**1、原型模式**

原型模式：用原型实例指向创建对象的类，使用于创建新的对象的类共享原型对象的属性以及方法。JavaScript是基于原型链实现对象之间的继承的，这种继承是基于一种对属性或者方法的共享，而不是对属性和方法的复制。

实例焦点图：（网页一种图片轮播效果，切换效果）如果一个页面中有多个这样的焦点图，其切换画面一般是多样化的，有的可能是上下切换，左右切换、渐隐切换，缩放切换等。

最佳的解决方案

**2、单例模式**

单例模式：又被称为单体模式，是只允许实例化一次的对象类。有时候可以用一个对象来规划一个命名空间，来管理对象上的属性和方法。

function g(id){

return document.getElementById(id);

}

function css(id,key,value){

g(id).style[key]=value;

}

function attr(id,key,value){

g(id)[key]=value;

}

function html(id,value){

g(id).innerHTML=value;

}

function on(id,type,fn){

g(id)['on'+type]=fn;

}

以上是鼠标滑动特效的部分代码，而如果有别人同时也写了一个一样的函数则，就有覆盖的可能，所以最好是使用单例模式

先来看看什么是命名空间：命名空间就是namespace，用它来解决这么一类的问题也就是按照这个原理，我们形成的单例模式。

var whq={

g:function (id){

return document.getElementById(id);

},

css:function (id,key,value){

g(id).style[key]=value;

}

……

}；

在使用的位置，加上空间名.属性名或者空间名.方法名

所以以上程序要改为：

var whq={

g:function (id){

return document.getElementById(id);

},

css:function (id,key,value){

this.g(id).style[key]=value;

}

……

}；

单例模式还可以用作模块划分：（管理代码库的各个模块）

如百度的js：

baidu.dom.addClass 添加元素类

baidu.dom.append

baidu.event.stopPropagation

baidu.event.preventDefault

baidu.string.encodeHTML

根据单例模式的特性，我们来创建一下自己的小型代码库

var whq={

Dom:{

getObject:function(){},

setAttr:function(){}

},

Event:{

say:function(){

return 'say';

}

},

Tool:{}

others:{}

};

使用方法： whq.Dom.getObject();

单列模式：实现静态变量是无法修改，但是可以提供使用

var Conf=（

function(){

var conf={

MAX\_NUM:100,

MIN\_NUM:1,

COUNT:1000

}

return {

get:function(name){

return conf[name]?conf[name]:null;

}

}

}）（）；

惰性单例：

单例对象延迟创建就是惰性创建

var LazySingle=(function(){

var \_instance=null;

function Single(){

return {

publicMethod:function(){},

publicProperty:'1.0';

}

}

//获取单例对象接口

return function(){

if(!\_instance)

{

\_instance=Single();

}

return \_instance;

}

})();

使用：LazySingle().publicProperty

以上是创建型设计模式

**结构型设计模式**

1、外观模式

外观模式：为一组复制的子系统接口提供一个更高级的统一接口，通过这个接口使得子系统接口访问更容易。在JavaScript中有时也会用于对底层机构兼容性做统一封装来简化用户使用

添加一个点击事件：

document.onclick=function(){

}

在开发中这样添加事件是很危险的，因为别的程序员也可以这样写，有被覆盖的危险。

处理办法：addEventListener

兼容方式：

//外观模式实现

function addEvent(dom,type,fn)

{

if(dom.addEventListener){ //chrome

dom.addEventListener(type,fn,false);

}else if(dom.attachEvent){

dom.attachEvent('on'+type,fn); //IE

}else

{

dom['on'+type]=fn; //other

}

}

这样就可以绑定多次了

还有个兼容问题也可以经过外观模式解决

//获取事件对象

//获取事件对象  
var getEvent=function(event){  
 return event||window.event;  
};  
  
//获取元素  
var getTarget=function(event){  
 var event=getEvent(event);  
 return event.target||event.scrElement;  
};  
  
//阻止默认行为  
var preventDefault=function(event){  
 var event=getEvent(event);  
 if(event.preventDefault)  
 {  
 event.preventDefault();  
 }  
 else  
 {  
 event.returnValue=false;  
 }  
};

也可以使用代码库

2、适配器模式

适配器模式：将一个类的接口转化成另外一个接口，以满足用户需求，使类之间接口不兼容问题通过适配器得以解决。

生活中的适配器：

jQuery适配器：

如果一个公司出现了一个框架A，与jQuery代码书写格式很像，就可以使用这种方式来添加适配器。

window.A=A=jQuery;

