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

```
/**
 * 最直白的绩效计算
 */
var calculateBouns = function(performanceLevel, salary){
    if(performanceLevel === 'S'){
        return salary * 4;
    }

    if(performanceLevel === 'A'){
        return salary * 3;
    }

    if(performanceLevel === 'B'){
        return salary * 2;
    }
}
```

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

```
/**
 * 组合模式重构
 */

var performanceS = function(salary){
    return salary * 4;
}

var performanceA = function(salary){
    return salary * 3;
}

var performanceB = function(salary){
    return salary * 2;
}

 var calculateBouns = function(performanceLevel, salary){
    if(performanceLevel === 'S'){
        return performanceS(salary);
    }

    if(performanceLevel === 'A'){
        return performanceA(salary);
    }

    if(performanceLevel === 'B'){
        return performanceB(salary);
    }
}
```

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

策略模式的目的就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

```
/**
 * JS版本的策略模式重构
 */
var strategies = {
    S: function(salary){
        return salary * 4
    },
    A: function(salary){
        return salary * 3;
    },
    B: function(salary){
        return salary * 2;
    }
}

var calculateBouns2 = function(level, salary){
    return strategies[level](salary)
}

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

使用策略模式实现缓动动画

早期的Web开发干的事情很多围绕：

- 表单验证

- 飞来飞去的动画

有一段时间H5版本的游戏可以达到不逊于Flash游戏的效果，很多时候游戏角色会进行跳跃和移动，实际上只是让div按照一定的缓动算法进行运动，这里我们以一个小球为例

实现动画效果的原理

其实跟动画片原理一样，通过连续改变某个元素的CSS熟悉，例如left，top，background-postion来实现动画效果

思路与准备工作

目标：编写一个动画类和一些缓动算法，让小球在页面中动起来

梳理需要的信息：

- 动画开始时小球的位置
- 小球移动时的目标位置
- 动画开始时的时间点
- 小球运动的持续时间

实现：

- 借助setInterval创建定时器，每个19ms循环一次
- 定时器的每一帧里，我们根据动画已经消耗的时间，小球原始位置，小球目标位置，动画总时间来传入缓动算法，更新小球对应的位置

让小球动起来

这里我们借助一些Flash中现成的缓动算法，配合动画类来实现效果：

```
// 缓动算法对象
var tween = {
    linear:function(t,b,c,d){
        return c*t/d + b;
    },
    easeIn:function(t,b,c,d){
        return c * (t/=d)*t + b;
    },
    // ... 其他算法可以网上搜索
}

// 定义Animate类
var Animate = function(dom){
    this.dom = dom;
    this.startTime = 0;
```

```
16      this.startPos = 0; // 动画开始时DOM位置
17      this.endPos = 0; // 动画结束时DOM位置
18      this.propertName = null; // dom节点需要改变的CSS名称
19      this.easing = null; // 缓动算法
20      this.duration = null;
21  }
22
23  // start方法负责启动动画并记录初始信息，供后续算法计算使用
24  Animate.prototype.start = function(propertName,  endPos, duration, easing){
25      this.startTime = +new Date;
26      this.startPos = this.dom.getBoundingClient()[propertName];
27      this.propertName = propertName;
28      this.endPos = endPos;
29      this.duration = duration;
30      this.easing = tween[easing];
31
32      var self = this;
33      var timeId = setInterval(function(){
34          if(self.step() === false){
35              clearInterval(timeId);
36          }
37      },19)
38  }
39
40  // step方法负责小球每一帧需要做的事情
41  Animate.prototype.step = function(){
42      var t = +new Date;
43      if(t >= this.startTime + this.duration){
44          this.update(this.endPos); // 更新小球CSS
45          return false; // 动画结束
46      }
47      // 得到算法更新后位置
48      var pos = this.easing(t - this.startTime, this.startPos, this.endPos - this.
49      // 更新CSS
50      this.update(pos);
51  }
52
53  // update方法负责更新CSS
54  Animate.prototype.update = function(pos){
55      this.dom.style[this.propertName] = pos + 'px'
56  }
57
58  // 测试一下
59  var div = document.getElementById('div');
60  var animate = new Animate(div);
61
62  animate.start('left', 500, 1000, 'linear');
```

这里我们使用了策略模式把算法传入动画类之中，达到各种不同的缓动效果，同时这些算法都可以进行替换。

策略模式的背后需要我们找到封装变化，委托和多态性思想的价值

更广义的“算法”

策略模式其实不仅仅只是可以封装我们之前讨论的那些“算法”，我们可以把算法的含义扩散开来，封装一系列“业务规则”。

只需要这些规则的目标一致，并且可以替换使用就可以用策略模式进行封装

例如Web开发中很常见的表单校验就可以使用策略模式来封装

表单校验

Web项目中客户端针对表单基本上都需要写一些校验逻辑，避免非法数据带来的网络开销

第一个版本

