

课程目标

内容定位

## 1. 策略模式

- 1.1. 策略模式的应用场景
- 1.2. 用策略模式实现促销优惠业务场景
- 1.3. 用策略模式实现选择支付方式的业务场景
- 1.4. 策略模式和委派模式结合使用
- 1.5. 策略模式在框架源码中的体现
- 1.6. 策略模式的优缺点

## 2. 责任链模式

- 2.1. 责任链模式的应用场景
- 2.2. 利用责任链模式进行数据校验拦截
- 2.3. 责任链模式和建造者模式结合使用
- 2.4. 责任链模式在源码中的体现
- 2.5. 责任链模式的优缺点

# 课程目标

- 1、掌握策略模式和责任链模式的应用场景；
- 2、通过学习策略模式来消除程序中大量的if..else..和switch语句；
- 3、掌握策略模式和委派模式的结合使用；
- 4、深刻理解责任链模式和建造者模式的结合应用。

# 内容定位

- 1、已经掌握建造者模式和委派模式的人群。
- 2、希望通过对策略模式的学习，来消除程序中大量的冗余代码和多重条件转移语句的人群。
- 3、希望通过学习责任链模式重构校验逻辑的人群。

## 1. 策略模式

策略模式（Strategy Pattern）又叫也叫政策模式（Policy Pattern），它是将定义的算法家族、分别封装起来，让它们之间可以互相替换，从而让算法的变化不会影响到使用算法的用户。属于行为型模式。

**原文：** Define a family of algorithms，encapsulate each one，and make them interchangeable.

策略模式使用的就是面向对象的继承和多态机制，从而实现同一行为在不同场景下具备不同实现。

## 1.1. 策略模式的应用场景

策略模式在生活场景中应用也非常多。比如一个人的交税比率与他的工资有关，不同的工资水平对应不同的税率。再比如我们在互联网移动支付的大背景下，每次下单后付款前，需要选择支付方式。



阶梯个税

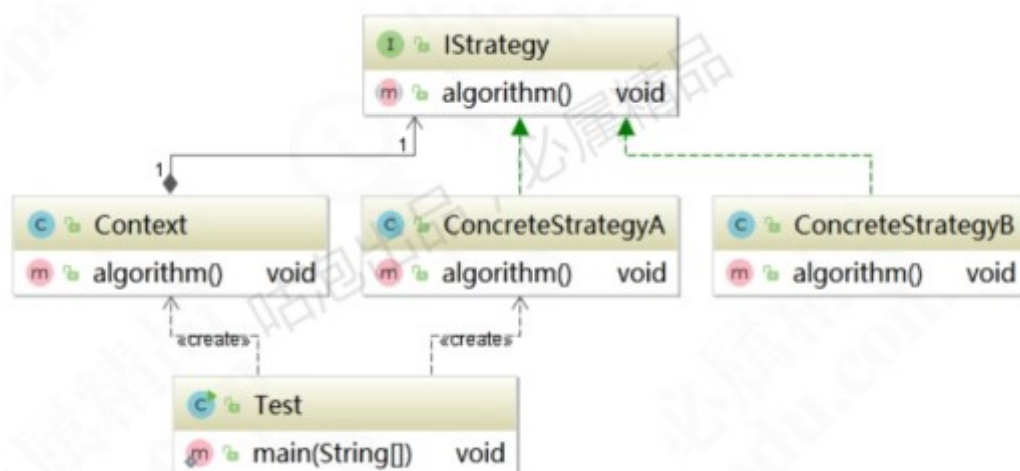


支付方式选择

策略模式可以解决在有多种算法相似的情况下，使用if..else或switch..case所带来的复杂性和臃肿性。在日常业务开发中，策略模式适用于以下场景：

- 1、针对同一类型问题，有多种处理方式，每一种都能独立解决问题；
- 2、算法需要自由切换的场景；
- 3、需要屏蔽算法规则的场景。

首先来看下策略模式的通用UML类图：



从UML类图中，我们可以看到，策略模式主要包含三种角色：

上下文角色（Context）：用来操作策略的上下文环境，屏蔽高层模块（客户端）对策略，算法的直接访问，封装可能存在的变化；

抽象策略角色（Strategy）：规定策略或算法的行为；

具体策略角色（ConcreteStrategy）：具体的策略或算法实现。

**注意：**策略模式中的上下文环境（Context），其职责本来是隔离客户端与策略类的耦合，让客户端完全与上下文环境沟通，无需关系具体策略。

## 1.2. 用策略模式实现促销优惠业务场景

大家都知道，我们咕泡学院的架构师课程经常会有优惠活动，优惠策略会有很多种可能如：领取优惠券抵扣、返现促销、拼团优惠。下面我们用代码来模拟，首先我们创建一个促销策略的抽象 IPromotionStrategy：

```

1 package cn.sitedev.promotion;
2
3 /**
4  * 促销策略抽象
5  */
6 public interface IPromotionStrategy {
7     void doPromotion();
8 }

```

然后分别创建优惠券抵扣策略 CouponStrategy类、返现促销策略CashbackStrategy类、拼团优惠策略 GroupbuyStrategy 类和无优惠策略EmptyStrategy类：

CouponStrategy类:

```

1 package cn.sitedev.promotion;
2
3 public class CouponStrategy implements IPromotionStrategy {
4     @Override
5     public void doPromotion() {
6         System.out.println("使用优惠券折扣");
7     }
8 }

```

CashbackStrategy类:

```

1 package cn.sitedev.promotion;
2
3 public class CashbackStrategy implements IPromotionStrategy {
4     @Override
5     public void doPromotion() {
6         System.out.println("使用返现");
7     }
8 }

```

GroupbuyStrategy 类:

```

1 package cn.sitedev.promotion;

```

```

2
3 public class GroupbuyStrategy implements IPromotionStrategy {
4     @Override
5     public void doPromotion() {
6         System.out.println("使用团购优惠");
7     }
8 }

```

EmptyStrategy类:

```

1 package cn.sitedev.promotion;
2
3 public class EmptyStrategy implements IPromotionStrategy {
4     @Override
5     public void doPromotion() {
6         System.out.println("无优惠");
7     }
8 }

```

然后创建促销活动方案PromotionActivity类：

```

1 package cn.sitedev.promotion;
2
3 public class PromotionActivity {
4     private IPromotionStrategy strategy;
5
6     public PromotionActivity(IPromotionStrategy strategy) {
7         this.strategy = strategy;
8     }
9
10    public void execute() {
11        this.strategy.doPromotion();
12    }
13 }

```

编写客户端测试类：

```

1 package cn.sitedev.promotion;

```

```

2
3 public class PromotionTest {
4     public static void main(String[] args) {
5         PromotionActivity activity = new PromotionActivity(new CouponStrategy());
6         activity.execute();
7     }
8 }

