## [前端攻略系列(一) - 前端各种优化(保证持续更新)](http://www.cnblogs.com/Darren_code/archive/2011/12/31/property.html)

2011-12-31 23:52 by 聂微东, 8127 阅读, 45 评论, [收藏](http://www.cnblogs.com/Darren_code/archive/2011/12/31/property.html), [编辑](http://www.cnblogs.com/Darren_code/admin/EditPosts.aspx?postid=2289248)

　　还有一个小时才到2012，那还能有点时间写点吐槽的东西，嘿嘿...2011年12月绝对是本人工作以来压力最大的一个月，已经忙到睡觉时间少了，业余看书时间少了，身体开始报警了，肩膀上的责任压的我实在喘不过气...作为一个普通的北漂，在北京类似我这样的人海的去了，尤其是在咱们这个行业。

　　我非常热爱生活，每分钟都无比珍贵；我热爱我现在所做的一切，哪怕确实让我精疲力尽(所以写博客暂时被我当成减压的方法)。

　　在我看来生活就是应该这样，既然我们在投胎的时候选择的生存模式or困难模式，那么不管发生什么困难都是意料之中。所以我在这不是想要表达自己的抱怨，而是想提醒自己和大家  “ **再忙再累也不要忘记自己的梦想和目标**”。

　　突然想起个例子不吐不快，关于我们80后应该都玩过的超级玛丽(mario)。不管你有多少生命多少金币，说到底都是浮云，游戏中你会碰到无数的敌人(乌龟、飞鱼、食人花...)，遇到无数的艰险(一个接一个的坑、下水道、火海...)，可是想要通关只有一个方法，那就是救到公主，这就是**目标**。

　　If you want to live a happy life, tie it to a goal. Not to people or things.

　　如果你想过得快乐，那就把生活和目标结合在一起，而不是来自他人或物质。

　　许久才出来冒一次泡，所以难免吐槽一下，不喜欢以上那段的就当看个热闹，下面咱们开始正题。

-------------------------------------------------我是分割线-------------------------------------------------

　　作为本人2011年的收尾文章，就来个总结性质的吧，^\_^...最近一直有给新同学做前端方面的培训，也有去参与公司前端的招聘，所以把自己资料库里面很多高效且有用的知识做了些规整分类，然后再分享一篇关于前端优化方面的总结。而且春节一过就又是招聘的高峰期了，在校的、跳槽的、浮躁的都开始进入面试的大军，前端优化也是前端面试一般会问到的东西，所以希望能对朋友们有用。

# 前端性能优化

**1.　　请减少HTTP请求**

**基本原理：**

　　在浏览器(客户端)和服务器发生通信时，就已经消耗了大量的时间，尤其是在网络情况比较糟糕的时候，这个问题尤其的突出。

　　一个正常HTTP请求的流程简述：如在浏览器中输入"www.xxxxxx.com"并按下回车，浏览器再与这个URL指向的服务器建立连接，然后浏览器才能向服务器发送请求信息，服务器在接受到请求的信息后再返回相应的信息，浏览器接收到来自服务器的应答信息后，对这些数据解释执行。

　　而当我们请求的网页文件中有很多图片、CSS、JS甚至音乐等信息时，将会频繁的与服务器建立连接，与释放连接，这必定会造成资源的浪费，且每个HTTP请求都会对服务器和浏览器产生性能负担。

　　网速相同的条件下，下载一个100KB的图片比下载两个50KB的图片要快。所以，**请减少HTTP请求。**

**解决办法：**

　　合并图片（css sprites），合并CSS和JS文件；图片较多的页面也可以使用 lazyLoad 等技术进行优化。

**2.　　请正确理解 Repaint 和 Reflow**

　　注：Repaint 和 Reflow 也就是重绘和重排，请允许我在这卖弄下我有限认识的那么几个英语单词...囧

**基本原理：**

　　Repaint(重绘)就是在一个元素的外观被改变，但没有改变布局(宽高)的情况下发生，如改变visibility、outline、背景色等等。

　　Reflow(重排)就是DOM的变化影响到了元素的几何属性（宽和高），浏览器会重新计算元素的几何属性，会使渲染树中受到影响的部分失效，浏览器会验证DOM树上的所有其它结点的visibility属性，这也是Reflow低效的原因。如：改变窗囗大小、改变文字大小、内容的改变、浏览器窗口变化，style属性的改变等等。如果Reflow的过于频繁，CPU使用率就会噌噌的往上涨，所以前端也就有必要知道 Repaint 和 Reflow的知识。

**减少性能影响的办法：**

　　上面提到通过设置style属性改变结点样式的话，每设置一次都会导致一次reflow，所以最好通过设置class的方式；　有动画效果的元素，它的position属性应当设为fixed或absolute，这样不会影响其它元素的布局；如果功能需求上不能设置position为fixed或absolute，那么就权衡速度的平滑性。

　　总之，因为 Reflow 有时确实不可避免，所以只能尽可能限制Reflow的影响范围。

**3.　　请减少对DOM的操作**

**基本原理：**

　　对DOM操作的代价是高昂的，这在网页应用中的通常是一个性能瓶颈。

　　天生就慢。在《高性能JavaScript》中这么比喻：“把DOM看成一个岛屿，把JavaScript(ECMAScript)看成另一个岛屿，两者之间以一座收费桥连接”。所以每次访问DOM都会教一个过桥费，而访问的次数越多，交的费用也就越多。所以一般建议尽量减少过桥次数。

**解决办法：**

　　修改和访问DOM元素会造成页面的Repaint和Reflow，循环对DOM操作更是罪恶的行为。所以请合理的使用JavaScript变量储存内容，考虑大量DOM元素中循环的性能开销，在循环结束时一次性写入。

　　减少对DOM元素的查询和修改，查询时可将其赋值给局部变量。

　　注：在IE中:hover会降低响应速度。

**4.　　使用JSON格式来进行数据交换**

**基本原理：**

　　JSON是一种轻量级的数据交换格式，采用完全独立于语言的文本格式，是理想的数据交换格式。同时，JSON是 JavaScript原生格式，这意味着在 JavaScript 中处理 JSON数据不需要任何特殊的 API 或工具包。

　　与XML序列化相比，JSON序列化后产生的数据一般要比XML序列化后数据体积小，所以在Facebook等知名网站中都采用了JSON作为数据交换方式。

**JS操作JSON：**

　　在JSON中，有两种结构：对象和数组。

　　1. 一个对象以 “ { ”  开始，“ } ” 结束。每个“名称”后跟一个 “ : ” ；“名称/值 对”之间使用 “ ,  ”（逗号）分隔。 名称用引号括起来；值如果是字符串则必须用引号括起来，数值型则不需要。如：

var obj={"name":"darren","age":24,"location":"beijing"}

　　2. 数组是值（value）的有序集合。一个数组以 “ [ ” 开始， “ ] ” 结束。值之间使用 “ , ” （逗号）分隔。如：

var jsonlist=[{"name":"darren","age":24,"location":"beijing"},{"name":"weidong.nie","age":24,"location":"hunan"}];

