js设计模式

2018年12月18日　　16:51

一、创建型设计模式

（1）简单工厂模式

简单工厂模式：又叫静态工厂方法，由一个工厂对象决定创建某一种产品对象类的实例，主要用于创建同一类对象。

```
function createFactory(name,time,type){
//创建一个对象,并对对象拓展属性和方法
var o=new Object();
o.name=name;
o.time=time;
o.type=type;
o.getName=function(){
console.log(this.name);
};
return o;
}
```

e.g.利用此模式实现各种框

```
function createPop(type,text){
var o=new Object();
o.content=text;
o.show=function(){//detail};
if(type=='alert'){
//alert detail
}
if(type=='prompt'){
//prompt detail
}
if(type=='confirm'){
//confirm detail
}
return o;
}

var userNameAlert=createPop('alert','警示框');
```

（2）工厂方法模式

工厂方法模式：通过对产品类的抽象使其创建业务，主要负责用于创建多类产品的实例。

为了避免出现问题，可对第一种模式加工，判断当前this是否指向当前对象

```
var Demo=function(){
if(!(this instanceof Demo)){
return new Demo();
```

```
}
}
Demo.prototype={
show:function(){
console.log('show');
}
}
```

e.g.利用此模式实现各个同学对应的样式

```
//安全模式创建的工厂类
var Factory=function(type,content){
if(this instanceof Factory){
var s=new this[type](content);
return s;
}else{
return new Factory(type,content);
}
//工厂原型中设置创建所有类型数据对象基类
Factory.prototype={
Java:function(content){
//java detail
},
UI:function(content){
this.content=content;
(function(content){
var div=document.createElement('div');
div.innerHTML=content;
div.style.border='1px solid red';
document.getElementById('container').appendChild(div);
})(content);
},
Php:function(content){
//php detail
}
}

Var data=[{type:'Java',content:'java'},
{type:'Php',content:'php'},
...];
for(var i=0;i<data.length;i++){
Factory(data[i].type,data[i].content);
}
```

（3）抽象工厂模式

抽象工厂模式：通过对类的工厂抽象使其业务用于对产品类簇的创建，而不负责创建某一类产品的实例。

e.g.利用此模式实现宝马

```
var VehicleFactory=function(subType,superType){
//判断抽象工厂中是否有该抽象类
```

```
if(typeof VehicleFactory[superType]==='function'){//缓存类
function F(){};
//继承父类属性和方法
F.prototype=new VehicleFactory[superType]();
//将子类constructor指向子类
subType.constructor=subType;
//子类原型继承“父类”
subType.prototype=new F();
}else{
throw new Error('未创建该抽象类');
}
}

//小汽车抽象类
VehicleFactory.Car=function(){
this.type='car';
};
VehicleFactory.Car.prototype={
getPrice:function(){
throw new Error('抽象方法不能调用');},
getSpeed:function(){
throw new Error('抽象方法不能调用');
}
}

//公交车抽象类
VehicleFactory.Bus=function(){
this.type='bus';
};
VehicleFactory.Bus.prototype={
getPrice:function(){
throw new Error('抽象方法不能调用');},
getSpeed:function(){
throw new Error('抽象方法不能调用');
}
}

//宝马汽车子类
var BMW=function(price,speed){
this.price-price;
this.speed=speed;
}
//抽象工厂实现对Car抽象类的继承
VehicleFactory(BMW,'Car');
BMW.prototype.getPrice=function(){
return this.price;
};
BMW.prototype.getPrice=function(){
return this.speed;
}
```

```
var bmw=new BMW(10000,1000);
console.log(bmw.getPrice());//10000
ocnsole.log(bmw.type);//car
```

（4）建造者模式

建造者模式：将一个复杂对象的构建层与其表示层相互分离，同样的构建过程可采用不同的表示。

e.g.利用此模式实现发布简历

```
var Human=function(param){
this.skill=param && param.skill||'保密';
this.hobby=param && param.hobby||'保密';
}
Human.prototype={
getSkill:function(){
return this.skill;
},
getHobby:function(){
return this.hobby;
}
};

var Named=function(name){
var that=this;
(function(name,that){
that.wholeName=name;
if(name.indexOf(" ")>-1){
that.FirstName=name.slice(0,name.indexOf(' '));
that.LastName=name.slice(name.indexOf(' '));
}
})(name,that);
};

var Work=function(work){
var that=this;
(function(work,that){
switch(work){
case 'code':
that.work='aaa';
that.workDescript='biancheng';
break;
case 'ui':
case 'ue':
that.work=''bbb';
that.workDescript='sheji';
break;
default:
that.work=work;
that.workDescript='unknown';
}
```

```
})(work,that);
};
Work.prototype.changeWork=function(work){
this.work=work;
};
Work.prototype.changeDescript=function(des){
this.workDescript=des;
};

var Person=function(name,work){
var _person=new Human();
_person.name=new Named(name);
_person.work=new Work(work);
return _person;
}

var person=new Person('test','code');
console.log(person);
```

（5）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及方法。

e.g.实现图片轮播

解决方案1：

```
var LoopImages=function(imgArr,container){
this.imagesArray=imgArr;//图片数组
this.container=container;//图片容器
this.createImage=function(){};//创建轮播图片
this.changeImage=function(){};//切换图片
}

var SlideLoopImg=function(imgArr,container){
LoopImages.call(this,imgArr,container);
this.changeImage=function(){
console.log('slide');
}
};
var FadeLoopImg=function(imgArr,container,arrow){
LoopImages.call(this,imgArr,container);
this.arrow=arrow;
this.changeImage=function(){
console.log('fade');
}
};

var fadeImg=new FadeLoopImg(['01.jpg','02.jpg'],'slide',['left.jpg','right.jpg']);
```

更优的解决方案（使用原型模式）：

将可复用的、可共享的、耗时大的从基类中提取出来放在原型中，子类通过组合继承或寄生组合继承继承方法和属性。

```
var LoopImages=function(imgArr,container){
this.imagesArray=imgArr;//图片数组
this.container=container;//图片容器
}
LoopImages.prototype={
createImage:function(){
console.log('createimg');
},
changeImage:function(){
console.log('changeimg');
}
};
var SlideLoopImg=function(imgArr,container){
LoopImages.call(this,imgArr,container);
};
SlideLoopImg.prototype=new LoopImages();
SlideLoopImg.prototype.changeImage=function(){
console.log('slide');
};

var FadeLoopImg=function(imgArr,container,arrow){
LoopImages.call(this,imgArr,container);
this.arrow=arrow;
};
FadeLoopImg.prototype=new LoopImages();
FadeLoopImg.prototype.changeImage=function(){
console.log('fade');
};

var fadeImg=new FadeLoopImg(['01.jpg','02.jpg'],'slide',['left.jpg','right.jpg']);
```

原型继承

```
 * 基于已经存在的模板对象克隆出新对象的模式
 * arguments[0], arguments[1], arguments[2]: 参数1，参数2，参数3 表示模板对象
 * 注意。这里对模板引用类型的属性实质上进行了浅复制（引用类型属性共享），当然根据需求可以自行深复制（引用类型属性复制）
 *****/
function prototypeExtend(){
var F=function(){},
args=arguments,
i=0;
len=args.length;
for(;i<len;i++){
for(var j in args[i]){
F.prototype[j]=args[i][j];
}
}
return new F();
}
```

（6）单例模式

单例模式 ：又名单体模式，是只允许实例化一次的对象类，可用于规划一个命名空间管理对象上的属性和方法。

e.g.实现一个小型代码库

```
var Demo={
Util:{
util_method1:function(){},
util_method2:function(){}
},
Tool:{
tool_method1:function(){},
tool_method2:function(){}
},
...
}
```

e.g.管理静态变量

```
var Conf=(function(){
var conf={
MAX:100,
MIN:0
};
return{
get:function(name){
return conf[name]?conf[name]:null;
}
};
})();
```

e.g.惰性单例

```
var LazySingle=(function(){
var _instance=null;

function Single(){
return{
publicMethod:function(){},
publicProperty:'1.0'
};
}

return function(){
if(!_instance){
_instance=Single();
}
return _instance;
};
})();

console.log(LazySingle().publicProperty);
```

二、结构型设计模式

（1）外观模式

外观模式：为一组复杂的子系统接口提供一个更高级的统一接口，通过这个接口使访问子系统接口更容易。

e.g.封装给元素添加事件

```
//外观模式
function addEvent(dom,type,fn){
if(dom.addEventListener){
dom.addEventListener(type,fn,false);
}else if(dom.attachEvent){
dom.attachEvent('on'+type,fn);
}else{
dom['on'+type]=fn;
}
}

addEvent(document.getElementById("aaa"),'click',function(){
console.log('click');
});
```

e.g.封装获取事件对象，获取元素，阻止默认行为

```
var getEvent=function(event){
return event || window.event;
};
var getTarget=function(event){
var event=getEvent(event);
return event.target||event.srcElement;
};
var preventDefault=function(event){
var event=getEvent(event);
if(event.preventDefault){
event.preventDefault();
}else{
event.returnValue=false;
}
};

document.onclick==function(e){
preventDefault(e);
if(getTarget(e) !== document.getElementById('aaa')){
//detail
}
}
```

e.g.代码库

```
var Test={
g:function(id){
```

```
return document.getElementById(id);
},
css:function(id,key,value){
document.getElementById(id).style[key]=value;
},
...
}
```

（2）适配器模式

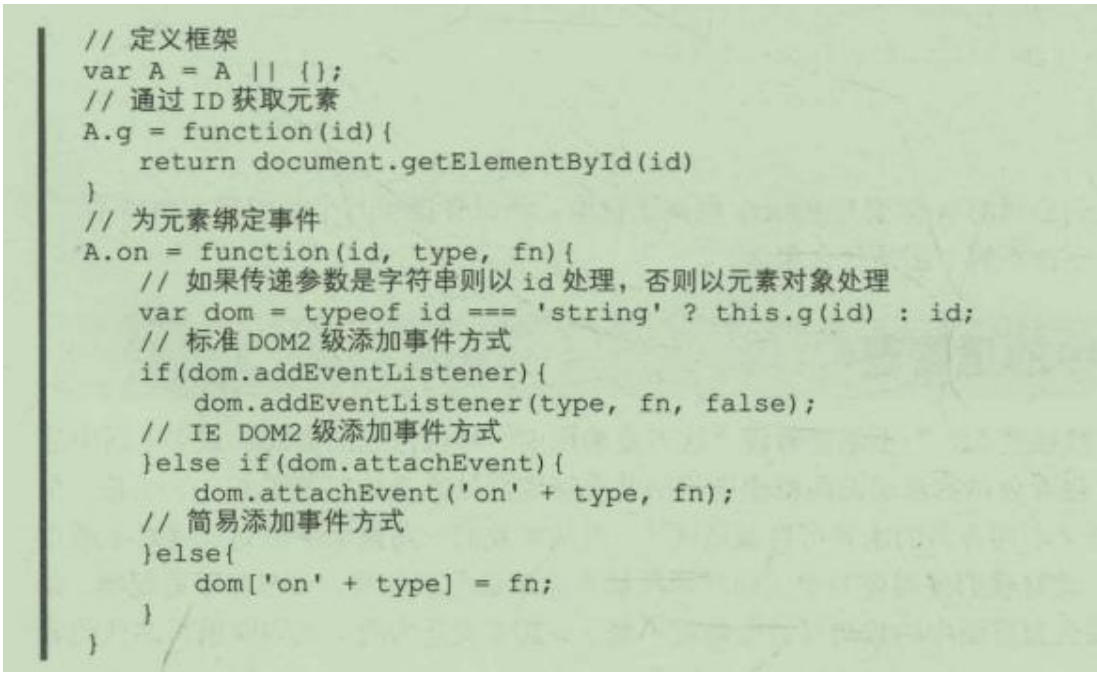
适配器模式：将一个类（对象）的接口（方法或者属性）转化成另一个接口，以满足用户需求，使类（对象）之间接口的不兼容问题通过适配器得以解决。

