**一 单例模式**

单例模式的定义是产生一个类的唯一实例，但js本身是一种“无类”语言。很多讲js设计模式的文章把{}当成一个单例来使用也勉强说得通。因为js生成对象的方式有很多种，我们来看下另一种更有意义的单例。

有这样一个常见的需求，点击某个按钮的时候需要在页面弹出一个遮罩层。比如web.qq.com点击登录的时候.

[](http://jbcdn2.b0.upaiyun.com/2012/10/image-11.jpg)

这个生成灰色背景遮罩层的代码是很好写的.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5 | var createMask = function(){       return document.body.appendChild(  document.createElement(div)  );    } |

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $( 'button' ).click( function(){       Var mask  = createMask();       mask.show();    }) |

问题是, 这个遮罩层是全局唯一的, 那么每次调用createMask都会创建一个新的div, 虽然可以在隐藏遮罩层的把它remove掉. 但显然这样做不合理.

再看下第二种方案, 在页面的一开始就创建好这个div. 然后用一个变量引用它.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var mask = document.body.appendChild( document.createElement( ''div' ) );    $( ''button' ).click( function(){       mask.show();    } ) |

这样确实在页面只会创建一个遮罩层div, 但是另外一个问题随之而来, 也许我们永远都不需要这个遮罩层, 那又浪费掉一个div, 对dom节点的任何操作都应该非常吝啬.

如果可以借助一个变量. 来判断是否已经创建过div呢?

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | var mask;    var createMask = function(){    if ( mask ) return mask;    else{    mask = document,body.appendChild(  document.createElement(div)  );    return mask;    }    } |

看起来不错, 到这里的确完成了一个产生单列对象的函数. 我们再仔细看这段代码有什么不妥.

首先这个函数是存在一定副作用的, 函数体内改变了外界变量mask的引用, 在多人协作的项目中, createMask是个不安全的函数. 另一方面, mask这个全局变量并不是非需不可. 再来改进一下.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var createMask = function(){    var mask;    return function(){         return mask || ( mask = document.body.appendChild( document.createElement('div') ) )    }  }() |

用了个简单的闭包把变量mask包起来, 至少对于createMask函数来讲, 它是封闭的.

可能看到这里, 会觉得单例模式也太简单了. 的确一些设计模式都是非常简单的, 即使从没关注过设计模式的概念, 在平时的代码中也不知不觉用到了一些设计模式. GOF里的23种设计模式, 也是在软件开发中早就存在并反复使用的模式. 如果程序员没有明确意识到他使用过某些模式, 那么下次他也许会错过更合适的设计 (这段话来自《松本行弘的程序世界》).

再回来正题, 前面那个单例还是有缺点. 它只能用于创建遮罩层. 假如我又需要写一个函数, 用来创建一个唯一的xhr对象呢? 能不能找到一个通用的singleton包装器.

js中函数是第一型, 意味着函数也可以当参数传递. 看看最终的代码.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var singleton = function( fn ){      var result;      return function(){          return result || ( result = fn .apply( this, arguments ) );      }  }    var createMask = singleton( function(){    return document.body.appendChild( document.createElement('div') );     }) |

用一个变量来保存第一次的返回值, 如果它已经被赋值过, 那么在以后的调用中优先返回该变量. 而真正创建遮罩层的代码是通过回调函数的方式传人到singleton包装器中的. 这种方式其实叫桥接模式. 关于桥接模式, 放在后面一点点来说.

然而singleton函数也不是完美的, 它始终还是需要一个变量result来寄存div的引用. 遗憾的是js的函数式特性还不足以完全的消除声明和语句.

**二 简单工厂模式**

简单工厂模式是由一个方法来决定到底要创建哪个类的实例, 而这些实例经常都拥有相同的接口. 这种模式主要用在所实例化的类型在编译期并不能确定， 而是在执行期决定的情况。 说的通俗点，就像公司茶水间的饮料机，要咖啡还是牛奶取决于你按哪个按钮。

简单工厂模式在创建ajax对象的时候也非常有用.

之前我写了一个处理ajax异步嵌套的库，地址在<https://github.com/AlloyTeam/DanceRequest.>

这个库里提供了几种ajax请求的方式，包括xhr对象的get, post, 也包括跨域用的jsonp和iframe. 为了方便使用, 这几种方式都抽象到了同一个接口里面.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var request1 = Request('cgi.xx.com/xxx' , ''get' );    request1.start();    request1.done( fn );    var request2 = Request('cgi.xx.com/xxx' , ''jsonp' );    request2.start();    request2.done( fn ); |