```

此时，小伙伴们会发现，如果把上面这段测试代码放到实际的业务场景其实并不实用。因为我们做活动时候往往是要根据不同的需求对促销策略进行动态选择的，并不会一次性执行多种优惠。所以，我们的代码通常会这样写：

```

1 package cn.sitedev.promotion;
2
3 public class PromotionTest2 {
4     public static void main(String[] args) {
5         PromotionActivity activity = null;
6         String promotionKey = "COUPON";
7
8         if ("COUPON".equals(promotionKey)) {
9             activity = new PromotionActivity(new CouponStrategy());
10        } else if ("CASHBACK".equals(promotionKey)) {
11            activity = new PromotionActivity(new CashbackStrategy());
12        } else if ("GROUPBUY".equals(promotionKey)) {
13            activity = new PromotionActivity(new GroupbuyStrategy());
14        } else {
15            activity = new PromotionActivity(new EmptyStrategy());
16        }
17
18        activity.execute();
19    }
20 }

```

这样改造之后，满足了业务需求，客户可根据自己的需求选择不同的优惠策略了。但是，经过一段时间的业务积累，我们的促销活动会越来越多。于是，我们的程序猿小哥哥就忙不赢了，每次上活动之前都要通宵改代码，而且要做重复测试，判断逻辑可能也变得越来越复杂。这时候，我们是不需要思考代码是不是应该重构了？回顾我们之前学过的设计模式应该如何来优化这段代码呢？其实，我们可以结合单例模式和工厂模式。创建 PromotionStrategyFactory 类：

```

1 package cn.sitedev.promotion.improved;

```

```

2
3 import cn.sitedev.promotion.*;
4
5 import java.util.HashMap;
6 import java.util.Map;
7
8 public class PromotionStrategyFactory {
9     private static Map<String, IPromotionStrategy> promotionStrategyMap = new HashMap<>()
10
11     private static final IPromotionStrategy EMPTY = new EmptyStrategy();
12
13     private PromotionStrategyFactory() {
14     }
15
16     static {
17         promotionStrategyMap.put(PromotionKey.COUPON, new CouponStrategy());
18         promotionStrategyMap.put(PromotionKey.CASHBACK, new CashbackStrategy());
19         promotionStrategyMap.put(PromotionKey.GROUPBUY, new GroupbuyStrategy());
20     }
21
22     private interface PromotionKey {
23         String COUPON = "COUPON";
24         String CASHBACK = "CASHBACK";
25         String GROUPBUY = "GROUPBUY";
26     }
27
28     public static IPromotionStrategy getPromotionStrategy(String promotionKey) {
29         IPromotionStrategy strategy = promotionStrategyMap.get(promotionKey);
30         return strategy == null ? EMPTY : strategy;
31     }
32
33     public static Set<String> getPromotionKeys() {
34         return promotionStrategyMap.keySet();
35     }
36 }

```

这时候我们客户端代码就应该这样写了：

```

1 package cn.sitedev.promotion.improved;
2
3 import cn.sitedev.promotion.IPromotionStrategy;

```

```

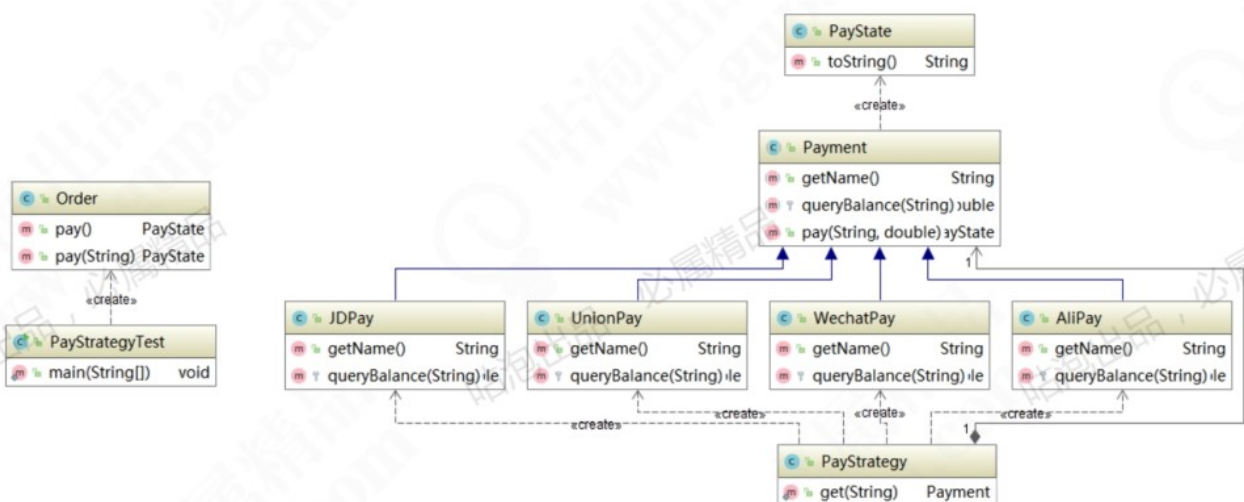
4
5 public class PromotionImprovedTest {
6     public static void main(String[] args) {
7         PromotionStrategyFactory.getPromotionKeys();
8
9         String promotionKey = "COUPON";
10
11         IPromotionStrategy strategy = PromotionStrategyFactory.getPromotionStrategy(promotionKey);
12         strategy.doPromotion();
13     }
14 }

```

代码优化之后，是不是我们程序猿小哥哥的维护工作就轻松了？每次上新活动，不影响原来的代码逻辑。

### 1.3. 用策略模式实现选择支付方式的业务场景

为了加深对策略模式的理解，我们再来举一个案例。相信小伙伴们都用过支付宝、微信支付、银联支付以及京东白条。一个常见的应用场景就是大家在下单支付时会提示选择支付方式，如果用户未选，系统也会默认好推荐的支付方式进行结算。来看一下类图，下面我们用策略模式来模拟此业务场景：



创建 `Payment` 抽象类，定义支付规范和支付逻辑，代码如下：

```

1 package cn.sitedev.pay.payport;
2
3 import cn.sitedev.pay.MsgResult;
4
5 /**
6  * 支付渠道
7  */

```

```

8 public abstract class Payment {
9     public abstract String getName();
10
11     /**
12      * 通用的逻辑放到抽象类里面实现
13      *
14      * @param uid
15      * @param amount
16      * @return
17      */
18     public MsgResult pay(String uid, double amount) {
19         // 余额是否足够
20         if (queryBalance(uid) < amount) {
21             return new MsgResult(500, "支付失败", "余额不足");
22         }
23         return new MsgResult(200, "支付成功", "支付金额:" + amount);
24     }
25
26     protected abstract double queryBalance(String uid);
27 }