e.g.适配相近框架（如适配jquery和某A框架）

```
window.A=A=jQuery;
```

e.g.适配异类框架（如适配jquery和某A框架）

A框架如下图：



```
// 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){
    // 如果传递参数是字符串则以 id 处理，否则以元素对象处理
    var dom = typeof id === 'string' ? this.g(id) : id;
    // 标准 DOM2 级添加事件方式
    if(dom.addEventListener){
        dom.addEventListener(type, fn, false);
    // IE DOM2 级添加事件方式
    }else if(dom.attachEvent){
        dom.attachEvent('on' + type, fn);
    // 简易添加事件方式
    }else{
        dom['on' + type] = fn;
    }
}
```

```
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);
}
```

e.g.适配参数

```
function dosth(obj){
var _adapter={
name:'aaa',
title:'bbb'
};
for(var i in _adapter){
```

```
_adapter[i]=obj[i] || _adapter[i];
}
//…
}
```

e.g.适配数据

返回数据为数组，需要数据为对象

```
function arrToObjAdapter(arr){
return{
name:arr[0],
type:arr[1]
};
}

var adapterData=arrToObjAdapter(['a','b']);
```

e.g.适配服务器端数据

```
function ajaxAdapter(data){
return [data["key1"],data["key2"]];
}

$.ajax({
url:'test.php',
success:function(data,status){
if(data){
dosth(ajaxAdapter(data));
}
}
});
```

（3）代理模式

代理模式：由于一个队象不能直接引用另一个对象，所以需要通过代理对象在这两个对象之间起到中介作用。

e.g.站长统计

```
var Count=(function(){
var _img=new Image();
return function(param){
var str='http://www.count.com/a.gif?';
for(var i in param){
str+=i+ '=' +param[i];
}
_img.src=str;
};
})();
Count({num:10});
```

e.g.JSONP

```
//前端
<script type="text/JavaScript">
function jsonpCallBack(res,req){
```

```
console.log(res,req);
}
</script>
<script type="text/JavaScript" src="http://test/jsonp.php?
callback=jsonpCallBack&data=getJsonPData"></script>

//服务器端
<?php
$data=$_GET["data"];
$callback=$_GET["callback"];
echo $callback."('success','".$data."')";
?>
```

e.g.代理模板

<script>标签”。小铭接着说，“与之类似的还有另外一种方案是被称之为代理模板的方案，他

的解决思路是这样的，既然不同域之间相互调用对方的页面是有限制的，那么自己域中的两个页面相互之间的调用是可以的，即代理页面 B 调用被代理的页面 A 中对象的方式是可以的。那么要实现这种方式我们只需要在被访问的域中，请求返回的 Header 重定向到代理页面，并在代理页面中处理被代理的页面 A 就可以了。”

“既然这样，是不是我们在自己的域中要有这样 A、B 两个页面了？”小白问。

“是的。比如我们将自己的域称为 X 域，另外的域称为 Y 域，X 域中要有一个被代理页面，即 A 页面。在 A 页面中应该具备三个部分，第一个部分是发送请求的模块，如 form 表单提交，负责向 Y 域发送请求，并提供额外两组数据，其一是要执行的回调函数名称，其二是 X 域中代理模板所在的路径，并将 target 目标指向内嵌框架。第二个部分是一个内嵌框架，如 iframe，负责提供第一个部分中 form 表单的响应目标 target 的指向，并将嵌入 X 域中的代理页面作为子页面，即 B 页面。第三个部分是一个回调函数，负责处理返回的数据。”

```
//x域中被代理页面A
<script type="text/JavaScript">
function callback(data){
console.log(data);
}
</script>
<iframe name="proxyIframe" id="proxyIframe" src="proxy.html">
</iframe>
<form action="http://localhost/test.php" method="post" target="proxyIframe">
<input type="text" name="callback" value="callback">
<input type="text" name="proxy" value="http://localhost/proxy.html">
<input type="submit">
```

```
</form>

//x域中代理页面B
//proxy.html
<script type="text/JavaScript">
window.onload=function(){
if(top==self) return;
var arr=location.search.substr(1).split('&'),fn,args;
for(var i=0;i<arr.length;i++){
item=arr[i].split('=');
if(item[0]=='callback') fn=item[1];
else if(item[0]=='arg') arg=item[1];
}
}
try{
eval('top.'+fn+'("'+args+'")');
}catch(e){ }
</script>

//服务器端
<?php
$proxy=$_POST["proxy"];
$callback=$_POST["callback"];
header("Location:".$proxy."?callback=".$callback."&arg=success");
?>
```

（4）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（添加属性和方法）使原有对象可以满足用户的更复杂需求。

e.g.为输入框添加需求

```
var decorator=function(input,fn){
var input=document.getElementById(input);
if(typeof input.onclick==='function'){
var oldClick=input.onclick;
input.onclick=function(){
oldClick();
fn();
};
}else{
input.onclick=fn;
}
//...
};
```

（5）桥接模式

桥接模式：在系统沿着多个维度变化时，不增加复杂度就能解耦。

e.g.添加事件

```
function changeColor(dom,color,bg){
dom.style.color=color;
dom.style.background=bg;
}
document.getElementById('#test').onmouseover=function(){
changeColor(this,'red','#fff');
};
```

e.g.多元化对象

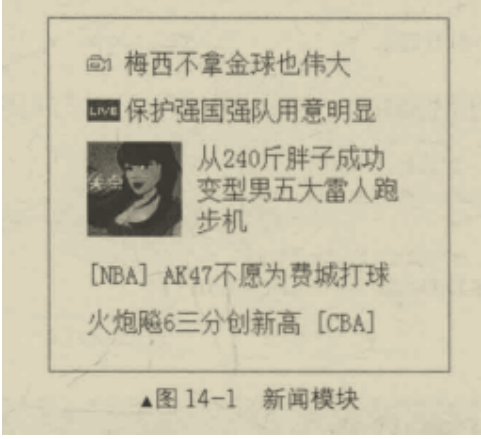
```
function Speed(x,y){
this.x=x;
this.y=y;
}
Speed.prototype.run=function(){
//detail
};
function Color(color){
this.color=color;
}
Color.prototype.draw=function(){
//detail
};


function People(x,y,c){
this.speed=new Speed(x,y);
this.color=new Color(c);
}
People.prototype.init=function(){
this.speed.run();
this.color.draw();
}
```

（6）组合模式

组合模式：又名部分-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。

e.g.实现新闻模块



▲图 14-1 新闻模块

```
//虚拟父类
 var News=function(){
this.children=[];
this.element=null;
};
News.prototype={
init:function(){
throw new Error("override");
},
add:function(){
throw new Error("override");
},
getElement:function(){
throw new Error("override");
}
};
//容器类
var Container=function(id,parent){
News.call(this);
this.id=id;
this.parent=parent;
this.init();
};
Container.prototype.init=function(){
this.element=document.createElement("ul");
this.element.id=this.id;
this.element.className='new-container';
};
Container.prototype.add=function(){
this.children.push(child);
this.element.appendChild(child.getElement());
return this;
}
Container.prototype.getElement=function(){
return this.element;
}
Container.prototype.show=function(){
this.parent.appendChild(this.element);
}
```

```
var Item=function(classname){
News.call(this);
this.classname=classname || '';
this.init();
};
Item.prototype.init=function(){
this.element=document.createElement("li");
this.element.className=this.classname;
};
Item.prototype.add=function(child){
this.children.push(child);
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();
};
NewsGroup.prototype.init=function(){
this.element=document.createElement("div");
this.element.className=this.classname;
};
NewsGroup.prototype.add=function(child){
this.children.push(child);
this.element.appendChild(child.getElement());
return this;
};
NewsGroup.prototype.getElement=function(){
return this.element;
};

var ImageNews=function(url,href,classname){
News.call(this);
this.url=url || '';
this.href= href || '#';
this.classname=classname || 'normal';
this.init();
};
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;
};

var IconNews=function(text,href,type){
News.call(this);
this.text=text || '';
this.href= href || '#';
this.type=type || 'video';
this.init();
};
IconNews.prototype.init=function(){
this.element=document.createElement("a");
this.element.innerHTML=this.text;
this.element.className='icon'+this.type;
this.element.href=this.href;
};
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();
};
EasyNews.prototype.init=function(){
this.element=document.createElement("a");
this.element.className='text';
this.element.href=this.href;
this.element.innerHTML=this.text;
};
EasyNews.prototype.add=function(){};
EasyNews.prototype.getElement=function(){
return this.element;
};

var news=new Container('news',document.body);
news.add(
new Item('normal').add(new IconNews('aaa','#','video')))
.add(new Item('normal').add(
new NewsGroup('has-img').add(
new ImageNews('img/1.jpg','#','small')
).add(
new EasyNews('ddd','#')
)
)).show();
```

（7）享元模式

享元模式：运用共享技术有效支持大量细粒度对象，避免对象间拥有相同内容造成多余开销。

e.g.优化分页

```
var Flyweight=function(){
var created=[];
function create(){
var dom=document.createElement('div');
document.getElementById('container').appendChild(dom);
created.push(dom);
return dom;
}

return{
getDiv:function(){
if(created.length<5){
return create();
}else{
var div=created.shift();
created.push(div);
return div;
}
}
};

var paper=0,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="";
}
}
};
```

e.g.享元动作

```
var Flyweight={
moveX:function(x){
this.x=x;
}
moveY:function(y){
```

```
this.y=y;
}
}

var Player=function(x,y,c){
this.x=x;
this.y=y;
this.color=c;
}
//继承移动
Player.prototype=Flyweight;

new Player(1,1,1).moveX(3);
```