　　对这种数组和对象字面量的操作是非常方便且高效的。如果预先知道JSON结构的情况下，使用JSON进行数据传递简直是太美妙了，可以写出很实用美观可读性强的代码。如果你是纯粹的前台开发人员，一定会非常喜欢JSON。

**5.　　高效使用HTML标签和CSS样式**

**基本原理：**

　　HTML是一门用来描述网页的一种**语言**，它使用标记标签来描述网页，作为一名合格的前端开发，你有必要去知道其常用标签代表的含义(SEO)和属性(表现形式)。

　　CSS指层叠样式表 (Cascading Style Sheets)，如果说把页面想象成一个人，HTML就是人的骨架，CSS就是人的衣装，一个人的品味从他的衣装就能一目了然。

　　一名专业的前端开发也是一名优秀的重构，因为在页面中经常会有各种不合理的嵌套和重复定义的CSS样式，我不是要你重构页面，只是希望你在碰到这种情况的时候解决这些问题。如这样的HTML：

<table><tr><td>  
　　<table><tr><td>  
　　　　...  
　　</td></tr></table>  
</td></tr></table>

或者这样的CSS:

body .box .border ul li p strong span{color:#000}

 　　以上都是对HTML和CSS非常糟糕的使用方法。

**正确理解：**

　　HTML是一门标记语言，使用合理的HTML标签前你必须了解其属性，比如Flow Elements，Metadata Elements ，Phrasing Elements。比较基础的就是得知道块级元素和内联元素、盒模型、SEO方面的知识。

　　CSS是用来渲染页面的，也是存在渲染效率的问题。CSS选择符是从右向左进行匹配的，这里对css选择符按照开销从小到大的顺序梳理一下：

　　ID选择符 #box  
　　类选择符 .box   
　　标签 div  
　　伪类和伪元素 a:hover

　　当页面被触发引起回流（reflow）的时候，低效的选择符依然会引发更高的开销，所以请避免低效。

**6.　　使用CDN加速（内容分发网络）**

**基本原理：**

　　CDN的全称是Content Delivery Network，即内容分发网络。

　　"其基本思路是尽可能避开互联网上有可能影响数据传输速度和稳定性的瓶颈和环节，使内容传输的更快、更稳定。通过在网络各处放置节点服务器所构成的在现有的互联网基础之上的一层智能虚拟网络，CDN系统能够实时地根据网络流量和各节点的连接、负载状况以及到用户的距离和响应时间等综合信息将用户的请求重新导向离用户最近的服务节点上。" - 百度百科。

　　上面几句话有多少能安安心心看完的，所以我还是通过说故事再来介绍一遍吧，顺便补一句，故事出处不明，^\_^：

　　古代打仗大家一定都知道，由于古代的交通很不发达，所以当外族进攻的时候往往不能及时的反击，等朝廷征完兵再把兵派往边境的时候那些侵略者却是早已不见了踪影，这个让古代的帝王很是郁闷。后来帝王们学聪明了，都将大量的兵员提前派往边境驻扎，让他们平时屯田，战时当兵，这样的策略起到了很显著的作用。

**不足之处：**

　　实时性不太好是CDN的致命缺陷。随着对CDN需求的逐渐升温，这一缺陷将得到改进，使来自于远程服务器的网络内容网页与复本服务器或缓存器中的网页保持同步。解决方法是在网络内容发生变化时将新的网络内容从服务器端直接传送到缓存器，或者当对网络内容的访问增加时将数据源服务器的网络内容尽可能实时地复制到缓存服务器。

**7.　　将CSS和JS放到外部文件中引用，CSS放头，JS放尾**

**基本原理：**

　　注：这个是很基础且必须遵循的知识点，可是为了文章的完整性勉为其难加进来吧，嘿嘿。

　　引入外部文件好处是显而易见的，而且是项目稍稍复杂一点的时候就有必要了这样做了。

　　易维护、易扩展，方便管理和重复利用。

**正确的方式：**

　　JavaScript是浏览器中的霸主，为什么这么说，因为在浏览器在执行JavaScript代码时，不能同时做其它事情，即<script>每次出现都会让页面等待脚本的解析和执行（不论JavaScript是内嵌的还是外链的），JavaScript代码执行完成后，才继续渲染页面。这个也就是JavaScript的阻塞特性。

　　因为这个阻塞的特点，建议把JavaScript代码放到</body>标签以前，这样既能有效的防止JavaScript的阻塞，又能使得页面的HTML结构能更快的释放。

　　HTML规范清楚指出CSS要放包含在页面的<head>区域内，这里就不多解释了。

**8.　　精简CSS和JS文件**

**基本原理：**

　　有一条非常重要的准则一直没有提到，就是CSS和JavaScript的压缩，直接减少下载的文件体积。我个人经常使用的方式是使用 **YUI Compressor**，它的特点是：移除注释；移除额外的空格；细微优化；标识符替换。

　　YUI Compressor是java程序，如果你对java很熟悉的话可快速的上手使用yuicompressor.jar；如果你对java很陌生也没关系，一样可以使用YUI Compressor，下面介绍其使用方式。

**YUI Compressor的配置和使用：**

　　先配置使用环境：

　　1.先确保电脑中是否安装了JDK

　　2.再配置必要的环境变量(细节不能三两句说清，所以不知道如何设置还是搜索吧)

　　3.在cmd界面，输入javac可测试是否安装成功

　　使用方法可从cmd到进入yuicompressor.jar所在磁盘，我以自己的yuicompressor-2.4.2.jar为例：

　　1.压缩JS

　　java -jar yuicompressor-2.4.2.jar api.js > api.min.js

　　2.压缩CSS

　　java -jar yuicompressor-2.4.2.jar style.css > style.min.css

　　当然，还有另一种更傻瓜式的使用方式，赶兴趣的朋友自己可去多尝试下。

**9.　　压缩图片和使用图片Sprite技术**

**基本原理:**

　　注：其实压缩图片和图片精灵是两个方面的技术，可是既然都是关于图片的优化还是放到一块吧。

　　现在由于工作的细分，专业的前端工程师已经少有机会去切图了，可是关于图片压缩还是得略微了解，一般图片压缩的方式有：

　　1.缩小图片分辨率；

　　2.改变图片格式；

　　3.降低图片保存质量。

　　关于图片精灵(Sprite)技术就和我们工作直接相关，不管是在CSS中的图片还是在HTML结构中的图片都会产生HTTP请求，前端优化的第一条就是减少请求数，最直接有效的方法是使用图片精灵（CSS Sprite）。图片精灵就是把许多图片放到一张大图片里面，通过CSS来显示图片的一部分。

