

5 策略模式

在现实中，很多时候都有很多途径来到达同一个目的地，程序设计中也是如此，例如实现一个压缩文件的程序，既可以选择zip算法也可以选择gzip算法

这些算法灵活多样并且可以随意互相替换，这种解决方案就是本章需要介绍的策略模式



定义：定义一系列算法并将他们一个个封装起来，同时使它们可以相互替换

使用策略模式计算奖金

最初的代码实现

```
1  /**
2   * 最直白的绩效计算
3   */
4  var calculateBouns = function(performanceLevel, salary){
5      if(performanceLevel === 'S'){
6          return salary * 4;
7      }
8
9      if(performanceLevel === 'A'){
10         return salary * 3;
11     }
12
13     if(performanceLevel === 'B'){
14         return salary * 2;
15     }
16 }
```


实现最简单，但是缺点很明显：

- 包含了很多if-else语句，必须覆盖所有的逻辑
- 缺乏弹性，如果新增了绩效C，该函数内部实现就必须修改，违反了开闭原则
- 复用性差，其他地方需要这个算法，我们只能复制和粘贴

使用组合模式重构

```
1  /**
2   * 组合模式重构
3   */
4
5  var performanceS = function(salary){
6      return salary * 4;
7  }
8
9  var performanceA = function(salary){
10     return salary * 3;
11 }
12
13 var performanceB = function(salary){
14     return salary * 2;
15 }
16
17 var calculateBouns = function(performanceLevel, salary){
18     if(performanceLevel === 'S'){
19         return performanceS(salary);
20     }
21
22     if(performanceLevel === 'A'){
23         return performanceA(salary);
24     }
25
26     if(performanceLevel === 'B'){
27         return performanceB(salary);
28     }
29 }
```

这种改善虽然将绩效与薪水的关联逻辑拆出去了，但是改善仍然有限，依然没有解决最大的问题：

 calculate函数未来可能越来越大，系统变化时候缺乏弹性，每一次都需要修改

使用策略模式重构代码

策略模式的目的是将算法的使用和算法的实现解耦开来，上面的例子中其实算法的使用方式是不变的（根据算法得到需要的绩效金额），但是算法的实现是各异的，每一种绩效都对应不同的计算规则。

基于策略模式的程序至少有两部分组成：

- 第一是一组策略类，封装了具体算法实现
- 第二是环境类Context，接受客户的请求并委托给某一个策略类，Context需要维持对策略对象的引用

```
1  /**
2   * JS版本的策略模式重构
3   */
4  var strategies = {
5      S: function(salary){
6          return salary * 4
7      },
8      A: function(salary){
9          return salary * 3;
10     },
11     B: function(salary){
12         return salary * 2;
13     }
14 }
15
16 var calculateBouns2 = function(level, salary){
17     return strategies[level](salary)
18 }
19
20 calculateBouns2('S', 20000); // 输出80000
```

在JS中，函数也是对象，所以简单直接的做法就是直接将策略定义为函数，而Context类可以直接由calculateBouns2函数来充当，并接受用户的请求

多态在策略模式中的体现

通过策略模式重构，我们消除了原先代码中大量的if-else条件分支，具体的实现也解耦到了策略对象中。Context不再负责具体计算过程而只是进行委托。

以上正式对象的多态性的体现，也是它们可以相互替换的目的，只需要替换Context类中当前保存的策略对象便能执行不同的算法来得到我们想要的效果