三、行为型设计模式

（1）模板方法模式

模板方法模式：父类中定义一组算法骨架，将一些实现步骤延迟到子类中，使子类不改变父类的算法结构的同时可以重新定义算法中某些实现步骤。

e.g.提示框归一化

```
var Alert=function(data){
if(!data) return;
this.content=data.content;
this.panel=document.createElement("div");
this.contentNode=document.createElement("p");
this.confirmBtn=document.createElement("span");
this.closeBtn=document.createElement("b");
this.panel.className='alert';
this.closeBtn.className='a-close';
this.confirmBtn.className='a-confirm';
this.confirmBtn.innerHTML=data.confirm || '确认';
this.contentNode.innerHTML=this.content;
this.success=data.success || function(){};
this.fail=data.fail || function(){};
};
Alert.prototype={
init:function(){
this.panel.appendChild(this.closeBtn);
this.panel.appendChild(this.contentNode);
this.panel.appendChild(this.confirmBtn);
document.body.appendChild(this.panel);
this.bindEvent();
this.show();
},
bindEvent:function(){
var me=this;
this.closeBtn.onclick=function(){
me.fail();
me.hide();
};
```

```
this.confirmBtn.onclick=function(){
me.success();
me.hide();
}
},
hide:function(){
this.panel.style.display='none';
},
show:function(){
this.panel.style.display='block';
}
};

//右侧按钮提示框
var RightAlert=function(data){
Alert.call(this,data);
this.confirmBtn.className=this.confirmBtn.className+'right';
};
RightAlert.prototype=new Alert();

//标题提示框
var TitleAlert=function(){
Alert.call(this,data);
this.title=data.title;
this.titleNode=document.createElement('h3');
this.titleNode.innerHTML=this.title;
};
TitleAlert.prototype=new Alert();
TitleAlert.prototype.init=function(){
this.panel.insertBefore(this.titleNode,this.panel.firstChild);
Alert.prototype.init.call(this);
};

//带有取消按钮的弹出框
var CancelAlert=function(data){
TitleAlert.call(this,data);
this.cancel=data.cancel;
this.cancelBtn=document.createElement('span');
this.cancelBtn.className='cancel';
this.cancelBtn.innerHTML=this.cancel || '取消';
}
CancelAlert.prototype=new Alert();
CancelAlert.prototype.init=function(){
TitleAlert.prototype.init.call(this);
this.panel.appendChild(this.cancelBtn);
}
CancelAlert.prototype.bindEvent=function(){
var me=this;
TitleAlert.prototype.bindEvent.call(me);
this.cancelBtn.onclick=function(){
me.fail();
me.hide();
```

```
}
}

new CancelAlert({
title:'aaa',
content:'bbb',
success:function(){
//detail
},
fail:function(){
//detail
}
}).init();
```

e.g.创建多类导航

```
function formateString(str,data){
return str.replace(/\{#(\w+)#\}/g,function(match,key){
return typeof data[key]==="undefined" ? '' :data[key]
});
}
//基础导航
var Nav=function(data){
this.item='<a href="{#href#}" title="{#title#}">{#name#}</a>';
this.html='';
for(var i=0,len=data.length;i<len;i++){
this.html+=formateString(this.item,data[i]);
}
return this.html;
}
//带有消息提醒信息导航
var NumNav=function(){
var tpl='<b>{#num#}</b>';
for(var i=data.length-1;i>=0;i--){
data[i].name+=data[i].name+formateString(tpl,data[i]);
}
return Nav.call(this,data);
};

document.getElementById('test').innerHTML=new Num
Nav([
{
href:'http://www.baidu.com',
title:'baidu',
name:'baidu',
num:'10'
},
{
href:'http://www.baidu.com',
title:'baidu',
name:'baidu',
num:'10'
}
```

]);

（2）观察者模式

观察者模式：又名发布-订阅者模式或消息机制，定义了一种依赖关系，解决了主体对象与观察者之间功能的耦合。

e.g.实现新闻评论模块

```
//观察者对象
var Observer=(function(){
var _message={};
return {
//注册信息接口
regist:function(type,fn){
if(typeof _message[type] === 'undefined'){
_message[type]=[fn];
}else{
_message[type].push(fn);
}
return this;
},
//发布信息接口
fire:function(type,args){
if(!_message[type]) return;

var events={
type:type,
args:args || {}
},i=0,len=_message[type].length;

for(;i<len;i++){
_message[type][i].call(this,events);
}
},
//移除信息接口
remove:function(type,fn){
if(_message[type] instanceof Array){
var i=_message[type].length-1;
for(;i>=0;i--){
_message[type][i] === fn && _message[type].splice(i,1);
}
}
}
}
})();

//外观模式 简化获取元素
function $(id){
return document.getElementById(id);
}
//工程师A
```

```
(function(){
function addMsgItem(e){
var text=e.args.text,ul=
$("msg"),li=document.createElement("li"),span=document.createElement('span');
li.innerHTML=text;
span.onclick=function(){
ul.removeChild(li);
Observer.fire('remove',{num:-1});
};
li.appendChild(span);
ul.appendChild(li);
}
Observer.regist('add',addMsgItem);
})();
//工程师B
(function(){
function changeMsgNum(e){
var num=e.args.num;
$('msg_num').innerHTML=parseInt($('msg_num').innerHTML)+num;
}
Observer.regist('add',changeMsgNum).regist('remove',changeMsgNum);
})();
//工程师C
(function(){
$('user_submit').onclick=function(){
var text=$('user_input');
if(text.value==='') return;
Observer.fire('add',{
text:text.value,
num:1
});
text.value='';
}
})();
```

e.g.对象间解耦

```
var Student=function(result){
var that=this;
that.result=result;
that.say=function(){
console.log(that.result);
}
}
Student.prototype.answer=function(question){
Observer.regist(question,this.say);
};
Student.prototype.sleep=function(question){
Observer.remove(question,this.say);
}

var Teacher=function(){};
Teacher.prototype.ask=function(question){
```

```
Observer.fire(question);
}

var stua=new Student('aaa'),
stub=new Student('bbb');

stua.answer('111');
stub.answer('111');
stua.answer('222');
stub.answer('222');

stub.sleep('111');

var tea=new Teacher();

tea.ask('111');
tea.ask('222');

//结果：
//aaa
//bbb
//aaa
```

（3）状态模式

状态模式：当一个对象的内部状态发生改变时，会导致行为改变，看起来是改变了对象。

e.g.实现超级玛丽

```
//超级玛丽状态类
var MarryState=function(){
//私有变量
var _currentState={},
states={
jump:function(){
//detail
},
move:function(){
//detail
},
shoot:function(){
//detail
}
};
//动作控制类
var Action={
changeState:function(){
var arg=arguments;
_currentState={};
if(arg.length){
for(var i=0;i<arg.length;i++){
```

```
_currentState[arg[i]]=true;
}
}
return  this;
},
goes:function(){
for(var I in currentState){
states[i] && states[i]();
}
return this;
}
};

return {
change:Action.changeState,
goes:Action.goes
}
};

var marry=new MarryState();
marry.change('jump','shoot').goes().goes().change('shoot').goes();
```

（4）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，封装的算法具有一定独立性，不会随客户端变化而变化。

e.g.商品促销

```
var PriceStrategy=function(){
var stragtegy={
return30:function(price){
return +price + parseInt(price / 100) * 30;
},
return50:function(price){
//detail
},
percent90:function(price){
//detail
}
};
return function(algo,price){
return stragtegy[algo] && stragtegy[algo](price);
};
};

var price=PriceStrategy('return50','312');
```

e.g.缓冲函数Jquery的animate

e.g.表单验证

```
var InputStrategy=function(){
```

```
var strategy={
notNull:function(){
//detail
},
number:function(){
//detail
},
...
};

return {
check:function(type,val){
return strategy[type]?strategy[type](value):'none';
},
//添加策略
add:function(type,fn){
strategy[type]=fn;
}
}
}
```

（5）职责链模式

职责链模式：解决请求的发送者与接收者之间的耦合，通过职责链上的多个对象分解请求流程，实现请求在多个对象之间的传递，直到最后一个对象完成请求的处理。

e.g.表单模块

```
//请求模块
var sendData=function(data,dealType,dom){
var xhr=new XMLHttpRequest(),
url='test.php?mod=userinfo';
xhr.onload=function(event){
if((xhr.status>=200&&xhr.status<300) || xhr.status==304){
dealData(xhr.responseText,dealType,dom);
}else{
//fail
}
};

for(var I in data){
url+='&'+i+'='+data[i];
}
xhr.open('get',url,true);
xhr.send(null);
};

//响应数据适配模块
var dealData=function(data,dealType,dom){
var dataType=Object.prototype.toString.call(data);
```

```
switch(dealType){
case 'sug':
if(dataType==='[Object Array]'){
return createSug(data,dom);
};
if(dataType==='[Object object]'){
var newData=[];
for(var i in data){
newData.push(data[i]);
}
return createSug(newData,dom);
}
return createSug([data],dom);
break;

case 'validate':
return createValidataResult(data,dom);
break;
}
};
//创建组件模块
var createSug=function(data,dom){
var i=0,len=data.length,html='';
for(;i<len;i++){
html+='<li>'+data[i]+'</li>';
}
dom.parentNode.getElementsByTagName('ul')[0].innerHTML=html;
};

var createValidataResult=function(data,dom){
dom.parentNode.getElementsByTagName('span')[0].innerHTML=data;
};

var input=document.getElementByTagName('input');
input[0].onchange=function(e){
sendData({value:input[0].value},'validate',input[0]);
}
input[1].onkeydown=function(e){
sendData({value:input[1].value},'sug',input[1]);
}
```

（6）命令模式

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

e.g.自由化创建视图

```
var viewCommand=(function(){
```

```
var tpl={
product:['<div>','<img src="{#src#}"/>','<p>{#text#}</p>','</div>'].join(''),
title:['<div>','<img src="{#src#}"/>','<p>{#text#}</p>','</div>'].join('')
},
html='';

function formateString(str,obj){
return str.replace(/\{#(\w+)#\}/g,function(match,key){
return obj[key];
});
}
var Action={
create:function(data,view){
if(data.length){
for(var i=0,len=data.length;i<len;i++)
html+=formateString(tpl[view],data[i]);
}else{
html+=formateString(tpl[view],data);
}
},
display:function(container,data,view){
if(data){
this.create(data,view);
}
document.getElementById(container).innerHTML=html;
html='';
}
};

return function excute(msg){
msg.param=Object.prototype.toString.call(msg.param)==="[Object Array]"?
msg.param:[msg.param];
Action[msg.command].apply(Action,msg.param);
};
})();

var productData=[{src:'01.jpg',text:'01'},{src:'02.jpg',text:'02'}];
viewCommand({
command:'display',
param:['product',productData,'product']
});
```

e.g.解耦对象——绘图命令

（7）访问者模式

访问者模式：针对对象结构中的元素，定义在不改变对象的前提下访问结构中的元素的新方法。

e.g.访问操作元素

```
function bindIEEvent(dom,type,fn,data){
```

```
var data=data || {};
dom.attachEvent('on'+type,function(e){
fn.call(dom,e,data);
});
};
```

e.g.原生对象构造器 Object.prototype.toString.call(…)

