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？

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。

```
public abstract class Handler {
  protected Handler successor = null;

  public void setSuccessor(Handler successor) {
    this.successor = successor;
  }

  public abstract void handle();
}
```

```
public class HandlerA extends Handler {
  @Override
  public boolean handle() {
    boolean handled = false;
    //...
    if (!handled && successor != null) {
      successor.handle();
    }
  }
}

public class HandlerB extends Handler {
  @Override
  public void handle() {
    boolean handled = false;
    //...
    if (!handled && successor != null) {
      successor.handle();
    }
  }
}

public class HandlerChain {
  private Handler head = null;
  private Handler tail = null;

  public void addHandler(Handler handler) {
    handler.setSuccessor(null);

    if (head == null) {
      head = handler;
      tail = handler;
      return;
    }

    tail.setSuccessor(handler);
    tail = handler;
  }

  public void handle() {
    if (head != null) {
      head.handle();
    }
  }
}

// 使用举例
public class Application {
  public static void main(String[] args) {
    HandlerChain chain = new HandlerChain();
    chain.addHandler(new HandlerA());
    chain.addHandler(new HandlerB());
    chain.handle();
  }
}
```

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：

```
public abstract class Handler {
  protected Handler successor = null;

  public void setSuccessor(Handler successor) {
    this.successor = successor;
  }

  public final void handle() {
    boolean handled = doHandle();
    if (successor != null && !handled) {
      successor.handle();
    }
  }

  protected abstract boolean doHandle();
}

public class HandlerA extends Handler {
  @Override
  protected boolean doHandle() {
    boolean handled = false;
    //...
    return handled;
  }
}

public class HandlerB extends Handler {
  @Override
  protected boolean doHandle() {
    boolean handled = false;
    //...
    return handled;
  }
}

// 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。

```
public interface IHandler {
  boolean handle();
}

public class HandlerA implements IHandler {
  @Override
  public boolean handle() {
    boolean handled = false;
    //...
    return handled;
  }
}
```

```
public class HandlerB implements IHandler {
  @Override
  public boolean handle() {
    boolean handled = false;
    //...
    return handled;
  }
}

public class HandlerChain {
  private List<IHandler> handlers = new ArrayList<>();

  public void addHandler(IHandler handler) {
    this.handlers.add(handler);
  }

  public void handle() {
    for (IHandler handler : handlers) {
      boolean handled = handler.handle();
      if (handled) {
        break;
      }
    }
  }
}

// 使用举例
public class Application {
  public static void main(String[] args) {
    HandlerChain chain = new HandlerChain();
    chain.addHandler(new HandlerA());
    chain.addHandler(new HandlerB());
    chain.handle();
  }
}
```

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。

```
public abstract class Handler {
  protected Handler successor = null;

  public void setSuccessor(Handler successor) {
    this.successor = successor;
  }

  public final void handle() {
    doHandle();
    if (successor != null) {
      successor.handle();
    }
  }
```

```
  protected abstract void doHandle();
}

public class HandlerA extends Handler {
  @Override
  protected void doHandle() {
    //...
  }
}

public class HandlerB extends Handler {
  @Override
  protected void doHandle() {
    //...
  }
}

public class HandlerChain {
  private Handler head = null;
  private Handler tail = null;

  public void addHandler(Handler handler) {
    handler.setSuccessor(null);

    if (head == null) {
      head = handler;
      tail = handler;
      return;
    }

    tail.setSuccessor(handler);
    tail = handler;
  }

  public void handle() {
    if (head != null) {
      head.handle();
    }
  }
}

// 使用举例
public class Application {
  public static void main(String[] args) {
    HandlerChain chain = new HandlerChain();
    chain.addHandler(new HandlerA());
    chain.addHandler(new HandlerB());
    chain.handle();
  }
}
```

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。

```
public interface SensitiveWordFilter {
  boolean doFilter(Content content);
}

public class SexyWordFilter implements SensitiveWordFilter {
  @Override
  public boolean doFilter(Content content) {
    boolean legal = true;
    //...
    return legal;
  }
}

// PoliticalWordFilter、AdsWordFilter类代码结构与SexyWordFilter类似

public class SensitiveWordFilterChain {
  private List<SensitiveWordFilter> filters = new ArrayList<>();

  public void addFilter(SensitiveWordFilter filter) {
    this.filters.add(filter);
  }

