

63 | 职责链模式（下）：框架中常用的过滤器、拦截器是如何实现的？

王争 · 设计模式之美



职责链模式（下）

上一节课，我们学习职责链模式的原理与实现，并且通过一个敏感词过滤框架的例子，展示了职责链模式的设计意图。本质上来说，它跟大部分设计模式一样，都是为了解耦代码，应对代码的复杂性，让代码满足开闭原则，提高代码的可扩展性。

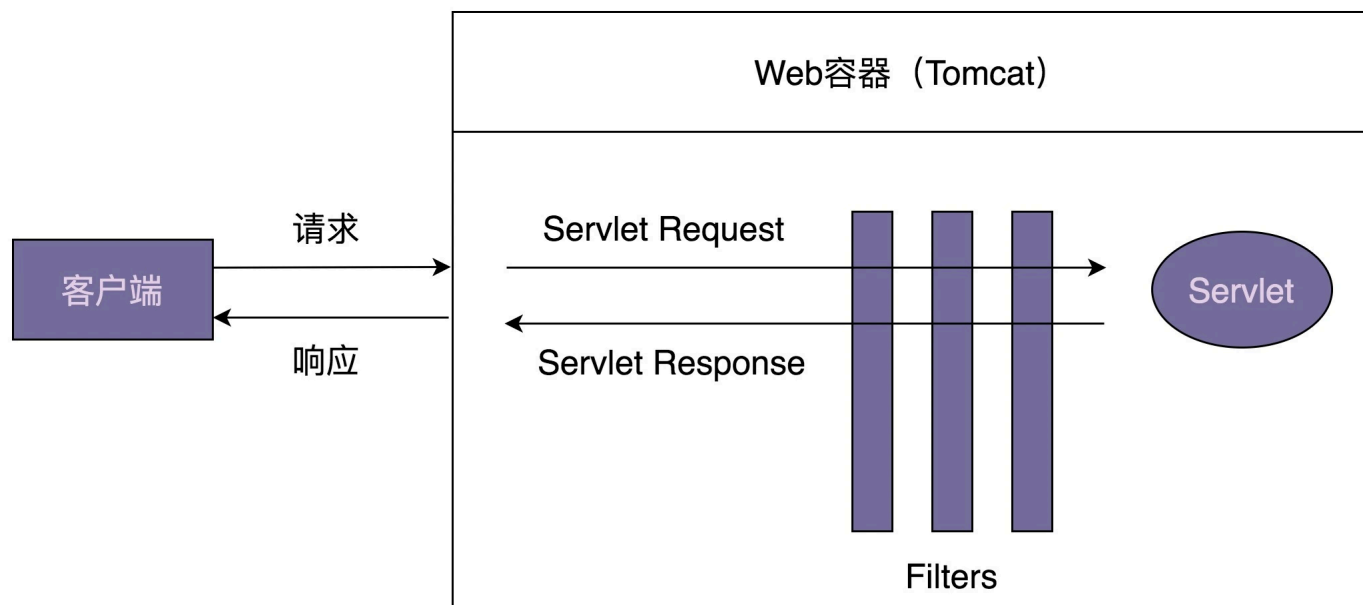
除此之外，我们还提到，职责链模式常用在框架的开发中，为框架提供扩展点，让框架的使用者在不修改框架源码的情况下，基于扩展点添加新的功能。实际上，更具体点来说，职责链模式最常用来开发框架的过滤器和拦截器。今天，我们就通过 Servlet Filter、Spring Interceptor 这两个 Java 开发中常用的组件，来具体讲讲它在框架开发中的应用。

话不多说，让我们正式开始今天的学习吧！

Servlet Filter

Servlet Filter 是 Java Servlet 规范中定义的组件，翻译成中文就是过滤器，它可以实现对 HTTP 请求的过滤功能，比如鉴权、限流、记录日志、验证参数等等。因为它是 Servlet 规范

的一部分，所以，只要是支持 Servlet 的 Web 容器（比如，Tomcat、Jetty 等），都支持过滤器功能。为了帮助你理解，我画了一张示意图阐述它的工作原理，如下所示。