适配异类框架

定义一个框架

//定义框架  
var *A*=*A*||{};  
//通过ID获取元素  
*A*.g=function(id){  
 return document.getElementById(id);  
};  
//为元素绑定事件  
*A*.on=function(id,type,fn){  
 var dom=typeof id==="string"?this.g(id):id;  
 if(dom.addEventListener)  
 {  
 dom.addEventListener(type,fn,false);  
 }else if(dom.attachEvent)  
 {  
 dom.attachEvent('on'+type,fn);  
 }  
 else  
 {  
 dom['on'+type]=fn;  
 }  
   
 //窗口加载完成事件  
 *A*.on(window,'load',function(){  
 *A*.on('mybutton','click',function(){  
   
 });  
 });  
   
   
};

引入jQuery来换原有的A库。

分析：首先G方法是通过ID获取元素，所有通过$方法获取jQuery对象，然后通过GEt获取第一个成员。通过ID on 函数就要加#了

A.g=function(id){

return $(id).get(0);

}

A.on=function(id,type,fn){

var dom=typeof id==='string' ?$('#'+id):$(id);

dom.on(type,fn);

}

适配器使用，如果有血缘关系比较近的就容易实现起来，如果有点远的适配器就有点复杂了。尽量引用相似框架。

参数适配器：

适用于方法需要传递多个参数

function dosomething(name,title,age,color,size){}

因为参数过多无法准确的记住顺序，所以我们以类的方式传入

形如：

obj.name:name

obj.title:title

obj.age:age

function dosomething(obj){}

当我们调用它的时候也不知道传递的参数是否完整，如有一些必须参数没有传入，一些参数是有默认值的等等。我们通常的做法就是使用适配器来适配传入的这个参数对象。

function doSomeThing(obj){  
 var \_adapter={  
 name:'西信传媒',  
 title:'设计模式',  
 age:24,  
 color:'red',  
 size:100,  
 prize:50  
 };  
   
 for(var i in \_adapter)  
 {  
 \_adapter[i]=obj[i]||\_adapter[i];  
 }  
}

数据适配

var arr=['javascript','book','编程语言','7月30日'];

因为数组中的每个成员代表的意义不同，所以这种数据结构语义不好，这种情况我们常会使用适配器模式。

如

var obj={

name:'',

type:'',

title:'',

time:''

}

我们的适配方式

function arrToObjAdapter(arr)

{

return {

name:arr[0],

type:arr[1],

title:arr[2],

date:arr[3]

}

}

var adapterData=arrToObjAdapter(arr);

服务器端数据适配

服务端适配器可以解决前后端的数据依赖，前端程序不再为后端传递的数据所束缚。如果后端因为架构改变导致传递的数据结构发生变化，我们只需要写个适配器就可以了。

function ajaxAdapter(data){

return [data['key1'],data['key2'],data['key3']];

}

$.ajax({

url:'',

sucess:function(data,status)

{

if(data){

doSomething(ajaxAdapter(data));

}

}

});

3、装饰者模式

装饰者模式：在不改变原对象的基础上，通过对其进行包装拓展（添加属性或者方法）使原有对象可以满足用户的更复杂需求。

因为需求在不断的改变。

需求：

用户信息表单需求，当用户点击输入框时，如果输入框输入的内容有限制，那么其后面显示用户输入内容的限制格式的提示文案，默认输入框上边显示一行提示文案，当用户点击输入框时文案消失。

<!DOCTYPE html>  
<html>  
<head lang=**"en"**>  
 <meta charset=**"UTF-8"**>  
 <title></title>  
 <script>  
 window.onload=function(){  
 var tel=document.getElementById('tel\_input');  
  
 tel.onclick=function(){  
 document.getElementById('tel\_warning').style.display='block';  
 };  
 decorator('tel\_input',function(){  
 //document.getElementById('tel\_warning').style.display='none';  
 alert('hello');  
 });  
 };  
  
 var decorator= function (input,fn) {  
 //获取事件源  
 var input=document.getElementById(input);  
  
 //若事件源已经绑定事件  
 if(typeof input.onclick==='function'){  
 //缓存事件源原有的回调函数  
 var oldClickfn=input.onclick;  
 //为新的事件源定义新的事件  
 input.onclick=function(){  
 //事件原有的回调函数  
 oldClickfn();  
 //执行事件新增的函数  
 fn();  
 };  
 }else  
 {  
 //如果事件源没有绑定事件，直接为事件源添加新的回调函数  
 input.onclick=fn;  
 }  
 };  
 </script>  
</head>  
<body>  
 <input type=**"text"** id=**"tel\_input"** />  
 <div id=**"tel\_warning"** style=**"**display:**none**;**"**>**输入框提示格式**</div>  
</body>  
</html>