Request实际上就是一个工厂方法, 至于到底是产生xhr的实例, 还是jsonp的实例. 是由后来的代码决定的。

实际上在js里面，所谓的构造函数也是一个简单工厂。只是批了一件new的衣服. 我们扒掉这件衣服看看里面。

通过这段代码, 在firefox, chrome等浏览器里，可以完美模拟new.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | function A( name ){          this.name = name;    }      function ObjectFactory(){       var obj = {},       Constructor = Array.prototype.shift.call( arguments );  obj.\_\_proto\_\_ =  typeof Constructor .prototype === 'number'  ? Object.prototype    :  Constructor .prototype;                  var ret = Constructor.apply( obj, arguments );                  return typeof ret === 'object' ? ret : obj;           }           var a = ObjectFactory( A, 'svenzeng' );           alert ( a.name );  //svenzeng |

这段代码来自es5的new和构造器的相关说明， 可以看到，所谓的new， 本身只是一个对象的复制和改写过程， 而具体会生成什么是由调用ObjectFactory时传进去的参数所决定的。

**三 观察者模式**

观察者模式( 又叫发布者-订阅者模式 )应该是最常用的模式之一. 在很多语言里都得到大量应用. 包括我们平时接触的dom事件. 也是js和dom之间实现的一种观察者模式.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5 | div.onclick  =  function click (){       alert ( ''click' )    } |

只要订阅了div的click事件. 当点击div的时候, function click就会被触发.

那么到底什么是观察者模式呢. 先看看生活中的观察者模式。

好莱坞有句名言. “不要给我打电话， 我会给你打电话”. 这句话就解释了一个观察者模式的来龙去脉。 其中“我”是发布者， “你”是订阅者。

再举个例子，我来公司面试的时候，完事之后每个面试官都会对我说：“请留下你的联系方式， 有消息我们会通知你”。 在这里“我”是订阅者， 面试官是发布者。所以我不用每天或者每小时都去询问面试结果， 通讯的主动权掌握在了面试官手上。而我只需要提供一个联系方式。

观察者模式可以很好的实现2个模块之间的解耦。 假如我正在一个团队里开发一个html5游戏. 当游戏开始的时候，需要加载一些图片素材。加载好这些图片之后开始才执行游戏逻辑. 假设这是一个需要多人合作的项目. 我完成了Gamer和Map模块, 而我的同事A写了一个图片加载器loadImage.

loadImage的代码如下

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | loadImage(  imgAry,  function(){    Map.init();    Gamer.init();    } ) |

当图片加载好之后, 再渲染地图, 执行游戏逻辑. 嗯, 这个程序运行良好. 突然有一天, 我想起应该给游戏加上声音功能. 我应该让图片加载器添上一行代码.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | loadImage(  imgAry,  function(){    Map.init();    Gamer.init();    Sount.init();    } ) |

可是写这个模块的同事A去了外地旅游. 于是我打电话给他, 喂. 你的loadImage函数在哪, 我能不能改一下, 改了之后有没有副作用. 如你所想, 各种不淡定的事发生了. 如果当初我们能这样写呢:

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | loadImage.listen( ''ready', function(){        Map.init();    })    loadImage.listen( ''ready', function(){       Gamer.init();    })    loadImage.listen( ''ready', function(){       Sount.init();    }) |

loadImage完成之后, 它根本不关心将来会发生什么, 因为它的工作已经完成了. 接下来它只要发布一个信号.

loadImage.trigger( ”ready’ );

那么监听了loadImage的’ready’事件的对象都会收到通知. 就像上个面试的例子. 面试官根本不关心面试者们收到面试结果后会去哪吃饭. 他只负责把面试者的简历搜集到一起. 当面试结果出来时照着简历上的电话挨个通知.

说了这么多概念, 来一个具体的实现. 实现过程其实很简单. 面试者把简历扔到一个盒子里， 然后面试官在合适的时机拿着盒子里的简历挨个打电话通知结果.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67 | Events = function() {               var listen, log, obj, one, remove, trigger, \_\_this;               obj = {};               \_\_this = this;               listen = function( key, eventfn ) {  //把简历扔盒子, key就是联系方式.                 var stack, \_ref;  //stack是盒子                 stack = ( \_ref = obj[key] ) != null ? \_ref : obj[ key ] = [];                 return stack.push( eventfn );               };               one = function( key, eventfn ) {                 remove( key );                 return listen( key, eventfn );               };               remove = function( key ) {                 var \_ref;                 return ( \_ref = obj[key] ) != null ? \_ref.length = 0 : void 0;               };               trigger = function() {  //面试官打电话通知面试者                 var fn, stack, \_i, \_len, \_ref, key;                 key = Array.prototype.shift.call( arguments );                 stack = ( \_ref = obj[ key ] ) != null ? \_ref : obj[ key ] = [];                 for ( \_i = 0, \_len = stack.length; \_i < \_len; \_i++ ) {                   fn = stack[ \_i ];                   if ( fn.apply( \_\_this,  arguments ) === false) {                     return false;                   }                 }                 return {                    listen: listen,                    one: one,                    remove: remove,                    trigger: trigger                 }               } |

最后用观察者模式来做一个成人电视台的小应用.