```
/**
 * 表单校验的第一个版本
 */

var registerForm = document.getElementById('registerForm')

registerForm.onsubmit = function () {
    if (registerForm.userName.value === '') {
        alert('用户名不能为空')
        return false;
    }

    // if...
}
```

缺点基本跟之前提到的计算奖金的一样：

- 很多if-else
- 缺乏弹性
- 难以复原

使用策略模式重构

校验这一块其实使用方式都是一样，不同的是具体的校验规则，我们可以使用策略模式来进行解耦和封装：

```
/**
 * 策略模式重构
 */
var strategies = {
    isNonEmpty: function(value, errorMsg){
        if(value === ''){
            return errorMsg;
        }
    },
    minLength: function(value, length, errorMsg){
        if(value. length < length){
            return errorMsg;
        }
    },
    // ...
}

// 实现Validator类作为Context的职责
var Validator = function(){
    this.cache = []; //保存校验规则
}

Validator.prototype.add = function(dom, rule, errorMsg){
    var ary = rule.split(':'); // 约定，分开策略名和参数
    this.cache.push(function(){
        var startegyName = ary.shift();
        ary.unshift(dom.value);
        ary.push(errorMsg);
        return strategies[startegyName].apply(dom, ary);
    })
}

Validator.prototype.start = function(){
    // 遍历执行校验 - 由返回值代表校验不通过
    for(i = 0, validatorFunc; validatorFunc = this.cache[i++];){
        var msg = validatorFunc();
        if(msg){
            return msg;
        }
    }
}
```

```
42
43 // 业务定义的校验逻辑
44 var validateFunction = function(){
45     var validator = new Validator();
46
47     validator.add(registerForm.userName, 'isNonEmpty', '用户名不能为空');
48     validator.add(registerForm.password, 'minLength:6', '密码长度大于6位');
49
50     var errorMsg = validator.start();
51     return errorMsg;
52 }
53
54 var registerForm = document.getElementById('registerForm')
55
56 registerForm.onsubmit = function () {
57     var errorMsg = validatorFunc();
58     if(errorMsg){
59         alert(errorMsg);
60         return false
61     }
62 }
```

重构完成代码之后，我们仅仅通过配置的方式就可以组合和复用校验规则了，也很方便移植

策略模式优缺点

优点

- 利用组合，委托，多态思想，避免多重if-else语句
- 提供了对开放-封闭原则支持，变化算法和逻辑封装在独立的startegy中，易于扩展切换
- 容易复用
- 利用组合和委托让Context拥有执行算法能力

缺点：不严重

- 新增许多策略类或者对象，实际上比堆叠在业务中好多了
- 使用策略需要对所有startegy有了解，尤其是差异，策略要向客户暴露所有事项，违反最少知识原则

一等函数对象与策略模式

在JS这种函数作为一等对象的语言中，其实策略模式是隐形的，strategy就是值为函数的变量，完全可以通过将“算法”封装到函数中并四处传递 -- 高阶函数

在JS中，函数对象的多态性来的更加简单和直接，之前的例子中如果我们把startegies去掉，你还能认出策略模式吗？：

```
/**
 * 隐形的策略模式
 */
    var S = function(salary){
        return salary * 4
    },
    var A = function(salary){
        return salary * 3;
    },
    var B = function(salary){
        return salary * 2;
    }

    var calculate = function(func,salary){return func(salary)}
```