** 的值。

所以在这里，我们直接手工实例化了 `FilterRegistrationBean` 实例，而且设置了其 `setOrder()`。同时不要忘记去掉 `AuthFilter` 和 `TimeCostFilter` 类中的 `@WebFilter`，这样问题就得以解决了。

## 案例 2：过滤器被多次执行


我们继续沿用上面的案例代码，要解决排序问题，可能有人就想了是不是有其他的解决方案呢？比如我们能否在两个过滤器中增加 `@Component`，从而让 `@Order` 生效呢？代码如下。

`AuthFilter`：

 复制代码


```
1 @WebFilter
2 @Slf4j
3 @Order(2)
4 @Component
5 public class AuthFilter implements Filter {
6     @SneakyThrows
7     @Override
8     public void doFilter(ServletRequest request, ServletResponse response, Fil
9         if(isPassAuth()){
10             System.out.println("通过授权");
11             chain.doFilter(request, response);
12         }else{
13             System.out.println("未通过授权");
14             ((HttpServletResponse)response).sendError(401);
15         }
16     }
17     private boolean isPassAuth() throws InterruptedException {
18         System.out.println("执行检查权限");
19         Thread.sleep(1000);
20         return true;
21     }
22 }
```

`TimeCostFilter` 类如下：

 复制代码


```
1 @WebFilter
2 @Slf4j
3 @Order(1)
4 @Component
5 public class TimeCostFilter implements Filter {
6     @Override
7     public void doFilter(ServletRequest request, ServletResponse response, Fil
8         System.out.println("#开始计算接口耗时");
9         long start = System.currentTimeMillis();
10        chain.doFilter(request, response);
11        long end = System.currentTimeMillis();
12        long time = end - start;
13        System.out.println("#执行时间(ms)：" + time);
14    }
15 }
```

最终执行结果如下：

 复制代码

```
1 #开始计算接口耗时
2 执行检查权限
3 通过授权
4 执行检查权限
5 通过授权
6 #开始计算接口耗时
7 .....用户注册成功
8 #执行时间(ms)：73
9 #执行时间(ms)：2075
```

更改 AuthFilter 类中的 Order 值为 0，继续测试，得到结果如下：

 复制代码

```
1 执行检查权限
2 通过授权
3 #开始计算接口耗时
4 执行检查权限
5 通过授权
6 #开始计算接口耗时
7 .....用户注册成功
8 #执行时间(ms)：96
9 #执行时间(ms)：1100
```


显然，通过 Order 的值，我们已经可以随意调整 Filter 的执行顺序，但是我们会惊奇地发现，过滤器本身被执行了 2 次，这明显不符合我们的预期！那么如何理解这个现象呢？

## 案例解析

从案例 1 中我们已经得知被 @WebFilter 的过滤器，会在 WebServletHandler 类中被重新包装为 FilterRegistrationBean 类的 BeanDefinition，而并非是 Filter 类型。

而我们在自定义过滤器中增加 @Component 时，我们可以大胆猜测下：理论上 Spring 会根据当前类再次包装一个新的过滤器，因而 doFilter() 被执行两次。因此看似奇怪的测试结果，也在情理之中了。

我们继续从源码中寻找真相，继续查阅 ServletContextInitializerBeans 的构造方法如下：

 复制代码

```
1 public ServletContextInitializerBeans(ListableBeanFactory beanFactory,
2     Class<? extends ServletContextInitializer>... initializerTypes) {
3     this.initializers = new LinkedMultiValueMap<>();
4     this.initializerTypes = (initializerTypes.length != 0) ? Arrays.asList(init
5         : Collections.singletonList(ServletContextInitializer.class);
6     addServletContextInitializerBeans(beanFactory);
7     addAdaptableBeans(beanFactory);
8     List<ServletContextInitializer> sortedInitializers = this.initializers.valu
9         .flatMap((value) -> value.stream().sorted(AnnotationAwareOrderCompara
10         .collect(Collectors.toList()));
11     this.sortedList = Collections.unmodifiableList(sortedInitializers);
12     logMappings(this.initializers);
13 }
```

上一个案例中，我们关注了 addServletContextInitializerBeans()，了解了它的作用是实例化并注册了所有 FilterRegistrationBean 类型的过滤器（严格说，是实例化并注册了所有的 ServletRegistrationBean、FilterRegistrationBean 以及 ServletListenerRegistrationBean，但这里我们只关注 FilterRegistrationBean）。

而第 7 行的 addAdaptableBeans()，其作用则是实例化所有实现 Filter 接口的类（严格说，是实例化并注册了所有实现 Servlet、Filter 以及 EventListener 接口的类），然后再逐一包装为 FilterRegistrationBean。



之所以 Spring 能够直接实例化 FilterRegistrationBean 类型的过滤器，这是因为：

WebFilterHandler 相关类通过扫描 @WebFilter，动态构建了 FilterRegistrationBean 类型的 BeanDefinition，并注册到 Spring；

或者我们自己使用 @Bean 来显式实例化 FilterRegistrationBean 并注册到 Spring，如案例 1 中的解决方案。

但 Filter 类型的过滤器如何才能被 Spring 直接实例化呢？相信你已经有了答案了：**任何通过 @Component 修饰的类，都可以自动注册到 Spring，且能被 Spring 直接实例化。**

现在我们直接查看 addAdaptableBeans()，其调用了 addAsRegistrationBean()，其 beanType 为 Filter.class：

[复制代码](#)

```
1 protected void addAdaptableBeans(ListableBeanFactory beanFactory) {
2     // 省略非关键代码
3     addAsRegistrationBean(beanFactory, Filter.class, new FilterRegistrationBean
4     // 省略非关键代码
5 }
```