//订阅者

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var adultTv = Event();    adultTv .listen(  ''play',  function( data ){       alert ( "今天是谁的电影" + data.name );    });    //发布者    adultTv .trigger(  ''play',  { 'name': '麻生希' }  ) |

**四 适配器模式**

去年年前当时正在开发dev.qplus.com, 有个存储应用分类id的js文件, 分类id的结构最开始设计的比较笨重. 于是我决定重构它. 我把它定义成一个json树的形式, 大概是这样:

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var category = {       music: {    id: 1,    children: [ , , , , ]       }    } |

dev.qplus.com里大概有4，5个页面都调用这个category对象. 春节前我休了1个星期假. 过年来之后发现邮箱里有封邮件, 设计数据库的同学把category..js也重构了一份, 并且其他几个项目里都是用了这份category.js, 我拿过来一看就傻眼了, 和我之前定的数据结构完全不一样.

当然这是一个沟通上的反面例子. 但接下来的重点是我已经在N个文件里用到了之前我定的category.js. 而且惹上了一些复杂的相关逻辑. 怎么改掉我之前的代码呢. 全部重写肯定是不愿意. 所以现在适配器就派上用场了.

只需要把同事的category用一个函数转成跟我之前定义的一样.

JavaScript

|  |  |
| --- | --- |
| 1 | my.category = adapterCategory ( afu.category ); |

适配器模式的作用很像一个转接口. 本来iphone的充电器是不能直接插在电脑机箱上的, 而通过一个usb转接口就可以了.

所以, 在程序里适配器模式也经常用来适配2个接口, 比如你现在正在用一个自定义的js库. 里面有个根据id获取节点的方法$id(). 有天你觉得jquery里的$实现得更酷, 但你又不想让你的工程师去学习新的库和语法. 那一个适配器就能让你完成这件事情.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5 | $id = function( id ){      return jQuery( '#' + id )[0];    } |

**五 代理模式**

代理模式的定义是把对一个对象的访问, 交给另一个代理对象来操作.

举一个例子, 我在追一个MM想给她送一束花，但是我因为我性格比较腼腆，所以我托付了MM的一个好朋友来送。

这个例子不是非常好， 至少我们没看出代理模式有什么大的用处，因为追MM更好的方式是送一台宝马。

再举个例子，假如我每天都得写工作日报( 其实没有这么惨 ). 我的日报最后会让总监审阅. 如果我们都直接把日报发给 总监 , 那可能 总监 就没法工作了. 所以通常的做法是把日报发给我的组长 ， 组长把所有组员一周的日报都汇总后再发给总监 .

实际的编程中， 这种因为性能问题使用代理模式的机会是非常多的。比如频繁的访问dom节点, 频繁的请求远程资源. 可以把操作先存到一个缓冲区, 然后自己选择真正的触发时机.

再来个详细的例子，之前我写了一个街头霸王的游戏, 地址在<http://alloyteam.github.com/StreetFighter/>

[](http://jbcdn2.b0.upaiyun.com/2012/10/image-2.jpg)

游戏中隆需要接受键盘的事件, 来完成相应动作.

于是我写了一个keyManage类. 其中在游戏主线程里监听keyManage的变化.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var keyMgr = keyManage();    keyMgr.listen( ''change', function( keyCode ){       console.log( keyCode );    }); |

图片里面隆正在放升龙拳, 升龙拳的操作是前下前+拳. 但是这个keyManage类只要发生键盘事件就会触发之前监听的change函数. 这意味着永远只能取得前，后，前，拳这样单独的按键事件，而无法得到一个按键组合。

好吧，我决定改写我的keyManage类, 让它也支持传递按键组合. 但是如果我以后写个html5版双截龙，意味着我每次都得改写keyManage. 我总是觉得, 这种函数应该可以抽象成一个更底层的方法, 让任何游戏都可以用上它.

所以最后的keyManage只负责映射键盘事件. 而隆接受到的动作是通过一个代理对象处理之后的.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var keyMgr = keyManage();    keyMgr.listen( ''change', proxy( function( keyCode ){       console.log( keyCode );  //前下前+拳    )} ); |

至于proxy里面怎么实现，完全可以自由发挥。

还有个例子就是在调用ajax请求的时候，无论是各种开源库，还是自己写的Ajax类, 都会给xhr对象设置一个代理. 我们不可能频繁的去操作xhr对象发请求, 而应该是这样.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var request = Ajax.get( 'cgi.xx.com/xxx' );    request.send();    request.done(function(){    }); |

**六 桥接模式**

桥接模式的作用在于将实现部分和抽象部分分离开来， 以便两者可以独立的变化。在实现api的时候， 桥接模式特别有用。比如最开始的singleton的例子.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var singleton = function( fn ){      var result;      return function(){          return result || ( result = fn .apply( this, arguments ) );      }  }    var createMask = singleton( function(){    return document.body.appendChild( document.createElement('div') );     }) |

singleton是抽象部分， 而createMask是实现部分。 他们完全可以独自变化互不影响。 如果需要再写一个单例的createScript就一点也不费力.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5 | var createScript = singleton( function(){    return document.body.appendChild( document.createElement('script') );     }) |