4、桥接模式

桥接模式是在系统沿着多个维度变化的同时，又不增加其复杂度并已达到解耦。

代码片段

var spans=document.getElementsByTagName('span');

spans[0].onmouseover=function(){

this.style.color='green';

this.style.background='#ccc';

}

spans[0].onmouseout=function(){

this.style.color='red';

this.style.background='#ddd';

}

……

上述代码的问题：是重复冗余，而属性变化多

这类的解决方案就是抽取共同点，让它解除与事件中的this 的耦合。

function changeStyle(obj,prop,value)

{

obj.style[prop]=value;

}

在使用中还要有一个方法让这个函数与事件方法链接起来，而这个连接的匿名函数就是我们所谓的桥梁，在设计模式中称为桥接模式

桥接模式的扩展：多元化对象

例如：一个canvas跑步游戏的试验，对于游戏中的人、小精灵、小球等一系列的实物都有动作单元，而他们的每个动作实现起来方式又都是统一的，比如人和精灵和球的运动其实就是坐标位置X和Y的变化，球的颜色与精灵的色彩的绘制方式都相似等。这样我们可以将这些多维变化部分，提取出来作为一个抽象运动单元进行保存，而当我们创建实体时，将需要的每个抽象动作单元通过桥接，链接在一起动作，这样他们之间不会相互影响并且该方式降低了他们之间的耦合。

<script>  
 //多维变量类  
 //运动单元  
 function Speed(x,y){  
 this.x=x;  
 this.y=y;  
 }  
 Speed.prototype.run=function(){  
 console.log('运动起来');  
 };  
  
 //着色单元  
 function Color(col)  
 {  
 this.color=col;  
 }  
  
 Color.prototype.draw=function()  
 {  
 console.log('绘制色彩');  
 };  
  
 //变形单元  
 function Shape(sp){  
 this.shape=sp;  
 }  
  
 Shape.prototype.change=function(){  
 console.log('改变行状');  
 };  
  
 //说话单元  
  
 function Speek(wd){  
 this.word=wd;  
 }  
  
 Speek.prototype.say=function(){  
 console.log('书写字体');  
 };  
  
 //创建一个人物类，他可以运动以及说话  
 function People(x,y,f){  
 this.speed=new Speed(x,y);  
 this.font=new Speek(f);  
 }  
  
 People.prototype.init=function(){  
 this.speed.run();  
 this.font.say();  
 };  
  
 //创建一个精灵类，让它可以运动可以着色可以改变形状  
 function Spirite(x,y,c,s)  
 {  
 this.speed=new Speed(x,y);  
 this.color=new Color(c);  
 this.shape=new Shape(s);  
 }  
  
 Spirite.prototype.init=function(){  
 this.speed.run();  
 this.color.draw();  
 this.shape.change();  
 };  
  
 //创建一个球类，它可以运动，着色  
 function Ball(x,y,c){  
 this.speed=new Speed(x,y);  
 this.color=new Color(c);  
 }  
  
 Ball.prototype.init=function(){  
 this.speed.run();  
 this.color.draw();  
 };  
  
 //实例化一个人物对象  
 var *p*=new People(10,20,32);  
 *p*.init();  
</script>

5、组合模式

组合模式：部分--整体模式，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。

新闻模块：

文字新闻、图片新闻、新闻链接、文字和图片的混合等

定义新闻虚拟父类News

<!DOCTYPE html>  
<html>  
<head lang=**"en"**>  
 <meta charset=**"UTF-8"**>  
 <title></title>  
  
 <style>  
 #news { width: 400**px**;  
 height: 400**px**;  
 background: #ccc;  
 }  
 </style>  
</head>  
<body>  
  
 <script>  
 function inheritObject(o)  
 {  
 function F(){}  
 F.prototype=o;  
 return new F();  
 }  
  
 function inheritPrototype(subClass,supperClass)  
 {  
 var p=inheritObject(supperClass.prototype);  
 p.constructor=subClass;  
 subClass.prototype=p;  
 }  
  
 var News=function(){  
 //子组件容器  
 this.children=[];  
 this.element=null;  
 };  
  
 News.prototype={  
 init:function(){  
 throw new Error('请重写你的方法');  
 },  
 add:function(){  
 throw new Error('请重写你的方法');  
 },  
 getElement:function(){  
 throw new Error('请重写你的方法');  
 }  
 };  
  