在实际项目中，我们该如何使用 Servlet Filter 呢？我写了一个简单的示例代码，如下所示。添加一个过滤器，我们只需要定义一个实现 `javax.servlet.Filter` 接口的过滤器类，并且将它配置在 `web.xml` 配置文件中。Web 容器启动的时候，会读取 `web.xml` 中的配置，创建过滤器对象。当有请求到来的时候，会先经过过滤器，然后才由 Servlet 来处理。

[复制代码](#)

```
1 public class LogFilter implements Filter {
2     @Override
3     public void init(FilterConfig filterConfig) throws ServletException {
4         // 在创建Filter时自动调用，
5         // 其中filterConfig包含这个Filter的配置参数，比如name之类的（从配置文件中读取的）
6     }
7
8     @Override
9     public void doFilter(ServletRequest request, ServletResponse response, FilterChain
10         System.out.println("拦截客户端发送来的请求.");
11         chain.doFilter(request, response);
12         System.out.println("拦截发送给客户端的响应.");
13     }
14
15     @Override
```


```
16     public void destroy() {
17         // 在销毁Filter时自动调用
18     }
19 }
20
21 // 在web.xml配置文件中如下配置:
22 <filter>
23     <filter-name>logFilter</filter-name>
24     <filter-class>com.xzg.cd.LogFilter</filter-class>
25 </filter>
26 <filter-mapping>
27     <filter-name>logFilter</filter-name>
28     <url-pattern>/*</url-pattern>
29 </filter-mapping>
```

从刚刚的示例代码中，我们发现，添加过滤器非常方便，不需要修改任何代码，定义一个实现 `javax.servlet.Filter` 的类，再改改配置就搞定了，完全符合开闭原则。那 Servlet Filter 是如何做到如此好的扩展性的呢？我想你应该已经猜到了，它利用的就是职责链模式。现在，我们通过剖析它的源码，详细地看看它底层是如何实现的。

在上一节课中，我们讲到，职责链模式的实现包含处理器接口（`IHandler`）或抽象类（`Handler`），以及处理器链（`HandlerChain`）。对应到 Servlet Filter，`javax.servlet.Filter` 就是处理器接口，`FilterChain` 就是处理器链。接下来，我们重点来看 `FilterChain` 是如何实现的。

不过，我们前面也讲过，Servlet 只是一个规范，并不包含具体的实现，所以，Servlet 中的 `FilterChain` 只是一个接口定义。具体的实现类由遵从 Servlet 规范的 Web 容器来提供，比如，`ApplicationFilterChain` 类就是 Tomcat 提供的 `FilterChain` 的实现类，源码如下所示。

为了让代码更易读懂，我对代码进行了简化，只保留了跟设计思路相关的代码片段。完整的代码你可以自行去 Tomcat 中查看。

 复制代码


```
1 public final class ApplicationFilterChain implements FilterChain {
2     private int pos = 0; //当前执行到了哪个filter
3     private int n; //filter的个数
4     private ApplicationFilterConfig[] filters;
```

```

5     private Servlet servlet;
6
7     @Override
8     public void doFilter(ServletRequest request, ServletResponse response) {
9         if (pos < n) {
10             ApplicationFilterConfig filterConfig = filters[pos++];
11             Filter filter = filterConfig.getFilter();
12             filter.doFilter(request, response, this);
13         } else {
14             // filter都处理完毕后, 执行servlet
15             servlet.service(request, response);
16         }
17     }
18
19     public void addFilter(ApplicationFilterConfig filterConfig) {
20         for (ApplicationFilterConfig filter:filters)
21             if (filter==filterConfig)
22                 return;
23
24         if (n == filters.length) { //扩容
25             ApplicationFilterConfig[] newFilters = new ApplicationFilterConfig[n + INCR];
26             System.arraycopy(filters, 0, newFilters, 0, n);
27             filters = newFilters;
28         }
29         filters[n++] = filterConfig;
30     }
31 }

```

ApplicationFilterChain 中的 doFilter() 函数的代码实现比较有技巧, 实际上是一个递归调用。你可以用每个 Filter (比如 LogFilter) 的 doFilter() 的代码实现, 直接替换 ApplicationFilterChain 的第 12 行代码, 一眼就能看出是递归调用了。我替换了一下, 如下所示。