　　至于图片精灵的操作细节就不多做介绍了，网上相关内容很多。

**10.　　注意控制Cookie大小和污染**

**基本原理和使用方法：**

　　有关Cookie的基础和高级知识可以去看本人写过的一篇文章[《JavaScript 操作 Cookie》](http://www.cnblogs.com/Darren_code/archive/2011/11/24/Cookie.html)。

　　因为Cookie是本地的磁盘文件，每次浏览器都会去读取相应的Cookie，所以建议去除不必要的Coockie，使Coockie体积尽量小以减少对用户响应的影响；

　　使用Cookie跨域操作时注意在适应级别的域名上设置coockie以便使子域名不受其影响

　　Cookie是有生命周期的，所以请注意设置合理的过期时间，合理地Expire时间和不要过早去清除coockie，都会改善用户的响应时间。

　　时间有限，今年就先到这里吧，哈哈。愿大家新年快乐，工作顺利。我会在以后的工作中慢慢完善这篇文章，也请大家多提宝贵建议。

　　如果大家也有什么好的优化方式也请留言，到时会补充到这篇文章当中，先在此谢谢了。

# 参考资料

　　百度百科

<http://dev.w3.org/html5/markup/common-models.html>

<http://www.w3schools.com/>

**前端攻城师攻略系列**

　　作为一名前端攻城师，越来越感受到在公司受到的重视，身边也有越来越多的朋友加入到前端开发这个岗位上，有种说不出的骄傲感。

　　由于闲暇时间喜欢看书，所以曾萌生过希望写本书的想法，虽然现在自己还有很多的坑需要踩过，真要写书也是将来很久以后的事了，可是有打算先尝试写点与前端紧密相关的攻略，于是就有了“前端攻略系列”。

　　由于现在本人表达能力和技术能力上的欠缺，难免有理解不对的地方，也请大家多多谅解。

　　如果觉得此文还算用心，请劳驾点击右下角的推荐^.^ 。

[前端攻略系列(三) - javascript 设计模式（文章很长，请自备瓜子，水果和眼药水）](http://www.cnblogs.com/Darren_code/archive/2011/08/31/JavascripDesignPatterns.html)

2011-08-31 23:55 by 聂微东, 26612 阅读, 146 评论, [收藏](http://www.cnblogs.com/Darren_code/archive/2011/08/31/JavascripDesignPatterns.html), [编辑](http://www.cnblogs.com/Darren_code/admin/EditPosts.aspx?postid=2144316)

　　一直都在考虑这个月分享大家什么东西最好，原计划是打算写一些HTML5中JS方面的内容或者是AJAX方面的，可是自己由于表达能力，时间，还有个人工作方面的问题，这个还是等下个月再做分享吧^.^。

　　老规矩，开始正文以前先交代自己写这篇文章的目的和一些注意事项：

　　1.首先本人一直从事前端开发，所以除了JavaScript其他的语言了解不深，所以文章只会以JavaScript语言的角度去论证；

　　2.其实我个人在项目用过的模式也不多，对模式的概念的理解也没有那么抽象，所以最近在面试中如果面试官问到与模式相关的问题，自己感觉在对答过程中很郁闷，很多东西表达不清楚，于是就找了些相关资料，才会有这篇文章分享；

　　3.JavaScript模式与前端的工作和成长密不可分，因为这确实不是一个简单的话题，所以我只能尽力用简单表达和例子阐明，而且园子里有很多的高手，所以希望大家踊跃发言(由于水平有限，请大家多多指教，希望嘴下留情)；

　　4.由于这篇文章更多的只是想起到一个介绍和讲解的作用，并不打算对每种模式进行细致的分析，所以每种模式只用到一个至二个例子，可能会造成这个例子的表达并不是最优的或者不够全面，如果各位看官觉得不过瘾，可以再去查找相关资料；

　　5.做任何事都需要坚持，写博客也是一样，嘿嘿，每月至少一篇(文章确实较长，希望能对朋友们有所帮助，**重点部分在前言中有介绍，大家可以选择感兴趣的模式进行深入**)。

　　6.欢迎转载，不过请注明出处，谢谢。

**了解JavaScript设计模式我们需要知道的一些必要知识点：（内容相对基础，高手请跳过）**

**闭包：**关于闭包这个月在园子里有几篇不错的分享了，在这我也从最实际的地方出发，说说我的理解。

　　　　1.闭包最常用的方式就是返回一个**内联函数**（何为内联函数？就是在函数内部声明的函数）；

　　　　2.在JavaScript中有作用域和执行环境的问题，在函数内部的变量在函数外部是无法访问的，在函数内部却可以得到全局变量。由于种种原因，我们有时候需要得到函数内部的变量，可是用常规方法是得不到的，这时我们就可以创建一个闭包，用来在外部访问这个变量。

　　　　3.闭包的用途 主要就是上一点提到的读取函数内部变量，还有一个作用就是可以使这些变量一直保存在内存中。

　　　　4.使用闭包要注意，由于变量被保存在内存中，所以会对内存造成消耗，所以不能滥用闭包。解决方法是 在退出函数之前，将不使用的局部变量全部删除。

　　　　最后还是上一套闭包的代码吧，这样更直观。

1 　　function f(){  
 2 　　var n = 999;  
 3 　　function f1(){  
 4 　　 alert(n+=1);  
 5 　　 }  
 6 　　 return f1;  
 7 　　}  
 8 　　var result = f();  
 9 　　result(); // 1000  
10 　　result(); // 1001  
11 　　result(); // 1002

　　**封装：**通过将一个方法或者属性声明为私用的，可以让对象的实现细节对其他对象保密以降低对象之间的耦合程度，可以保持数据的完整性并对其修改方式加以约束，这样可以是代码更可靠，更易于调试。封装是面向对象的设计的基石。

　　尽管JavaScript是一门面向对象的语言，可它并不具备将成员声明为公用或私用的任何内部机制，所以我们只能自己想办法实现这种特性。下面还是通过一套完整的代码去分析，介绍什么是**私有**属性和方法，什么是**特权**属性和方法，什么是**公有**属性和方法，什么是**公有静态**属性和方法。