另外一个常见的例子就是forEach函数的实现, 用来迭代一个数组.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | forEach = function( ary, fn ){    for ( var i = 0, l = ary.length; i < l; i++ ){      var c = ary[ i ];      if ( fn.call( c, i, c ) === false ){        return false;      }     }  } |

可以看到, forEach函数并不关心fn里面的具体实现. fn里面的逻辑也不会被forEach函数的改写影响.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | forEach( [1,2,3], function( i, n ){     alert ( n\*2 )    } )    forEach( [1,2,3], function( i, n ){      alert ( n\*3 )    } ) |

**七 外观模式**

外观模式(门面模式)，是一种相对简单而又无处不在的模式。外观模式提供一个高层接口，这个接口使得客户端或子系统更加方便调用。  
用一段再简单不过的代码来表示

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var getName = function(){    return ''svenzeng"  }  var getSex = function(){     return 'man'  } |

如果你需要分别调用getName和getSex函数. 那可以用一个更高层的接口getUserInfo来调用.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4 | var getUserInfo = function(){    var info = a() + b();    return info;  } |

也许你会问为什么一开始不把getName和getSex的代码写到一起, 比如这样

JavaScript

|  |  |
| --- | --- |
| 1  2  3 | var getNameAndSex = function(){    return 'svenzeng" + "man";  } |

答案是显而易见的，饭堂的炒菜师傅不会因为你预定了一份烧鸭和一份白菜就把这两样菜炒在一个锅里。他更愿意给你提供一个烧鸭饭套餐。同样在程序设计中，我们需要保证函数或者对象尽可能的处在一个合理粒度，毕竟不是每个人喜欢吃烧鸭的同时又刚好喜欢吃白菜。  
外观模式还有一个好处是可以对用户隐藏真正的实现细节，用户只关心最高层的接口。比如在烧鸭饭套餐的故事中，你并不关心师傅是先做烧鸭还是先炒白菜，你也不关心那只鸭子是在哪里成长的。

最后写个我们都用过的外观模式例子

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4 | var stopEvent = function( e ){   //同时阻止事件默认行为和冒泡    e.stopPropagation();    e.preventDefault();  } |

**八 访问者模式**

GOF官方定义： 访问者模式是表示一个作用于某个对象结构中的各元素的操作。它使可以在不改变各元素的类的前提下定义作用于这些元素的新操作。我们在使用一些操作对不同的对象进行处理时，往往会根据不同的对象选择不同的处理方法和过程。在实际的代码过程中，我们可以发现，如果让所有的操作分散到各个对象中，整个系统会变得难以维护和修改。且增加新的操作通常都要重新编译所有的类。因此，为了解决这个问题，我们可以将每一个类中的相关操作提取出来，包装成一个独立的对象，这个对象我们就称为访问者（Visitor）。利用访问者，对访问的元素进行某些操作时，只需将此对象作为参数传递给当前访问者，然后，访问者会依据被访问者的具体信息，进行相关的操作。

据统计，上面这段话只有5%的人会看到最后一句。那么通俗点讲，访问者模式先把一些可复用的行为抽象到一个函数(对象)里，这个函数我们就称为访问者（Visitor）。如果另外一些对象要调用这个函数，只需要把那些对象当作参数传给这个函数，在js里我们经常通过call或者apply的方式传递this对象给一个Visitor函数.  
访问者模式也被称为GOF总结的23种设计模式中最难理解的一种。不过这有很大一部分原因是因为《设计模式》基于C++和Smalltalk写成. 在强类型语言中需要通过多次重载来实现访问者的接口匹配。

而在js这种基于鸭子类型的语言中，访问者模式几乎是原生的实现, 所以我们可以利用apply和call毫不费力的使用访问者模式，这一小节更关心的是这种模式的思想以及在js引擎中的实现。

我们先来了解一下什么是鸭子类型，说个故事：  
很久以前有个皇帝喜欢听鸭子呱呱叫，于是他召集大臣组建一个一千只鸭子的合唱团。大臣把全国的鸭子都抓来了，最后始终还差一只。有天终于来了一只自告奋勇的鸡，这只鸡说它也会呱呱叫，好吧在这个故事的设定里，它确实会呱呱叫。 后来故事的发展很明显，这只鸡混到了鸭子的合唱团中。— 皇帝只是想听呱呱叫，他才不在乎你是鸭子还是鸡呢。

这个就是鸭子类型的概念，在js这种弱类型语言里，很多方法里都不做对象的类型检测，而是只关心这些对象能做什么。  
Array构造器和String构造器的prototype上的方法就被特意设计成了访问者。这些方法不对this的数据类型做任何校验。这也就是为什么arguments能冒充array调用push方法.

