**状态模式：**

状态模式（State）当一个对象内部状态发生改变时，会导致其行为的改变，这样看起来像是改变了对象。

状态对象的实现

<script>  
 //投票结果状态对象  
 var *ResultState*=function(){  
 //每种状态作为一种独立方法保存起来  
 var States= {  
 state0: function () {  
 //处理结果0  
 console.log('这是第1中情况');  
 },  
 state1: function () {  
 //处理结果0  
 console.log('这是第2中情况');  
 },  
 state2: function () {  
 //处理结果0  
 console.log('这是第3中情况');  
 },  
 state3: function () {  
 //处理结果0  
 console.log('这是第4中情况');  
 }  
 };  
 //获取某一种状态并执行对应的方法  
 function show(result){  
 States['state'+result]&& States['state'+result]();  
 }  
  
 return{ //返回调用状态方法接口  
 show:show  
 }  
 }();  
  
  
 *ResultState*.show(3); //测试  
</script>

上述是状态模式的基本雏形。

对于状态模式，主要目的就是将条件判断的不同结果转化为状态对象的内部状态，既然是内部对象的状态，所以一般作为状态对象内部的私有变量，然后提供一个能够调用状态对象内部状态的接口方法对象。这样当我们需要增加、修改、删除、调用某种状态方法时就会很容易，也方便了我们对状态对象中内部状态的管理。

**超级玛丽**：

超级玛丽游戏，玛丽要吃蘑菇，那么他就跳起，顶出墙壁里面的蘑菇；玛丽想到悬崖的另一边，他就要跳起；玛丽想避免被前面的乌龟咬到，他就要开枪将其打掉；前方飞过了炮弹，玛丽要蹲下躲避；时间不够了，玛丽要加速奔跑……。

跳跃、开枪、蹲下、奔跑等这些都是个一个一个的状态，如果我们用if或者switch条件判断语句写的代码，这些的代码很难维护的，因为增加或删除一个状态需要改动的地方太多了。此时使用状态模式就是再好不过了。对于玛丽，有的时候它需要跳跃开枪，有的时候它需要蹲下开枪，有的时候适合需要奔跑开枪等。如果这些组合状态用if或者switch分支判断去实现，无疑增加的成本是无法想象的。例如：

//但动作条件判断，每增加一个动作就需要添加一个判断

var lastAction='';

function changMarry(action){

if(action=='jump'){

//跳跃动作

}else if(action=='move'){

//移动动作

}else

{

//默认情况

}

lastAction=action;

}

//复合动作对条件判断的开销是翻倍的

var lastActio1='';

var lastAction2='';

function changeMarry(action1,action2){

if(action1=='shoot'){

//射击

}else if(action1=='jump'){

//跳跃

}else if(action1=='move' && action2=='shoot')

{

//移动射击

}else if(action1=='jump' && action2=='shoot')

{

//跳跃中射击

}

//保留上一个动作

lastActio1=action1||'';

lastAction2=action2||'';

}

**状态的优化：**

状态模式设计

//创建超级玛丽状态类  
var MarryState=function(){  
 //内部状态私有变量  
 var \_currentState={};  
 //动作与状态方法映射  
 states={  
 jump:function(){  
 //跳跃  
 console.log('jump');  
 },  
 move:function(){  
 //移动  
 console.log('move');  
 },  
 shoot:function(){  
 //射击  
 console.log('shoot');  
 },  
 squat:function(){  
 //蹲下  
 console.log('squat');  
 }  
 };  
   
 //动作控制类  
 var Action={  
 //改变状态方法  
 changeState: function () {  
 //组合动作通过传递多个参数实现  
 var arg=arguments;  
 //重置内部状态  
 \_currentState={};  
 //如果有动作则添加动作  
 if(arg.length) {  
 //遍历动作  
 for(var i=0,len=arg.length;i<len;i++){  
 //向内部状态中添加动作  
 \_currentState[arg[i]]=true;  
 }  
 //返回动作控制类  
 return this;  
 }  
   
