

62 | 职责链模式（上）：如何实现可灵活扩展算法的敏感信息过滤框架？

王争 2020-03-25



00:00 讲述：冯永奇 大小：6.46M 09:23

前几节课中，我们学习了模板模式、策略模式，今天，我们来学习职责链模式。这三种模式具有相同的作用：复用和扩展，在实际的项目开发中比较常用，特别是框架开发中，我们可以利用它们来提供框架的扩展点，能够让框架的使用者在不修改框架源码的情况下，基于扩展点定制化框架的功能。

今天，我们主要讲解职责链模式的原理和实现。除此之外，我还会利用职责链模式，带你实现一个可以灵活扩展算法的敏感词过滤框架。下一节课，我们会更加贴近实战，通过剖析 Servlet Filter、Spring Interceptor 来看，如何利用职责链模式实现框架中常用的过滤器、拦截器。

话不多说，让我们正式开始今天的学习吧！

职责链模式的原理和实现

职责链模式的英文翻译是 Chain Of Responsibility Design Pattern。在 GoF 的《设计模式》中，它是这么定义的：

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

翻译成中文就是：将请求的发送和接收解耦，让多个接收对象都有机会处理这个请求。将这些接收对象串成一条链，并沿着这条链传递这个请求，直到链上的某个接收对象能够处理它为止。

这么说比较抽象，我用更加容易理解的话来进一步解读一下。

在职责链模式中，多个处理器（也就是刚刚定义中说的“接收对象”）依次处理同一个请求。一个请求先经过 A 处理器处理，然后再把请求传递给 B 处理器，B 处理器处理完后再传递给 C 处理器，以此类推，形成一个链条。链条上的每个处理器各自承担各自的处理职责，所以叫作职责链模式。

关于职责链模式，我们先来看看它的代码实现。结合代码实现，你会更容易理解它的定义。职责链模式有多种实现方式，我们这里介绍两种比较常用的。

第一种实现方式如下所示。其中，Handler 是所有处理器类的抽象父类，handle() 是抽象方法。每个具体的处理器类（HandlerA、HandlerB）的 handle() 函数的代码结构类似，如果它能处理该请求，就不继续往下传递；如果不能处理，则交由后面的处理器来处理（也就是调用 successor.handle()）。HandlerChain 是处理器链，从数据结构的角度来看，它就是一个记录了链头、链尾的链表。其中，记录链尾是为了方便添加处理器。

```
1 public abstract class Handler {
2     protected Handler successor = null;
3
4     public void setSuccessor(Handler successor) {
5         this.successor = successor;
6     }
7
8     public abstract void handle();
9 }
10
11 public class HandlerA extends Handler {
12     @Override
13     public boolean handle() {
14         boolean handled = false;
15         //...
16         if (!handled && successor != null) {
17             successor.handle();
18         }
19     }
20 }
21
22 public class HandlerB extends Handler {
23     @Override
24     public void handle() {
25         boolean handled = false;
26         //...
27         if (!handled && successor != null) {
28             successor.handle();
29         }
30     }
31 }
32
33 public class HandlerChain {
34     private Handler head = null;
35     private Handler tail = null;
36
37     public void addHandler(Handler handler) {
38         handler.setSuccessor(null);
39
40         if (head == null) {
41             head = handler;
42             tail = handler;
43             return;
44         }
45
46         tail.setSuccessor(handler);
47         tail = handler;
48     }
49
50     public void handle() {
51         if (head != null) {
52             head.handle();
53         }
54     }
55 }
56
57 // 使用示例
58 public class Application {
59     public static void main(String[] args) {
60         HandlerChain chain = new HandlerChain();
61         chain.addHandler(new HandlerA());
62         chain.addHandler(new HandlerB());
63         chain.handle();
64     }
65 }
```

实际上，上面的代码实现不够优雅。处理器类的 handle() 函数，不仅包含自己的业务逻辑，还包含对下一个处理器的调用，也就是代码中的 successor.handle()。一个不熟悉这种代码结构的程序员，在添加新的处理器类的时候，很有可能忘记在 handle() 函数中调用 successor.handle()，这就会导致代码出现 bug。

针对这个问题，我们对代码进行重构，利用模板模式，将调用 successor.handle() 的逻辑从具体的处理器类中剥离出来，放到抽象父类中。这样具体的处理器类只需要实现自己的业务逻辑就可以了。重构之后的代码如下所示：

```
1 public abstract class Handler {
2     protected Handler successor = null;
3
4     public void setSuccessor(Handler successor) {
5         this.successor = successor;
6     }
7
8     public final void handle() {
9         boolean handled = doHandle();
10        if (successor != null && !handled) {
11            successor.handle();
12        }
13    }
14
15    protected boolean doHandle() {
16        boolean handled = false;
17        //...
18        return handled;
19    }
20 }
21
22 public class HandlerB extends Handler {
23     @Override
24     protected boolean doHandle() {
25         boolean handled = false;
26         //...
27         return handled;
28     }
29 }
30
31 // HandlerChain和Application代码不变
```