 //组合的容器类  
 var Container=function(id,parent){  
 //构造函数继承父类  
 News.call(this);  
 //模块id  
 this.id=id;  
 //模块父容器  
 this.parent=parent;  
 //构建方法  
 this.init();  
 };  
  
 //寄生式继承父类原型方法  
 inheritPrototype(Container,News);  
 //构建方法  
 Container.prototype.init=function(){  
 this.element=document.createElement('ul');  
 this.element.id=this.id;  
 this.element.className='new-container';  
 };  
  
 //添加子元素方法  
 Container.prototype.add=function(child){  
 //在子元素容器中插入子元素  
 this.children.push(child);  
 //插入当前组件元素树中  
 this.element.appendChild(child.getElement());  
 return this;  
 };  
 // 获取当前元素方法  
  
 Container.prototype.getElement=function(){  
 return this.element;  
 };  
  
 //显示方法  
 Container.prototype.show=function(){  
 this.parent.appendChild(this.element);  
 };  
  
 //下一层级的行成员集合类以及后面的新闻组合体类实现的方式与之类似  
 var Item=function(classname){  
 News.call(this);  
 this.classname =classname||'';  
 this.init();  
 };  
  
 inheritPrototype(Item,News);  
  
 Item.prototype.init=function(){  
 this.element=document.createElement('li');  
 this.element.className=this.classname;  
 };  
  
 Item.prototype.add=function(child){  
 //在子元素容器中插入子元素  
 this.children.push(child);  
 //插入当前组件元素树中  
 this.element.appendChild(child.getElement());  
 return this;  
 };  
  
 Item.prototype.getElement=function(){  
 return this.element;  
 };  
  
 var NewsGroup=function(classname){  
 News.call(this);  
 this.classname=classname||'';  
 this.init();  
 };  
  
 inheritPrototype(NewsGroup,News);  
 NewsGroup.prototype.init=function(){  
 this.element=document.createElement('div');  
 this.element.className=this.classname;  
 };  
  
 NewsGroup.prototype.add=function(child){  
 //在子元素容器中插入子元素  
 this.children.push(child);  
 //插入当前组件元素树中  
 this.element.appendChild(child.getElement());  
 return this;  
 };  
  
 NewsGroup.prototype.getElement=function(){  
 return this.element;  
 };  
  
 //创建新闻类(创建图片新闻类)  
  
 var ImageNews=function(url,href,classname){  
 News.call(this);  
 this.url=url||'';  
 this.href=href||'#';  
 this.classname=classname||'nomal';  
 this.init();  
 };  
  
 inheritPrototype(ImageNews,News);  
 ImageNews.prototype.init=function(){  
 this.element=document.createElement('a');  
 var img=new Image();  
 img.src=this.url;  
 this.element.appendChild(img);  
 this.element.className='image-news'+this.classname;  
 this.element.href=this.href;  
 };  
  
 ImageNews.prototype.add=function(){};  
 ImageNews.prototype.getElement=function(){  
 return this.element;  
 };  
  
 //其他基类的创建  
 var IconNews=function(text,href,type){  
 News.call(this);  
 this.text=text||'';  
 this.href=href||'#';  
 this.type=type||'video';  
 this.init();  
 };  
  
 inheritPrototype(IconNews,News);  
 IconNews.prototype.init=function(){  
 this.element=document.createElement('a');  
 this.element.innerHTML=this.text;  
 this.element.href=this.href;  
 this.element.className='icon'+this.type;  
 };  
  
 IconNews.prototype.add=function(){};  
 IconNews.prototype.getElement=function(){  
 return this.element;  
 };  
  
 var EasyNews=function(text,href){  
 News.call(this);  
 this.text=text||'';  
 this.href=href||'#';  
 this.init();  
 };  
 inheritPrototype(EasyNews,News);  
 EasyNews.prototype.init=function(){  
 this.element=document.createElement('a');  
 this.element.innerHTML=this.text;  
 this.element.href=this.href;  
 this.element.className='text';  
 };  
  
 EasyNews.prototype.add=function(){};  
 EasyNews.prototype.getElement=function(){  
 return this.element;  
 };  
  
 var TypeNews=function(text,href,type,pos){  
 News.call(this);  
 this.text=text||'';  
 this.href=href||'#';  
 this.type=type||'';  
 this.pos=pos||'left';  
 this.init();  
 };  
 inheritPrototype(TypeNews,News);  
 TypeNews.prototype.init=function(){  
 this.element=document.createElement('a');  
 if(this.pos==='left'){  
 this.element.innerHTML='['+this.type+']'+this.text;  
 }else  
 {  
 this.element.innerHTML=this.text+'['+this.type+']';  
 }  
 this.element.href=this.href;  
 this.element.className='text';  
 };  
  