（8）中介者模式

中介者模式：通过中介者对象封装一系列对象之间的交互，是对象之间不再相互引用，降低耦合。

e.g.设置导航

```
var Mediator=(function(){
var _msg={};
return {
register:function(type,action){
if(_msg[type])
 _msg[type].push(action);
else{
_msg[type]=[];
_msg[type].push(action);
}
},
send:function(type){
if(_msg[type]){
for(var i=0,len=_msg[type].length;i<len;i++){
_msg[type][i] && _msg[type][i]();
}
}
}
};
})();

var showHideNavWidget=function(mod,tag,showOrHide){
var mod=document.getElementById(mod),
tag=mod.getElementByTagName(tag),
showOrHide=(!showOrHide || showOrHide=='hide') ? 'hidden':'visible';
for(var i=tag.length-1;i>=0;i--){
tag.style.visibility=showOrHide;
}
};

(function(){
Mediator.register('hide',function(){
showHideNavWidget('test','b',false);
});
Mediator.register('show',function(){
showHideNavWidget('test','b',true);
});
```

```
var hideNum=document.getElementById('rrr');
hideNum.onchange=function(){
if(hideNum.checked){
Mediator.send('hide');
}else{
Mediator.send('show');
}
}
})();
```

（9）备忘录模式

备忘录模式：在不破坏对象的封装性的前提下，在对象之外捕获并保存该对象内部的状态以便日后对象使用或对象恢复到以前的某个状态。

e.g.新闻展示

```
var Page=(function(){
var cache={};
return function(page,fn){
if(cache[page]){
showPage(page,cache[page]);
fn && fn();
}else{
$.post('test.php',{
page:page
},function(res){
if(res.errNo==0){
showpage(page,res.data);
cache[page]=res.data;
fn && fn();
}else{
//fail
}//if
});//func
}//if
}//func
})();

$('#test').click(function(){
var page=$('#news').data('page');
Page(page,function(){
$('#news').data('page',page+1);
});
});
```

（10）迭代器模式

迭代器模式：在不暴露对象内部结构的同时，可以顺序地访问聚合对象内部的元素。

e.g.实现焦点图

```
var Iterator=function(items,container){
var container=container && document.getElementById(container) || document,
items=container.getElementsByTagName(items),
length=items.length,
index=0;

var splice=[].splice;

return{
first:function(){
index=0;
return items[index];
},
second:function(){
index=length-1;
return items[index];
},
pre:function(){
if(--index>0){
return items[index];
}else{
index=0;
return null;
}
},
next:function(){
if(++index<length){
return items[index];
}else{
index=length-1;
return null;
}
},
get:function(){
//负数逆向获取
index=num>=0 ? num % length : num % length+length;
return items[index];
},
dealEach:function(){
var args=splice.call(arguments,1);
for(var i=0;i<length;i++){
fn.apply(items[i],args);
}
},
dealItem:function(num,fn){
fn.apply(this.get(num),splice.call(arguments,2));
},
exclusive:function(num,allFn,numFn){
this.dealEach(allFn);
if(Object.prototype.toString.call(num) === '[Object Array]'){
for(var i=0,len=num.length;i<len;i++){
this.dealItem(num[i],numFn);
}
```

```
}else{
this.dealItem(num,numFn);
}
}
};
}
```

e.g.数组迭代器

```
var eachArray=function(arr,fn){
var i=0,len=arr.length;
for(;i<len;i++){
if(fn.call(arr[i],i,arr[i])===false){
break;
}
}
}
```

e.g.对象迭代器

```
var eachObject=function(obj,fn){
for(var i in obj){
if(fn.call(obj[i],i,obj[i])===false){
break;
}
}
}
```

e.g.同步变量迭代器

```
var Agetter=function(key){
if(!A) return undefined;
var result=A;
key=key.split('.');
for(var i=0,len=key.length;i<len;i++){
if(result[key[i]] !== undefined){
result=result[key[i]];
}else{
return undefined;
}
}
return result;
}
```

（11）解释器模式

解释器模式：对于一种语言，给出文法表示形式，定义一种解释器，通过使用解释器解释语言中定义的句子。

e.g.统计元素路径

```
var Interpreter=(function(){
function getSublingName(node){
if(node.previousSibling){
```

```
var name='',count=1,nodeName=node.nodeName,sibling=node.previousSibling;

while(sibling){
if(sibling.nodeType==1 && sibling.nodeType===node.nodeType &&
sibling.nodeName){
if(nodeName==sibling.nodeName){
name+=++count;
}else{
count=1;
name+='|'+sibling.nodeName.toUpperCase();
}
}
sibling=sibling.previousSibling;
}
return name;
}else{
return '';
}
}

return function(node,wrap){
var path=[],
wrap=wrap || document;
if(node===wrap){
if(wrap.nodeType==1){
path.push(wrap.nodeName.toUpperCase());
}
return path;
}

if(node.parentNode !== wrap){
path=arguments.callee(node.parentNode,wrap);
}else{
if(wrap.nodeType==1){
path.push(wrap.nodeName.toUpperCase());
}
}

var sublingsNames=getSublingName(node);
if(node.nodeType==1){
path.push(node.nodeName.toUpperCase()+sublingsNames);
}
return path;
}
})();

console.log(Interpreter(…))
```

五、技巧型设计模式

（1）链模式

链模式：通过在对象方法中将当前对象返回，实现对同一个对象多个方法的链式调用。

e.g.获取元素

```
var A=function(selector,context){
return new A.fn.init(selector,contex);
}

A.fn=A.prototype={
constructor:A,
init:function(selector,context){
//detail
return this;
},
……,
//增强数组
push:[].push,
sort:[].sort,
splice:[].splice
};
A.fn.init.prototype=A.fn;

//对象拓展
A.extend=A.fn.extend=function(){
var i=1,len=argument.length,target=argument[0],j;
if(i==len){
target=this;
i--;
}
for(;i<len;i++){
for(j in arguments[i]){
target[j]=arguments[i][j]
}
}
return target;
}
```

（2）委托模式

委托模式：多个对象接收并处理同一请求，将请求委托给另一个对象统一处理请求。

e.g.点击日历交互

```
ul.onclick=function(e){
var e=e||window.event,target=e.target||e.srcElement;
if(target.nodeName.toLowerCase()==='li'){
target.style.color="red";
}
```

```
};

e.g.数据分发
var deal={
banner:function(res){},
aside:function(res){},
...
}

$.get('deal.php?',function(res){
for(var i in res){
deal[i] && deal[i](res[i]);
}
});
```

（3）数据访问对象模式

数据访问对象模式：抽象和封装对数据源的访问和存储，DAO通过对数据源链接的管理方便访问与存储数据。

```
e.g.实现本地存储类
var BaseLocalStorage=function(preId,timeSign){
this.preId=preId;//本地存储数据库前缀
this.timeSign=timeSign || '|-|';//时间戳与存储数据之间的拼接符
}

BaseLocalStorage.prototype={
status:{
SUCCESS:0,
FAILURE:1,
OVERFLOW:2,//溢出
TIMEOUT:3//过期
},
storage:localStorage || window.localStorage,
getKey:function(key){
return this.preId + key;
},
set:function(key,value,callback,time){
//detail
},
get:function(key,callback){
//detail
},
remove:function(key,callback){
//detail
}
}
```

（4）节流模式

节流模式：对重复的业务逻辑进行节流控制，执行最后一次操作并取消其他操作来提高性能。

e.g.实现节流器

```
var throttle=function(){
var isClear=arguments[0],fn;//获取第一个参数
//第一个参数是boolean类型表示清除定时器
if(typeof isClear === 'boolean'){
fn = arguments[1];
fn._throttleId && clearTimeout(fn._throttleId);
}else{
//通过计时器延迟函数执行，第一个参数为函数
fn = isClear;
//第二个参数为函数执行时的参数
param =arguments[1];

var p=extend({
context:null,//执行时的作用域
args:[],//相关参数
time:300// 延迟执行的时间
},param);
//清除执行函数定时器
arguments.callee(true,fn);
//重新绑定定时器，延迟执行函数
fn._throttleId=setTimeout(function(){
fn.apply(p.context,p.args);
},p.time);

}
}
```

e.g.解决返回顶部不断执行动画的问题

```
// 首先引入 jquery.js 与 easing.js 方便返回顶部动画实现
// 返回顶部按钮动画
function moveScroll(){
    var top = $(document).scrollTop();
    $('#back').animate({top : top + 300}, 400, 'easeOutCubic')
}
// 监听页面滚动条事件
$(window).on('scroll', function(){
    // 节流执行返回顶部按钮动画
    throttle(moveScroll);
})
```

e.g.优化浮层

```
<div id="icon" class="icon">
    <ul  class="icon">
        <li class="weixin"></li>
        <li class="weibo"></li>
    </ul>
    <div class="">
        <img class="show" src="img/weixin.png" alt="">
        <img src="img/weibo.png" alt="">
        <span class="arrow"><em></em></span>
    </div>
</div>
```

```
// 外观模式封装获取元素方法
function $(id){return document.getElementById(id)}
function $tag(tag, container){
    container = container || document;
    return container.getElementsByTagName(tag)
}
// 浮层类
var Layer = function(id){
    // 获取容器
    this.container = $(id);
    // 获取容器中的浮层容器
    this.layer = $tag('div', this.container)[0];
    // 获取 icon 容器
    this.lis = $tag('li', this.container);
    // 获取二维码图片
    this.imgs = $tag('img', this.container);
    // 绑定事件
    this.bindEvent();
}
Layer.prototype = {
    // 绑定交互事件
    bindEvent : function(){
        // ......
    },
    // 事件绑定方法
    on : function(ele, type, fn){
        ele.addEventListener ? ele.addEventListener(type, fn, false) :
ele.attachEvent('on' + type, fn);
        return this;
    }
}
```

```
// 绑定交互事件
bindEvent : function(){
    // 缓存当前对象
    var that = this;
    // 隐藏浮层
    function hideLayer(){
        that.layer.className = '';
    }
    // 显示浮层
    function showLayer(){
        that.layer.className = 'show';
    }
    // 鼠标光标移入事件
    that.on(that.container, 'mouseenter', function(){
        // 清除隐藏浮层方法计时器
        throttle(true, hideLayer);
        // 延迟显示浮层方法
        throttle(showLayer);
    // 鼠标光标移出事件
    }).on(that.container, 'mouseleave', function(){
        // 延迟浮层隐藏方法
        throttle(hideLayer);
        // 清除显示浮层方法计时器
        throttle(true, showLayer);
    });
    // 遍历 icon 绑定事件
    for(var i = 0; i < that.lis.length; i++){
        // 自定义属性 index
        that.lis[i].index = i;
        // 为每一个 li 元素绑定鼠标移入事件
        that.on(that.lis[i], 'mouseenter', function(){
            // 获取自定义属性 index
            var index = this.index;
            // 排除所有 img 的 show 类
            for(var i = 0; i < that.imgs.length; i++){
                that.imgs[i].className = '';
            }
            // 为目标图片设置 show 类
            that.imgs[index].className = 'show';
            // 从新定义浮层位置
            that.layer.style.left = -22 + 60 * index + 'px';
        })
    }

}
```