**私有属性和方法：**函数有作用域，在函数内用var 关键字声明的变量在外部无法访问，私有属性和方法本质就是你希望在对象外部无法访问的变量。

**特权属性和方法：**创建属性和方法时使用的this关键字，因为这些方法定义在构造器的作用域中，所以它们可以访问到私有属性和方法；只有那些需要直接访问私有成员的方法才应该被设计为特权方法。

**共有属性和方法：**直接链在prototype上的属性和方法，不可以访问构造器内的私有成员，可以访问特权成员，子类会继承所有的共有方法。

**共有静态属性和方法：**最好的理解方式就是把它想象成一个命名空间，实际上相当于把构造器作为命名空间来使用。

1 　　/\* -- 封装 -- \*/  
 2 　　var \_packaging =function(){  
 3 　　//私有属性和方法  
 4 　　var name ='Darren';  
 5 　　var method1 =function(){  
 6 　　 //...  
 7 　　}  
 8 　　//特权属性和方法  
 9 　　this.title ='JavaScript Design Patterns' ;  
10 　　this.getName =function(){  
11 　　 return name;  
12 　　}  
13 　　}  
14 　　//共有静态属性和方法  
15 　　\_packaging.\_name ='Darren code';  
16 　　\_packaging.alertName =function(){  
17 　　alert(\_packaging.\_name);  
18 　　}  
19 　　//共有属性和方法  
20 　　\_packaging.prototype = {  
21 　　init:function(){  
22 　　//...  
23 　　}  
24 　　}

　　**继承：**继承本身就是一个抽象的话题，在JavaScript中继承更是一个复杂的话题，因为JavaScript想要实现继承有两种实现方式，分别是类式继承和原型式继承，每种实现的方式都需要采取不少措施，下面本人通过分析例子的方式讲解JavaScript中这个很重要的话题。

1 　　/\* -- 类式继承 -- \*/  
 2 　　//先声明一个超类  
 3 　　function Person(name){  
 4 　　　　this.name = name;  
 5 　　}  
 6 　　//给这个超类的原型对象上添加方法 getName   
 7 　　Person.prototype.getName =function(){  
 8 　　returnthis.name;  
 9 　　}  
10 　　//实例化这个超类  
11 　　var a =new Person('Darren1')  
12 　　alert(a.getName());  
13 　　//再声明类  
14 　　function Programmer(name,sex){  
15 　　//这个类中要调用超类Person的构造函数，并将参数name传给它  
16 　　Person.call(this,name);  
17 　　this.sex = sex;  
18 　　}  
19　　 //这个子类的原型对象等于超类的实例  
20 　　Programmer.prototype =new Person();  
21 　　//因为子类的原型对象等于超类的实例，所以prototype.constructor这个方法也等于超类构造函数，你可以自己测试一下，如果没这一步，alert(Programmer.prototype.constructor)，这个是Person超类的引用，所以要从新赋值为自己本身  
22 　　Programmer.prototype.constructor = Programmer;  
23 　　//子类本身添加了getSex 方法  
24 　　Programmer.prototype.getSex =function(){  
25 　　returnthis.sex;  
26 　　}  
27 　　//实例化这个子类  
28 　　var \_m =new Programmer('Darren2','male');  
29 　　//自身的方法  
30 　　alert(\_m.getSex());  
31 　　//继承超类的方法  
32 　　alert(\_m.getName());

　　代码都不难，只要对 原型链 有基础就能理解。类式继承模式是JavaScript继承主要的模式，几乎所有用面向对象方式编写的JavaScript代码中都用到了这种继承，又因为在各种流行语言中只有JavaScript使用原型式继承，因此最好还是使用类式继承。可是要熟悉JavaScript语言，原型继承也是我们必须所了解的，至于在项目中是否使用就得看个人编码风格了。

1 　　/\* -- 原型式继承 -- \*/  
 2 　　//clone()函数用来创建新的类Person对象  
 3 　　var clone =function(obj){  
 4 var \_f =function(){};  
 5 　　**//这句是原型式继承最核心的地方，函数的原型对象为对象字面量**  
 6 　　\_f.prototype = obj;   
 7 　　returnnew \_f;  
 8 　　}  
 9 　　//先声明一个对象字面量  
10 　　var Person = {  
11 　　name:'Darren',  
12 　　getName:function(){  
13 　　returnthis.name;  
14 　　}  
15 　　}  
16 　　//不需要定义一个Person的子类，只要执行一次克隆即可  
17 　　var Programmer = clone(Person);  
18 　　//可以直接获得Person提供的默认值，也可以添加或者修改属性和方法  
19 　　alert(Programmer.getName())  
20 　　Programmer.name ='Darren2'  
21 　　alert(Programmer.getName())  
22   
23 　　//声明子类,执行一次克隆即可  
24 　　var Someone = clone(Programmer);

　　*------------------------------------------　　正文开始了，我是分割线　　------------------------------------------*  
   
 　**前言：**

　　JavaScript设计模式的作用 - 提高代码的重用性，可读性，使代码更容易的维护和扩展。

　　1.单体模式，工厂模式，桥梁模式个人认为这个一个优秀前端必须掌握的模式，对抽象编程和接口编程都非常有好处。

　　2.装饰者模式和组合模式有很多相似的地方，它们都与所包装的对象实现同样的接口并且会把任何方法的调用传递给这些对象。装饰者模式和组合模式是本人描述的较吃力的两个模式，我个人其实也没用过，所以查了很多相关资料和文档，请大家海涵。

　　3.门面模式是个非常有意思的模式，几乎所有的JavaScript库都会用到这个模式，假如你有逆向思维或者逆向编程的经验，你会更容易理解这个模式（听起来有挑战，其实一接触你就知道这是个很简单的模式）；还有配置器模式得和门面模式一块拿来说，这个模式对现有接口进行包装，合理运用可以很多程度上提高开发效率。这两个模式有相似的地方，所以一块理解的话相信都会很快上手的。

　　4.享元模式是一种以优化为目的的模式。

　　5.代理模式主要用于控制对象的访问，包括推迟对其创建需要耗用大量计算资源的类得实例化。

　　6.观察者模式用于对对象的状态进行观察，并且当它发生变化时能得到通知的方法。用于让对象对事件进行监听以便对其作出响应。观察者模式也被称为“订阅者模式”。

　　7.命令模式是对方法调用进行封装的方式，用命名模式可以对方法调用进行参数化和传递，然后在需要的时候再加以执行。

　　8.职责链模式用来消除请求的发送者和接收者之间的耦合。  
 

　　JavaScript设计模式都有哪些？

**单体（Singleton）模式： 绝对是JavaScript中最基本最有用的模式。**

　　单体在JavaScript的有多种用途，**它用来划分命名空间**。可以减少网页中全局变量的数量(在网页中使用全局变量有风险)；可以在多人开发时避免代码的冲突(使用合理的命名空间)等等。

　　在中小型项目或者功能中，单体可以用作命名空间把自己的代码组织在一个全局变量名下；在稍大或者复杂的功能中，单体可以用来把相关代码组织在一起以便日后好维护。

　　使用单体的方法就是用一个命名空间包含自己的所有代码的全局对象，示例：

1 　　var functionGroup = {  
2 　　　　name:'Darren',  
3 　　　　method1:function(){  
4 　　　　　　//code  
5 　　　　},  
6 　　　　init:function(){  
7 　　　　　　//code  
8 　　　　}  
9 　　}