看下v8引擎里面Array.prototype.push的代码:

JavaScript

|  |  |
| --- | --- |
| 1  2 | function ArrayPush() {  var n = TO\_UINT32( this.length );    var m = %\_ArgumentsLength();    for (var i = 0; i < m; i++) {    this[i+n] = %\_Arguments(i);    //属性拷贝  }  this.length = n + m;             //修正length  return this.length;} |

可以看到，ArrayPush方法没有对this的类型做任何显示的限制，所以理论上任何对象都可以被传入ArrayPush这个访问者。

不过在代码的执行期，还是会受到一些隐式限制，在上面的例子很容易看出要求:  
1、 this对象上面可储存属性. //反例: 值类型的数据  
2、 this的length属性可写. //反例: functon对象, function有一个只读的length属性, 表示形参个数.

如果不符合这2条规则的话，代码在执行期会报错. 也就是说, Array.prototype.push.call( 1, ‘first’ )和Array.prototoype.push.call( function(){}, ‘first’ )都达不到预期的效果.

利用访问者，我们来做个有趣的事情. 给一个object对象增加push方法.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var Visitor = {}  Visitor .push  =  function(){      return Array.prototype.push.apply( this, arguments );  }  var obj = {};  obj.push = Visitor .push;  obj.push( '"first" );  alert ( obj[0] )  //"first"  alert ( obj.length );  //1 |

**九 策略模式**

策略模式的意义是定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。  
一个小例子就能让我们一目了然。  
回忆下jquery里的animate方法.

JavaScript

|  |  |
| --- | --- |
| 1  2 | $( div ).animate( {"left: 200px"}, 1000, 'linear' );  //匀速运动  $( div ).animate( {"left: 200px"}, 1000, 'cubic' );  //三次方的缓动 |

这2句代码都是让div在1000ms内往右移动200个像素. linear(匀速)和cubic(三次方缓动)就是一种策略模式的封装.  
再来一个例子. 上半年我写的dev.qplus.com, 很多页面都会有个即时验证的表单. 表单的每个成员都会有一些不同的验证规则.

[](http://jbcdn2.b0.upaiyun.com/2012/10/image-34.png)

比如姓名框里面， 需要验证非空，敏感词，字符过长这几种情况。 当然是可以写3个if else来解决，不过这样写代码的扩展性和维护性可想而知。如果表单里面的元素多一点，需要校验的情况多一点，加起来写上百个if else也不是没有可能。  
所以更好的做法是把每种验证规则都用策略模式单独的封装起来。需要哪种验证的时候只需要提供这个策略的名字。就像这样：

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | nameInput.addValidata({     notNull: true,     dirtyWords: true,     maxLength: 30  })  而notNull，maxLength等方法只需要统一的返回true或者false，来表示是否通过了验证。  validataList = {    notNull: function( value ){       return value !== '';    },    maxLength: function( value, maxLen ){       return value.length() > maxLen;    }  } |

可以看到，各种验证规则很容易被修改和相互替换。如果某天产品经理建议字符过长的限制改成60个字符。那只需要0.5秒完成这次工作。

**十 模版方法模式**

模式方法是预先定义一组算法，先把算法的不变部分抽象到父类，再将另外一些可变的步骤延迟到子类去实现。听起来有点像工厂模式( 非前面说过的简单工厂模式 ).  
最大的区别是,工厂模式的意图是根据子类的实现最终获得一种对象. 而模版方法模式着重于父类对子类的控制.

按GOF的描叙，模版方法导致一种反向的控制结构，这种结构有时被称为“好莱坞法则”，即“别找我们，我们找你”。这指的是一个父类调用一个子类的操作，而不是相反。  
一个很常用的场景是在一个公司的项目中，经常由架构师搭好架构，声明出抽象方法。下面的程序员再去分头重写这些抽象方法。

在深入了解之前，容许我先扯远一点。  
作为一个进化论的反对者，假设这个世界是上帝用代码创造的。那么上帝创造生命的时候可能就用到了模版方法模式。看看他是怎么在生命构造器中声明模版方法的：

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | var Life = function(){  }  Life.prototype.init = function(){     this.DNA复制();     this.出生();     this.成长();     this.衰老();     this.死亡();  }  this.prototype.DNA复制 = function(){    &\*$%&^%^&(&(&(&&(^^(\*)  //看不懂的代码  }  Life.prototype.出生 = function(){  }  Life.prototype.成长 = function(){  }  Life.prototype.衰老 = function(){  }  Life.prototype.死亡 = function(){  } |

其中DNA复制是预先定义的算法中不变部分. 所有子类都不能改写它. 如果需要我们可以写成protected的类型.  
而其他的函数在父类中会被先定义成一个空函数(钩子). 然后被子类重写，这就是模版方法中所谓的可变的步骤。  
假设有个子类哺乳动物类继承了Life类.