e.g.图片延迟加载

```
/***
 * 节流延迟加载图片类
 * param id   延迟加载图片的容器 id
 * 注：图片格式如下 <img src="img/loading.gif" alt="" data-src="img/1.jpg">
 ****/
function LazyLoad(id){
    // 获取需要节流延迟加载图片的容器
```

```
    this.container = document.getElementById(id);
    // 缓存图片
    this.imgs = this.getImgs();
    // 执行逻辑
    this.init();
}
// 节流延迟加载图片类原型方法
LazyLoad.prototype = {
    // 起始执行逻辑
    init : function(){},
    // 获取延迟加载图片
    getImgs : function(){},
    // 加载图片
    update : function(){},
    // 判断图片是否在可视范围内
    shouldShow : function(i){},
    // 获取元素页面中的纵坐标位置
    pageY : function(element){},
    // 绑定事件（简化版）
    on : function(element, type, fn){},
    // 为窗口绑定 resize 事件与 scroll 事件
    bindEvent : function(){}
}
```

```
// 起始执行逻辑
init : function(){
    // 加载当前视图图片
    this.update();
    // 绑定事件
    this.bindEvent();
}
```

```
// 获取延迟加载图片
getImgs : function(){
    // 新数组容器
    var arr = [];
    // 获取图片
    var imgs = this.container.getElementsByTagName('img');
     // 将获取的图片转化为数组(IE 下通过 Array.prototype.slice 会报错)
    for(var i = 0, len = imgs.length; i < len; i++){
        arr.push(imgs[i])
    }
    return arr;
}
```

```
// 加载图片
update : function(){
    // 如果图片都加载完成，返回
    if(!this.imgs.length){
        return;
    }
    // 获取图片长度
    var i = this.imgs.length;
    // 遍历图片
    for(--i; i >= 0; i--){
        // 如果图片在可视范围内
        if(this.shouldShow(i)){
            // 加载图片
            this.imgs[i].src = this.imgs[i].getAttribute('data-src');
            // 清除缓存中的此图片
            this.imgs.splice(i, 1);
        }
    }
}
```

```
// 判断图片是否在可视范围内
shouldShow : function(i){
        // 获取当前图片
    var img = this.imgs[i],
        // 可视范围内顶部高度(页面滚动条 top 值)
        scrollTop        =        document.documentElement.scrollTop        ||
document.body.scrollTop,
        // 可视范围内底部高度
        scrollBottom = scrollTop + document.documentElement.clientHeight;
        // 图片的顶部位置
        imgTop = this.pageY(img),
        // 图片的底部位置
        imgBottom = imgTop + img.offsetHeight;
    // 判断图片是否在可视范围内：图片底部高度大于可视视图顶部高度并且图片底部高度小于可视视
    // 图底部高度，或者图片顶部高度大于可视视图顶部高度并且图片顶部高度小于可视视图底部高度
    if(imgBottom > scrollTop && imgBottom < scrollBottom || (imgTop > scrollTop
&& imgTop < scrollBottom))
        return true;

    // 不满足上面条件则返回 false
    return false;

}

// 获取元素页面中的纵坐标位置
pageY : function(element){
    // 如果元素有父元素
    if(element.offsetParent){
        // 返回元素+父元素高度
        return element.offsetTop + this.pageY(element.offsetParent);
    }else{
        //否则返回元素高度
        return element.offsetTop;
    }
}
// 绑定事件（简化版）
on : function(element, type, fn){
    if(element.addEventListener){
        addEventListener(type, fn, false);
    }else{
        element.attachEvent('on' + type, fn, false);
    }
}
// 为窗口绑定 resize 事件与 scroll 事件
bindEvent : function(){
    var that = this;
    this.on(window, 'resize', function(){
        // 节流处理更新图片逻辑
        throttle(that.update, {context : that});
    });
    this.on(window, 'scroll', function(){
        // 节流处理更新图片逻辑
        throttle(that.update, {context : that});
    })
}

// 延迟加载 container 容器内的图片
new LazyLoad('container');
```

e.g.统计打包

```
// 打包统计对象
var LogPack = function(){
    var data = [],                  // 请求缓存数组
        MaxNum = 10,                // 请求缓存最大值
        itemSplitStr = '|',         // 统计项统计参数间隔符
        keyValueSplitStr = '*',     // 统计项统计参数键值对间隔符
        img = new Image();          // 请求触发器，通过图片 src 属性实现简单的 get 请求
    // 发送请求方法
    function sendLog(){}
    // 统计方法
    return function(param){
        // 如果无参数则发送统计
        if(!param){
            sendLog();
            return;
        }
        // 添加统计项
        data.push(param);
        // 如果统计项大于请求缓存最大值则发送统计请求包
        data.length >= MaxNum && sendLog();
    }
}();
// 发送请求方法
function sendLog(){
        // 请求参数
    var logStr = '';
        // 截取 MaxNum 个统计项发送
        fireData = data.splice(0, MaxNum);
    // 遍历统计项
    for(var i = 0, len = fireData.length; i < len; i++){
        // 添加统计项顺序索引
        logStr += 'log' + i + '=';
        // 遍历统计项内的统计参数
        for(var j in fireData[i]){
            // 添加统计项参数键 + 间隔符 + 值
            logStr += j + keyValueSplitStr + fireData[i][j];
            // 添加统计项统计参数间隔符
            logStr += itemSplitStr;
        }
        // &符拼接多个统计项
        logStr = logStr.replace(/\|$/, '') + '&';
    }
    // 添加统计项打包长度
    logStr += 'logLength=' + len;
    // 请求触发器发送统计
    img.src = 'a.gif?' + logStr;
}
// 点击统计
btn.onclick = function(){
    LogPack({
        btnId : this.id,
        context : this.innerHTML,
        type : 'click'
    });
};
// 点击统计
btn.onmouseover = function(){
    LogPack({
        btnId : this.id,
        context : this.innerHTML,
        type : 'mouseover'
    });
};
```

（5）简单模板模式

简单模板模式：通过格式化字符串拼凑视图优化内存开销。

e.g.实现模板生成器

```
var A=A || {};
A.formateString=function(str,data){
return str.replace(/\{#(\w+)#\}/g,function(match,key){return typeof data[key] ===
undefined ? '' :data[key]});
}
A.view=function(name){
//模板库
var v={
code:'<pre><code>{#code#}</code></pre>',
…
};
if(Object.prototype.toString.call(name) === "[object Array]"){
var tpl='';
for(var i=0;i<name.length;i++){
tpl +=arguments.callee(name[i]);
}
return tpl;
}else{
return v[name] ? v[name] : ('<'+name+'>{#'+name+'#}</'+name+'>');
}
}

A.strategy={
'A':function(data){
//…
tpl=A.view(['h2','p','ul']),
liTpl=A.formateString(A.view('li'),{li:A.view(['strong','span'])}),
…
},
'B':…
}
```

（6）惰性模式

惰性模式：减少每次代码执行时的重复性分支判断，通过重定义屏蔽分支判断。

e.g.优化事件

初始代码：

```
// 单体模式定义命名空间
var A = {};
// 添加绑定事件方法 on
A.on = function(dom, type, fn){
    if(dom.addEventListener){
        dom.addEventListener(type, fn, false);
    }else if(dom.attachEvent){
        dom.attachEvent('on' + type, fn);
    }else{
        dom['on' + type] = fn;
    }
}
```

第一种优化：

```
// 添加绑定事件方法 on
A.on = function(dom, type, fn){
    // 如果支持 addEventListener 方法
    if(document.addEventListener){
        // 返回新定义方法
        return function(dom, type, fn){
            dom.addEventListener(type, fn, false);
        }
    // 如果支持 attachEvent 方法（IE）
    }else if(document.attachEvent){
        // 返回新定义方法
        return function(dom, type, fn){
            dom.attachEvent('on' + type, fn);
        }
    // 定义 on 方法
    }else{
        // 返回新定义方法
        return function(dom, type, fn){
            dom['on' + type] = fn;
        }
    }
}();
```

第二种优化：

```
// 添加绑定事件方法 on
A.on = function(dom, type, fn){
    // 如果支持 addEventListener 方法
    if(dom.addEventListener){
        // 显示重定义 on 方法
        A.on = function(dom, type, fn){
            dom.addEventListener(type, fn, false);
        }
    // 如果支持 attachEvent 方法（IE）
    }else if(dom.attachEvent){
        // 显示重定义 on 方法
        A.on = function(dom, type, fn){
            dom.attachEvent('on' + type, fn);
        }
    // 如果支持 DOM0 级事件绑定
    }else{
        // 显示重定义 on 方法
        A.on = function(dom, type, fn){
            dom['on' + type] = fn;
        }
    }
    // 执行重定义 on 方法
    A.on(dom, type, fn);
};
```

（7）参与者模式

参与者模式：在特定的作用域中执行给定的函数，并将参数原封不动地传递。

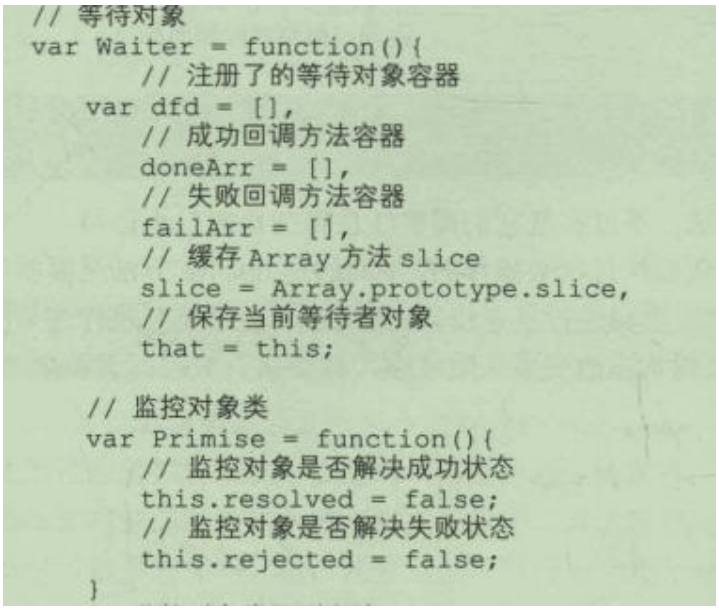
e.g.实现bind函数

```
function bind(fn,context){
var Slice=[].slice,
args=Slice.call(arguments,2);
return function(){
var addArgs=Slice.call(arguments),
allArgs=addArgs.concat(args);
return fn.apply(context,allArgs);
}
}
```