我们再来看第二种实现方式，代码如下所示。这种实现方式更加简单。HandlerChain 类用数组而非链表来保存所有的处理器，并且需要在 HandlerChain 的 handle() 函数中，依次调用每个处理器的 handle() 函数。

```
1 public interface IHandler {
2     boolean handle();
3 }
4
5 public class HandlerA implements IHandler {
6     @Override
7     public boolean handle() {
8         boolean handled = false;
9         //...
10        return handled;
11    }
12 }
13
14 public class HandlerB implements IHandler {
15     @Override
16     public boolean handle() {
17         boolean handled = false;
18         //...
19         return handled;
20     }
21 }
22
23 public class HandlerChain {
24     private List<IHandler> handlers = new ArrayList<>();
25
26     public void addHandler(IHandler handler) {
27         this.handlers.add(handler);
28     }
29
30     public void handle() {
31         for (IHandler handler : handlers) {
32             boolean handled = handler.handle();
33             if (handled) {
34                 break;
35             }
36         }
37     }
38 }
39
40 // 使用示例
41 public class Application {
42     public static void main(String[] args) {
43         HandlerChain chain = new HandlerChain();
44         chain.addHandler(new HandlerA());
45         chain.addHandler(new HandlerB());
46         chain.handle();
47     }
48 }
```

在 GoF 给出的定义中，如果处理器链上的某个处理器能够处理这个请求，那就不会继续往下传递请求。实际上，职责链模式还有一种变体，那就是请求会被所有的处理器都处理一遍，不存在中途终止的情况。这种变体也有两种实现方式：用链表存储处理器和用数组存储处理器，跟上面的两种实现方式类似，只需要稍微修改即可。

我这里只给出其中一种实现方式，如下所示。另外一种实现方式你对照着上面的实现自行修改。

```
1 public abstract class Handler {
2     protected Handler successor = null;
3
4     public void setSuccessor(Handler successor) {
5         this.successor = successor;
6     }
7
8     public final void handle() {
9         doHandle();
10        if (successor != null) {
11            successor.handle();
12        }
13    }
14
15    protected void doHandle() {
16        //...
17    }
18 }
19
20 public class HandlerB extends Handler {
21     @Override
22     protected void doHandle() {
23         //...
24     }
25 }
26
27 public class HandlerChain {
28     private Handler head = null;
29     private Handler tail = null;
30
31     public void addHandler(Handler handler) {
32         handler.setSuccessor(null);
33
34         if (head == null) {
35             head = handler;
36             tail = handler;
37             return;
38         }
39
40         tail.setSuccessor(handler);
41         tail = handler;
42     }
43
44     public void handle() {
45         if (head != null) {
46             head.handle();
47         }
48     }
49 }
50
51 // 使用示例
52 public class Application {
53     public static void main(String[] args) {
54         HandlerChain chain = new HandlerChain();
55         chain.addHandler(new HandlerA());
56         chain.addHandler(new HandlerB());
57         chain.handle();
58     }
59 }
```

职责链模式的应用场景举例

职责链模式的原理和实现讲完了，我们再通过一个实际的例子，来学习一下职责链模式的应用场景。

对于支持 UGC（User Generated Content，用户生成内容）的应用（比如论坛）来说，用户生成的内容（比如，在论坛中发表的帖子）可能会包含一些敏感词（比如涉黄、广告、反动等词汇）。针对这个应用场景，我们就可以利用职责链模式来过滤这些敏感词。

对于包含敏感词的内容，我们有两种处理方式，一种是直接禁止发布，另一种是给敏感词打马赛克（比如，用 *** 替换敏感词）之后再发布。第一种处理方式符合 GoF 给出的职责链模式的定义，第二种处理方式是职责链模式的变体。

我们这里只给出第一种实现方式的代码示例，如下所示，并且，我们只给出了代码实现的骨架，具体的敏感词过滤算法并没有给出，你可以参看我的另一个专栏 [《数据结构与算法之美》](#) 中多模式字符串匹配的相关章节自行实现。

```
1 public interface SensitiveWordFilter {
2     boolean doFilter(Content content);
3 }
4
5 public class SexyWordFilter implements SensitiveWordFilter {
6     @Override
7     public boolean doFilter(Content content) {
8         boolean legal = true;
9         //...
10        return legal;
11    }
12 }
13
14 // PoliticalWordFilter、AdsWordFilter 类代码结构与 SexyWordFilter 类似
15
16 public class SensitiveWordFilterChain {
17     private List<SensitiveWordFilter> filters = new ArrayList<>();
18
19     public void addFilter(SensitiveWordFilter filter) {
20         this.filters.add(filter);
21     }
22
23     // return true if content doesn't contain sensitive words.
24     public boolean filter(Content content) {
25         for (SensitiveWordFilter filter : filters) {
26             if (!filter.doFilter(content)) {
27                 return false;
28             }
29         }
30         return true;
31     }
32 }
33
34 public class ApplicationDemo {
35     public static void main(String[] args) {
36         SensitiveWordFilterChain filterChain = new SensitiveWordFilterChain();
37         filterChain.addFilter(new AdsWordFilter());
38         filterChain.addFilter(new SexyWordFilter());
39         filterChain.addFilter(new PoliticalWordFilter());
40
41         boolean legal = filterChain.filter(new Content());
42         if (!legal) {
43             // 不发表
44         } else {
45             // 发表
46         }
47     }
48 }
```

