招 聘 技 术 部

JavaScript 设计模式之策略模式

贾磊



DEEP THINKING 深入思考

**目录**

1. 定义
2. 使用策略模式计算奖金
3. 使用策略模式计算奖金——if-else/case-switch 的简单粗暴方式
4. 使用策略模式计算奖金——使用组合函数
5. 使用策略模式计算奖金——使用策略模式
6. 使用策略模式计算奖金——使用策略模式(面向对象)
7. 使用策略模式计算奖金——使用策略模式(javascript)
8. 策略模式的优缺点



DEEP THINKING 深入思考

**一、定义**

在程序设计中，我们也常常遇到类似的情况，要实现某一个功能有多种方案可以选择。比如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，而且可以随意互相替换。这种解决方案就是本章将要介绍的策略模式。

策略模式的定义是：定义一系列的算法，把它们一个个封装起来，并且使它们可以相互替换。



DEEP THINKING 深入思考

**二、使用策略模式计算奖金**



DEEP THINKING 深入思考

策略模式有着广泛的应用。本节我们就以年终奖的计算为例进行介绍。

很多公司的年终奖是根据员工的工资基数和年底绩效情况来发放的。例如，绩效为S的人年终奖有6倍工资，绩效为A的人年终奖有5倍工资，而绩效为B的人年终奖是4倍工资。



DEEP THINKING 深入思考

我们可以编写一个名为calculateBonus的函数来计算每个人的奖金数额。很显然，calculateBonus函数要正确工作，就需要接收两个参数：员工的工资数额和他的绩效考核等级。代码如下：

**三、使用策略模式计算奖金——if-else/case-switch 的简单粗暴方式**



DEEP THINKING 深入思考

{

// 计算奖金 —— if-else/case-switch 的简单粗暴方式

const calculateBonus = function (performanceLevel, salary) {

if (performanceLevel === 'S') return salary \* 6

if (performanceLevel === 'A') return salary \* 5

if (performanceLevel === 'B') return salary \* 4

}

console.log(calculateBonus('A', 1500)) // 7500

console.log(calculateBonus('B', 1500)) // 6000

}

可以发现，这段代码十分简单，但是存在着显而易见的缺点。

* calculateBonus函数比较庞大，包含了很多if-else语句，这些语句需要覆盖所有的逻辑分支。
* calculateBonus函数缺乏弹性，如果增加了一种新的绩效等级C，或者想把绩效S的奖金系数改为8，那我们必须深入calculateBonus函数的内部实现，这是违反开放-封闭原则的。
* 算法的复用性差，如果在程序的其他地方需要重用这些计算奖金的算法呢？我们的选择只有复制和粘贴。

因此，我们需要重构这段代码。

**四、使用策略模式计算奖金——使用组合函数**



DEEP THINKING 深入思考

一般最容易想到的办法就是使用组合函数来重构代码，我们把各种算法封装到一个个的小函数里面，这些小函数有着良好的命名，可以一目了然地知道它对应着哪种算法，它们也可以被复用在程序的其他地方。代码如下：

{

// 计算奖金 —— 使用组合函数

const

performanceS = function (salary) { return salary \* 6 },

performanceA = function (salary) { return salary \* 5 },

performanceB = function (salary) { return salary \* 4 },

calculateBonus = function (performanceLevel, salary) {

if (performanceLevel === 'S') return performanceS(salary)

if (performanceLevel === 'A') return performanceA(salary)

if (performanceLevel === 'B') return performanceB(salary)

}

console.log(calculateBonus('A', 1500)) // 7500

console.log(calculateBonus('B', 1500)) // 6000

}

目前，我们的程序得到了一定的改善，但这种改善非常有限，我们依然没有解决最重要的问题：calculateBonus函数有可能越来越庞大，而且在系统变化的时候缺乏弹性。



DEEP THINKING 深入思考

**五、使用策略模式计算奖金——使用策略模式**

经过思考，我们想到了更好的办法——使用策略模式来重构代码。策略模式指的是定义一系列的算法，把它们一个个封装起来。将不变的部分和变化的部分隔开是每个设计模式的主题，策略模式也不例外，策略模式的目的就是将算法的使用与算法的实现分离开来。

在这个例子里，算法的使用方式是不变的，都是根据某个算法取得计算后的奖金数额。而算法的实现是各异和变化的，每种绩效对应着不同的计算规则。

一个基于策略模式的程序至少由两部分组成。第一个部分是一组策略类，策略类封装了具体的算法，并负责具体的计算过程。第二个部分是环境类Context, Context接受客户的请求，随后把请求委托给某一个策略类。要做到这点，说明Context中要维持对某个策略对象的引用。

