**介绍**

策略模式定义了算法家族，分别封装起来，让他们之间可以互相替换，此模式让算法的变化不会影响到使用算法的客户。

**正文**

在理解策略模式之前，我们先来一个例子，一般情况下，如果我们要做数据合法性验证，很多时候都是按照swith语句来判断，但是这就带来几个问题，首先如果增加需求的话，我们还要再次修改这段代码以增加逻辑，而且在进行单元测试的时候也会越来越复杂，代码如下：

validator = {  
 validate: function (value, type) {  
 switch (type) {  
 case 'isNonEmpty ':  
 {  
 return true; // NonEmpty 验证结果  
 }  
 case 'isNumber ':  
 {  
 return true; // Number 验证结果  
 break;  
 }  
 case 'isAlphaNum ':  
 {  
 return true; // AlphaNum 验证结果  
 }  
 default:  
 {  
 return true;  
 }  
 }  
 }  
 };  
 // 测试  
 alert(validator.validate("123", "isNonEmpty"));

那如何来避免上述代码中的问题呢，根据策略模式，我们可以将相同的工作代码单独封装成不同的类，然后通过统一的策略处理类来处理，OK，我们先来定义策略处理类，代码如下：

var validator = {  
  
 // 所有可以的验证规则处理类存放的地方，后面会单独定义  
 types: {},  
  
 // 验证类型所对应的错误消息  
 messages: [],  
  
 // 当然需要使用的验证类型  
 config: {},  
  
 // 暴露的公开验证方法  
 // 传入的参数是 key => value对  
 validate: function (data) {  
  
 var i, msg, type, checker, result\_ok;  
  
 // 清空所有的错误信息  
 this.messages = [];  
  
 for (i in data) {  
 if (data.hasOwnProperty(i)) {  
  
 type = this.config[i]; // 根据key查询是否有存在的验证规则  
 checker = this.types[type]; // 获取验证规则的验证类  
  
 if (!type) {  
 continue; // 如果验证规则不存在，则不处理  
 }  
 if (!checker) { // 如果验证规则类不存在，抛出异常  
 throw {  
 name: "ValidationError",  
 message: "No handler to validate type " + type  
 };  
 }  
  
 result\_ok = checker.validate(data[i]); // 使用查到到的单个验证类进行验证  
 if (!result\_ok) {  
 msg = "Invalid value for \*" + i + "\*, " + checker.instructions;  
 this.messages.push(msg);  
 }  
 }  
 }  
 return this.hasErrors();  
 },  
  
 // helper  
 hasErrors: function () {  
 return this.messages.length !== 0;  
 }  
};

然后剩下的工作，就是定义types里存放的各种验证类了，我们这里只举几个例子：

// 验证给定的值是否不为空  
validator.types.isNonEmpty = {  
 validate: function (value) {  
 return value !== "";  
 },  
 instructions: "传入的值不能为空"  
};  
  
// 验证给定的值是否是数字  
validator.types.isNumber = {  
 validate: function (value) {  
 return !isNaN(value);  
 },  
 instructions: "传入的值只能是合法的数字，例如：1, 3.14 or 2010"  
};  
  
// 验证给定的值是否只是字母或数字  
validator.types.isAlphaNum = {  
 validate: function (value) {  
 return !/[^a-z0-9]/i.test(value);  
 },  
 instructions: "传入的值只能保护字母和数字，不能包含特殊字符"  
};

使用的时候，我们首先要定义需要验证的数据集合，然后还需要定义每种数据需要验证的规则类型，代码如下：

var data = {  
 first\_name: "Tom",  
 last\_name: "Xu",  
 age: "unknown",  
 username: "TomXu"  
};  
  
validator.config = {  
 first\_name: 'isNonEmpty',  
 age: 'isNumber',  
 username: 'isAlphaNum'  
};

最后，获取验证结果的代码就简单了：

validator.validate(data);  
  
if (validator.hasErrors()) {  
 console.log(validator.messages.join("\n"));  
}

**总结**

策略模式定义了一系列算法，从概念上来说，所有的这些算法都是做相同的事情，只是实现不同，他可以以相同的方式调用所有的方法，减少了各种算法类与使用算法类之间的耦合。

从另外一个层面上来说，单独定义算法类，也方便了单元测试，因为可以通过自己的算法进行单独测试。

实践中，不仅可以封装算法，也可以用来封装几乎任何类型的规则，是要在分析过程中需要在不同时间应用不同的业务规则，就可以考虑是要策略模式来处理各种变化。