JavaScript

|  |  |
| --- | --- |
| 1  2  3 | var Mammal = function(){  }  Mammal.prototype = Life.prototype;   //继承Life |

然后重写出生和衰老这两个钩子函数.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | Mammal.prototope.出生 = function(){    '胎生()  }  Mammal.prototype.成长 = function(){    //再留给子类去实现  }  Mammal.prototope.衰老 = function(){    自由基的过氧化反应()  }  Life.prototype.死亡 = function(){   //再留给子类去实现  }  //再实现一个Dog类  var = Dog = function(){  }  //Dog继承自哺乳动物.  Dog.prototype = Mammal.prototype;  var dog = new Dog();  dog.init(); |

至此，一只小狗的生命会依次经历DNA复制，出生，成长，衰老，死亡这几个过程。这些步骤早在它出生前就决定了。所幸的是，上帝没有安排好它生命的所有细节。它还是能通过对成长函数的重写，来成为一只与众不同的小狗。

举个稍微现实点的例子，游戏大厅中的所有游戏都有登录，游戏中，游戏结束这几个过程，而登录和游戏结束之后弹出提示这些函数都是应该公用的。  
那么首先需要的是一个父类。

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | var gameCenter = function(){  }  gameCenter.ptototype.init = function(){    this.login();    this.gameStart();    this.end();  }  gameCenter.prototype.login= function(){     //do something  }  gameCenter.prototype.gameStart= function(){     //空函数, 留给子类去重写  }  gameCenter.prototype.end= function(){    alert ( "欢迎下次再来玩" );  } |

接下来创建一个斗地主的新游戏, 只需要继承gameCenter然后重写它的gameStart函数.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var 斗地主 = function(){  }  斗地主.prototype = gameCenter.prototype;  //继承  斗地主.prototype.gameStart = function(){    //do something  }  (new 斗地主).init(); |

这样一局新的游戏就开始了.

**十一 中介者模式**

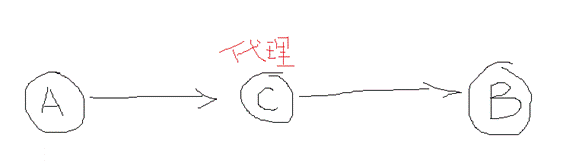
中介者对象可以让各个对象之间不需要显示的相互引用，从而使其耦合松散，而且可以独立的改变它们之间的交互。

打个比方，军火买卖双方为了安全起见，找了一个信任的中介来进行交易。买家A把钱交给中介B，然后从中介手中得到军火，卖家C把军火卖给中介，然后从中介手中拿回钱。一场交易完毕，A甚至不知道C是一只猴子还是一只猛犸。因为中介的存在，A也未必一定要买C的军火，也可能是D，E，F。

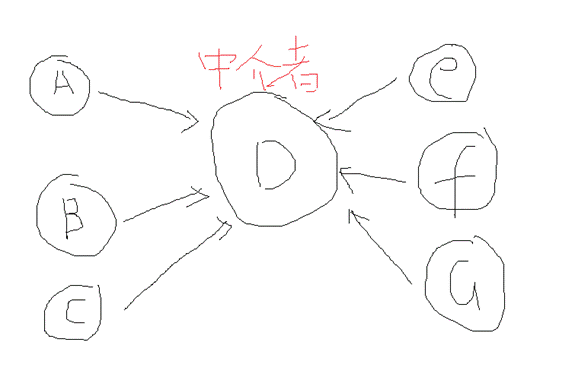
银行在存款人和贷款人之间也能看成一个中介。存款人A并不关心他的钱最后被谁借走。贷款人B也不关心他借来的钱来自谁的存款。因为有中介的存在，这场交易才变得如此方便。

中介者模式和代理模式有一点点相似。都是第三者对象来连接2个对象的通信。具体差别可以从下图中区别。

代理模式：

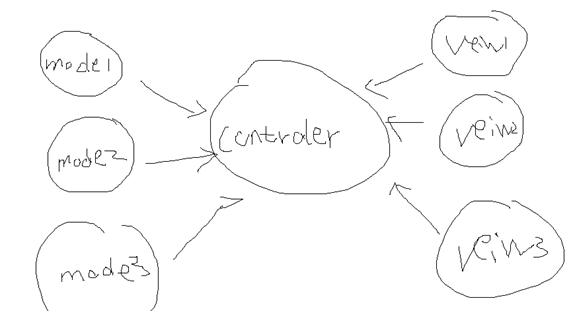
[](http://jbcdn2.b0.upaiyun.com/2012/10/image-4.png)

中介者模式

[](http://jbcdn2.b0.upaiyun.com/2012/10/image-5.png)

代理模式中A必然是知道B的一切，而中介者模式中A,B,C对E,F,G的实现并不关心.而且中介者模式可以连接任意多种对象。

切回到程序世界里的mvc，无论是j2ee中struts的Action. 还是js中backbone.js和spine.js里的Controler. 都起到了一个中介者的作用.  
拿backbone举例. 一个mode里的数据并不确定最后被哪些view使用. view需要的数据也可以来自任意一个mode. 所有的绑定关系都是在controler里决定. 中介者把复杂的多对多关系, 变成了2个相对简单的1对多关系.