 复制代码

```

1     @Override
2     public void doFilter(ServletRequest request, ServletResponse response) {
3         if (pos < n) {
4             ApplicationFilterConfig filterConfig = filters[pos++];
5             Filter filter = filterConfig.getFilter();
6             //filter.doFilter(request, response, this);
7             //把filter.doFilter的代码实现展开替换到这里
8             System.out.println("拦截客户端发送来的请求.");
9             chain.doFilter(request, response); // chain就是this
10            System.out.println("拦截发送给客户端的响应.");

```

```
11     } else {
12         // filter都处理完毕后，执行servlet
13         servlet.service(request, response);
14     }
15 }
```

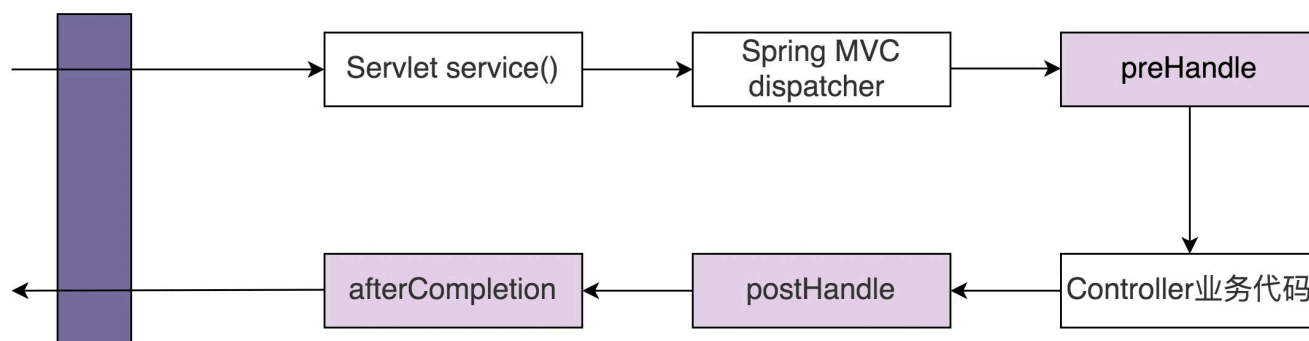
这样实现主要是为了在一个 `doFilter()` 方法中，支持双向拦截，既能拦截客户端发送来的请求，也能拦截发送给客户端的响应，你可以结合着 `LogFilter` 那个例子，以及对比待会要讲到的 `Spring Interceptor`，来自己理解一下。而我们上一节课给出的两种实现方式，都没法做到在业务逻辑执行的前后，同时添加处理代码。

Spring Interceptor


刚刚讲了 `Servlet Filter`，现在我们来讲一个功能上跟它非常类似的东西，`Spring Interceptor`，翻译成中文就是拦截器。尽管英文单词和中文翻译都不同，但这两者基本上可以看作一个概念，都用来实现对 `HTTP` 请求进行拦截处理。

它们不同之处在于，`Servlet Filter` 是 `Servlet` 规范的一部分，实现依赖于 `Web` 容器。`Spring Interceptor` 是 `Spring MVC` 框架的一部分，由 `Spring MVC` 框架来提供实现。客户端发送的请求，会先经过 `Servlet Filter`，然后再经过 `Spring Interceptor`，最后到达具体的业务代码中。我画了一张图来阐述一个请求的处理流程，具体如下所示。

Servlet Filter



在项目中，我们该如何使用 Spring Interceptor 呢？我写了一个简单的示例代码，如下所示。LogInterceptor 实现的功能跟刚才的 LogFilter 完全相同，只是实现方式上稍有区别。LogFilter 对请求和响应的拦截是在 doFilter() 一个函数中实现的，而 LogInterceptor 对请求的拦截在 preHandle() 中实现，对响应的拦截在 postHandle() 中实现。