```

分别创建具体的支付方式，支付宝AliPay类：

```

1 package cn.sitedev.pay.payport;
2
3 public class AliPay extends Payment {
4     @Override
5     public String getName() {
6         return "支付宝";
7     }
8
9     @Override
10    protected double queryBalance(String uid) {
11        return 900;
12    }
13 }

```

京东白条JDPay类：

```

1 package cn.sitedev.pay.payport;

```



```
2
3 public class JDPay extends Payment {
4     @Override
5     public String getName() {
6         return "京东白条";
7     }
8
9     @Override
10    protected double queryBalance(String uid) {
11        return 500;
12    }
13 }
```

微信支付WechatPay类：

```
1 package cn.sitedev.pay.payport;
2
3 public class WechatPay extends Payment {
4     @Override
5     public String getName() {
6         return "微信支付";
7     }
8
9     @Override
10    protected double queryBalance(String uid) {
11        return 263;
12    }
13 }
```

银联支付UnionPay类：

```
1 package cn.sitedev.pay.payport;
2
3 public class UnionPay extends Payment {
4     @Override
5     public String getName() {
6         return "银联支付";
7     }
8
9     @Override
```

```

10     protected double queryBalance(String uid) {
11         return 120;
12     }
13 }

```

创建支付状态的包装类MsgResult：

```

1 package cn.sitedev.pay;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6
7 /**
8  * 支付完成以后的状态
9  */
10 @Data
11 @AllArgsConstructor
12 @NoArgsConstructor
13 public class MsgResult {
14     private int code;
15     private String msg;
16     private Object data;
17
18 }

```

创建支付策略管理类：

```

1 package cn.sitedev.pay.payport;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 /**
7  * 支付策略管理
8  */
9 public class PayStrategy {
10
11     public static final String ALI_PAY = "AliPay";
12     public static final String JD_PAY = "JdPay";

```

```

13     public static final String WECHAT_PAY = "WechatPay";
14     private static final String UNION_PAY = "UnionPay";
15     public static final String DEFAULT_PAY = ALI_PAY;
16
17     private static Map<String, Payment> strategyMap = new HashMap<>();
18
19     static {
20         strategyMap.put(ALI_PAY, new AliPay());
21         strategyMap.put(JD_PAY, new JDPay());
22         strategyMap.put(WECHAT_PAY, new WechatPay());
23         strategyMap.put(UNION_PAY, new UnionPay());
24     }
25
26     public static Payment get(String payKey) {
27         if (strategyMap.containsKey(payKey)) {
28             return strategyMap.get(payKey);
29         }
30         return strategyMap.get(DEFAULT_PAY);
31     }
32
33 }

```

创建订单Order类：

```

1  package cn.sitedev.pay;
2
3  import cn.sitedev.pay.payport.PayStrategy;
4  import cn.sitedev.pay.payport.Payment;
5  import lombok.AllArgsConstructor;
6  import lombok.Data;
7  import lombok.NoArgsConstructor;
8
9  @Data
10 @AllArgsConstructor
11 @NoArgsConstructor
12 public class Order {
13     private String uid;
14     private String orderId;
15     private double amount;
16
17     public MsgResult pay() {

```

```

18         return pay(PayStrategy.DEFAULT_PAY);
19     }
20
21     public MsgResult pay(String payKey) {
22         Payment payment = PayStrategy.get(payKey);
23         System.out.println("欢迎使用:" + payment.getName());
24         System.out.println("本地交易金额为:" + amount + ", 开始扣款");
25         return payment.pay(uid, amount);
26     }
27 }

```

测试代码：

```

1 package cn.sitedev.pay;
2
3 import cn.sitedev.pay.payport.PayStrategy;
4
5 public class PayTest {
6     public static void main(String[] args) {
7         Order order = new Order("1", "20200315", 325.4);
8         System.out.println(order.pay());
9         System.out.println("=====");
10        System.out.println(order.pay(PayStrategy.WECHAT_PAY));
11    }
12 }

```

运行结果：

```

Run: PayTest x
D:\DevSoftWare\Java\jdk8\bin\java.exe ...
欢迎使用:支付宝
本地交易金额为:325.4, 开始扣款
MsgResult(code=200, msg=支付成功, data=支付金额:325.4)
=====
欢迎使用:微信支付
本地交易金额为:325.4, 开始扣款
MsgResult(code=500, msg=支付失败, data=余额不足)

Process finished with exit code 0

```

希望通过大家耳熟能详的业务场景来举例，让小伙伴们更深刻地理解策略模式。希望小伙伴们在面试和工作体现出自己的优势。

## 1.4. 策略模式和委派模式结合使用

在上面的代码中我们列举了几个非常常见的业务场景，相信小伙伴对委派模式和策略模式有了非常深刻的理解了。现在，我们再来回顾一下，DispatcherServlet的委派逻辑，代码如下：

```
1 package cn.sitedev.delegate.mvc.controller;
2
3 import javax.servlet.http.HttpServletResponse;
4 import java.io.IOException;
5
6 public class MemberController {
7     public void getMemberById(HttpServletResponse response, String mid) {
8         try {
9             response.getWriter().write("getMemberById=>" + mid);
10        } catch (IOException e) {
11            e.printStackTrace();
12        }
13    }
14 }
15 ///////////////////////////////////////////////////
16 package cn.sitedev.delegate.mvc.controller;
17
18 import javax.servlet.http.HttpServletResponse;
19 import java.io.IOException;
20
21 public class OrderController {
22     public void getOrderById(HttpServletResponse response, String mid) {
23         try {
24             response.getWriter().write("getOrderById=>" + mid);
25        } catch (IOException e) {
26            e.printStackTrace();
27        }
28    }
29 }
30 ///////////////////////////////////////////////////
31 package cn.sitedev.delegate.mvc.controller;
32
33 import javax.servlet.http.HttpServletResponse;
34 import java.io.IOException;
```

```

35
36 public class SystemController {
37     public void logout(HttpServletResponse response) {
38         try {
39             response.getWriter().write("logout");
40         } catch (IOException e) {
41             e.printStackTrace();
42         }
43     }
44 }
45 //////////////////////////////////////
46 package cn.sitedev.delegate.mvc;
47
48 import cn.sitedev.delegate.mvc.controller.MemberController;
49 import cn.sitedev.delegate.mvc.controller.OrderController;
50 import cn.sitedev.delegate.mvc.controller.SystemController;
51
52 import javax.servlet.ServletException;
53 import javax.servlet.http.HttpServlet;
54 import javax.servlet.http.HttpServletRequest;
55 import javax.servlet.http.HttpServletResponse;
56 import java.io.IOException;
57
58 public class DispatcherServlet extends HttpServlet {
59
60     private void doDispatch(HttpServletRequest request, HttpServletResponse response) {
61         String uri = request.getRequestURI();
62         String mid = request.getParameter("mid");
63
64         if ("/getMemberById".equals(uri)) {
65             new MemberController().getMemberById(response, mid);
66         } else if ("/getOrderById".equals(uri)) {
67             new OrderController().getOrderById(response, mid);
68         } else if ("/logout".equals(uri)) {
69             new SystemController().logout(response);
70         } else {
71             response.getWriter().write("404 NOT FOUND");
72         }
73     }
74
75     @Override
76     protected void service(HttpServletRequest req, HttpServletResponse resp) throws ServletException {
77         try {