（8）等待者模式

等待者模式：通过对多个异步进程的监听，触发未来发生的动作。

e.g.实现等待者对象



```
// 等待对象
var Waiter = function(){
        // 注册了的等待对象容器
    var dfd = [],
        // 成功回调方法容器
        doneArr = [],
        // 失败回调方法容器
        failArr = [],
        // 缓存 Array 方法 slice
        slice = Array.prototype.slice,
        // 保存当前等待者对象
        that = this;

    // 监控对象类
    var Primise = function(){
        // 监控对象是否解决成功状态
        this.resolved = false;
        // 监控对象是否解决失败状态
        this.rejected = false;
    }
```

```
// 监控对象类原型方法
Primise.prototype = {
    // 解决成功
    resolve : function(){ },
    // 解决失败
    reject : function(){}
}

// 创建监控对象
that.Deferred = function(){
    return new Primise();
}

// 回调执行方法
function _exec(arr){}

// 监控异步方法 参数：监控对象
that.when = function(){};

// 解决成功回调函数添加方法
that.done = function(){};

// 解决失败回调函数添加方法
that.fail = function(){}
```

```
// 监控对象原型方法
Primise.prototype = {
    // 解决成功
    resolve : function(){
        // 设置当前监控对象解决成功
        this.resolved = true;
        // 如果没有监控对象则取消执行
        if(!dfd.length)
            return;
        // 遍历所有注册了的监控对象
        for(var i = dfd.length - 1; i >= 0; i--){
            // 如果有任意一个监控对象没有被解决或者解决失败则返回
            if(dfd[i] && !dfd[i].resolved || dfd[i].rejected){
                return;
            }
            // 清除监控对象
            dfd.splice(i,1);
        }
        // 执行解决成功回调方法
        _exec(doneArr);
    },
    // 解决失败
    reject : function(){
        // 设置当前监控对象解决失败
        this.rejected = true;
        // 如果没有监控对象则取消执行
        if(!dfd.length)
            return;
        // 清除所有监控对象
        dfd.splice(0);
        // 执行解决成功回调方法
        _exec(failArr);
    }
```

```
// 回调执行方法
function _exec(arr){
    var i = 0,
        len = arr.length;
    // 遍历回调数组执行回调
    for(; i < len; i++){
        try{
            // 执行回调函数
            arr[i] && arr[i]();
        }catch(e){}
    }
}
```

```
// 监控异步方法 参数：监控对象
that.when = function(){
    // 设置监控对象
    dfd = slice.call(arguments);
    // 获取监控对象数组长度
    var i = dfd.length;
    // 向前遍历监控对象，最后一个监控对象的索引值为 length-1
    for(--i; i >= 0; i--){
        // 如果不存在监控对象，或者监控对象已经解决，或者不是监控对象
        if(!dfd[i] || dfd[i].resolved || dfd[i].rejected || !dfd[i] instanceof
        Promise){
            // 清理内存 清除当前监控对象
            dfd.splice(i,1);
        }
    }
    // 返回等待者对象
    return that;
};
```

```
// 解决成功回调函数添加方法
that.done = function(){
    // 向成功回调函数容器中添加回调方法
```

```
    doneArr = doneArr.concat(slice.call(arguments));
    // 返回等待者对象
    return that;
};
// 解决失败回调函数添加方法
that.fail = function(){
    // 向失败回调函数容器中添加回调方法
    failArr = failArr.concat(slice.call(arguments));
    // 返回等待者对象
    return that;
}
```

e.g.封装异步请求

```
// 封装 get 请求
var ajaxGet = function(url, success, fail){
    var xhr = new XMLHttpRequest();
    // 创建检测对象
    var dtd = waiter.Deferred();
    xhr.onload = function(event){
```

```
        // 请求成功
        if((xhr.status >= 200 && xhr.status < 300) || xhr.status == 304){
            success && success()
            dtd.resolve();
        // 请求失败
        }else{
            dtd.reject();
            fail && fail();
        }
    };
    xhr.open("get", url, true);
    xhr.send(null);
};
```

六、架构型设计模式

（1）同步模块模式

同步模块模式：请求发出后，无论模块是否存在，立即执行后续逻辑，实现模块开发中对模块的立即引用。

e.g.实现模块管理器

```
// 定义模块管理器单体对象
var F = F || {};
/***
 * 定义模块方法（理论上，模块方法应放在闭包中实现，可以隐蔽内部信息，这里我们为让读者能够看明白，我们忽略此步骤）
 * @param str 模块路由
 * @param fn  模块方法
 **/
F.define = function(str, fn){
    // 解析模块路由
    var parts = str.split('.'),
        // old 当前模块的祖父模块，parent 当前模块父模块
        // 如果在闭包中，为了屏蔽对模块直接访问，建议将模块添加给闭包内部私有变量
        old = parent = this,
        // i 模块层级，len 模块层级长度
        i = len = 0;
    // 如果第一个模式是模块管理器单体对象，则移除
    if(parts[0] === 'F'){
        parts = parts.slice(1);
    }
    // 屏蔽对 define 与 module 模块方法的重写
    if(parts[0] === 'define' || parts[0] === 'module'){
        return;
    }
    // 遍历路由模块并定义每层模块
    for(len = parts.length; i < len; i++){
        // 如果父模块中不存在当前模块
        if(typeof parent[parts[i]] === 'undefined'){
            // 声明当前模块
            parent[parts[i]] = {};
        }
        // 缓存下一层级的祖父模块
        old = parent;
        // 缓存下一层级父模块
        parent = parent[parts[i]];
    }
    // 如果给定模块方法则定义该模块方法
    if(fn){
        // 此时 i 等于 parts.length，故减一
        old[parts[--i]] = fn();
    }
    // 返回模块管理器单体对象
    return this;
}
```

```
// 模块调用方法
F.module = function(){
        // 将参数转化为数组
    var args = [].slice.call(arguments),
        // 获取回调执行函数
        fn = args.pop(),
        // 获取依赖模块，如果args[0]是数组,则依赖模块为args[0]。否则依赖模块为arg
        parts = args[0] && args[0] instanceof Array ? args[0] : args,
        // 依赖模块列表
        modules = [],
        // 模块路由
        modIDs = '',
        // 依赖模块索引
        i = 0,
        // 依赖模块长度
        ilen = parts.length,
        // 父模块，模块路由层级索引，模块路由层级长度
        parent, j, jlen;
    // 遍历依赖模块
    while(i < ilen){
        // 如果是模块路由
        if(typeof parts[i] === 'string'){
            // 设置当前模块父对象（F）
            parent = this;
            // 解析模块路由，并屏蔽掉模块父对象
            modIDs = parts[i].replace(/^F\./, '').split('.');
            // 遍历模块路由层级
            for(j = 0, jlen = modIDs.length; j < jlen; j++){
                // 重置父模块
                parent = parent[modIDs[j]] || false;
            }
            // 将模块添加到依赖模块列表中
            modules.push(parent);
        // 如果是模块对象
        }else{
            // 直接加入依赖模块列表中
            modules.push(parts[i]);
        }
        // 取下一个依赖模块
        i++;

    }
    // 执行回调执行函数
    fn.apply(null, modules);
}
```

```
// F.string 模块
F.define('string', function(){
    // 接口方法
    return {
        // 清楚字符串两边空白
        trim : function(str){
            return str.replace(/^\s+|\s+$/g, '');
        }
    }
});
```

```
F.define('dom', function(){
    // 简化获取元素方法(重复获取可被替代，此设计用于演示模块添加)
    var $ = function(id){
        $.dom = document.getElementById(id);
        // 返回构造函数对象
        return $;
    }
    // 获取或者设置元素内容
    $.html = function(html){
        // 如果传参则设置元素内容，否则获取元素内容
        if(html){
            this.dom.innerHTML = html;
            return this;
        }else{
            return this.dom.innerHTML;
        }
    }
    // 返回构造函数
    return $;
});
// 引用 dom 模块与 document 对象（注意，依赖模块对象通常为已创建的模块对象）
F.module(['dom', document], function(dom, doc){
    // 通过 dom 模块设置元素内容
    dom('test').html('new add!');
    // 通过 document 设置 body 元素背景色
    doc.body.style.background = 'red';
});
// 依赖引用 dom 模块，string.trim 方法
F.module('dom', 'string.trim', function(dom, trim){
    // 测试元素 <div id="test">  test  </div>
    var html = dom('test').html();     // 获取元素内容
    var str = trim(html);              // 去除字符串两边空白符
    console.log("*" + html + "*", "*" + str + "*"); // *  test  * *test*
});
```

（2）异步模块模式

异步模块模式：请求发出后，继续其他业务逻辑，知道模块加载完成执行后续逻辑，实现模块开发中对模块加载完成后的引用。

e.g.实现异步模块模式

```
// 向闭包中传入模块管理器对象 F（~屏蔽压缩文件时，前面漏写;报错）
~(function(F){
    // 模块缓存器。存储已创建模块
    var moduleCache = {}
})((function(){
    // 创建模块管理器对象 F，并保存在全局作用域中
    return window.F = {};
})());
```

```
/**
 * 创建或调用模块方法
 * @param      url            参数为模块 url
 * @param      deps           参数为依赖模块
 * @param      callback       参数为模块主函数
 **/
F.module = function(url, modDeps, modCallback){
        // 将参数转化为数组
    var args = [].slice.call(arguments),
        // 获取模块构造函数（参数数组中最后一个参数成员）
        callback = args.pop(),
        // 获取依赖模块（紧邻回调函数参数，且数据类型为数组）
        deps = (args.length && args[args.length - 1] instanceof Array) ? args.
pop() : [],
        // 该模块 url（模块 ID）
        url = args.length ? args.pop() : null,
        // 依赖模块序列
        params = [],
        // 未加载的依赖模块数量统计
        depsCount = 0,
        // 依赖模块序列中索引值
        i = 0,
        // 依赖模块序列长度
        len;
  // 获取依赖模块长度
  if(len = deps.length){
      // 遍历依赖模块
      while(i < len){
          // 闭包保存 i
          (function(i){
              // 增加未加载依赖模块数量统计
              depsCount++;
              // 异步加载依赖模块
              loadModule(deps[i], function(mod){
                  // 依赖模块序列中添加依赖模块接口引用
                  params[i] = mod;
                  // 依赖模块加载完成，依赖模块数量统计减一
                  depsCount--;
                  // 如果依赖模块全部加载
                  if(depsCount === 0){
                      // 在模块缓存器中矫正该模块，并执行构造函数
                      setModule(url, params, callback);
                  }

              });


          })(i);
          // 遍历下一依赖模块
          i++
      }
    // 无依赖模块，直接执行回调函数
    }else{
        // 在模块缓存器中矫正该模块，并执行构造函数
        setModule(url, [], callback);
    }
}
```

```
var moduleCache = {},
    setModule = function(moduleName, params, callback){},
    /***
     * 异步加载依赖模块所在文件
     * @param moduleName      模块路径（id）
     * @param callback        模块加载完成回调函数
     **/
    loadModule = function(moduleName, callback){
        // 依赖模块
        var _module;
        // 如果依赖模块被要求加载过
        if(moduleCache[moduleName]){
            // 获取该模块信息
            _module = moduleCache[moduleName];
            // 如果模块加载完成
            if(_module.status === 'loaded'){
                // 执行模块加载完成回调函数
                setTimeout(callback(_module.exports), 0);
            }else{
                // 缓存该模块所处文件加载完成回调函数
                _module.onload.push(callback);
            }
        // 模块第一次被依赖引用
        }else{
            // 缓存该模块初始化信息
            moduleCache[moduleName] = {
                moduleName : moduleName,        // 模块 Id

                status : 'loading',             // 模块对应文件加载状态（默认加载中）
                exports : null,                 // 模块接口
                onload : [callback]             // 模块对应文件加载完成回调函数缓冲器
            };
            // 加载模块对应文件
            loadScript(getUrl(moduleName));
        }

    },
    getUrl = function(){},
    loadScript = function(){};
```

```
// 获取文件路径
getUrl = function(moduleName){
    // 拼接完整的文件路径字符串，如'lib/ajax' => 'lib/ajax.js'
    return String(moduleName).replace(/\.js$/g, '') + '.js';
},
// 加载脚本文件
loadScript = function(src){
    // 创建 script 元素
    var _script = document.createElement('script');
    _script.type = 'text/JavaScript';       // 文件类型
    _script.charset = 'UTF-8';              // 确认编码
    _script.async = true;                   // 异步加载
    _script.src = src;                      // 文件路径
    document.getElementsByTagName('head')[0].appendChild(_script);  // 插入页面中
};
```

```
/***
 * 设置模块并执行模块构造函数
 * @param moduleName          模块 id 名称
 * @param params              依赖模块
 * @param callback            模块构造函数
 **/
setModule = function(moduleName, params, callback){
    // 模块容器，模块文件加载完成回调函数
    var _module, fn;
    // 如果模块被调用过
    if(moduleCache[moduleName]){
        // 获取模块
        _module = moduleCache[moduleName];
        // 设置模块已经加载完成
        _module.status = 'loaded';
        // 矫正模块接口
        _module.exports = callback ? callback.apply(_module, params) : null;
        // 执行模块文件加载完成回调函数
        while(fn = _module.onload.shift()){
            fn(_module.exports);
        }
    }else{
        // 模块不存在（匿名模块），则直接执行构造函数
        callback && callback.apply(null, params);
    }
}
F.module('lib/dom', function(){
    return {
        // 获取元素方法
        g : function(id){
            return document.getElementById(id);
        },
        // 获取或者设置元素内容方法
        html : function(id, html){
            if(html)
                this.g(id).innerHTML = html;
            else
                return this.g(id).innerHTML;
        }
    }
});

F.module('lib/event', ['lib/dom'], function(dom){
    var events = {
        // 绑定事件
        on : function(id, type, fn){
            dom.g(id)['on' + type] = fn;
        }
    }
    return events;
});
//index.html 页面中
F.module(['lib/event', 'lib/dom'], function(events, dom){
    events.on('demo', 'click', function(){
        dom.html('demo', 'success');
    })
});
```