 复制代码

```
1 public class LogInterceptor implements HandlerInterceptor {
2
3     @Override
4     public boolean preHandle(HttpServletRequest request, HttpServletResponse respon
5         System.out.println("拦截客户端发送来的请求.");
6         return true; // 继续后续的处理
7     }
8
9     @Override
10    public void postHandle(HttpServletRequest request, HttpServletResponse response
11        System.out.println("拦截发送给客户端的响应.");
12    }
13
14    @Override
15    public void afterCompletion(HttpServletRequest request, HttpServletResponse res
16        System.out.println("这里总是被执行.");
17    }
18 }
19
20 //在Spring MVC配置文件中配置interceptors
21 <mvc:interceptors>
22     <mvc:interceptor>
23         <mvc:mapping path="/*"/>
24         <bean class="com.xzg.cd.LogInterceptor" />
25     </mvc:interceptor>
26 </mvc:interceptors>
```

同样，我们还是来剖析一下，Spring Interceptor 底层是如何实现的。

当然，它也是基于职责链模式实现的。其中，HandlerExecutionChain 类是职责链模式中的处理器链。它的实现相较于 Tomcat 中的 ApplicationFilterChain 来说，逻辑更加清晰，不需要使用递归来实现，主要是因为它将请求和响应的拦截工作，拆分到了两个函数中实现。HandlerExecutionChain 的源码如下所示，同样，我对代码也进行了一些简化，只保留了关键代码。

```
1 public class HandlerExecutionChain {
2     private final Object handler;
3     private HandlerInterceptor[] interceptors;
4
5     public void addInterceptor(HandlerInterceptor interceptor) {
6         initInterceptorList().add(interceptor);
7     }
8
9     boolean applyPreHandle(HttpServletRequest request, HttpServletResponse response)
10        HandlerInterceptor[] interceptors = getInterceptors();
11        if (!ObjectUtils.isEmpty(interceptors)) {
12            for (int i = 0; i < interceptors.length; i++) {
13                HandlerInterceptor interceptor = interceptors[i];
14                if (!interceptor.preHandle(request, response, this.handler)) {
15                    triggerAfterCompletion(request, response, null);
16                    return false;
17                }
18            }
19        }
20        return true;
21    }
22
23    void applyPostHandle(HttpServletRequest request, HttpServletResponse response, M
24        HandlerInterceptor[] interceptors = getInterceptors();
25        if (!ObjectUtils.isEmpty(interceptors)) {
26            for (int i = interceptors.length - 1; i >= 0; i--) {
27                HandlerInterceptor interceptor = interceptors[i];
28                interceptor.postHandle(request, response, this.handler, mv);
29            }
30        }
31    }
32
33    void triggerAfterCompletion(HttpServletRequest request, HttpServletResponse resp
34        throws Exception {
35        HandlerInterceptor[] interceptors = getInterceptors();
36        if (!ObjectUtils.isEmpty(interceptors)) {
37            for (int i = this.interceptorIndex; i >= 0; i--) {
38                HandlerInterceptor interceptor = interceptors[i];
39                try {
40                    interceptor.afterCompletion(request, response, this.handler, ex);
41                } catch (Throwable ex2) {
42                    logger.error("HandlerInterceptor.afterCompletion threw exception", ex2);
43                }
44            }
45        }
46    }
47 }
```

在 Spring 框架中，DispatcherServlet 的 doDispatch() 方法来分发请求，它在真正的业务逻辑执行前后，执行 HandlerExecutionChain 中的 applyPreHandle() 和 applyPostHandle() 函数，用来实现拦截的功能。具体的代码实现很简单，你自己应该能脑补出来，这里就不罗列了。感兴趣的话，你可以自行去查看。

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

职责链模式常用在框架开发中，用来实现框架的过滤器、拦截器功能，让框架的使用者在不需要修改框架源码的情况下，添加新的过滤拦截功能。这也体现了之前讲到的对扩展开放、对修改关闭的设计原则。

今天，我们通过 Servlet Filter、Spring Interceptor 两个实际的例子，给你展示了在框架开发中职责链模式具体是怎么应用的。从源码中，我们还可以发现，尽管上一节课中我们有给出职责链模式的经典代码实现，但在实际的开发中，我们还是要具体问题具体对待，代码实现会根据不同的需求有所变化。实际上，这一点对于所有的设计模式都适用。

