**介绍**

组合模式（Composite）将对象组合成树形结构以表示“部分-整体”的层次结构，组合模式使得用户对单个对象和组合对象的使用具有一致性。

常见的场景有asp.net里的控件机制（即control里可以包含子control，可以递归操作、添加、删除子control），类似的还有DOM的机制，一个DOM节点可以包含子节点，不管是父节点还是子节点都有添加、删除、遍历子节点的通用功能。所以说组合模式的关键是要有一个抽象类，它既可以表示子元素，又可以表示父元素。

**正文**

举个例子，有家餐厅提供了各种各样的菜品，每个餐桌都有一本菜单，菜单上列出了该餐厅所偶的菜品，有早餐糕点、午餐、晚餐等等，每个餐都有各种各样的菜单项，假设不管是菜单项还是整个菜单都应该是可以打印的，而且可以添加子项，比如午餐可以添加新菜品，而菜单项咖啡也可以添加糖啊什么的。

这种情况，我们就可以利用组合的方式将这些内容表示为层次结构了。我们来逐一分解一下我们的实现步骤。

**第一步，先实现我们的“抽象类”函数MenuComponent：**

var MenuComponent = function () {  
};  
MenuComponent.prototype.getName = function () {  
 throw new Error("该方法必须重写!");  
};  
MenuComponent.prototype.getDescription = function () {  
 throw new Error("该方法必须重写!");  
};  
MenuComponent.prototype.getPrice = function () {  
 throw new Error("该方法必须重写!");  
};  
MenuComponent.prototype.isVegetarian = function () {  
 throw new Error("该方法必须重写!");  
};  
MenuComponent.prototype.print = function () {  
 throw new Error("该方法必须重写!");  
};  
MenuComponent.prototype.add = function () {  
 throw new Error("该方法必须重写!");  
};  
MenuComponent.prototype.remove = function () {  
 throw new Error("该方法必须重写!");  
};  
MenuComponent.prototype.getChild = function () {  
 throw new Error("该方法必须重写!");  
};

该函数提供了2种类型的方法，一种是获取信息的，比如价格，名称等，另外一种是通用操作方法，比如打印、添加、删除、获取子菜单。

**第二步，创建基本的菜品项：**

var MenuItem = function (sName, sDescription, bVegetarian, nPrice) {  
 MenuComponent.apply(this);  
 this.sName = sName;  
 this.sDescription = sDescription;  
 this.bVegetarian = bVegetarian;  
 this.nPrice = nPrice;  
};  
MenuItem.prototype = new MenuComponent();  
MenuItem.prototype.getName = function () {  
 return this.sName;  
};  
MenuItem.prototype.getDescription = function () {  
 return this.sDescription;  
};  
MenuItem.prototype.getPrice = function () {  
 return this.nPrice;  
};  
MenuItem.prototype.isVegetarian = function () {  
 return this.bVegetarian;  
};  
MenuItem.prototype.print = function () {  
 console.log(this.getName() + ": " + this.getDescription() + ", " + this.getPrice() + "euros");  
};

由代码可以看出，我们只重新了原型的4个获取信息的方法和print方法，没有重载其它3个操作方法，因为基本菜品不包含添加、删除、获取子菜品的方式。

**第三步，创建菜品：**

var Menu = function (sName, sDescription) {  
 MenuComponent.apply(this);  
 this.aMenuComponents = [];  
 this.sName = sName;  
 this.sDescription = sDescription;  
 this.createIterator = function () {  
 throw new Error("This method must be overwritten!");  
 };  
};  
Menu.prototype = new MenuComponent();  
Menu.prototype.add = function (oMenuComponent) {  
 // 添加子菜品  
 this.aMenuComponents.push(oMenuComponent);  
};  
Menu.prototype.remove = function (oMenuComponent) {  
 // 删除子菜品  
 var aMenuItems = [];  
 var nMenuItem = 0;  
 var nLenMenuItems = this.aMenuComponents.length;  
 var oItem = null;  
  