   
 },  
 //执行动作  
 goes:function(){  
 console.log('触发一次动作');  
 //遍历内部状态保存的动作  
 for(var i in \_currentState){  
 //如果该动作存在则执行  
 states[i] && states[i]();  
 }  
 return this;  
 }  
   
 };  
   
 //返回接口方法 change gose  
 return {  
 change:Action.changeState,  
 goes:Action.goes  
 }  
};

两种使用方式

函数方式

MarryState().change('jump','shoot')  
 .goes()  
 .goes()  
 .change('shoot')  
 .goes();

状态类

//创建一个超级玛丽  
var *marry*=new MarryState();  
*marry*.change('jump','shoot')  
 .goes()  
 .goes()  
 .change('shoot')  
 .goes();

**策略模式**

策略模式将定义的一组算法封装起来，使其相互之间可以替换。封装的算法具有一个独立性，不会随客户端变化而变化。

商品促销（打折）一部分商品5折，一部分商品8折，一部分商品9折。还可以有反馈活动，普通用户满100返30，高级VIP用户满100返50……

结构上，它与状态模式很像，也是在内部封装一个对象，然后通过返回的接口对象实现对内部对象的调用，不同点是，策略模式不需要管理状态、状态间没有依赖关系、策略之间可以相互替换、在策略对象内部保存的是相互独立的一些算法。为了实现对每种商品策略调用。首先要将这些算法封装在一个策略对象内，然后对每种商品的策略调用时，直接对策略对象中的算法调用即可，而策略算法又独立地分装在策略对象内。为方便管理与使用，需要一个调用接口对象来实现对策略算法的调用。

**策略对象**

<script>  
 //价格策略对象  
 var *PriceStrategy*=function(){  
 //内部算法对象  
 var stragtegy={  
 return30:function(price){  
 //100返30  
 return price+parseInt(price/100)\*30;  
 },  
 return50:function(price){  
 //100返50  
 return price+parseInt(price/100)\*50;  
 },  
 return90:function(price){  
 //9折  
 return price\*100\*90/10000;  
 },  
 return80:function(price){  
 //8折  
 return price\*100\*80/10000;  
 },  
 return50:function(price){  
 //5折  
 return price\*100\*50/10000;  
 }  
  
 }  
 //策略算法调用接口  
 return function(algorithm,price){  
 //如果算法存在，则调用算法，否则返回false  
 return stragtegy[algorithm] && stragtegy[algorithm](price)  
 }  
 }();  
 //策略应用  
  
 var *price*=*PriceStrategy*('return80','345.67');  
 console.log(*price*);  
  
  
</script>

**缓冲函数**

其实缓冲函数就是策略算法

$('div').animate({width:'200px'},1000,'linear');

$('div').animate({width:'200px'},1000,'swing');

运动的方式就是策略函数

**表单验证**

表单验证策略对象

//表单正则验证策略对象  
var InputStrategy=function(){  
 var strategy={  
 //是否为空  
 notNull:function(value){  
 return /\s+/.test(value)?'请输入内容':'';  
 },  
 //是否是一个数字  
 number:function(value){  
 return /^[0-9]+(\.[0-9]+)?$/.test(value)?'':'请输入数字';  
 },  
 //是否是本地电话  
 phone:function(value){  
 return /^\d{3}\-\d{8}$|^\d{4}\-\d{7}$/.test(value)?'':'请输入正确的电话号码，如010-12345678或0402-1234567';  
  
 }  
 };  
  
  
 return{  
 //验证接口 type算法 value表单值  
 check:function(type,value){  
 //去除收尾空白字符  
 value=value.replace(/^\s+|\s+$/g,'');  
 return strategy[type]?strategy[type](value):'没有该类型的检测方法'  
 },  
 //添加策略  
 addStrategy:function(type,fn){  
 strategy[type]=fn;  
 }  
 }  
};

