[设计模式之装饰者模式](http://www.cnblogs.com/TomXu/archive/2012/02/24/2353434.html)

**介绍**

装饰者提供比继承更有弹性的替代方案。 装饰者用用于包装同接口的对象，不仅允许你向方法添加行为，而且还可以将方法设置成原始对象调用（例如装饰者的构造函数）。

装饰者用于通过重载方法的形式添加新功能，该模式可以在被装饰者前面或者后面加上自己的行为以达到特定的目的。

**正文**

那么装饰者模式有什么好处呢？前面说了，装饰者是一种实现继承的替代方案。当脚本运行时，在子类中增加行为会影响原有类所有的实例，而装饰者却不然。取而代之的是它能给不同对象各自添加新行为。如下代码所示：

//需要装饰的类（函数）  
function Macbook() {  
 this.cost = function () {  
 return 1000;  
 };  
}  
  
function Memory(macbook) {  
 this.cost = function () {  
 return macbook.cost() + 75;  
 };  
}  
  
function BlurayDrive(macbook) {  
 this.cost = function () {  
 return macbook.cost() + 300;  
 };  
}  
  
  
function Insurance(macbook) {  
 this.cost = function () {  
 return macbook.cost() + 250;  
 };  
}  
  
  
// 用法  
var myMacbook = new Insurance(new BlurayDrive(new Memory(new Macbook())));  
console.log(myMacbook.cost());

下面是另一个实例，当我们在装饰者对象上调用performTask时，它不仅具有一些装饰者的行为，同时也调用了下层对象的performTask函数。

function ConcreteClass() {  
 this.performTask = function () {  
 this.preTask();  
 console.log('doing something');  
 this.postTask();  
 };  
}  
  
function AbstractDecorator(decorated) {  
 this.performTask = function () {  
 decorated.performTask();  
 };  
}  
  
function ConcreteDecoratorClass(decorated) {  
 this.base = AbstractDecorator;  
 this.base(decorated);  
  
 decorated.preTask = function () {  
 console.log('pre-calling..');  
 };  
  
 decorated.postTask = function () {  
 console.log('post-calling..');  
 };  
  
}  
  
var concrete = new ConcreteClass();  
var decorator1 = new ConcreteDecoratorClass(concrete);  
var decorator2 = new ConcreteDecoratorClass(decorator1);  
decorator2.performTask();

再来一个彻底的例子：

var tree = {};  
tree.decorate = function () {  
 console.log('Make sure the tree won\'t fall');  
};  
  
tree.getDecorator = function (deco) {  
 tree[deco].prototype = this;  
 return new tree[deco];  
};  
  
tree.RedBalls = function () {  
 this.decorate = function () {  
 this.RedBalls.prototype.decorate(); // 第7步：先执行原型（这时候是Angel了）的decorate方法  
 console.log('Put on some red balls'); // 第8步 再输出 red  
 // 将这2步作为RedBalls的decorate方法  
 }  
};  
  
tree.BlueBalls = function () {  
 this.decorate = function () {  
 this.BlueBalls.prototype.decorate(); // 第1步：先执行原型的decorate方法，也就是tree.decorate()  
 console.log('Add blue balls'); // 第2步 再输出blue  
 // 将这2步作为BlueBalls的decorate方法  
 }  
};  
  
tree.Angel = function () {  
 this.decorate = function () {  
 this.Angel.prototype.decorate(); // 第4步：先执行原型（这时候是BlueBalls了）的decorate方法  
 console.log('An angel on the top'); // 第5步 再输出angel  
 // 将这2步作为Angel的decorate方法  
 }  
};  
  
tree = tree.getDecorator('BlueBalls'); // 第3步：将BlueBalls对象赋给tree，这时候父原型里的getDecorator依然可用  
tree = tree.getDecorator('Angel'); // 第6步：将Angel对象赋给tree，这时候父原型的父原型里的getDecorator依然可用  
tree = tree.getDecorator('RedBalls'); // 第9步：将RedBalls对象赋给tree  
  
tree.decorate(); // 第10步：执行RedBalls对象的decorate方法

**总结**

装饰者模式是为已有功能动态地添加更多功能的一种方式，把每个要装饰的功能放在单独的函数里，然后用该函数包装所要装饰的已有函数对象，因此，当需要执行特殊行为的时候，调用代码就可以根据需要有选择地、按顺序地使用装饰功能来包装对象。优点是把类（函数）的核心职责和装饰功能区分开了。