```

```

78         doDispatch(req, resp);
79     } catch (Exception e) {
80         e.printStackTrace();
81     }
82 }
83 }

```

这样的代码扩展性不太优雅，也不现实，因为我们实际项目中一定不止这几个Controller，往往是成千上万个Controller，显然，我们不能写成成千上万个if..else.。那么我们如何来改造呢？小伙伴们一定首先就想到了策略模式，来看一下我是怎么优化的：

```

1  package cn.sitedev.delegate.mvc;
2
3  import cn.sitedev.delegate.mvc.controller.MemberController;
4  import lombok.AllArgsConstructor;
5  import lombok.Data;
6
7  import javax.servlet.ServletException;
8  import javax.servlet.http.HttpServlet;
9  import javax.servlet.http.HttpServletRequest;
10 import javax.servlet.http.HttpServletResponse;
11 import java.io.IOException;
12 import java.lang.reflect.InvocationTargetException;
13 import java.lang.reflect.Method;
14 import java.util.ArrayList;
15 import java.util.List;
16
17 public class DispatcherServletUsingStrategyPattern extends HttpServlet {
18     private List<Handler> handlerMapping = new ArrayList<>();
19
20     @Override
21     public void init() {
22         Class<?> memberControllerClass = MemberController.class;
23         try {
24             handlerMapping.add(new Handler(memberControllerClass.newInstance(),
25                 memberControllerClass.getMethod("getMemberById",
26                     new Class[]{HttpServletResponse.class, String.class}),
27                     "/getMemberById"));
28         } catch (InstantiationException e) {
29             e.printStackTrace();
30         } catch (IllegalAccessException e) {

```

```

31         e.printStackTrace();
32     } catch (NoSuchMethodException e) {
33         e.printStackTrace();
34     }
35 }
36
37 private void doDispatch(HttpServletRequest request, HttpServletResponse response) throws ServletException {
38     // 1. 获取用户请求的url
39     // 如果按照J2EE的标准，每个url对对应一个Servlet，url由浏览器输入
40     String uri = request.getRequestURI();
41
42     // 2. Servlet拿到url以后，要做权衡，
43     // 根据用户请求的url，去找到这个url对应的某一个Java类的方法
44
45     // 3. 通过拿到的url去handlerMapping(我们把它认为是策略常量)
46     Handler handler = null;
47     for (Handler h : handlerMapping) {
48         if (uri.equals(h.getUrl())) {
49             handler = h;
50             break;
51         }
52     }
53     if (handler == null) {
54         response.getWriter().write("<h1>404 NOT FOUND.....</h1>");
55         return;
56     }
57     // 4. 将具体的任务分发给Method(通过反射去调用其对应的方法)
58     // Object object = null;
59     try {
60         handler.getMethod().invoke(handler.getController(), response, request.getParameterNames().toArray(new String[0]));
61     } catch (IllegalAccessException e) {
62         e.printStackTrace();
63     } catch (InvocationTargetException e) {
64         e.printStackTrace();
65     }
66 }
67
68 // 5. 获取到Method执行的结果，通过Response返回回去
69 // response.getWriter().write(object.toString());
70 }
71
72 @Override
73 protected void service(HttpServletRequest req, HttpServletResponse resp) throws ServletException {

```



```

74         try {
75             doDispatch(req, resp);
76         } catch (Exception e) {
77             e.printStackTrace();
78         }
79     }
80
81     @Data
82     @AllArgsConstructor
83     public class Handler<T> {
84         private T controller;
85         private Method method;
86         private String url;
87     }
88 }

```

上面的代码我结合了策略模式、工厂模式、单例模式。当然，我的优化方案不一定是最完美的，仅代表个人观点。感兴趣的小伙伴可以继续思考，如何让这段代码变得更优雅。

## 1.5. 策略模式在框架源码中的体现

首先来看JDK中一个比较常用的比较器Comparator 接口我们看到的一个大家常用的compare()方法，就是一个策略抽象实现：

```

1 @FunctionalInterface
2 public interface Comparator<T> {
3     ...
4     int compare(T o1, T o2);
5     ...

```

Comparator抽象下面有非常多的实现类我们经常会把Comparator作为参数传入作为排序策略，例如Arrays 类的 parallelSort方法等：

```

1 public class Arrays {
2     ...
3     public static <T> void parallelSort(T[] a, int fromIndex, int toIndex,
4                                         Comparator<? super T> cmp) {
5         rangeCheck(a.length, fromIndex, toIndex);
6         if (cmp == null)
7             cmp = NaturalOrder.INSTANCE;

```

```

8      int n = toIndex - fromIndex, p, g;
9      if (n <= MIN_ARRAY_SORT_GRAN ||
10         (p = ForkJoinPool.getCommonPoolParallelism()) == 1)
11         TimSort.sort(a, fromIndex, toIndex, cmp, null, 0, 0);
12     else
13         new ArraysParallelSortHelpers.FJObject.Sorter<T>
14             (null, a,
15              (T[])Array.newInstance(a.getClass().getComponentType(), n),
16              fromIndex, n, 0, ((g = n / (p << 2)) <= MIN_ARRAY_SORT_GRAN) ?
17              MIN_ARRAY_SORT_GRAN : g, cmp).invoke();
18     }
19     ...

```

还有TreeMap的构造方法：

```

1 public class TreeMap<K,V>
2     extends AbstractMap<K,V>
3     implements NavigableMap<K,V>, Cloneable, java.io.Serializable
4 {
5     ...
6     public TreeMap(Comparator<? super K> comparator) {
7         this.comparator = comparator;
8     }

```

这就是Comparator在JDK源码中的应用。那我们来看策略模式在Spring 源码中的应用，来看Resource类:

```

1 package org.springframework.core.io;
2
3 import java.io.File;
4 import java.io.IOException;
5 import java.io.InputStream;
6 import java.net.URI;
7 import java.net.URL;
8 import java.nio.channels.Channels;
9 import java.nio.channels.ReadableByteChannel;
10
11 import org.springframework.lang.Nullable;
12
13 public interface Resource extends InputStreamSource {