[](http://jbcdn2.b0.upaiyun.com/2012/10/image-6.png)

一段简单的示例代码：

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var mode1 = Mode.create(),  mode2 = Mode.create();  var view1 = View.create(),   view2 = View.create();  var controler1 = Controler.create( mode1, view1, function(){    view1.el.find( ''div' ).bind( ''click', function(){      this.innerHTML = mode1.find( 'data' );    } )  })  var controler2 = Controler.create( mode2 view2, function(){    view1.el.find( ''div' ).bind( ''click', function(){      this.innerHTML = mode2.find( 'data' );    } )  }) |

**十二 迭代器模式**

迭代器模式提供一种方法顺序访问一个聚合对象中各个元素，而又不需要暴露该方法中的内部表示。  
js中我们经常会封装一个each函数用来实现迭代器。  
array的迭代器：

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4 | forEach = function( ary, fn ){  for ( var i = 0, l = ary.length; i < l; i++ ){    var c = ary[ i ];    if ( fn.call( c, i , c ) === false ){      return false;    }   }}  forEach( [ 1, 2, 3 ], function( i, n ){    alert ( i );  }) |

obejct的迭代器:

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4 | forEach = function( obj, fn ){  for ( var i in obj ){    var c = obj[ i ];    if ( fn.call( c, i, c ) === false ){      return false;    }   }}  forEach( {"a": 1,"b": 2}, function( i, n ){    alert ( i );  }) |

**十三 组合模式**

组合模式又叫部分-整体模式，它将所有对象组合成树形结构。使得用户只需要操作最上层的接口，就可以对所有成员做相同的操作。  
一个再好不过的例子就是jquery对象，大家都知道1个jquery对象其实是一组对象集合。比如在这样一个HTML页面

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4 | <div>     <span></span>     <span></span>  </div> |

我们想取消所有节点上绑定的事件, 需要这样写

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5 | var allNodes = document.getElementsByTagName("\*");  var len = allNodes.length;  while( len-- ){    allNodes.unbind("\*");  } |

但既然用了jquery，就肯定不会再做这么搓的事情。我们只需要$( ‘body’ ).unbind( ‘\*’ );  
当每个元素都实现unbind接口, 那么只需调用最上层对象$( ‘body’ )的unbind, 便可自动迭代并调用所有组合元素的unbind方法.  
再来个具体点的例子， 还是dev.qplus.com这个网站的即时验证表单。

[](http://jbcdn2.b0.upaiyun.com/2012/10/image-7.png)

注意下面那个修改资料的按钮，如果有任意一个field的验证没有通过，修改资料的按钮都将是灰色不可点的状态。 这意味着我们重新填写了表单内容后, 都得去校验每个field, 保证它们全部OK.  
这代码不难实现.

JavaScript

|  |  |
| --- | --- |
| 1  2  3 | if ( nameField.validata() && idCard.validata() && email.validata() && phone.validata() ){     alert ( "验证OK" );  } |

似乎我们用一个外观模式也能勉强解决这里条件分支堆砌的问题，但真正的问题是，我们并不能保证表单里field的数量，也许明天产品经理就让你删掉一个或者增加两个.那么这样的维护方式显然不能被接受.  
更好的实现是有一个form.validata函数, 它负责把真正的validata操作分发给每个组合对象.  
form.validata函数里面会依次遍历所有需要校验的field. 若有一个field校验未通过, form.validata都会返回false. 伪代码如下.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | form.validata = function(){    forEach( fields, function( index, field ){      if ( field.validata() === false  ){         return false;      }    })    return true;  } |

**十四 备忘录模式**

备忘录模式在js中经常用于数据缓存. 比如一个分页控件, 从服务器获得某一页的数据后可以存入缓存。以后再翻回这一页的时候，可以直接使用缓存里的数据而无需再次请求服务器。  
实现比较简单，伪代码：

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | var Page = function(){     var page = 1,        cache = {},        data;     return function( page ){        if ( cache[ page ] ){                 data =  cache[ page ];                 render( data );        }else{                 Ajax.send( 'cgi.xx.com/xxx', function( data ){                     cache[ page ] = data;                     render( data );                 })        }      }  }() |

**十五 职责链模式**  
职责链模式是一个对象A向另一个对象B发起请求，如果B不处理，可以把请求转给C，如果C不处理，又可以把请求转给D。一直到有一个对象愿意处理这个请求为止。

打个比方，客户让老板写个php程序。老板肯定不写，然后老板交给了部门经理。部门经理不愿意写，又交给项目经理。项目经理不会写，又交给程序员。最后由码农来完成。

在这个假设里， 有几条职责链模式的特点。

1 老板只跟部门经理打交道，部门经理只联系项目经理，项目经理只找码农的麻烦。  
2 如果码农也不写，这个项目将会流产。  
3 客户并不清楚这个程序最后是由谁写出来的。  
js中的事件冒泡就是作为一个职责链来实现的。一个事件在某个节点上被触发，然后向根节点传递， 直到被节点捕获。

**十六 享元模式**

享元模式主要用来减少程序所需的对象个数. 有一个例子, 我们这边的前端同学几乎人手一本《javascript权威指南》. 从省钱的角度讲, 大约三本就够了. 放在部门的书柜里, 谁需要看的时候就去拿, 看完了还回去. 如果同时有4个同学需要看, 此时再去多买一本.