  // return true if content doesn't contain sensitive words.
  public boolean filter(Content content) {
    for (SensitiveWordFilter filter : filters) {
      if (!filter.doFilter(content)) {
        return false;
      }
    }
    return true;
  }
}

public class ApplicationDemo {
  public static void main(String[] args) {
    SensitiveWordFilterChain filterChain = new SensitiveWordFilterChain();
    filterChain.addFilter(new AdsWordFilter());
    filterChain.addFilter(new SexyWordFilter());
    filterChain.addFilter(new PoliticalWordFilter());

    boolean legal = filterChain.filter(new Content());
    if (!legal) {
      // 不发表
    } else {
      // 发表
    }
  }
}
```

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？这是不是过度设计呢？

```
public class SensitiveWordFilter {
  // return true if content doesn't contain sensitive words.
  public boolean filter(Content content) {
    if (!filterSexyWord(content)) {
      return false;
    }

    if (!filterAdsWord(content)) {
      return false;
    }

    if (!filterPoliticalWord(content)) {
      return false;
    }

    return true;
  }

  private boolean filterSexyWord(Content content) {
    //....
  }

  private boolean filterAdsWord(Content content) {
    //...
  }

  private boolean filterPoliticalWord(Content content) {
    //...
  }
}
```

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。

当我们要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，即便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器了。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。

精选留言：

- 小晏子 2020-03-25 07:29:06
  如果希望客户端代码也满足开闭原则，不修改任何代码，那么有个办法是不需要用户手动添加处理器，让框架代码能自动发现处理器，然后自动调用，要实现这个，就需要框架代码中自动发现接口实现类，可以通过注解和反射实现，然后将所有实现类都放到调用链中。这有个问题就是不够灵活，所有调用链可能都被执行，用户不能自由选择和组合处理器。 [7赞]

- 葫芦娃 2020-03-25 07:18:48
  通过配置文件配置需要的处理器，客户端代码也可以不改，通过反射动态加载 [5赞]

- 唔多志 2020-03-25 00:30:19
  职责链模式和装饰器模式太像了... [5赞]

- Michael 2020-03-25 09:13:39
  之前在公司做的一个关于金融日历的需求，就用到了老师说的指责链模式，一个用户有各种金融日历提醒，每个提醒逻辑不一样，通过给各个提醒服务打上注解标记，通过spring ioc容器中动态获取提醒服务对象，再利用Java中的future，并行调用，最终得到的提醒汇聚成了一个提醒列表，再通过排序规则返给前端，之前这么做了，代码复合开闭原则了，但不知道是责任链模式，老师讲了，才恍然大悟，是责任链的变体，所有链条都执行一遍。 [4赞]

- Geek_54edc1 2020-03-25 17:57:38
  通过配置文件，配置需要的过滤处理器，利用java的反射机制，动态的加载处理器类，创建处理器对象。 [2赞]

- ， 2020-03-25 09:20:44
  课后题:在项目中遇到跟责任链模式很相像的内容,业务是将天线扫描到的电子标签分发到不同的类中,这些类都实现了一个接口,有同样的方法,项目中的做法是在对象中维护一个ArrayList容器,在对象与天线建立连接后开始回调,将数据发送给各个实现类