```

```
14
15     boolean exists();
16
17     default boolean isReadable() {
18         return exists();
19     }
20
21     default boolean isOpen() {
22         return false;
23     }
24
25     default boolean isFile() {
26         return false;
27     }
28
29     URL getURL() throws IOException;
30
31     URI getURI() throws IOException;
32
33     File getFile() throws IOException;
34
35     default ReadableByteChannel readableChannel() throws IOException {
36         return Channels.newChannel(getInputStream());
37     }
38
39     long contentLength() throws IOException;
40
41     long lastModified() throws IOException;
42
43     Resource createRelative(String relativePath) throws IOException;
44
45     @Nullable
46     String getFilename();
47
48     String getDescription();
49
50 }
```

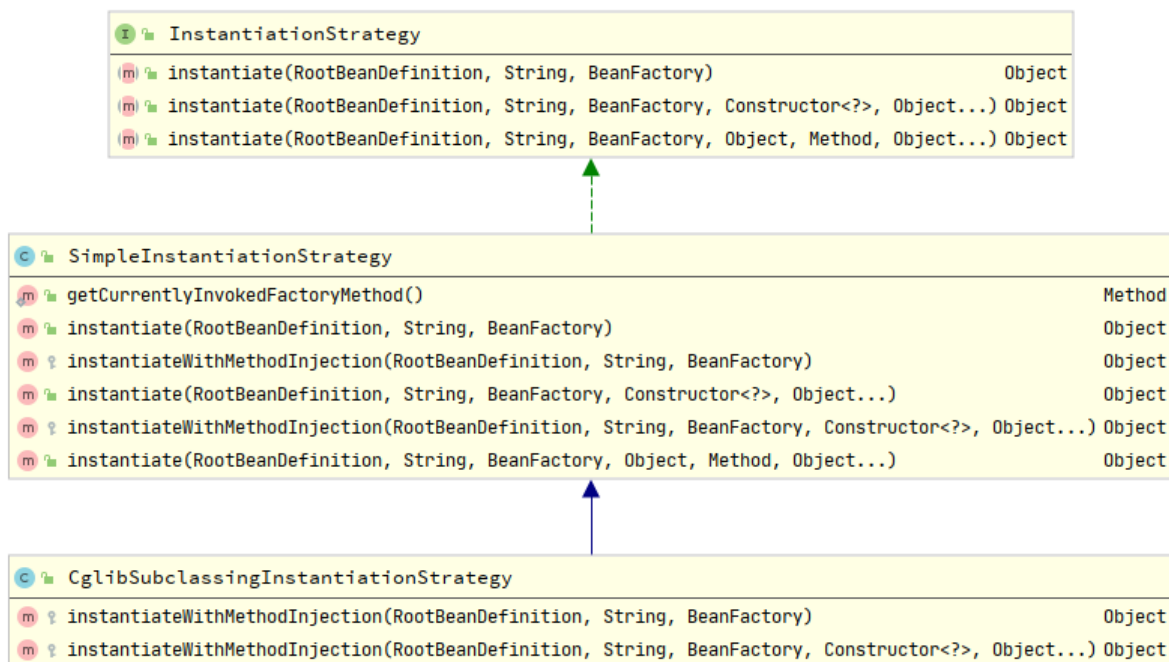
我们虽然没有直接使用Resource类，但是我们经常使用它的子类，例如：

```
* @see WritableResource
* @see ContextResource
* @see UrlResource
* @see FileUrlResource
* @see FileSystemResource
* @see ClassPathResource
* @see ByteArrayResource
* @see InputStreamResource
```

还有一个非常典型的场景，Spring的初始化也采用了策略模式，不同的类型的类采用不同的初始化策略。首先有一个InstantiationStrategy接口，我们来看一下源码：

```
1 package org.springframework.beans.factory.support;
2
3 import java.lang.reflect.Constructor;
4 import java.lang.reflect.Method;
5
6 import org.springframework.beans.BeansException;
7 import org.springframework.beans.factory.BeanFactory;
8 import org.springframework.lang.Nullable;
9
10 public interface InstantiationStrategy {
11
12     Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFactory ow
13         throws BeansException;
14
15     Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFactory ow
16         Constructor<?> ctor, Object... args) throws BeansException;
17
18     Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFactory ow
19         @Nullable Object factoryBean, Method factoryMethod, Object... args)
20         throws BeansException;
21
22 }
```

顶层的策略抽象非常简单，但是它下面有两种策略 SimpleInstantiationStrategy和 CglibSubclassingInstantiationStrategy，我们看一下类图：



打开类图我们还发现CglibSubclassingInstantiationStrategy 策略类还继承了 SimpleInstantiationStrategy类，说明在实际应用中多种策略之间还可以继承使用。小伙伴们可以作为一个参考，在实际业务场景中，可以根据需要来设计。

## 1.6. 策略模式的优缺点

优点：

- 1、策略模式符合开闭原则。
- 2、避免使用多重条件转移语句，如if..else...语句、switch 语句
- 3、使用策略模式可以提高算法的保密性和安全性。

缺点：

- 1、客户端必须知道所有的策略，并且自行决定使用哪一个策略类。
- 2、代码中会产生非常多策略类，增加维护难度。

## 2. 责任链模式

责任链模式（Chain of Responsibility Pattern）是将链中每一个节点看作是一个对象，每个节点处理的请求均不同，且内部自动维护一个下一节点对象。当一个请求从链式的首端发出时，会沿着链的路径依次传递给每一个节点对象，直至有对象处理这个请求为止。属于行为型模式。

**原文：** 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.

**解释：** 使多个对象都有机会处理请求，从而避免了请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

### 2.1. 责任链模式的应用场景

在日常生活中责任链模式还是比较常见的，我们平时工作处理一些事务，往往是各部门协同合作的完成某一个任务。而每个部门都有各自的职责，因此，很多时候事情完成一半，便会转交给下一个部门，直到所有部门都过一遍之后事情才能完成。还有我们平时俗话说的过五关、斩六将其实也是一种责任链。



