

13 职责链模式

职责链（责任链）模式的定义：使很多对象都有机会处理请求，从而避免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，沿着链子传递请求直到有对象处理它



现实中的职责链模式

- 上面的审批流就是一个很好的例子
- 再例如学生作弊传纸条，遇到不会答的题目就往后传递纸条，直到有人会回答为止

优点就是请求发送者只需要知道链中的第一个节点，弱化了发送者和一组接收者之间的强联系。例如上面的审批流我们就必须知道我手上的审批单到底应该谁审批，作弊学生也必须知道到底谁会答案

实际开发中的职责链模式

假设这样一个业务：

我们负责一个购买网站，这里预购期间缴纳500定金预定之后可以在正式购买之后收到100元的商城优惠券，缴纳200元定金的用户可以收到50元的商城优惠券，而没有给定金的进入普通购买模式，没有优惠券。

php会传递下面几个字段用于渲染页面：

- orderType: code为1,2,3，分别表示500定金用户，200定金用户，普通用户
- pay: 表示用户是否已经付过定金
- stock: 表示普通购买的商品库存数量，支付定金的不受限制

我们尝试书写一下代码：

```
1  /**
2   * order过程
3   */
4  let order = (orderType, pay, stock) => {
5      if(orderType === 1){
6          if(pay){
7              console.log('500元定金预购，得到100优惠券')
8          } else {
9              if(stock > 0){
10                 console.log('普通购买，无优惠券');
11             } else {
12                 console.log('手机库存不足');
13             }
14         }
15     }
16
17     // if(orderType === 2){...}
18     // if(orderType === 3){...}
19 }
```

这样的代码很难阅读，而且会经常修改，难以维护

利用职责链来重构

我们尝试采用职责链模式重构，首先我们需要根据职责将500,200，普通分成3个函数，接着还需要一个传递的过程：

```
1  /**
2   * 第一次重构
3   */
4  let order500 = (orderType, pay, stock) => {
5      if(orderType === 1 && pay){
6          console.log('500元定金预购，得到100优惠券')
7      } else {
8          // 传递
9          order200(orderType, pay, stock);
10     }
11 }
12
13 let order200 = (orderType, pay, stock) => {
14     if(orderType === 2 && pay){
```

```

15     console.log('200元定金预购，得到50优惠券')
16   } else {
17     // 传递
18     orderNormal(orderType, pay, stock);
19   }
20 }
21
22 let orderNormal = (orderType, pay, stock) => {
23   if(stock > 0){
24     console.log('普通购买')
25   } else {
26     console.log('库存不足')
27   }
28 }
29
30 order500(1, false, 500);
31 // ...

```

可以看到这里我们将巨大的order函数根据职责拆分了3个小函数，去掉了冗余的嵌套if-else语句

但是目前的传递其实非常僵硬，并且传递的逻辑耦合到了业务逻辑里面，依然是违反开放-封闭原则的，如果有哪天去掉200元的逻辑的话，那么500元内部的业务代码就必须修改了

灵活可拆分的职责链节点

这里我们尝试使用一种更加灵活的方式处理，这里我们约定，某一个节点如果不能处理就返回“nextSuccessor”来表示：

```

1  /**
2   * 灵活可拆分
3   */
4  var order500 = (orderType, pay, stock) => {
5    if(orderType === 1 && pay){
6      console.log('500元定金预购，得到100优惠券')
7    } else {
8      return 'nextSuccessor'
9    }
10 }
11
12 var order200 = (orderType, pay, stock) => {
13   if(orderType === 2 && pay){
14     console.log('200元定金预购，得到50优惠券')
15   } else {
16     return 'nextSuccessor'
17   }

```

```

18 }
19
20 var orderNormal = (orderType, pay, stock) => {
21     if(stock > 0){
22         console.log('普通购买')
23     } else {
24         console.log('库存不足')
25     }
26 }

```

接着我们一个Chain来包装上述函数为职责链的节点，同时他还应该可以描述职责链的传递顺序与传递动作：

```

1 class Chain {
2     constructor(fn){
3         this.fn = fn;
4         this.successor = null;
5     }
6
7     // 指定链中下一个节点
8     setNextSuccessor = function(successor){
9         return this.successor = successor;
10    }
11
12    // 传递请求
13    pastRequest = function(){
14        const ret = this.fn.apply(this, arguments);
15        if(ret = 'nextSuccessor'){
16            return this.successor && this.successor.pastRequest.apply(this, succe
17        }
18        return ret;
19    }
20 }
21
22 let chainOrder500 = new Chain(order500);
23 let chainOrder200 = new Chain(order200);
24 let chainOrderNormal = new Chain(orderNormal);
25
26 // 指定顺序
27 chainOrder500.setNextSuccessor(chainOrder200);
28 chainOrder200.setNextSuccessor(chainOrderNormal);
29
30 // 只需要关注第一个节点
31 chainOrder500.pastRequest(1,true, 500);
32 // ...

```

通过这个改进，我们可以灵活的增加和移除链子中节点以及顺序



对于程序员来说，我们总是喜欢去改动那些容易改动的地方，就像改动框架的配置文件远比改动框架源代码简单得多。在这里完全不用理会原来的订单函数代码，我们要做的只是调整节点，设置Chain顺序就行了