看了上面的实现，你可能会说，我像下面这样也可以实现敏感词过滤功能，而且代码更加简单，为什么非要使用职责链模式呢？这不是过度设计呢？

```
1 public class SensitiveWordFilter {
2     // return true if content doesn't contain sensitive words.
3     public boolean filter(Content content) {
4         if (!filterSexyWord(content)) {
5             return false;
6         }
7
8         if (!filterAdsWord(content)) {
9             return false;
10        }
11
12        if (!filterPoliticalWord(content)) {
13            return false;
14        }
15        return true;
16    }
17 }
18
19 private boolean filterSexyWord(Content content) {
20     //...
21 }
22
23 private boolean filterAdsWord(Content content) {
24     //...
25 }
26
27 private boolean filterPoliticalWord(Content content) {
28     //...
29 }
30 }
```

我们前面多次讲过，应用设计模式主要是为了应对代码的复杂性，让其满足开闭原则，提高代码的扩展性。这里应用职责链模式也不例外。实际上，我们在讲解 [策略模式](#) 的时候，也讲过类似的问题，比如，为什么要用策略模式？当时的给出的理由，与现在应用职责链模式的理由，几乎是一样的，你可以结合着当时的讲解一块来看下。

首先，我们来看，职责链模式如何应对代码的复杂性。

将大块代码逻辑拆分成函数，将大类拆分成小类，是应对代码复杂性的常用方法。应用职责链模式，我们把各个敏感词过滤函数继续拆分出来，设计成独立的类，进一步简化了 SensitiveWordFilter 类，让 SensitiveWordFilter 类的代码不会过多，过复杂。

其次，我们再来看，职责链模式如何让代码满足开闭原则，提高代码的扩展性。

当我们扩展新的过滤算法的时候，比如，我们还需要过滤特殊符号，按照非职责链模式的代码实现方式，我们需要修改 SensitiveWordFilter 的代码，违反开闭原则。不过，这样的修改还算比较集中，也是可以接受的。而职责链模式的实现方式更加优雅，只需要新添加一个 Filter 类，并且通过 addFilter() 函数将它添加到 FilterChain 中即可，其他代码完全不需要修改。

不过，你可能会说，即使使用职责链模式来实现，当添加新的过滤算法的时候，还是要修改客户端代码（ApplicationDemo），这样做也没有完全符合开闭原则。

实际上，细化一下的话，我们可以把上面的代码分成两类：框架代码和客户端代码。其中，ApplicationDemo 属于客户端代码，也就是使用框架的代码。除 ApplicationDemo 之外的代码属于敏感词过滤框架代码。

假设敏感词过滤框架并不是我们开发维护的，而是我们引入的一个第三方框架，我们要扩展一个新的过滤算法，不可能直接去修改框架的源码。这个时候，利用职责链模式就能达到开篇所说的，在不修改框架源码的情况下，基于职责链模式提供的扩展点，来扩展新的功能。换句话说，我们在框架这个代码范围内实现了开闭原则。

除此之外，利用职责链模式相对于不用职责链的实现方式，还有一个好处，那就是配置过滤算法更加灵活，可以只选择使用某几个过滤算法。

重点回顾

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

在职责链模式中，多个处理器依次处理同一个请求。一个请求先经过 A 处理器处理，然后再把请求传递给 B 处理器，B 处理器处理完后再传递给 C 处理器，以此类推，形成一个链条。链条上的每个处理器各自承担各自的处理职责，所以叫作职责链模式。

在 GoF 的定义中，一旦某个处理器能处理这个请求，就不会继续将请求传递给后续的处理了。当然，在实际的开发中，也存在对这个模式的变体，那就是请求不会中途终止传递，而是会被所有的处理器都处理一遍。

职责链模式有两种常用的实现。一种是使用链表来存储处理器，另一种是使用数组来存储处理器，后面一种实现方式更加简单。

课堂讨论

今天讲到利用职责链模式，我们可以让框架代码满足开闭原则。添加一个新的处理器，只需要修改客户端代码。如果我们希望客户端代码也满足开闭原则，不修改任何代码，你有什么办法可以做到呢？

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。

学习计划

学习 6 小时，
「免费」领课程！

3月23日 - 3月29日

【点击】图片，查看详情，参与学习

© 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。