（3）Widget模式

Widget模式：借用web组件化思想将页面分解成部件，针对部件开发，组合成完整的页面。

e.g.实现组件化模式

```
// 模板引擎模块
F.module('lib/template', function(){
        // 模板引擎 处理数据与编译模板入口
    var _TplEngine = function(){},
        // 获取模板
        _getTpl = function(){},
        // 处理模板
        _dealTpl = function(){},
        // 编译执行
        _compileTpl = function(){};
    return _TplEngine;
});
/***
 * 模板引擎 处理数据与编译模板入口
 * @param str 模板容器 id 或者模板字符串
 * @param data      渲染数据
 **/
_TplEngine = function(str, data){
    // 如果数据是数组
    if(data instanceof Array){
            // 缓存渲染模板结果
        var html = '',
            // 数据索引
            i = 0,
            // 数据长度
            len = data.length;
        // 遍历数据
        for(; i < len; i++){
            // 缓存模板渲染结果，也可写成 html += arguments.callee(str, data[i]);
            html += _getTpl(str)(data[i]);
        }
        // 返回模板渲染最终结果
        return html;
    }else{
        // 返回模板渲染结果
        return _getTpl(str)(data);
    }
}

/***
 * 获取模板
 * @param str 模板容器 id，或者模板字符串
 **/
_getTpl = function(str){
    // 获取元素
    var ele = document.getElementById(str);
    // 如果元素存在
    if(ele){
        // 如果是 input 或者 textarea 表单元素，则获取该元素的 value 值，否则获取元素的内容
        var html = /^(textarea|input)$/i.test(ele.nodeName) ? ele.value :
ele.innerHTML;
        // 编译模板
        return _compileTpl(html);
    }else{
        // 编译模板
        return _compileTpl(str);
    }
}
```

```
_dealTpl = function(str){
    var _left = '{%',   // 左分隔符
        _right = '%}';   // 右分隔符
        // 显式转化为字符串
    return String(str)
        // 转义标签内的< 如: <div>{%if(a&lt;b)%}</div> -> <div>{%if(a<b)%}</div>
        .replace(/&lt;/g,'<')
        // 转义标签内的>
        .replace(/&gt;/g,'>')
        // 过滤回车符，制表符，回车符
        .replace(/[\r\t\n]/g,'')
        // 替换内容
        .replace(new RegExp(_left+'=(.*?)'+_right, 'g'), "',typeof($1)===
'undefined'?'': $1,'")
        // 替换左分隔符
        .replace(new RegExp(_left, 'g'), "');")
        // 替换右分隔符
        .replace(new RegExp(_right, 'g'), "template_array.push('");
}
```

```
/***
 * 编译执行
 * @param str 模板数据
 **/
_compileTpl = function(str){
    // 编译函数体
    var fnBody = "var template_array=[];\nvar fn=(function(dat
key='';\ nfor(key in data){\ntemplate_key+=('var '+key+'=data
\n}\neval (template_key);\ntemplate_array.push('"+_dealTpl(st
key=null;\n})(templateData);\nfn = null;\nreturn template_a
    // 编译函数
    return new Function("templateData", fnBody);
},
```

```
// 页面元素内容
<div id="demo_tag" class="template">
    <div id="tag_cloud">
        {% for(var i = 0, len = tagCloud.length; i < len; i++){
            var ctx = tagCloud[i];%}
            <a href="#" class="tag_item
            {% if(ctx['is_selected']){ %}
                selected
            {% } %}
            " title="{%=ctx["title"]%}">{%=ctx["text"]%}</a>
        {% } %}
    </div>
</div>
// 表单元素内的内容
<textarea id="demo_textarea" class="template">
    <div id="tag_cloud">
        {% for(var i = 0, len = tagCloud.length; i < len; i++){
            var ctx = tagCloud[i];%}
            <a href="#" class="tag_item
            {% if(ctx["is_selected"]){ %}
                selected
            {% } %}
            " title="{%=ctx["title"]%}">{%=ctx["text"]%}</a>
        {% } %}
    </div>
</textarea>// script 模板内容
<script type="text/template" id="demo_script">
    <div id="tag_cloud">
        {% for(var i = 0, len = tagCloud.length; i < len; i++){
            var ctx = tagCloud[i];%}
            <a href="#" class="tag_item
            {% if(ctx["is_selected"]){ %}
```

```
            selected
        {% } %}
        " title="{%=ctx["title"]%}">{%=ctx["text"]%}</a>
    {% } %}
</div>
</script>
// 自定义模板
var demo_tpl = ['<div id="tag_cloud">',
    '{% for(var i = 0, len = tagCloud.length; i < len; i++){',
        'var ctx = tagCloud[i];%}',
        '<a href="#" class="tag_item ',
        '{% if(ctx["is_selected"]){ %}',
            'selected',
        '{% } %}',
        '" title="{%=ctx["title"]%}">{%=ctx["text"]%}</a>',
    '{% } %}',
'</div>'].join('');

var data = {
    tagCloud : [
        {is_selected:true, title:'这是一本设计模式书', text:'设计模式'},
        {is_selected:false, title:'这是一本 HTML 书', text:'HTML'},
        {is_selected:null, title:'这是一本 CSS 书', text:'CSS'},
        {is_selected:'', title:'这是一本 JavaScript 书', text:'JavaScript'},
    ]
};
```

```
/*widget/tag_cloud.js*/
F.module(['lib/template', 'lib/dom'], function(template, dom){
    // 服务器端获取到 data 数据逻辑
    // 创建组件视图逻辑
    var str = template('demo_script', data);
    dom.html('test', str);
    //  组件其他交互逻辑
});
```

（4）MVC模式

MVC模式：模型-视图-控制器，将数据，视图，业务逻辑分离。

e.g.实现MVC

```
// 为简化页面操作逻辑，这里引用链模式中实现的 A 框架，具体方法参考附录 A
// 页面加载后创建 MVC 对象
$(function(){
    // 初始化 MVC 对象
    var MVC = MVC || {};
    // 初始化 MVC 数据模型层
    MVC.model = function(){}();
    // 初始化 MVC 视图层
    MVC.view = function(){}();
    // 初始化 MVC 控制器层
    MVC.ctrl = function(){}();
});
```

```
// MVC 数据模型层
MVC.model = function(){
    // 内部数据对象
    var M = {};
    // 服务器端获取的数据，通常通过 Ajax 获取并存储，后面的案例为简化实现，直接作为同步数据
    // 写在页面中，减少服务器端异步请求操作
    M.data = {};
    // 配置数据，页面加载时即提供
    M.conf = {};
    // 返回数据模型层对象操作方法
    return {
        // 获取服务器端数据
        getData : function(m){
```

```
            // 根据数据字段获取数据
            return M.data[m];
        },
        // 获取配置数据
        getConf : function(c){
            // 根据配置数据字段获取配置数据
            return M.conf[c]
        },
        // 设置服务器端数据（通常将服务器端异步获取到的数据，更新该数据）
        setData : function(m, v){
            // 设置数据字段 m 对应的数据 v
            M.data[m] = v;
            return this;
        }
        // 设置配置数据（通常在页面中执行某些操作，为做记录而更新配置数据）
        setConf : function(c, v){
            // 设置配置数据字段 c 对应的配置数据 v
            M.conf[c] = v;
            return this;
        }
    }
}();
```

```
// MVC 视图层
MVC.view = function(){
    // 模型数据层对象操作方法引用
    var M = MVC.model;
    // 内部视图创建方法对象
    var V = {}
    // 获取视图接口方法
    return function(v){
        // 根据视图名称返回视图（由于获取的是一个方法，这里需要将该方法执行一遍以获取相应视图）
        V[v]();
    }
}();
```

```
// MVC 控制器层
MVC.ctrl = function(){
    // 模型数据层对象操作方法引用
    var M = MVC.model;
    // 视图数据层对象操作方法引用
    var V = MVC.view;
    // 控制器创建方法对象
    var C = {};
}();
```