　　或者

1 　　var functionGroup =newfunction myGroup(){  
2 　　　　this.name ='Darren';  
3 　　　　this.getName =function(){  
4 　　　　　　returnthis.name  
5 　　　　}  
6 　　　　this.method1 =function(){}  
7 　　　　...  
8 　　}

**工厂（Factory）模式：提供一个创建一系列相关或相互依赖对象的接口，而无需指定他们具体的类。**

　　工厂就是把成员对象的创建工作转交给一个外部对象，好处在于消除对象之间的耦合(何为耦合？就是相互影响)。**通过使用工厂方法而不是new关键字及具体类，可以把所有实例化的代码都集中在一个位置**，有助于创建模块化的代码，这才是工厂模式的目的和优势。

　　举个例子：你有一个大的功能要做，其中有一部分是要考虑扩展性的，那么这部分代码就可以考虑抽象出来，当做一个全新的对象做处理。好处就是将来扩展的时候容易维护 - 只需要操作这个对象内部方法和属性，达到了动态实现的目的。非常有名的一个示例 - **XHR工厂**：

1　　 var XMLHttpFactory =function(){};　　　　　　//这是一个简单工厂模式  
 2 　　XMLHttpFactory.createXMLHttp =function(){  
 3　　　　 var XMLHttp = null;  
 4 　　　　if (window.XMLHttpRequest){  
 5 　　　　　　XMLHttp = new XMLHttpRequest()  
 6　　　　 }elseif (window.ActiveXObject){  
 7 　　　　　　XMLHttp = new ActiveXObject("Microsoft.XMLHTTP")  
 8 　　　　}  
10 　　return XMLHttp;  
11 　　}  
12 　　//XMLHttpFactory.createXMLHttp()这个方法根据当前环境的具体情况返回一个XHR对象。  
13 　　var AjaxHander =function(){  
14 　　　　var XMLHttp = XMLHttpFactory.createXMLHttp();  
15 　　　　...  
16 　　}

　　工厂模式又区分简单工厂模式和抽象工厂模式，上面介绍的是简单工厂模式，这种模式用的更多也更简单易用。抽象工厂模式的使用方法就是 - 先设计一个抽象类，这个类不能被实例化，只能用来派生子类，最后通过对子类的扩展实现工厂方法。 示例：

1 　　var XMLHttpFactory =function(){};　 //这是一个抽象工厂模式  
 2 　　XMLHttpFactory.prototype = {  
 3 　　//如果真的要调用这个方法会抛出一个错误，它不能被实例化，只能用来派生子类  
 4 　　createFactory:function(){  
 5 　　thrownew Error('This is an abstract class');  
 6 　　}  
 7 　　}  
 8 　　//派生子类，文章开始处有基础介绍那有讲解继承的模式，不明白可以去参考原理  
 9 　　var XHRHandler =function(){  
10 　　XMLHttpFactory.call(this);  
11 　　};  
12 　　XHRHandler.prototype =new XMLHttpFactory();  
13 　　XHRHandler.prototype.constructor = XHRHandler;  
14 　　//重新定义createFactory 方法  
15 　　XHRHandler.prototype.createFactory =function(){  
16 　　var XMLHttp =null;  
17 　　if (window.XMLHttpRequest){  
18 　　XMLHttp =new XMLHttpRequest()  
19 　　}elseif (window.ActiveXObject){  
20 　　XMLHttp =new ActiveXObject("Microsoft.XMLHTTP")  
21 　　}  
22 　　return XMLHttp;  
23 　　}

**桥接（bridge）模式：在实现API的时候，桥梁模式灰常有用。在所有模式中，这种模式最容易立即付诸实施。**

　　桥梁模式可以用来弱化它与使用它的类和对象之间的耦合，就是将抽象与其实现隔离开来，以便二者独立变化；这种模式对于JavaScript中常见的时间驱动的编程有很大益处，桥梁模式最常见和实际的应用场合之一是时间监听器回调函数。先分析一个不好的示例：

1 　　element.onclick =function(){  
2 　　new setLogFunc();  
3 　　};

　　为什么说这个示例不好，因为从这段代码中无法看出那个LogFunc方法要显示在什么地方，它有什么可配置的选项以及应该怎么去修改它。换一种说法就是，桥梁模式的要诀就是让接口“可桥梁”，实际上也就是可配置。把页面中一个个功能都想象成模块，接口可以使得模块之间的耦合降低。

　　掌握桥梁模式的正确使用收益的不只是你，还有那些负责维护你代码的人。把抽象于其实现隔离开，可独立地管理软件的各个部分，bug也因此更容易查找。

　　桥梁模式目的就是让API更加健壮，提高组件的模块化程度，促成更简洁的实现，并提高抽象的灵活性。一个好的示例：

1 　　element.onclick =function(){　　//API可控制性提高了，使得这个API更加健壮  
2 　　new someFunction(element,param,callback);  
3 　　}

　　注：桥梁模式还可以用于连接公开的API代码和私有的实现代码，还可以把多个类连接在一起。在文章封装介绍的部分提到过**特权方法**，也是桥梁模式的一种特例。《JS设计模式》上找的示例，加深大家对这个模式的理解：

1 　　//错误的方式  
 2 　　//这个API根据事件监听器回调函数的工作机制，事件对象被作为参数传递给这个函数。本例中并没有使用这个参数，而只是从this对象获取ID。  
 3 　　addEvent(element,'click',getBeerById);  
 4 　　function(e){  
 5 　　var id =this.id;  
 6 　　asyncRequest('GET','beer.url?id='+ id,function(resp){  
 7 　　//Callback response  
 8 　　 console.log('Requested Beer: '+ resp.responseText);  
 9 　　});  
10 　　}  
11   
12 　　//好的方式  
13 　　//从逻辑上分析，把id传给getBeerById函数式合情理的，且回应结果总是通过一个毁掉函数返回。这么理解，我们现在做的是针对接口而不是实现进行编程，用桥梁模式把抽象隔离开来。  
14 　　function getBeerById(id,callback){  
15 　　asyncRequest('GET','beer.url?id='+ id,function(resp){  
16 　　callback(resp.responseText)  
17 　　});  
18 　　}  
19 　　addEvent(element,'click',getBeerByIdBridge);  
20 　　function getBeerByIdBridge(e){  
21 　　getBeerById(this.id,function(beer){  
22 　　console.log('Requested Beer: '+ beer);  
23 　　});  
24 　　}