此例子与上面例子的区别：

它在最后返回的接口中添加一个策略接口，因为已有的策略即使再多，有时候也是不能满足其他工程师的需求的。此时我们为增加策略对象的拓展性，我们增加一个为了策略对象添加策略算法的接口，这样我们如果有新的策略算法只需要通过addStrategy方法即可实现策略算法的添加，而不用修改策略对象内部的代码。

//拓展 可以延伸算法

InputStrategy.addStrategy('nickname',function(value){

return /^[a-zA-Z]\w{3,7}$/.test(value)?'':'请输入4-8位昵称，如：xiaomomo';

});

算法调用：

//外观模式简化元素的获取

function $tag(tag,context){

context=context||document;

return context.getElementsByTagName(tag);

}

//提交按钮点击

Stag('input')[1].onclick=function(){

//获取输入框内的内容

var value=$tag('input')[0].value;

//获取日期格式验证结果

$tag('span')[0].innerHTML=InputStrategy.check('nickname',value);

}

**命令模式**

命令模式：将请求与实现解耦并封装成独立对象，从而使不同的请求对客户端的实现参数化。

自由化创建视图

需求：要创建的图片，可以有一张也可以有多张。所以你需要在命令对象内部合理封装这些处理方法，但是要提供一个命令接口，来合理化接受并处理你的命令

**命名对象**

//模块实现模块

var viewCommand=( function(){

//方法集合

var Action={

//创建方法

create:function(){},

//展示方法

display:function(){}

};

//命令接口

return function excute(){}

})();

**视图创建**

<script>  
 //模块实现模块  
 var viewCommand=(function(){  
 var tpl={  
 //展示图片结构模板  
 product:[  
 '<div>',  
 '<img src="{#src#}"/>',  
 '<p>{#text#}</p>'  
 '</div>'  
 ].join(''),  
 //展示标题结构模板  
 title:[  
 '<div class="title">',  
 '<div class="main">',  
 '<h2>{#title#}</h2>',  
 '<p>{#tips#}</p>',  
 '</div>',  
 '</div>'  
 ].join('')  
 },  
 //格式化字符串缓存字符串  
 html='';  
 //格式化字符串 如'<div>{#content#}</div>'用{content:'demo'}  
 //替换后可得到字符串：'<div>demo</div>'  
   
 function formateString(str,obj){  
 //替换'{#'与'#}'之间的字符串  
 return str.replace(/\{#(\w+)#\}/g,function(match,key){  
 return obj[key];  
 })  
 }  
   
 //方法集合  
 var Action={}  
 //命令接口  
 return function excute(){}  
   
 })();  
</script>

创建模块视图就可通过对模块视图数据进行字符串模板格式化来获取，并封装在create方法里。

create:function(data,view){  
 //解析数据如果数据是一个数组  
 if(data.length){  
 //遍历数组  
 for(var i= 0,len=data.length;i<len;i++){  
 //将格式化后字符串缓存到HTML中  
 html+=formateString(tpl(view),data[i]);  
 }  
 }else{  
 //直接格式化字符串缓存到HTML中  
 html+=formateString(tpl(view),data);  
 }  
}

在create方法data参数指定了模块视图数据，view参数指定了视图模板，用数据去格式化视图模板即可获得视图字符串。不过注意的是传入的data参数可能是对象也可能是数组，所以并适配传入data的数据是很必要的，如果是数组，一一格式化每个数组成员数据并将结果缓存在html中，否则直接格式化传入的data数据。

视图展示

//视图展示  
display:function(container,data,view){  
 //如果传入数据  
 if(data){  
 //根据给定数据创建视图  
 this.create(data,view);  
 }  
 //展示模块  
 document.getElementById(container).innerHTML=html;  
 //展示后清空缓存的字符串  
 html='';  
}