课堂讨论

1. 前面在讲代理模式的时候，我们提到，Spring AOP 是基于代理模式来实现的。在实际的项目开发中，我们可以利用 AOP 来实现访问控制功能，比如鉴权、限流、日志等。今天我们又讲到，Servlet Filter、Spring Interceptor 也可以用来实现访问控制。那在项目开发中，类似权限这样的访问控制功能，我们该选择三者（AOP、Servlet Filter、Spring Interceptor）中的哪个来实现呢？有什么参考标准吗？
2. 除了我们讲到的 Servlet Filter、Spring Interceptor 之外，Dubbo Filter、Netty ChannelPipeline 也是职责链模式的实际应用案例，你能否找一个你熟悉的并且用到职责链模式的框架，像我一样分析一下它的底层实现呢？

欢迎留言和我分享你的想法。如果有收获，欢迎你把这篇文章分享给你的朋友。

AI智能总结

本文深入介绍了Java开发中常用的过滤器和拦截器的实现方式，重点讲解了Servlet Filter和Spring Interceptor。通过示例代码和源码剖析，详细解释了它们的工作原理和底层实现，以及如何利用职责链模式实现了良好的扩展性。文章强调了职责链模式在框架开发中的重要性，以及如何实现对扩展开放、对修改关闭的设计原则。此外，还提出了在项目开发中选择AOP、Servlet Filter、Spring Interceptor来实现访问控制功能时的参考标准，并引出了其他职责链模式的实际应用案例。整体而言，本文为读者提供了深入了解过滤器和拦截器实现方式的重要知识，以及在框架开发中如何应用职责链模式的指导。

© 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。

全部留言 (64)

最新 精选



万历十五年

2020-11-28

三者应用范围不同: web filter 作用于容器，应用范围影响最大；spring interceptor 作用于框架，范围影响适中；aop 作用于业务逻辑，精细化处理，范围影响最小。

作者回复: 嗯嗯 🤔🤔🤔🤔🤔🤔

共 6 条评论 >

👍 108



cricket1981

2020-03-27

Filter 可以拿到原始的http请求，但是拿不到你请求的控制器和请求控制器中的方法的信息; Interceptor 可以拿到你请求的控制器和方法，却拿不到请求方法的参数; Aop 可以拿到方法的参数，但是却拿不到http请求和响应的对象

共 20 条评论 >

👍 176



常清静

2020-03-27

针对问题1而言，其实要实现一个鉴权的过滤器，通过以上3种方式都是可以去实现的，然而从粒度，场景，和方式上边有所区别，主要采取用哪个，还是有业务来决定去用，没有统一的参考标准。比如要对所有的web接口，进行统一的权限处理，不需要区分动作，写或者读，所有一视同仁，这种情况下，servlet的更加适合。针对一些存在状态的，比如做一些统一的去参数转换，cookie转uid之类，以及通用检验uid是否符合当前权限，则很用mvc较好，而ao

p粒度就可以分的更加细致了，在一些更新需要，查询不需要的，如分控，日志记录等，就比较适合

共 1 条评论 >

👍 62



webmin

2020-03-29

AOP、Servlet Filter、Spring Interceptor这三者可以从不同权限检查的范围大小的视角来应用：

1. Servlet Filter

运维部门需要对只供内部访问的服务进行IP限制或访问审查时，在容器这一层增加一个Filter，在发布时发布系统自动加挂这个Filter，这样对上层应用就是透明的，内网IP地址段增减或审查规则调整都不需要上层应用的开发人员去关心。

2. Spring Interceptor

由框架或基础服务部门来提供的微服务间相互调用的授权检查时，可以提供统一的SDK，由程序员在需要的服务上配置。

3. AOP

业务应用内权限检查，可以把权限检查在统一模块中实现，通过配置由AOP加插拦截检查。

共 1 条评论 >

👍 57



小晏子

2020-03-27

首先需要明确“访问控制功能”的粒度，如果访问控制功能要精确到每个请求，那么要使用AOP，AOP可以配置每个controller的访问权限。而Spring interceptor和servlet filter的粒度会粗一些，控制HttpRequest, HttpResponse的访问。另外servlet filter不能够使用 Spring 容器资源，只能在容器（如tomcat）启动时调用一次，而Spring Interceptor是一个Spring的组件，归Spring管理，配置在Spring文件中，因此能使用Spring里的任何资源、对象，例如 Service对象、数据源、事务管理等，通过IoC注入到Interceptor即可。相比较而言，Spring interceptor更灵活一些。