工作生活中的审批流程

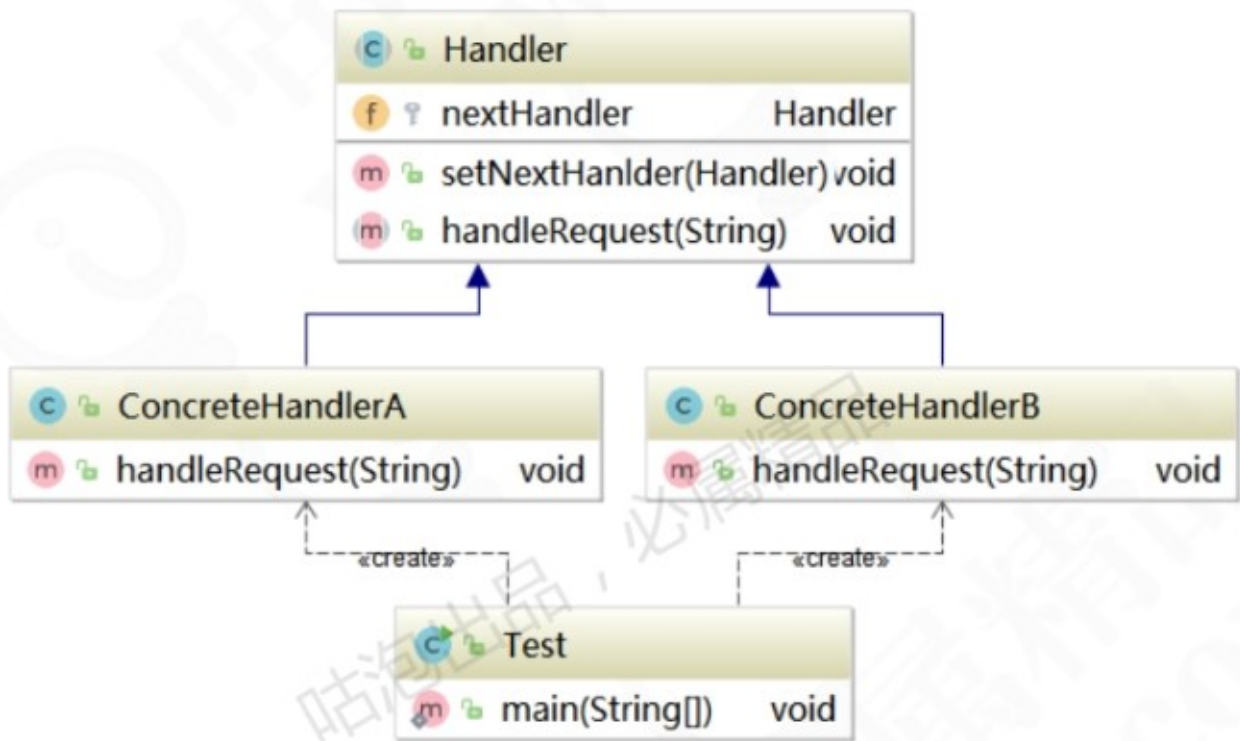


过五关、斩六将

责任链模式主要是解耦了请求与处理，客户只需将请求发送到链上即可，无需关心请求的具体内容和处理细节，请求会自动进行传递直至有节点对象进行处理。适用于以下应用场景：

- 1、多个对象可以处理同一请求，但具体由哪个对象处理则在运行时动态决定；
- 2、在不明确指定接收者的情况下，向多个对象中的一个提交一个请求；
- 3、可动态指定一组对象处理请求。

首先来看下责任链模式的通用UML类图：



从UML类图中，我们可以看到，责任链模式主要包含两种角色：

- 抽象处理者（Handler）：定义一个请求处理的方法，并维护一个下一个处理节点Handler对象的引用；
- 具体处理者（ConcreteHandler）：对请求进行处理，如果不感兴趣，则进行转发。

责任链模式的本质是解耦请求与处理，让请求在处理链中能进行传递与被处理；理解责任链模式应当理解的是其模式（道）而不是其具体实现（术），责任链模式的独到之处是其将节点处理者组合成了链式结构，并允许节点自身决定是否进行请求处理或转发，相当于让请求流动了起来。

## 2.2. 利用责任链模式进行数据校验拦截

首先，创建一个实体类Member：

```
1 package cn.sitedev.auth;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6
7 @Data
8 @AllArgsConstructor
9 @NoArgsConstructor
10 public class Member {
11     private String loginName;
12     private String loginPass;
13     private String roleName;
14
15     public Member(String loginName, String loginPass) {
16         this.loginName = loginName;
17         this.loginPass = loginPass;
18     }
19 }
```

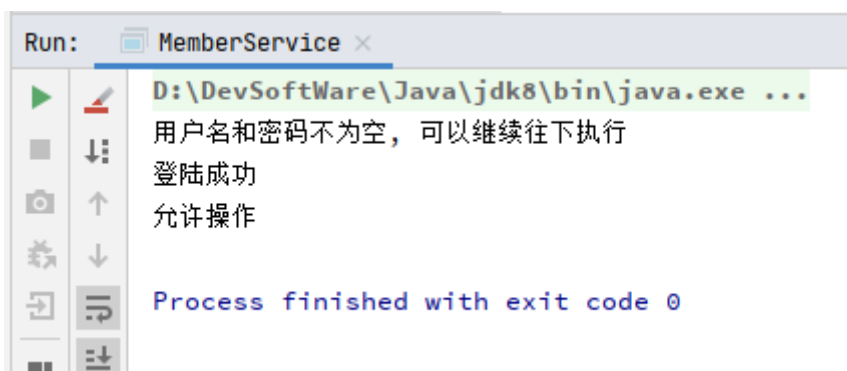
然后来看一段我们经常写的代码：

```
1 package cn.sitedev.auth.old;
2
3 import cn.sitedev.auth.Member;
4
5 public class MemberService {
6     public void login(String loginName, String loginPass) {
7         if (loginName == null || loginName.isEmpty() || loginPass == null || loginPass.
8             System.out.println("用户名或密码为空");
9             return;
10         }
11         System.out.println("用户名和密码不为空，可以继续往下执行");
12     }
13 }
```

```

12
13     Member member = checkExists(loginName, loginPass);
14     if (member == null) {
15         System.out.println("用户不存在");
16         return;
17     }
18     System.out.println("登陆成功");
19
20     if (!"管理员".equals(member.getRoleName())) {
21         System.out.println("您不是管理员，没有操作权限");
22         return;
23     }
24     System.out.println("允许操作");
25
26 }
27
28 private Member checkExists(String loginName, String loginPass) {
29     Member member = new Member(loginName, loginPass);
30     member.setRoleName("管理员");
31     return member;
32 }
33
34 public static void main(String[] args) {
35     MemberService memberService = new MemberService();
36     memberService.login("sitedev", "sitedev");
37 }
38 }

```



请问各位小伙伴，你们是不是经常这么干？上面的代码中，主要功能是做了登录前的数据验证，然后，判断逻辑是有先后顺序的。首先做非空判断，然后检查账号是否有效，最终获得用户角色。然后根据用户角色所拥有的权限再匹配是否有操作权限。那么这样的检验性代码一般都是必不可少的，但是写在具体的业务代码又显得代码非常臃肿，因此我们可以用责任链模式，将这些检查步骤串联起来，而且不影响代码美观。可以使得我们在编码时可以更加专注于某一个具体的业务逻辑处理。



下面我们用责任链模式来优化一下代码，首先创建一个Handler类：

```
1 package cn.sitedev.auth.chain;
2
3 import cn.sitedev.auth.Member;
4
5 public abstract class Handler {
6     protected Handler chain;
7
8     public void next(Handler handler) {
9         this.chain = handler;
10    }
11
12    public abstract void doHandle(Member member);
13 }
```