 TypeNews.prototype.add=function(){};  
 TypeNews.prototype.getElement=function(){  
 return this.element;  
 };  
  
 //创建新闻模块  
 var *news1*=new Container('news',document.body);  
  
 *news1*.add(  
 new Item('normal').add(new IconNews('我想和你唱','#','video'))  
 ).add(  
 new Item('normal').add(new IconNews('保护水源','#','live'))  
 ).add(  
 new NewsGroup('has-img').add(  
 new ImageNews('img/1.jpg','#','small')  
 ).add(new EasyNews('娱乐新闻报道','#')  
 ).add(new EasyNews('西安大雁塔','#'))  
 ).add(  
 new Item('normal').add(  
 new TypeNews('科比的世界','#','NBA','left')  
 )  
 ).add(  
 new Item('normal').add(  
 new TypeNews('中国的足球真的是那么差吗','#','football','right')  
 )  
 ).show();  
  
 </script>  
</body>  
</html>

此种方法经常在表单中应用

开发：创建表单基类，然后3个组合类FormItem FieldsetItem Group,以及成员类 InputItem LabelItem SpanItem TextareaItem.

6、享元模式

享元模式：运用共享技术有效地支持大量的细粒度的对象，避免对象间拥有相同内容造成多余的开销。

享元模式主要还是对数据、方法共享分离，它将数据和方法分成内部数据、内部方法和外部数据、外部方法。内部方法与内部数据指的是相似或者共有的数据和方法，所以将这一部分提取出来减少开销，以提高性能。

<!DOCTYPE html>  
<html>  
<head lang=**"en"**>  
 <meta charset=**"UTF-8"**>  
 <title></title>  
  
</head>  
<body>  
 <div id=**"container"**></div>  
 <script>  
 var *Flyweight*=function(){  
 //已创建的元素  
 var created=[];  
 //创建一个新闻包装容器  
 function create(){  
 var dom=document.createElement('div');  
 //将容器插入新闻列表容器中  
 document.getElementById('container').appendChild(dom);  
 //缓存新创建的元素  
 created.push(dom);  
 //返回创建的新元素  
 return dom;  
 }  
  
 return{  
 //获取创建新闻元素方法  
 getDiv:function(){  
 //如果已创建的元素小于当前页元素总个数，则创建  
 if(created.length<5)  
 {  
 return create();  
 }else  
 {  
 //获取第一个元素，并插入最后面  
 var div=created.shift();  
 created.push(div);  
 return div;  
 }  
  
 }  
 }  
 }();  
  
 var *paper*=0;  
 var *article*=['1','2','3','4','5','6'];  
 num=5;  
  
 len=*article*.length;  
 for(var *i*=0;*i*<5;*i*++){  
 if(*article*[*i*])  
 *Flyweight*.getDiv().innerHTML=*article*[*i*];  
 }  
  
 document.getElementById('next\_page').onclick=function(){  
 if(*article*.length<5)  
 return ;  
  
 var n=++*paper*\*num%len,  
 j=0;  
 for(;j<5;j++)  
 {  
 if(*article*[n+j]){  
 *Flyweight*.getDiv().innerHTML=*article*[n+j];  
 }else if(*article*[n+j-len])  
 {  
 *Flyweight*.getDiv().innerHTML=*article*[n+j-len];  
 }else  
 {  
 *Flyweight*.getDiv().innerHTML='';  
 }  
 }  
 };  
 </script>  
</body>  
</html>

享元动作

<script>  
 var *Flyweight*={  
 moveX:function(x){  
 this.x=x;  
 },  
 moveY:function(y){  
 this.y=y;  
 }  
 };  
 //让人继承移动的方法  
 var Player=function(x,y,c){  
 this.x=x;  
 this.y=y;  
 this.color=c;  
 };  
 Player.prototype= *Flyweight*;  
 Player.prototype.changeC=function(c){  
 this.color=c;  
 };  
  
 //让精灵继承移动的方法  
 var Spirit=function(x,y,r){  
 this.x=x;  
 this.y=y;  
 this.r=r;  
 };  
 Spirit.prototype=*Flyweight*;  
 Spirit.prototype.changeR=function(r){  
 this.r=r;  
 };  
 // 创建一个人  
 var *player1*=new Player(10,15,'blue');  
 console.log(*player1*);  
 *player1*.moveX(20);  
 *player1*.moveY(39);  
 *player1*.changeC('pink');  
 console.log(*player1*);  
   
</script>