共 2 条评论 >

👍 28



楊_宵夜

2020-03-29

针对问题1，一把泪水想起了项目中的坑. 个人觉得最大的不同还是生效粒度的问题.

1. Servlet Filter是针对Servlet容器里的方法都能生效. 就是说Servlet容器里就算要把Spring换成别的框架，鉴权代码依然能生效.

2. Spring开头的就只能在Spring中生效，

2.1. 但更好还是在interceptor，因为interceptor天然的设计背景就是[在请求前，在相应后.]

2.2. 如果用AOP实现，就很依赖于AOP的pointcut设置，一不小心就会在[一次请求响应里]执行了[多次重复的鉴权服务].....



👍 20



筱乐乐哦

2020-03-27

1、个人感觉权限的话，属于api的调用，应该放在调用链比较靠前的位置，早发现早处理，所以用Servlet Filter会更好一些吧，如果是rpc层的话，例如dubbo，就需要 在实现filter的时候通过order吧filter得优先级提高一些，让这个filter先执行，个人感觉哈

2、Dubbo Filter的核心处理逻辑在ProtocolFilterWrapper类下的buildInvokerChain这个方法中，属于把所有的filter的类对象搞成一个list，通过遍历list去调用所有的filter，Netty ChannelPipeline我记得是一个双向链表，pipeline 中的节点的数据结构是 ChannelHandlerContext类，每个 ChannelHandlerContext 包含一个 ChannelHandler这种，支持从头尾开始传播事件，也就是触发调用，也可以从中间节点进行调用，入栈(read)是从head开始传播，也就是开始依次调用，出栈(write)是从tail开始传播，倒着调用。感觉算是对责任链的一个拓展使用，记不清了，得去看看代码，如果说错了，欢迎指点

共 3 条评论 >

👍 18



xk_

2020-04-26

课后题1，当然是全部都是用啊。

filter，可以控制所有的请求，用来处理网络攻击什么的。

interceptor可以控制用户和非用户的登录啊。

AOP可以控制用户角色对方方法的访问权限。

详情请见shiro，或者spring security。



👍 13



鹤涵

2020-12-30

Servlet Filter，Spring Interceptor，Spring AOP三者粒度是越来越细的。根据业务场景的覆盖度选择。

1. 比如限流就可以在Filter层去做，因为全局都需要限流防止服务被压垮。

2. 用户是否登录权限等可以使用Interceptor做。

3. 细粒度到类或者方法的控制使用AOP去做，比如日志 事务 方法级别权限。



11



Geek_3e636e

2020-12-18

一个请求从客户端到服务端再到响应，假设Filter、Interceptor、AOP都存在，经过的路径大概是：请求->Filter->Interceptor->AOP->核心业务处理->AOP->Interceptor->Filter->响应。

Filter、Interceptor、AOP在不同的节点所能感知到的数据状态都是不同的，姑且理解为域不同吧，要实现权限访问控制，肯定是在到达核心业务前植入权限控制逻辑，那就在“请求->Filter->Interceptor->AOP->核心业务处理”。

权限控制逻辑需要三个核心属性：资源、角色、角色资源映射。资源：一般我们用uri来标识某一个资源，或者可以通过注解等方式在方法上声明一个资源标识；角色和角色资源映射一般通过读取Session获取。那么权限控制逻辑放在那里取决于哪里可以拿到这两个信息？理论上角色和角色资源映射在哪里都可以读取到的。就看资源怎么表示了，如果你的资源是标识的servlet，那就通过Filter控制，如果你的资源是标识的Controller，可以在Interceptor控制，如果你的资源是标识的很深层的方法，可以在AOP控制



7