  我们需要做的就是给某个接口添加一个实现类,然后将它存入对象的容器中,通过利用框架的依赖注入的方式,在类这个粒度上可以满足开闭原则 [2赞]

- test 2020-03-25 08:56:36
  在静态初始化块里面定义好字符串常量与处理类的对应关系，用户使用的时候在配置文件里面配置好 [2赞]

- Liam 2020-03-25 08:48:05
  1 工厂模式创建chain
  2 使用配置文件或注解添加节点
  3 反射自动装配chain [2赞]

- Monday 2020-03-25 08:20:13
  思考题，客户端组装过滤器时，从配置文件获取 [2赞]

- 韵呀 2020-03-25 02:28:09
  在项目开发中，无意用到过滤链思维，也就是老师说的职责链模式的变体。理解更深刻了。
  收获总结：
  标准的职责链模式，链上的处理器顺序执行，有一个处理器可以处理，就终止传递执行
  变体的职责链模式，链上的处理器会顺序执行，不会终止。

  职责链模式的两种实现方式：
  1.链表，只记录head和tail，结合模板方法模式，显式调用下一个处理器，具体处理器只要实现自己的处理逻辑即可。
  2.数组列表，将处理器放进一个list里，Java的arraylist底层就是一个数组，for循环调用所有的处理器 [2赞]

- 小刀 2020-03-25 08:47:06
  配置文件+反射 [1赞]

- 墨雨 2020-03-25 08:46:51

可以使用自定义注解来添加责任链 [1赞]

- 平凡世界 2020-03-27 09:11:02
  请求中间件，也算职责链的一种变体吧

- 陈天柱 2020-03-27 07:50:43
  mybatis里的插件机制就用到了责任链模式，且是整个链条都加工处理一次，同时使用配置扩展点的方式让客户端可以动态扩展插件，结合反射和动态代理创建插件

- 昌哥 2020-03-26 14:45:38
  职责链模式有两种形式
  1、多个处理器依次处理请求，某个处理器处理完成后后续处理器不在处理请求；
  2、多个处理器依次处理请求，直到调用链中所有请求处理完成；
  职责链有两种存储处理器的方式
  1、链表方式；
  2、数组方式；
  个人比较倾向于数据方式，实现起来比较简单，而且每个处理器的职责更加单一，无需存储后继处理器（易错点），用注解和反射的方式更容易实现处理器的自动加载

- 朱晋君 2020-03-26 10:50:42
  使用注解方式，定义一个注解接口，里面定义2个参数，一个是处理器在职责链的顺序order，第二个是是否开启处理open，在所有的处理器类上面加上这个注解并且指定这2个参数。这样新增加一个处理器时，只需要加上注解，并且定义好这2个参数就行了。

- Frank 2020-03-25 23:00:52
  打卡 今日学习 职责链模式，收获如下：
  将请求的发送和处理解耦合，使得多个处理器能够有机会依次处理请求。将这些处理器使用一个组件管理起来，使之在逻辑上形成一条链，请求在该链上依次被处理。职责链模式的变体有很多，比如Core J2ee Patern 中的 Intercepting Filter,Netty 中的Handler 处理器链 都可以看作是职责链的不同变体。GoF 中定义的职责链 与 Intercepting Filter在一定程度上是类似的，但也有所不同。 请求会经过Intercepting Filter中的所有过滤器，最后到达目标组件，被目标组件进行处理，个人觉得这种从“过滤”角度来考虑。而职责链模式中，请求会经过链上的处理器，这里的处理器需要从“处理”角度来思考，此处的处理器从两个角度来考虑，其一是是否要处理这个请求，其二是是否要将这个请求传递给下一个处理器。在Netty中处理器这个领域稍有不同，在Netty中处理器的角色划分是细粒度的，比如处理器A是进站处理器（只负责进来的I/O事件），处理器B是出站处理器（只负责出去的操作），处理器C既是进站处理器又是出站处理器。而在Servlet Filter请求和响应都要经过过滤器。Netty中这种划分方式体现了单一职责原则，将不同的事件回调拆分出来，减少复杂度。
  对于思考题：可以使用自定义注解来标识过滤器，在客户端应用启动时，去扫描指定包下指定注解的类，拿到所有过滤器的Class对象，遍历这些Class对象，通过反射机制创建实例。当添加新的过滤器时，只需要开发新的过滤器，并打上自定义注解即可。

- 子夜 2020-03-25 19:31:47
  之前面试被问到的问题：使用职责链设计模式和直接用for循环进行处理有什么不一样呢？
  我现在觉得使用设计模式有很好的扩展性，单独的for循环难以应对变化。请问老师和同学们有什么看法呢？

- Thinking 2020-03-25 14:55:02
  既然观察者模式也可以链式传递请求 责任链模式也可以广播请求 那他们有何区别？

- Jxin 2020-03-25 12:59:30

  1.客户端代码做了两件事。创建职责链实例和编排其执行顺序。所以要不改动客户端代码，就是说怎么把这两个职能从客户端代码抽离。

  1.创建职责链实例，这个简单。用spi或则基于spring都可以自动创建实例。扩展也不用动客户端代码，添加配置即可。至于，使用这组自动创建的实例集，依赖注入想必无需赘述。

  2.编排职责链执行顺序，这个也不难，但有点耦合。可以在入参数据中做文章。让入参数据带有 执行哪些责任链标识，并且呈现执行顺序即可。一个type数组字段即可。但这样业务代码逻辑就依赖了入参，这就感觉不是很友好，不知栏主有什么好方案吗？