**装饰者（Decorator）模式：这个模式就是为对象增加功能(或方法)。**

**动态地给一个对象添加一些额外的职责。就扩展功能而言，它比生成子类方式更为灵活。**

**装饰者模式和组合模式有很多共同点，它们都与所包装的对象实现统一的接口并且会把任何方法条用传递给这些对象。可是组合模式用于把众多子对象组织为一个整体，而装饰者模式用于在不修改现有对象或从派生子类的前提下为其添加方法。**

**装饰者的运作过程是透明的，这就是说你可以用它包装其他对象，然后继续按之前使用那么对象的方法来使用，从下面的例子中就可以看出。还是从代码中理解吧：**

1 　　//创建一个命名空间为myText.Decorations  
 2 　　var myText= {};  
 3 　　myText.Decorations={};  
 4 　　myText.Core=function(myString){  
 5 　　this.show =function(){return myString;}  
 6 　　}  
 7 　　//第一次装饰  
 8 　　myText.Decorations.addQuestuibMark =function(myString){  
 9 　　this.show =function(){return myString.show()+'?';};  
10 　　}  
11 　　//第二次装饰  
12 　　myText.Decorations.makeItalic =function(myString){  
13 　　this.show =function(){return'<li>'+myString.show()+'</li>'};  
14 　　}  
15 　　//得到myText.Core的实例  
16 　　var theString =new myText.Core('this is a sample test String');  
17 　　alert(theString.show());　　//output 'this is a sample test String'  
18 　　theString =new myText.Decorations.addQuestuibMark(theString);  
19 　　alert(theString.show());　　//output 'this is a sample test String?'  
20 　　theString =new myText.Decorations.makeItalic (theString);  
21 　　alert(theString.show());　　//output '<li>this is a sample test String</li>'

　　从这个示例中可以看出，这一切都可以不用事先知道组件对象的接口，甚至可以动态的实现，在为现有对象增添特性这方面，装饰者模式有极大的灵活性。

　　如果需要为类增加特性或者方法，而从该类派生子类的解决办法并不实际的话，就应该使用装饰者模式。派生子类之所以会不实际最常见的原因是需要添加的特性或方法的数量要求使用大量子类。

**组合（Composite）模式：将对象组合成树形结构以表示“部分-整体”的层次结构。它使得客户对单个对象和复合对象的使用具有一致性。**

　　组合模式是一种专为创建Web上的动态用户界面而量身定制的模式。使用这种模式，可以用一条命令在多个对象上激发复杂的或递归的行为。组合模式擅长于对大批对象进行操作。

　　组合模式的好处：1.程序员可以用同样的方法处理对象的集合与其中的特定子对象；2.它可以用来把一批子对象组织成树形结构，并且使整棵树都可被便利。

　　组合模式适用范围：1.存在一批组织成某处层次体系的对象（具体结构可能在开发期间无法知道）；2.希望对这批对象或其中的一部分对象实话一个操作。

　　其实组合模式就是将一系列相似或相近的对象组合在一个大的对象，由这个大对象提供一些常用的接口来对这些小对象进行操作，代码可重用，对外操作简单。例如：对form内的元素，不考虑页面设计的情况下，一般就剩下input了，对于这些input都有name和value的属性，因此可以将这些input元素作为form对象的成员组合起来，form对象提供对外的接口，便可以实现一些简单的操作，比如设置某个input的value，添加/删除某个input等等。

　　这种模式描述起来比较吃力，我从《JS设计模式》上找个一个实例，大家还是看代码吧：**先创建组合对象类**

1 　　// DynamicGallery Class  
 2 　　var DynamicGallery =function (id) { // 实现Composite，GalleryItem组合对象类   
 3 　　this.children = [];  
 4 　　this.element = document.createElement('div');  
 5 　　this.element.id = id;  
 6 　　this.element.className ='dynamic-gallery';  
 7 　　}  
 8 　　DynamicGallery.prototype = {  
 9 　　// 实现Composite组合对象接口   
10 　　add: function (child) {  
11 　　this.children.push(child);  
12 　　this.element.appendChild(child.getElement());  
13 　　},  
14 　　remove: function (child) {  
15 　　for (var node, i =0; node =this.getChild(i); i++) {  
16 　　if (node == child) {  
17 　　this.children.splice(i, 1);  
18 　　 break;  
19 　　}  
20 　　}  
21 　　this.element.removeChild(child.getElement());  
22 　　},  
23 　　getChild: function (i) {  
24 　　returnthis.children[i];  
25 　　},  
26 　　// 实现DynamicGallery组合对象接口   
27 　　hide: function () {  
28 　　for (var node, i =0; node =this.getChild(i); i++) {  
29 　　node.hide();  
30 　　}  
31 　　this.element.style.display ='none';  
32 　　},  
33 　　show: function () {  
34 　　this.element.style.display ='block';  
35 　　for (var node, i =0; node = getChild(i); i++) {  
36 　　node.show();  
37 　　}  
38 　　},  
39 　　// 帮助方法   
40 　　getElement: function () {  
41 　　returnthis.element;  
42 　　}  
43 　　}

　　再创建叶对象类

1 　　var GalleryImage =function (src) { // 实现Composite和GalleryItem组合对象中所定义的方法   
 2 　　this.element = document.createElement('img');  
 3 　　this.element.className ='gallery-image';  
 4 　　this.element.src = src;  
 5 　　}  
 6 　　GalleryImage.prototype = {  
 7 　　// 实现Composite接口   
 8 　　// 这些是叶结点，所以我们不用实现这些方法，我们只需要定义即可   
 9 　　add: function () { },  
10 　　remove: function () { },  
11 　　getChild: function () { },  
12 　　// 实现GalleryItem接口   
13 　　hide: function () {  
14 　　this.element.style.display ='none';  
15 　　},  
16 　　show: function () {  
17 　　this.element.style.display ='';  
18 　　},  
19 　　// 帮助方法   
20 　　getElement: function () {  
21 　　returnthis.element;  
22 　　}  
23 　　}

　　现在我们可以使用这两个类来管理图片：

1 　　var topGallery =new DynamicGallery('top-gallery');   
 2 　　topGallery.add(new GalleryImage('/img/image-1.jpg'));   
 3 　　topGallery.add(new GalleryImage('/img/image-2.jpg'));   
 4 　　topGallery.add(new GalleryImage('/img/image-3.jpg'));   
 5 　　var vacationPhotos =new DyamicGallery('vacation-photos');   
 6 　　for(var i =0, i <30; i++){   
 7 　　　　vacationPhotos.add(new GalleryImage('/img/vac/image-'+ i +'.jpg'));   
 8 　　}   
 9 　　topGallery.add(vacationPhotos);   
10 　　topGallery.show();   
11 　　vacationPhotos.hide();

**门面（facade）模式：门面模式是几乎所有JavaScript库的核心原则**　　子系统中的一组接口提供一个一致的界面，门面模式定义了一个高层接口，这个接口使得这一子系统更加容易使用，简单的说这是一种组织性的模式，它可以用来修改类和对象的接口，使其更便于使用。