在webqq里面, 打开QQ好友列表往下拉的时候，会为每个好友创建一个div( 如果算上div中的子节点, 还远不只1个元素 ).

[](http://jbcdn2.b0.upaiyun.com/2012/10/image-8.jpg)

如果有1000个QQ好友, 意味着如果从头拉到尾, 会创建1000个div, 这时候有些浏览器也许已经假死了. 这还只是一个随便翻翻好友列表的操作.

所以我们想到了一种解决办法, 当滚动条滚动的时候, 把已经消失在视线外的div都删除掉. 这样页面可以保持只有一定数量的节点. 问题是这样频繁的添加与删除节点, 也会造成很大的性能开销, 而且这种感觉很不对味.

现在享元模式可以登场了. 顾名思义, 享元模式可以提供一些共享的对象以便重复利用. 仔细看下上图, 其实我们一共只需要10个div来显示好友信息,也就是出现在用户视线中的10个div.这10个div就可以写成享元.  
伪代码如下.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | var getDiv = (function(){      var created = [];      var create = function(){            return document.body.appendChild( document.createElement( 'div' ) );      }      var get = function(){           if ( created.length ){                return created.shift();            }else{                  return create();             }       }  /\* 一个假设的事件，用来监听刚消失在视线外的div，实际上可以通过监听滚                                     动条位置来实现 \*/        userInfoContainer.disappear(function( div ){                created.push( div );          })   })()    var div = getDiv();    div.innerHTML = "${userinfo}"; |

原理其实很简单, 把刚隐藏起来的div放到一个数组中, 当需要div的时候, 先从该数组中取, 如果数组中已经没有了, 再重新创建一个. 这个数组里的div就是享元, 它们每一个都可以当作任何用户信息的载体.

当然这只是个示例,实际的情况要复杂一些, 比如快速拖动的时候, 我们可能还得为节点设置一个缓冲区.

**十七 状态模式**

状态模式主要可以用于这种场景  
1 一个对象的行为取决于它的状态  
2 一个操作中含有庞大的条件分支语句

回想下街头霸王的游戏。

[](http://jbcdn2.b0.upaiyun.com/2012/10/image-2.jpg)

隆有走动，攻击，防御，跌倒，跳跃等等多种状态，而这些状态之间既有联系又互相约束。比如跳跃的时候是不能攻击和防御的。跌倒的时候既不能攻击又不能防御，而走动的时候既可以攻击也可以跳跃。

要完成这样一系列逻辑, 常理下if else是少不了的. 而且数量无法估计, 特别是增加一种新状态的时候, 可能要从代码的第10行一直改到900行.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | if ( state === 'jump' ){     if ( currState === 'attack' || currState === 'defense' ){       return false;     }  }else if ( state === 'wait' ){     if ( currState === 'attack' || currState === 'defense' ){       return true;     }  } |

为了消灭这些if else, 并且方便修改和维护, 我们引入一个状态类.

JavaScript

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | var StateManager = function(){    var currState = 'wait';    var states = {      jump: function( state ){      },      wait: function( state ){      },      attack: function( state ){      },      crouch: function( state ){      },      defense: function( state ){        if ( currState === 'jump'  ){            return false;  //不成功，跳跃的时候不能防御        }      //do something;     //防御的真正逻辑代码, 为了防止状态类的代码过多, 应该把这些逻辑继续扔给真正的fight类来执行.      currState = 'defense'; //  切换状态      }    }    var changeState = function( state ){      states[ state ] && states[ state ]();    }    return {        changeState  : changeState    }  }  var stateManager = StateManager();  stateManager.changeState( 'defense' ); |

通过这个状态类，可以把散落在世界各地的条件分支集中管理到一个类里，并且可以很容易的添加一种新的状态。而作为调用者，只需要通过暴露的changeState接口来切换人物的状态。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*分界线1\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

GOF提出的23种设计模式，至此已经写完大半。还有一些要么是js里不太适用，要么是js中已有原生自带的实现，所以就没再去深究。这2篇文章里的大部分例子都来自或改写自工作和学习中的代码。我对设计模式的看法是不用刻意去学习设计模式，平时我们接触的很多代码里已经包含了一些设计模式的实现。我的过程是读过prototype和jquery的源码后，回头翻设计模式的书，发现不知觉中已经接触过十之六七。

同样在实际的编码中也没有必要刻意去使用一些设计模式。就如同tokyo hot 32式一样，在一场友好的papapa过程中，没有必要去刻意使用某种姿势。一切还是看需求和感觉。