现在用策略模式来重构上面的代码。第一个版本是模仿传统面向对象语言中的实现。我们先把每种绩效的计算规则都封装在对应的策略类里面：



DEEP THINKING 深入思考

**六、使用策略模式计算奖金——使用策略模式(面向对象)**

{

// 计算奖金 —— 使用策略模式(面向对象)

const performanceS = function () { }

performanceS.prototype.calculate = function (salary) { return salary \* 6 }

const performanceA = function () { }

performanceA.prototype.calculate = function (salary) { return salary \* 5 }

const performanceB = function () { }

performanceB.prototype.calculate = function (salary) { return salary \* 4 }

const Bonus = function (strategy = null, salary = null) {

this.strategy = strategy

this.salary = salary

}

Bonus.prototype.setSalary = function (salary) { this.salary = salary }

Bonus.prototype.setStrategy = function (strategy) { this.strategy = strategy }

Bonus.prototype.getBonus = function () { return this.strategy.calculate(this.salary) }

const bonus = new Bonus()

bonus.setSalary(10000)

bonus.setStrategy(new performanceS())

console.log(bonus.getBonus()) // 60000

bonus.setStrategy(new performanceB())

console.log(bonus.getBonus()) // 40000

}



DEEP THINKING 深入思考

**六、使用策略模式计算奖金——使用策略模式(面向对象)**

创建一个bonus对象，并且给bonus对象设置一些原始的数据，比如员工的原始工资数额。接下来把某个计算奖金的策略对象也传入bonus对象内部保存起来。当调用bonus.getBonus()来计算奖金的时候，bonus对象本身并没有能力进行计算，而是把请求委托给了之前保存好的策略对象。

**策略模式的思想：定义一系列的算法，把它们一个个封装起来，并且使它们可以相互替换。**这句话如果说得更详细一点，就是：定义一系列的算法，把它们各自封装成策略类，算法被封装在策略类内部的方法里。在客户对Context发起请求的时候，Context总是把请求委托给这些策略对象中间的某一个进行计算。

刚刚用策略模式重构了这段计算年终奖的代码，可以看到通过策略模式重构之后，代码变得更加清晰，各个类的职责更加鲜明。但这段代码是基于传统面向对象语言的模仿，下面用JavaScript实现策略模式。



DEEP THINKING 深入思考

**七、使用策略模式计算奖金——使用策略模式(javascript)**

前面的strategy对象从各个策略类中创建而来，这是模拟一些传统面向对象语言的实现。实际上在JavaScript语言中，函数也是对象，所以更简单和直接的做法是把strategy直接定义为函数：

{

// 计算奖金 —— 使用策略模式(javascript)

const strategies = {

'S': function (salary) { return salary \* 6 },

'A': function (salary) { return salary \* 5 },

'B': function (salary) { return salary \* 4 },

}

const calculateBonus = function (level, salary) { return strategies[level](salary) }

console.log(calculateBonus('S', 10000)) // 60000

console.log(calculateBonus('B', 10000)) // 40000

}

通过使用策略模式重构代码，我们消除了原程序中大片的条件分支语句。所有跟计算奖金有关的逻辑不再放在Context中，而是分布在各个策略对象中。Context并没有计算奖金的能力，而是把这个职责委托给了某个策略对象。每个策略对象负责的算法已被各自封装在对象内部。当我们对这些策略对象发出“计算奖金”的请求时，它们会返回各自不同的计算结果，这正是对象**多态性**的体现，也是“它们可以相互替换”的目的。替换Context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结果。



DEEP THINKING 深入思考

**八、策略模式的优缺点**

策略模式的一些优点：

策略模式利用组合、委托和多态等技术和思想，可以有效地避免多重条件选择语句。

策略模式提供了对开放—封闭原则的完美支持，将算法封装在独立的strategy中，使得它们易于切换，易于理解，易于扩展。

策略模式中的算法也可以复用在系统的其他地方，从而避免许多重复的复制粘贴工作。

在策略模式中利用组合和委托来让Context拥有执行算法的能力，这也是继承的一种更轻便的替代方案。

当然，策略模式也有一些缺点，但这些缺点并不严重。

首先，使用策略模式会在程序中增加许多策略类或者策略对象，但实际上这比把它们负责的逻辑堆砌在Context中要好。

其次，要使用策略模式，必须了解所有的strategy，必须了解各个strategy之间的不同点，这样才能选择一个合适的strategy。比如，我们要选择一种合适的旅游出行路线，必须先了解选择飞机、火车、自行车等方案的细节。此时strategy要向客户暴露它的所有实现，这是违反最少知识原则的。