我们分别创建非空校验ValidateHandler类、登录校验LoginHandler类和权限校验AuthHandler类，来看代码ValidateHandler类：

```
1 package cn.sitedev.auth.chain;
2
3 import cn.sitedev.auth.Member;
4
5 public class ValidateHandler extends Handler {
6     @Override
7     public void doHandle(Member member) {
8         if (member.getLoginName() == null || member.getLoginName().isEmpty() || member.
9             System.out.println("用户名或密码为空");
10            return;
11        }
12        System.out.println("用户名和密码校验成功，可以往下执行");
13        chain.doHandle(member);
14    }
15 }
```

LoginHandler类：

```
1 package cn.sitedev.auth.chain;
2
```

```

3 import cn.sitedev.auth.Member;
4
5 public class LoginHandler extends Handler {
6     @Override
7     public void doHandle(Member member) {
8         System.out.println("登陆成功");
9         member.setRoleName("管理员");
10        chain.doHandle(member);
11    }
12 }

```

AuthHandler类：

```

1 package cn.sitedev.auth.chain;
2
3 import cn.sitedev.auth.Member;
4
5 public class AuthHandler extends Handler {
6     @Override
7     public void doHandle(Member member) {
8         if (!"管理员".equals(member.getRoleName())) {
9             System.out.println("您不是管理员，没有操作权限");
10            return;
11        }
12        System.out.println("您是管理员，允许操作");
13    }
14 }

```

接下来，修改MemberService中的代码，其实我们只需要将前面定义好的几个Handler根据业务需求串联起来，形成一条链即可

```

1 package cn.sitedev.auth.chain;
2
3 import cn.sitedev.auth.Member;
4
5 public class MemberService {
6     public void login(String loginName, String loginPass) {
7         Handler validateHandler = new ValidateHandler();
8         Handler loginHandler = new LoginHandler();
9         Handler authHandler = new AuthHandler();

```

```

10
11     validateHandler.next(loginHandler);
12     loginHandler.next(authHandler);
13
14     validateHandler.doHandle(new Member(loginName, loginPass));
15 }
16 }

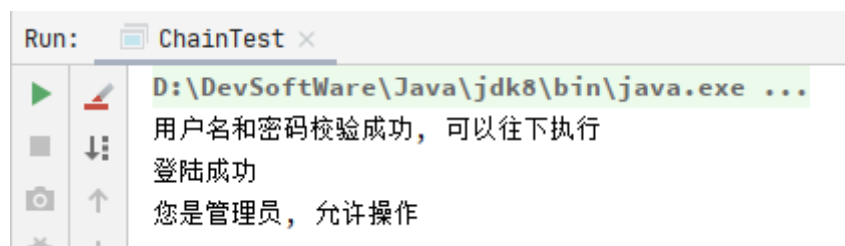
```

来看客户端调用代码：

```

1 package cn.sitedev.auth.chain;
2
3 public class ChainTest {
4     public static void main(String[] args) {
5         MemberService service = new MemberService();
6         service.login("sitedev", "sitedev");
7     }
8 }

```



其实我们平时使用的很多权限校验框架都是运用这样一个原理，将各个维度的权限处理解耦之后再串联起来，各自只处理各自相关的职责。如果职责与自己不相关则抛给链上的下一个Handler，俗称踢皮球。

## 2.3. 责任链模式和建造者模式结合使用

因为责任链模式具备链式结构，而上面代码中，我们看到，负责组装链式结构的角色是 MemberService，当链式结构较长时，MemberService的工作会非常繁琐，并且MemberService 代码相对臃肿，且后续更改处理者或消息类型时，都必须在MemberService中进行修改，不符合开闭原则。产生这些问题的原因就是链式结构的组装过于复杂，而对于复杂结构的创建，很自然我们会想到建造者模式，使用建造者模式，我们完全可以对MemberService 指定的处理节点对象进行自动链式组装，客户只需指定处理节点对象，其他任何事情都无需关心，并且客户指定的处理节点对象顺序不同，构造出来的链式结构也随之不同。我们来改造一下，修改Handler的代码：

```

1 package cn.sitedev.auth.chainbuilder;

```

```

2
3 import cn.sitedev.auth.Member;
4
5 public abstract class Handler<T> {
6     protected Handler chain;
7
8     public void next(Handler handler) {
9         this.chain = handler;
10    }
11
12    public abstract void doHandle(Member member);
13
14    public static class Builder<T> {
15        private Handler<T> head;
16        private Handler<T> tail;
17
18        public Builder<T> addHandler(Handler<T> handler) {
19            if (this.head == null) {
20                // 节点为空
21                this.head = this.tail = handler;
22                return this;
23            }
24            // 往尾部添加节点
25            this.tail.next(handler);
26            this.tail = handler;
27            return this;
28        }
29
30        public Handler<T> build() {
31            return this.head;
32        }
33    }
34
35 }

```

然后，修改MemberService的代码：

```

1 package cn.sitedev.auth.chainbuilder;
2
3 import cn.sitedev.auth.Member;
4

```

```

5 public class MemberService {
6     public void login(String loginName, String loginPass) {
7         Handler.Builder builder = new Handler.Builder();
8         builder.addHandler(new ValidateHandler()).addHandler(new LoginHandler()).addHan
9
10        builder.build().doHandle(new Member(loginName, loginPass));
11
12    }
13 }

```

因为建造者模式要构建的是节点处理者，因此我们把Builder作为Handler的静态内部类，并且因为客户端无需进行链式组装，因此我们还可以把链式组装方法next()方法设置为private，使Handler更加高内聚，代码如下：

```

1 package cn.sitedev.auth.chainbuilder;
2
3 import cn.sitedev.auth.Member;
4
5 public abstract class Handler<T> {
6     protected Handler chain;
7
8     private void next(Handler handler) {
9         this.chain = handler;
10    }
11
12    public abstract void doHandle(Member member);
13
14    public static class Builder<T> {
15        private Handler<T> head;
16        private Handler<T> tail;
17
18        public Builder<T> addHandler(Handler<T> handler) {
19            if (this.head == null) {
20                // 节点为空
21                this.head = this.tail = handler;
22                return this;
23            }
24            // 往尾部添加节点
25            this.tail.next(handler);
26            this.tail = handler;
27            return this;

```

```

28     }
29
30     public Handler<T> build() {
31         return this.head;
32     }
33 }
34
35 }

```

通过这个案例，小伙伴们应该已经感受到责任链和建造者结合的精髓了。

## 2.4. 责任链模式在源码中的体现

首先我们来看责任链模式在JDK中的应用，来看一个J2EE标准中非常常见的Filter类：

```

1 package javax.servlet;
2
3 import java.io.IOException;
4
5 public interface Filter {
6
7     public void init(FilterConfig filterConfig) throws ServletException;
8
9     public void doFilter(ServletRequest request, ServletResponse response, FilterChain
10
11     public void destroy();
12 }