　　门面模式的两个作用：1.简化类的接口；2.消除类与使用它的客户代码之间的耦合。

　　门面模式的使用目的就是图方面。

　　想象一下计算机桌面上的那些快捷方式图标，它们就是在扮演一个把用户引导至某个地方的接口的角色，每次操作都是间接的执行一些幕后的命令。

　　你在看这篇的博客的时候我就假设你已经有JavaScript的使用经验了，那么你一定写过或者看过这样的代码：

1 　　var addEvent =function(el,type,fn){  
2 　　if(window.addEventListener){  
3 　　el.addEventListener(type,fn);  
4 　　}elseif(window.attachEvent){  
5 　　el.attachEvent('on'+type,fn);  
6 　　}else{  
7 　　el['on'+type] = fn;  
8 　　}  
9 　　}

　　这个就是一个JavaScript中常见的事件监听器函数，这个函数就是一个基本的门面，有了它，就有了为DOM节点添加事件监听器的简便方法。

　　现在要说门面模式的精华部分了，为什么说JavaScript库几乎都会用这种模式类。假如现在要设计一个库，那么最好把其中所有的工具元素放在一起，这样更好用，访问起来更简便。看代码：

1 　　//\_model.util是一个命名空间  
 2 　　\_myModel.util.Event = {  
 3 　　getEvent:function(e){  
 4 　　return e|| window.event;  
 5 　　},  
 6 　　getTarget:function(e){  
 7 　　return e.target||e.srcElement;  
 8 　　},  
 9 　　preventDefault:function(e){  
10 　　if(e.preventDefault){  
11 　　e.preventDefault();  
12 　　}else{  
13 　　e.returnValue =false;  
14 　　}  
15 　　}  
16 　　};  
17 　　//事件工具大概就是这么一个套路，然后结合addEvent函数使用  
18 　　addEvent(document.getElementsByTagName('body')[0],'click',function(e){  
19 　　alert(\_myModel.util.Event.getTarget(e));  
20 　　});

　　个人认为，在处理游览器差异问题时最好的解决办法就是把这些差异抽取的门面方法中，这样可以提供一个更一致的接口，addEvent函数就是一个例子。

**适配置器（Adapter）模式：**将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作，使用这种模式的对象又叫包装器，因为他们是在用一个新的接口包装另一个对象。

　　从表面上看，它和门面模式有点相似，差别在于它们如何改变接口，门面模式展现的是一个简化的接口，它并不提供额外的选择，而适配器模式则要把一个接口转换为另一个接口，它并不会滤除某些能力，也不会简化接口。先来一个简单的示例看看：

1 　　//假如有一个3个字符串参数的函数，但是现在拥有的却是一个包含三个字符串元素的对象，那么就可以用一个配置器来衔接二者  
 2 　　var clientObject = {  
 3 　　str1:'bat',  
 4 　　str2:'foo',  
 5 　　str3:'baz'  
 6 　　}  
 7 　　function interfaceMethod(str1,str2,str3){  
 8 　　　　alert(str1)  
 9 　　}  
10 　　//配置器函数  
11 　　function adapterMethod(o){  
12 　　interfaceMethod(o.str1, o.str2, o.str3);  
13 　　}  
14 　　adapterMethod(clientObject)  
15 　　//adapterMethod函数的作为就在于对interfaceMethod函数进行包装，并把传递给它的参数转换为后者需要的形式。

　　适配器模式的工作机制是：用一个新的接口对现有类得接口进行包装。

　　示例：适配两个库。下面的例子要实现的是从Prototype库的$函数到YUI的get方法的转换。

1 　　//先看它们在接口方面的差别  
 2 　　//Prototype $ function   
 3 　　function $(){  
 4 　　var elements =new Array();  
 5 　　for(var i=0;i<arguments.length;i++){  
 6 　　var element = arguments[i];  
 7 　　if(typeof element =='string'){  
 8 　　element = document.getElementById(element);  
 9 　　}  
10 　　if(typeof.length ==1) return element;  
11 　　elements.push(element);  
12 　　}  
13 　　return elements;  
14 　　}  
15 　　//YUI get method  
16 　　YAHOO.util.Dom.get =function(el){  
17 　　if(YAHOO.lang.isString(el)){  
18 　　return document.getElementById(el);  
19 　　}  
20 　　if(YAHOO.lang.isArray(el)){  
21 　　var c =[];  
22 　　for(var i=0,len=el.length;i<len;++i){  
23 　　c[c.length] = YAHOO.util.Dom.get(el[i]);  
24 　　}  
25 　　return c;  
26 　　}  
27 　　if(el){  
28 　　return el;  
29 　　}  
30 　　returnnull;  
31 　　}  
32 　　//二者区别就在于get具有一个参数，且可以是HTML,字符串或者数组；而$木有正是的参数，允许使用者传入任意数目的参数，不管HTML还是字符串。  
33 　　//如果需要从使用Prototype的$函数改为使用YUI的get方法（或者相反，那么用适配器模式其实很简单）  
34 　　function PrototypeToYUIAdapter(){  
35 　　return YAHOO.util.Dom.get(arguments);  
36 　　}  
37 　　function YUIToPrototypeAdapter(el){  
38 　　return $.apply(window,el instanceof Array?el:[el]);  
39 　　}

**享元（Flyweight）模式：运用共享技术有效地支持大量细粒度的对象。**

**享元模式可以避免大量非常相似类的开销。在程序设计中有时需要生成大量细粒度的类实例来表示数据。如果发现这些实例除了几个参数外基本伤都是相同的，有时就能够受大幅度第减少需要实例化的类的数量。如果能把这些参数移到类实例外面，在方法调用时将他们传递进来，就可以通过共享大幅度地减少单个实例的数目。**

**从实际出发说说自己的理解吧。**

1 　　**组成部分**  
 2 　　“享元”：抽离出来的外部操作和数据；  
 3 　　“工厂”：创造对象的工厂；  
 4 　　“存储器”：存储实例对象的对象或数组，供“享元”来统一控制和管理。  
 5   
 6 　　**应用场景**  
 7 　　1. 页面存在大量资源密集型对象；  
 8 　　2. 这些对象具备一定的共性，可以抽离出公用的操作和数据  
 9   
10 　　**关键**  
11 　　1. 合理划分内部和外部数据。  
12 　　既要保持每个对象的模块性、保证享元的独立、可维护，又要尽可能多的抽离外部数据。  
13 　　2. 管理所有实例  
14　　 既然抽离出了外部数据和操作，那享元就必须可以访问和控制实例对象。在JavaScript这种动态语言中，这个需求是很容易实现的：我们可以把工厂生产出的对象简单的扔在一个数组中。为每个对象设计暴露给外部的方法，便于享元的控制。  
15   
16 　　**优点**  
17 　　1. 将能耗大的操作抽离成一个，在资源密集型系统中，可大大减少资源和内存占用；  
18 　　2. 职责封装，这些操作独立修改和维护；  
19   
20 　　**缺点**  
21　　 1. 增加了实现复杂度。  
22 　　将原本由一个工厂方法实现的功能，修改为了一个享元+一个工厂+一个存储器。  
23 　　2. 对象数量少的情况，可能会增大系统开销。