继续查看最终调用到的方法 addAsRegistrationBean()：

[复制代码](#)

```
1 private <T, B extends T> void addAsRegistrationBean(ListableBeanFactory beanFa
2     Class<B> beanType, RegistrationBeanAdapter<T> adapter) {
3     List<Map.Entry<String, B>> entries = getOrderedBeansOfType(beanFactory, bea
4     for (Entry<String, B> entry : entries) {
5         String beanName = entry.getKey();
6         B bean = entry.getValue();
7         if (this.seen.add(bean)) {
8             // One that we haven't already seen
9             RegistrationBean registration = adapter.createRegistrationBean(beanNa
10             int order = getOrder(bean);
11             registration.setOrder(order);
12             this.initializers.add(type, registration);
13             if (logger.isTraceEnabled()) {
14                 logger.trace("Created " + type.getSimpleName() + " initializer for
15                     + order + ", resource=" + getResourceDescription(beanName, b
16             }
17         }
18     }
```

```
19 }
```

主要逻辑如下：

通过 `getOrderedBeansOfType()` 创建了所有 `Filter` 子类的实例，即所有实现 `Filter` 接口且被 `@Component` 修饰的类；

依次遍历这些 `Filter` 类实例，并通过 `RegistrationBeanAdapter` 将这些类包装为 `RegistrationBean`；

获取 `Filter` 类实例的 `Order` 值，并设置到包装类 `RegistrationBean` 中；

将 `RegistrationBean` 添加到 `this.initializers`。

到这，我们了解到，当过滤器同时被 `@WebFilter` 和 `@Component` 修饰时，会导致两个 `FilterRegistrationBean` 实例的产生。`addServletContextInitializerBeans()` 和 `addAdaptableBeans()` 最终都会创建 `FilterRegistrationBean` 的实例，但不同的是：

`@WebFilter` 会让 `addServletContextInitializerBeans()` 实例化，并注册所有动态生成的 `FilterRegistrationBean` 类型的过滤器；

`@Component` 会让 `addAdaptableBeans()` 实例化所有实现 `Filter` 接口的类，然后再逐一包装为 `FilterRegistrationBean` 类型的过滤器。

## 问题修正

解决这个问题提及的顺序问题，自然可以继续参考案例 1 的问题修正部分。另外我们也可以去掉 `@WebFilter` 保留 `@Component` 的方式进行修改，修改后的 `Filter` 示例如下：

 复制代码

```
1 // @WebFilter
2 @Slf4j
3 @Order(1)
4 @Component
5 public class TimeCostFilter implements Filter {
6     // 省略非关键代码
7 }
```

## 重点回顾

这节课我们分析了过滤器在 Spring 框架中注册、包装以及实例化的整个流程，最后我们再次回顾下重点。

@WebFilter 和 @Component 的相同点是：

它们最终都被包装并实例化成为了 FilterRegistrationBean；

它们最终都是在 ServletContextInitializerBeans 的构造器中开始被实例化。

@WebFilter 和 @Component 的不同点是：

被 @WebFilter 修饰的过滤器会被提前在 BeanFactoryPostProcessors 扩展点包装成 FilterRegistrationBean 类型的 BeanDefinition，然后在 ServletContextInitializerBeans.addServletContextInitializerBeans() 进行实例化；而使用 @Component 修饰的过滤器类，是在 ServletContextInitializerBeans.addAdaptableBeans() 中被实例化成 Filter 类型后，再包装为 RegistrationBean 类型。

被 @WebFilter 修饰的过滤器不会注入 Order 属性，但被 @Component 修饰的过滤器会在 ServletContextInitializerBeans.addAdaptableBeans() 中注入 Order 属性。

## 思考题

这节课的两个案例，它们都是在 Tomcat 容器启动时发生的，但你了解 Spring 是如何整合 Tomcat，使其在启动时注册这些过滤器吗？

期待你的思考，我们留言区见！

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 赞 1  提建议

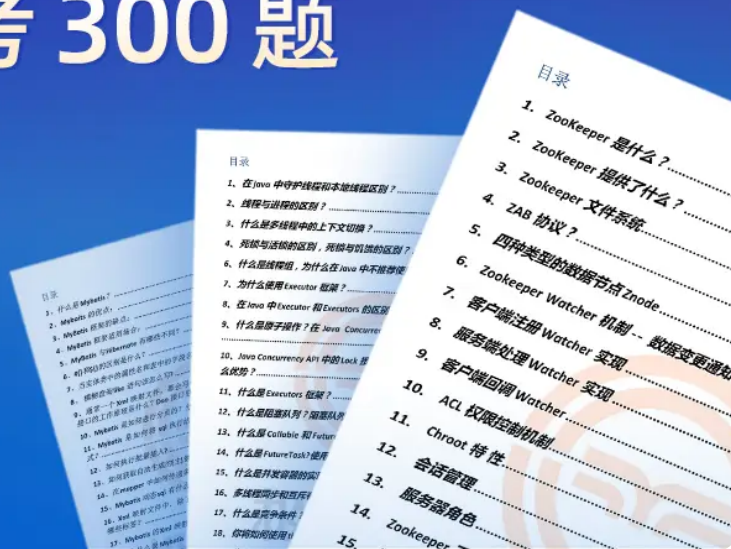
© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

更多学习推荐

# Java 面试必考 300 题

## 最新汇总

限时免费领取



### 精选留言 (3)

写留言



一只幸运的小码畜

2021-05-24

过滤器这两章给我一个启发，能不用@WebFilter就不用，没啥大用还出一堆问题

5

10



ImYours°

2021-07-14

实现order接口的话是有效的吗？

展开



xiaomifeng1010

2021-07-13

使用@Component注解替换@WebFilter，是不是启动类上的@ServletComponentScan注解也要去掉呢？