```

这个Filter 接口的定义非常简单，相当于责任链模型中的Handler抽象角色，那么它是如何形成一条责任链的呢？我们来看另外一个类，其实在doFilter()方法的最后一个参数我们已经看到其类型是FilterChain类，来看它的源码：

```

1 package javax.servlet;
2
3 import java.io.IOException;
4
5 public interface FilterChain {
6
7     public void doFilter(ServletRequest request, ServletResponse response) throws IOExc
8 }

```

FilterChain类中也只定义了一个doFilter()方法，那么它们是怎么串联成一个责任链的呢？实际上J2EE只是定义了一个规范，具体处理逻辑是由使用者自己来实现。我们来看一个Spring的实现MockFilterChain类：

```
1 public class MockFilterChain implements FilterChain {
2
3     @Nullable
4     private ServletRequest request;
5
6     @Nullable
7     private ServletResponse response;
8
9     private final List<Filter> filters;
10
11     @Nullable
12     private Iterator<Filter> iterator;
13
14     ...
15
16     @Override
17     public void doFilter(ServletRequest request, ServletResponse response) throws IOException {
18         Assert.notNull(request, "Request must not be null");
19         Assert.notNull(response, "Response must not be null");
20         Assert.state(this.request == null, "This FilterChain has already been called!");
21
22         if (this.iterator == null) {
23             this.iterator = this.filters.iterator();
24         }
25
26         if (this.iterator.hasNext()) {
27             Filter nextFilter = this.iterator.next();
28             nextFilter.doFilter(request, response, this);
29         }
30
31         this.request = request;
32         this.response = response;
33     }
34     ...
}
```

它把链条中的所有 Filter 放到List中，然后在调用doFilter()方法时循环迭代List，也就是说List中的Filter会顺序执行。

再来看一个在Netty中非常经典的串行化处理 Pipeline就采用了责任链设计模式。它底层采用双向链表的数据结构，将链上的各个处理器串联起来。客户端每一个请求的到来，Netty都认为Pipeline中的所有的处理器都有机会处理它。因此，对于入栈的请求全部从头节点开始往后传播，一直传播到尾节点才会把消息释放掉。来看一个Netty的责任处理器接口ChannelHandler：

```
1 package io.netty.channel;
2
3 import java.lang.annotation.Documented;
4 import java.lang.annotation.ElementType;
5 import java.lang.annotation.Inherited;
6 import java.lang.annotation.Retention;
7 import java.lang.annotation.RetentionPolicy;
8 import java.lang.annotation.Target;
9
10 public interface ChannelHandler {
11
12     // 当handler被添加到真实的上下文中,并且准备处理事件时被调用
13     // handler被添加进去的回调
14     void handlerAdded(ChannelHandlerContext var1) throws Exception;
15
16     // handler被移出后的回调
17     void handlerRemoved(ChannelHandlerContext var1) throws Exception;
18
19     /** @deprecated */
20     @Deprecated
21     void exceptionCaught(ChannelHandlerContext var1, Throwable var2) throws Exception;
22
23     @Inherited
24     @Documented
25     @Target({ElementType.TYPE})
26     @Retention(RetentionPolicy.RUNTIME)
27     public @interface Sharable {
28     }
29 }
```

Netty对责任处理接口做了更细粒度的划分，处理器被分成了两种，一种是入栈处理器ChannelInboundHandler，另一种是出栈处理器ChannelOutboundHandler，这两个接口都继承自ChannelHandler。而所有的处理器最终都在添加在Pipeline上。所以，添加删除责任处理器的接口的行为在Netty的ChannelPipeline中的进行了规定：



```

1 public interface ChannelPipeline
2     extends ChannelInboundInvoker, ChannelOutboundInvoker, Iterable<Entry<String, C
3         ChannelPipeline addFirst(String name, ChannelHandler handler);
4
5     ChannelPipeline addFirst(EventExecutorGroup group, String name, ChannelHandler hand
6
7     ChannelPipeline addLast(String name, ChannelHandler handler);
8
9     ChannelPipeline addLast(EventExecutorGroup group, String name, ChannelHandler handl
10
11     ChannelPipeline addBefore(String baseName, String name, ChannelHandler handler);
12
13     ChannelPipeline addBefore(EventExecutorGroup group, String baseName, String name, C
14
15     ChannelPipeline addAfter(String baseName, String name, ChannelHandler handler);
16
17     ChannelPipeline addAfter(EventExecutorGroup group, String baseName, String name, Ch
18
19     ChannelPipeline addFirst(ChannelHandler... handlers);
20
21     ChannelPipeline addFirst(EventExecutorGroup group, ChannelHandler... handlers);
22
23     ChannelPipeline addLast(ChannelHandler... handlers);
24
25     ChannelPipeline addLast(EventExecutorGroup group, ChannelHandler... handlers);
26
27     ChannelPipeline remove(ChannelHandler handler);
28
29     ChannelHandler remove(String name);
30
31     <T extends ChannelHandler> T remove(Class<T> handlerType);
32
33     ChannelHandler removeFirst();
34
35     ChannelHandler removeLast();
36     ...

```

在默认实现类中将所有的Handler都串成了一个链表：

```

1 public class DefaultChannelPipeline implements ChannelPipeline {

```

```
2    ...
3    final AbstractChannelHandlerContext head;
4    final AbstractChannelHandlerContext tail;
5    ...
```

在Pipeline中的任意一个节点，只要我们不手动的往下传播下去，这个事件就会终止传播在当前节点。对于入栈数据，默认会传递到尾节点进行回收。如果我们不进行下一步传播，事件就会终止在当前节点。对于出栈数据把数据写回客户端也意味着事件的终止。

当然在很多安全框架中也会大量使用责任链模式，比如Spring Security、Apache Shiro都会用到设计模式中的责任链模式，感兴趣的小伙伴可以尝试自己去研究一下。

大部分框架中无论怎么实现，所有的实现都是大同小异的。其实如果我们是站在设计者这个角度看源码的话，对我们学习源码和编码内功是非常非常有益处的，因为这样，我们可以站在更高的角度来学习优秀的思想，而不是钻到某一个代码细节里边。我们需要对所有的设计必须有一个宏观概念，这样学习起来才更加轻松。

## 2.5. 责任链模式的优缺点

优点：

- 1、将请求与处理解耦；
- 2、请求处理者（节点对象）只需关注自己感兴趣的请求进行处理即可，对于不感兴趣的请求，直接转发给下一级节点对象；
- 3、具备链式传递处理请求功能，请求发送者无需知晓链路结构，只需等待请求处理结果；
- 4、链路结构灵活，可以通过改变链路结构动态地新增或删减责任；
- 5、易于扩展新的请求处理类（节点），符合开闭原则。

缺点：

- 1、责任链太长或者处理时间过长，会影响整体性能
- 2、如果节点对象存在循环引用时，会造成死循环，导致系统崩溃；