（5）MVP模式

MVP模式：模型-视图-管理器，视图层不直接引用模型层的数据，而是通过管理器层实现对模型层数据的访问，所有层次交互发生在管理器层。

e.g.实现MVP

```
// MVP 模块
~(function(window){
    // MVP 构造函数
    var MVP = function(){};
    // 数据层
    MVP.model = function(){};
    // 视图层
    MVP.view = function(){};
    // 管理层
    MVP.presenter = function(){};
    // MVP 入口
    MVP.init = function(){}
    // 暴露 MVP 对象，这样即可在外部访问 MVP
    window.MVP = MVP;
})(window)

// 数据层 与 MVC 模式中的数据层相似
MVP.model = function(){
    var M = {};
    M.data = {}
    M.conf = {}
    return {
        getData : function(m){
            return M.data[m];
        },
        /***
         * 设置数据
         * @param    m     模块名称
         * @param    v     模块数据
         **/
        setData : function(m, v){
            M.data[m] = v;
            return v;
        },
        getConf : function(c){
            return M.conf[c]
        },
        /***
         * 设置配置
         * @param    c     配置项名称
         * @param    v     配置项值
         **/
        setConf : function(c, v){
            M.conf[c] = v;
            return v;
        }
    }
}();
```

```
MVP.view = function(){
    // 子元素或者兄弟元素替换模板
    var REPLACEKEY = '__REPLACEKEY__';
    // 获取完整元素模板
    function getHTML(str, replacePos){}
    /***
     * 数组迭代器
     * @param arr   数组
     * @param fn    回调函数
     **/
    function eachArray(arr, fn){
        // 遍历数组
        for(var i = 0, len = arr.length; i < len; i++){
            // 将索引值、索引对应值、数组长度传入回调函数中并执行
            fn(i, arr[i], len);
        }
    }
    /***
     * 替换兄弟元素模板或者子元素模板
     * @param str   原始字符串
     * @param rep   兄弟元素模板或者子元素模板
     **/
    function formateItem(str, rep){
        // 用对应元素字符串替换兄弟元素模板或者子元素模板
        return str.replace(new RegExp(REPLACEKEY, 'g'), rep);
    }
// 模板解析器
return function(str){
        // 模板层级数组
    var part = str
    // 去除首尾空白符
    .replace(/^\s+|\s+$/g, '')
    // 去除>两端空白符
    .replace(/^\s+(>)\s+/g, '$1')
    // 以>分组
    .split('>'),
        // 模块视图根模板
        html = REPLACEKEY,
        // 同层元素
        item,
        // 同级元素模板
        nodeTpl;
    // 遍历每组元素
    eachArray(part, function(partIndex, partValue, partLen){
        // 为同级元素分组
        item = partValue.split('+');
        // 设置同级元素初始模板
        nodeTpl = REPLACEKEY;
        // 遍历同级每一个元素
        eachArray(item, function(itemIndex, itemValue, itemLen){
```

```
            // 用渲染元素得到的模板去渲染同级元素模板，此处简化逻辑处理
            // 如果 itemIndex（同级元素索引）对应元素不是最后一个 则作为兄弟元素处理
            // 否则，如果 partIndex（层级索引）对应的层级不是最后一层 则作为父层级处理
            // （该层级有子
            //层级，即该元素是父元素）
            // 否则，该元素无兄弟元素，无子元素
            nodeTpl = formateItem(nodeTpl, getHTML(itemValue, itemIndex ===
            itemLen - 1?(partIndex === partLen - 1?'':'in'):'add'));
        });
        // 用渲染子层级得到的模板去渲染父层级模板
        html = formateItem(html, nodeTpl);
    })
    // 返回期望视图模板
    return html;
  }
}();

/***
 * 获取完整元素模板
 * @param str       元素字符串
 * @param type      元素类型
 **/
function getHTML(str, type){
    // 简化实现，只处理字符串中第一个{}里面的内容
    return str
        .replace(/^(\w+)([^\{\}]*)?(\{([@\w]+)\})?(.*?)$/, function(match, $1,
$2, $3, $4, $5){
            $2 = $2 || '';     // 元素属性参数容错处理
            $3 = $3 || '';     // {元素内容}参数容错处理
            $4 = $4 || '';     // 元素内容参数容错处理
            $5 = $5.replace(/\{([@\w]+)\}/g, ''); // 去除元素内容后面添加的元素属性
                                                  // 中的{}内容
// 以 str=div 举例：如果 div 元素有子元素则表示成<div>__REPLACEKEY__</div>,如果 div 有
            //兄弟元素则表示成<div></div>__REPLACEKEY__,否则表示成<div></div>
            return type === 'in' ?
                '<' + $1 + $2 + $5 + '>' + $4 + REPLACEKEY + '</' + $1 + '>' :
            type === 'add' ?
                '<' + $1 + $2 + $5 + '>' + $4 + '</' + $1 + '>' + REPLACEKEY :
                '<' + $1 + $2 + $5 + '>' + $4 + '</' + $1 + '>';
```

```
        })
        // 处理特殊标识#--id 属性
        .replace(/#([@\-\w]+)/g, ' id="$1"')
        // 处理特殊标识.--class 属性
        .replace(/\.([@\-\s\w]+)/g, ' class="$1"')
        // 处理其他属性组
        .replace(/\[(.+)\]/g, function(match, key){
                // 元素属性组
            var a = key
                    // 过滤其中引号
                    .replace(/'|"/g, '')
                    // 以空格分组
                    .split(' '),
                // 属性模板字符串
                h = '';
            // 遍历属性组
            for(var j = 0, len = a.length; j < len; j++){
                // 处理并拼接每一个属性
                h += ' ' + a[j].replace(/=(.*)/g, '="$1"');
            }
            // 返回属性组模板字符串
            return h;
        })
        // 处理可替换内容，可根据不同模板渲染引擎自由处理
        .replace(/@(\w+)/g, '{#$1#}');
}
```

```
// 管理器层
MVP.presenter = function(){
    var V = MVP.view;
    var M = MVP.model;
    var C = {};
    return {
        // 执行方法
        init : function(){
            // 遍历内部管理器
            for(var i in C){
                // 执行所有管理器内部逻辑
                C[i] && C[i](M, V, i);
            }
        }
    };
}();
```

```
var C = {
    /***
     * 导航管理器
     * @param  M    数据层对象
     * @param  V    视图层对象
     **/
    nav : function(M, V){
        // 获取导航渲染数据
        var data = M.getData('nav');
        // 处理导航渲染数据
        data[0].choose = 'choose';
        data[data.length - 1].last = 'last';
        // 获取导航渲染模板
        var tpl = V('li.@mode @choose @last[data-mode=@mode]>a#nav_@mode.nav-@mode
[href=@url title=@text]>i.nav-icon-@mode+span{@text}');
        $
        // 创建导航容器
        .create('ul', {
            'class' : 'navigation',
            'id' : 'nav'
        })
        // 插入导航视图
        .html(
            // 渲染导航视图
            A.formateString(tpl, data)
        )
        // 导航模块添加到页面中
        .appendTo('#container');

        // 其他交互逻辑与动画逻辑
        // ......
    }
};

M.data = {
    // 导航模块渲染数据
    nav : [
        {
            text : '新闻头条',
            mode : 'news',
            url : 'http://www.example.com/01'


        },
        {
            text : '最新电影',
            mode : 'movie',
            url : 'http://www.example.com/02'
        },
        {
            text : '热门游戏',
            mode : 'game',
            url : 'http://www.example.com/03'
        },
        {
            text : '今日特价',
            mode : 'price',
            url : 'http://www.example.com/04'
        }
    ]
};

// MVP入口
MVP.init = function(){
    this.presenter.init();
}
```

```
window.onload = function(){
    //执行管理器
    MVP.init();
}

F.module('lib/MVP', function(){
    // MVP 构造函数
    var MVP = function(){};
    // MVP 实现


    // ……
    return MVP;
});

// MVP 构造函数
var MVP = function(modName, pst, data){
    // 在数据层中添加 modName 渲染数据模块
    MVP.model.setData(modName, data);
    // 在管理器层中添加 modName 管理器模块
    MVP.presenter.add(modName, pst);
};

// 管理器层
MVP.presenter = function(){
    // ……
    return {
        init : function(){},
        /***
         * 为管理器添加模块
         * @param   modName   模块名称
         * @param   pst       模块管理器
         **/
        add : function(modName, pst){
            C[modName] = pst;
            return this;
        }
    };
}();
```

（6）MVVM模式

MVVM模式：模型-视图-视图模型，为视图层量身定做视图模型，在视图模型中创建属性方法，为视图层绑定数据并实现交互。

e.g.实现MVVM

```
<div class="first" data-bind="type : 'slider', data : demo1"></div>
<div class="second" data-bind="type : 'slider', data : demo2">test</div>
<div class="third" data-bind="type : 'progressbar', data : demo3"></div>
```

```
// ~屏蔽压缩报错
~(function(){
    // 在闭包中获取全局变量
    var window = this || (0, eval)('this');
```

```
    // 获取页面字体大小，作为创建页面 UI 尺寸参照物
    var FONTSIZE = function(){
        // 获取页面 body 元素字体大小并转化成整数。
        return parseInt(document.body.currentStyle ? document.body.currentStyle
        ['fontSize'] : getComputedStyle(document.body, false)['fontSize']);
    }();
    // 视图模型对象
    var VM = function(){}();
    // 将视图模型对象绑定在 Window 上，供外部获取
    window.VM = VM;
})();
```

```
var VM = function(){
    // 组件创建策略对象
    var Method = {
        // 进度条组件创建方法
        progressbar : function(){},
        // 滑动条组件创建方法
        slider : function(){}
    }
}();
```

```
/***
 * 进度条组件创建方法
 * dom    进度条容器
 * data 进度条数据模型
 **/
progressbar : function(dom, data){
        // 进度条进度完成容器

    var progress = document.createElement('div'),
        // 数据模型数据，结构：{position : 50}
        param = data.data;
    // 设置进度完成容器尺寸
    progress.style.width = (param.position || 100) + '%';
    // 为进度条组件添加 UI 样式
    dom.className += ' ui-progressbar';
    // 进度完成容器元素插入进度条容器中
    dom.appendChild(progress);
}
```

```
var VM = function(){
    var Method = {
        // ......
    }
    /***
     * 获取视图层组件渲染数据的映射信息
     * dom      组件元素
     **/
    function getBindData(dom){
        // 获取组件自定义属性 data-bind 值
        var data = dom.getAttribute('data-bind');
        // 将自定义属性 data-bind 值转化为对象
        return !!data && (new Function("return ({" + data + "})"))();
    }
}();
```

```
var VM = function(){
    var Method = {
        // ......
    }
    function getBindData(){}

    // 组件实例化方法
    return function(){
            // 获取页面中所有元素
        var doms = document.body.getElementsByTagName('*'),
            // 元素自定义数据句柄
            ctx = null;
        // ui 处理是会向页面中插入元素，此时 doms.length 会改变，此时动态获取 dom.length
        for(var i = 0; i < doms.length; i++){

            // 获取元素自定义数据
            ctx = getBindData(doms[i]);
            // 如果元素是 UI 组件，则根据自定义属性中组件类型，渲染该组件
            ctx.type && Method[ctx.type] && Method[ctx.type](doms[i], ctx);
        }

    }
}();
```

```
// 数据模型层中获取到的组件渲染数据
    // 带有提示文案的滑动条
var demo1 = {
        position : 60,
        totle : 200
    },
    // 简易版滑动条
    demo2 = {
        position : 20
    },
    // 进度条
    demo3 = {position : 50};

window.onload = function(){
    // 渲染组件
    VM();
}
```