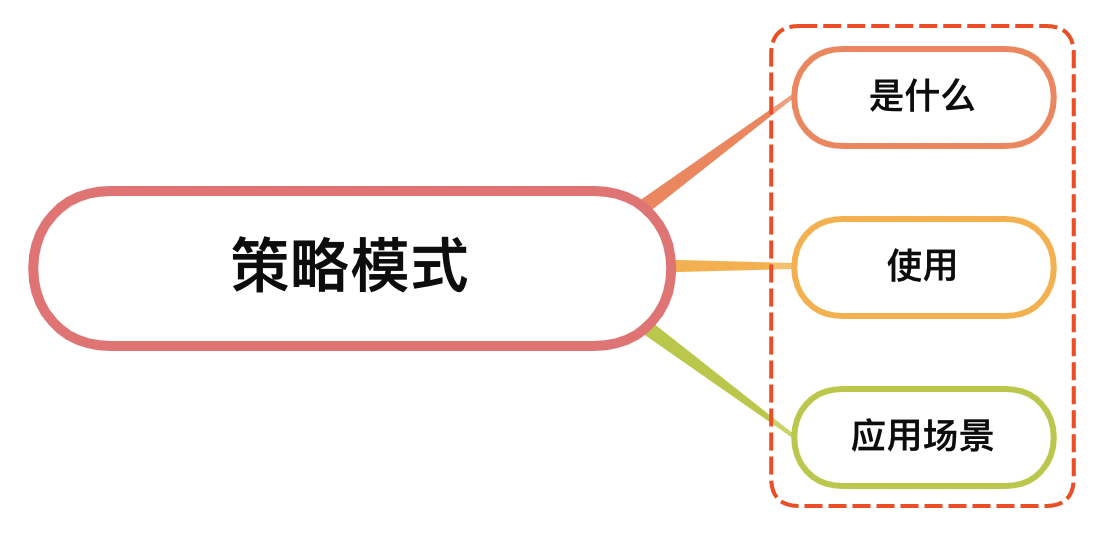
# 面试官：说说你对策略模式的理解？应用场景？



## 一、是什么

策略模式（Strategy Pattern）指的是定义一系列的算法，把它们一个个封装起来，目的就是将算法的使用与算法的实现分离开来

一个基于策略模式的程序至少由两部分组成：

* 策略类，策略类封装了具体的算法，并负责具体的计算过程
* 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

## 二、使用

举个例子，公司的年终奖是根据员工的工资和绩效来考核的，绩效为A的人，年终奖为工资的4倍，绩效为B的人，年终奖为工资的3倍，绩效为C的人，年终奖为工资的2倍

若使用if来实现，代码则如下：

var calculateBouns = function(salary,level) {  
 if(level === 'A') {  
 return salary \* 4;  
 }  
 if(level === 'B') {  
 return salary \* 3;  
 }  
 if(level === 'C') {  
 return salary \* 2;  
 }  
};  
// 调用如下：  
console.log(calculateBouns(4000,'A')); // 16000  
console.log(calculateBouns(2500,'B')); // 7500

从上述可有看到，函数内部包含过多if...else，并且后续改正的时候，需要在函数内部添加逻辑，违反了开放封闭原则

而如果使用策略模式，就是先定义一系列算法，把它们一个个封装起来，将不变的部分和变化的部分隔开，如下：

var obj = {  
 "A": function(salary) {  
 return salary \* 4;  
 },  
 "B" : function(salary) {  
 return salary \* 3;  
 },  
 "C" : function(salary) {  
 return salary \* 2;  
 }   
};  
var calculateBouns =function(level,salary) {  
 return obj[level](salary);  
};  
console.log(calculateBouns('A',10000)); // 40000

上述代码中，obj对应的是策略类，而calculateBouns对应上下通信类

又比如实现一个表单校验的代码，常常会像如下写法：

var registerForm = document.getElementById("registerForm");  
registerForm.onsubmit = function(){  
 if(registerForm.userName.value === '') {  
 alert('用户名不能为空');  
 return;  
 }  
 if(registerForm.password.value.length < 6) {  
 alert("密码的长度不能小于6位");  
 return;  
 }  
 if(!/(^1[3|5|8][0-9]{9}$)/.test(registerForm.phoneNumber.value)) {  
 alert("手机号码格式不正确");  
 return;  
 }  
}

上述代码包含多处if语句，并且违反了开放封闭原则，如果应用中还有其他的表单，需要重复编写代码

此处也可以使用策略模式进行重构校验，第一步确定不变的内容，即策略规则对象，如下：

var strategy = {  
 isNotEmpty: function(value,errorMsg) {  
 if(value === '') {  
 return errorMsg;  
 }  
 },  
 // 限制最小长度  
 minLength: function(value,length,errorMsg) {  
 if(value.length < length) {  
 return errorMsg;  
 }  
 },  
 // 手机号码格式  
 mobileFormat: function(value,errorMsg) {  
 if(!/(^1[3|5|8][0-9]{9}$)/.test(value)) {  
 return errorMsg;  
 }  
 }   
};

然后找出变的地方，作为环境类context，负责接收用户的要求并委托给策略规则对象，如下Validator类：

var Validator = function(){  
 this.cache = []; // 保存效验规则  
};  
Validator.prototype.add = function(dom,rule,errorMsg) {  
 var str = rule.split(":");  
 this.cache.push(function(){  
 // str 返回的是 minLength:6   
 var strategy = str.shift();  
 str.unshift(dom.value); // 把input的value添加进参数列表  
 str.push(errorMsg); // 把errorMsg添加进参数列表  
 return strategys[strategy].apply(dom,str);  
 });  
};  
Validator.prototype.start = function(){  
 for(var i = 0, validatorFunc; validatorFunc = this.cache[i++]; ) {  
 var msg = validatorFunc(); // 开始效验 并取得效验后的返回信息  
 if(msg) {  
 return msg;  
 }  
 }  
};

通过validator.add方法添加校验规则和错误信息提示，使用如下：

var validateFunc = function(){  
 var validator = new Validator(); // 创建一个Validator对象  
 /\* 添加一些效验规则 \*/  
 validator.add(registerForm.userName,'isNotEmpty','用户名不能为空');  
 validator.add(registerForm.password,'minLength:6','密码长度不能小于6位');  
 validator.add(registerForm.userName,'mobileFormat','手机号码格式不正确');  
  
 var errorMsg = validator.start(); // 获得效验结果  
 return errorMsg; // 返回效验结果  
};  
var registerForm = document.getElementById("registerForm");  
registerForm.onsubmit = function(){  
 var errorMsg = validateFunc();  
 if(errorMsg){  
 alert(errorMsg);  
 return false;  
 }  
}

上述通过策略模式完成表单的验证，并且可以随时调用，在修改表单验证规则的时候，也非常方便，通过传递参数即可调用

## 三、应用场景

从上面可以看到，使用策略模式的优点有如下：

* 策略模式利用组合，委托等技术和思想，有效的避免很多if条件语句
* 策略模式提供了开放-封闭原则，使代码更容易理解和扩展
* 策略模式中的代码可以复用

策略模式不仅仅用来封装算法，在实际开发中，通常会把算法的含义扩散开来，使策略模式也可以用来封装 一系列的“业务规则”

只要这些业务规则指向的目标一致，并且可以被替换使用，我们就可以用策略模式来封装它们

## 参考文献

* https://segmentfault.com/a/1190000021883055
* https://juejin.cn/post/6844903504109109262
* https://juejin.cn/post/6844903751225081864