 for (; nMenuItem < nLenMenuItems; ) {  
 oItem = this.aMenuComponents[nMenuItem];  
 if (oItem !== oMenuComponent) {  
 aMenuItems.push(oItem);  
 }  
 nMenuItem = nMenuItem + 1;  
 }  
 this.aMenuComponents = aMenuItems;  
};  
Menu.prototype.getChild = function (nIndex) {  
 //获取指定的子菜品  
 return this.aMenuComponents[nIndex];  
};  
Menu.prototype.getName = function () {  
 return this.sName;  
};  
Menu.prototype.getDescription = function () {  
 return this.sDescription;  
};  
Menu.prototype.print = function () {  
 // 打印当前菜品以及所有的子菜品  
 console.log(this.getName() + ": " + this.getDescription());  
 console.log("--------------------------------------------");  
  
 var nMenuComponent = 0;  
 var nLenMenuComponents = this.aMenuComponents.length;  
 var oMenuComponent = null;  
  
 for (; nMenuComponent < nLenMenuComponents; ) {  
 oMenuComponent = this.aMenuComponents[nMenuComponent];  
 oMenuComponent.print();  
 nMenuComponent = nMenuComponent + 1;  
 }  
};

注意上述代码，除了实现了添加、删除、获取方法外，打印print方法是首先打印当前菜品信息，然后循环遍历打印所有子菜品信息。

**第四步，创建指定的菜品：**

我们可以创建几个真实的菜品，比如晚餐、咖啡、糕点等等，其都是用Menu作为其原型，代码如下：

var DinnerMenu = function () {  
 Menu.apply(this);  
};  
DinnerMenu.prototype = new Menu();  
  
var CafeMenu = function () {  
 Menu.apply(this);  
};  
CafeMenu.prototype = new Menu();  
  
var PancakeHouseMenu = function () {  
 Menu.apply(this);  
};  
PancakeHouseMenu.prototype = new Menu();

**第五步，创建最顶级的菜单容器——菜单本：**

var Mattress = function (aMenus) {  
 this.aMenus = aMenus;  
};  
Mattress.prototype.printMenu = function () {  
 this.aMenus.print();  
};

该函数接收一个菜单数组作为参数，并且值提供了printMenu方法用于打印所有的菜单内容。

**第六步，调用方式：**

var oPanCakeHouseMenu = new Menu("Pancake House Menu", "Breakfast");  
var oDinnerMenu = new Menu("Dinner Menu", "Lunch");  
var oCoffeeMenu = new Menu("Cafe Menu", "Dinner");  
var oAllMenus = new Menu("ALL MENUS", "All menus combined");  
  
oAllMenus.add(oPanCakeHouseMenu);  
oAllMenus.add(oDinnerMenu);  
  
oDinnerMenu.add(new MenuItem("Pasta", "Spaghetti with Marinara Sauce, and a slice of sourdough bread", true, 3.89));  
oDinnerMenu.add(oCoffeeMenu);  
  
oCoffeeMenu.add(new MenuItem("Express", "Coffee from machine", false, 0.99));  
  
var oMattress = new Mattress(oAllMenus);  
console.log("---------------------------------------------");  
oMattress.printMenu();  
console.log("---------------------------------------------");

熟悉asp.net控件开发的同学，是不是看起来很熟悉？

**总结**

组合模式的使用场景非常明确：

1. 你想表示对象的部分-整体层次结构时；
2. 你希望用户忽略组合对象和单个对象的不同，用户将统一地使用组合结构中的所有对象（方法）

另外该模式经常和装饰者一起使用，它们通常有一个公共的父类（也就是原型），因此装饰必须支持具有add、remove、getChild操作的 component接口。

参考：https://github.com/tcorral/Design-Patterns-in-Javascript/blob/master/Composite/index.html