对于视图模块展示方法display，首先要判断是否传入了数据，如果传入了数据则解析这组数据，生产并缓存视图模块对应的字符串，然后将缓存的模块字符串插入页面中显示模块，最后将展示后的视图缓存字符串清空。

**命令接口**

对于这个接口的参数应该包括两部分，第一部分是命令对象内部的方法名称，第二部分是命令对象内部方法对应的参数，通过这两个参数你就可以实现自由地对视图模块创建或者展示了。

return function excute(msg){  
 //解析命令，如果msg.param不是数据则将其转化为数组（apply方法要求第二个参数为数组）  
 msg.param=Object.propertype.toString.call(msg.param)==="[objectArray]"?  
 msg.param:[msg.param];  
 //Action 内部调用的方法引用this，所以此处为保证作用域this执行传入action  
 Action[msg.command].apply(Action,msg.param)  
}

示例

测试

<script>  
 //模块实现模块  
 var *viewCommand*=(function(){  
 var tpl={  
 //展示图片结构模板  
 product:[  
 '<div>',  
 '<img src="{#src#}"/>',  
 '<p>{#text#}</p>',  
 '</div>'  
 ].join(''),  
 //展示标题结构模板  
 title:[  
 '<div class="title">',  
 '<div class="main">',  
 '<h2>{#title#}</h2>',  
 '<p>{#tips#}</p>',  
 '</div>',  
 '</div>'  
 ].join('')  
 };  
 //格式化字符串缓存字符串  
 html='';  
 //格式化字符串 如'<div>{#content#}</div>'用{content:'demo'}  
 //替换后可得到字符串：'<div>demo</div>'  
  
 function formatString(str,obj){  
 //替换'{#'与'#}'之间的字符串  
 return str.replace(/\{#(\w+)#\}/g,function(match,key){  
 return obj[key];  
 })  
 }  
  
 //方法集合  
 var Action={  
 create:function(data,view){  
 //解析数据如果数据是一个数组  
 if(data.length){  
 //遍历数组  
 for(var i= 0,len=data.length;i<len;i++){  
 //将格式化后字符串缓存到HTML中  
 html+=formatString(tpl[view],data[i]);  
 }  
 }else{  
 //直接格式化字符串缓存到HTML中  
 html+=formatString(tpl[view],data);  
 }  
 },  
 //视图展示  
 display:function(container,data,view){  
 //如果传入数据  
 if(data){  
 //根据给定数据创建视图  
 this.create(data,view);  
 }  
 //展示模块  
 document.getElementById(container).innerHTML=html;  
 //展示后清空缓存的字符串  
 html='';  
 }  
 };  
 //命令接口  
 return function excute(msg){  
 //解析命令，如果msg.param不是数据则将其转化为数组（apply方法要求第二个参数为数组）  
 msg.param=Object.prototype.toString.call(msg.param)==="[objectArray]"?  
 msg.param:[msg.param];  
 //Action 内部调用的方法引用this，所以此处为保证作用域this执行传入action  
 Action[msg.command].apply(Action,msg.param)  
 }  
  
 })();  
 //产品展示数据  
 var *productData*=[{src:'img/1.jpg',text:'绽放的桃花'},  
 {src:'img/1.jpg',text:'阳光如此温暖'},  
 {src:'img/1.jpg',text:'鲜花如此美丽'}];  
  
 //模板标题数据  
 titleData={  
 title:'三伏天，真的受不了了',  
 tips:'给你一份温情'  
 };  
  
 *viewCommand*({  
 command:'display',  
 param:['product',titleData,'title']  
 });  
  
 *viewCommand*({  
 command:'create',  
 param:[{  
 src:'img/1.jpg',  
 text:'如此美景'},'product']  
 });  
  
 *viewCommand*({  
 command:'display',  
 param:['product',*productData*,'product']  
 });  
  
</script>