　　示例：

1 　　//汽车登记示例  
 2 　　var Car =function(make,model,year,owner,tag,renewDate){  
 3 　　　　this.make=make;  
 4 　　　　this.model=model;  
 5 　　　　this.year=year;  
 6 　　　　this.owner=owner;  
 7 　　　　this.tag=tag;  
 8 　　　　this.renewDate=renewDate;  
 9 　　}  
10 　　Car.prototype = {  
11 　　　　getMake:function(){  
12 　　　　　　returnthis.make;  
13 　　　　},  
14 　　　　getModel:function(){  
15 　　　　　　returnthis.model;  
16 　　　　},  
17 　　　　getYear:function(){  
18 　　　　　　returnthis.year;  
19 　　　　},  
20 　　　　transferOwner:function(owner,tag,renewDate){  
21 　　　　　　this.owner=owner;  
22 　　　　　　this.tag=tag;  
23 　　　　　　this.renewDate=renewDate;  
24 　　　　},  
25 　　　　renewRegistration:function(renewDate){  
26 　　　　　　this.renewDate=renewDate;  
27 　　　　}  
28 　　}  
29 　　//数据量小到没多大的影响，数据量大的时候对计算机内存会产生压力，下面介绍享元模式优化后  
30 　　//包含核心数据的Car类  
31 　　var Car=function(make,model,year){  
32 　　　　this.make=make;  
33 　　　　this.model=model;  
34 　　　　this.year=year;  
35 　　}  
36 　　Car.prototype={  
37 　　　　getMake:function(){  
38 　　　　　　returnthis.make;  
39 　　　　},  
40 　　　　getModel:function(){  
41 　　　　　　returnthis.model;  
42 　　　　},  
43 　　　　getYear:function(){  
44 　　　　　　returnthis.year;  
45 　　　　}  
46 　　}  
47 　　//中间对象，用来实例化Car类  
48 　　var CarFactory=(function(){  
49 　　　　var createdCars = {};  
50 　　　　return {  
51 　　　　　　createCar:function(make,model,year){  
52 　　　　　　　　var car=createdCars[make+"-"+model+"-"+year];  
53 　　　　　　　　return car ? car : createdCars[make +'-'+ model +'-'+ year] =(new Car(make,model,year));  
54 　　　　　　}  
55 　　　　}  
56 　　})();  
57 　　//数据工厂，用来处理Car的实例化和整合附加数据  
58 　　var CarRecordManager = (function() {  
59 　　　　var carRecordDatabase = {};  
60 　　　　return {  
61 　　　　　　addCarRecord:function(make,model,year,owner,tag,renewDate){  
62 　　　　　　　　var car = CarFactory.createCar(make, model, year);  
63 　　　　　　　　carRecordDatabase[tag]={  
64 　　　　　　　　　　owner:owner,  
65 　　　　　　　　　　tag:tag,  
66 　　　　　　　　　　renewDate:renewDate,  
67 　　　　　　　　　　car:car  
68 　　　　　　}  
69 　　　　},  
70 　　　　　　transferOwnership:function(tag, newOwner, newTag, newRenewDate){  
71 　　　　　　　　var record=carRecordDatabase[tag];  
72 　　　　　　　　record.owner = newOwner;  
73 　　　　　　　　record.tag = newTag;  
74 　　　　　　　　record.renewDate = newRenewDate;  
75 　　　　　　},  
76 　　　　　　renewRegistration:function(tag,newRenewDate){  
77 　　　　　　　　carRecordDatabase[tag].renewDate=newRenewDate;  
78 　　　　　　},  
79 　　　　　　getCarInfo:function(tag){  
80 　　　　　　　　return carRecordDatabase[tag];  
81 　　　　　　}  
82 　　　　}  
83 　　})();

**代理（Proxy）模式：此模式最基本的形式是对访问进行控制。代理对象和另一个对象（本体）实现的是同样的接口，可是实际上工作还是本体在做，它才是负责执行所分派的任务的那个对象或类，代理对象不会在另以对象的基础上修改任何方法，也不会简化那个对象的接口。**

**举一个具体的情况：如果那个对象在某个远端服务器上，直接操作这个对象因为网络速度原因可能比较慢，那我们可以先用Proxy来代替那个对象。  
　　总之对于开销较大的对象，只有在使用它时才创建，这个原则可以为我们节省很多内存。《JS设计模式》上的图书馆示例：**

1 　　var Publication =new Interface('Publication', ['getIsbn', 'setIsbn', 'getTitle', 'setTitle', 'getAuthor', 'setAuthor', 'display']);  
 2　　 var Book =function(isbn, title, author) {  
 3 　　 //...  
 4 　　}   
 5 　　// implements Publication  
 6 　　implements(Book,Publication);  
 7   
 8 　　/\* Library interface. \*/  
 9 　　var Library =new Interface('Library', ['findBooks', 'checkoutBook', 'returnBook']);  
10   
11 　　/\* PublicLibrary class. \*/  
12 　　var PublicLibrary =function(books) {  
13 　　 //...  
14 　　};  
15 　　// implements Library  
16 　　implements(PublicLibrary,Library);   
17   
18 　　PublicLibrary.prototype = {  
19 　　 findBooks: function(searchString) {  
20 　　 //...  
21 　　},  
22 　　checkoutBook: function(book) {  
23 　　 //...  
24 　　 },  
25 　　returnBook: function(book) {  
26 　　 //...  
27 　　 }  
28 　　};  
29   
30 　　/\* PublicLibraryProxy class, a useless proxy. \*/  
31 　　var PublicLibraryProxy =function(catalog) {   
32 　　 this.library =new PublicLibrary(catalog);  
33 　　};  
34 　　// implements Library  
35 　　implements(PublicLibraryProxy,Library);  
36   
37 　　PublicLibraryProxy.prototype = {  
38 　　 findBooks: function(searchString) {  
39 　　 returnthis.library.findBooks(searchString);  
40 　　},  
41 　　checkoutBook: function(book) {  
42 　　 returnthis.library.checkoutBook(book);  
43 　　 },  
44 　　 returnBook: function(book) {  
45 　　 returnthis.library.returnBook(book);  
46 　　 }  
47 　　};

**观察者（Observer）模式：定义对象间的一种一对多的依赖关系，以便当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动刷新。**

**观察者模式中存在两个角色,观察者和被观察者。在DOM的编程环境中的高级事件模式中，事件监听器说到底就是一种内置的观察者。事件处理器(handler)和时间监听器(listener)并不是一回事，前者就是一种把事件传给与其关联的函数的手段，而在后者中，一个时间可以与几个监听器关联，每个监听器都能独立于其他监听器而改变。**

1 　　//使用时间监听器可以让多个函数相应一个事件  
 2 　　var fn1 =function(){  
 3 　　//code  
 4 　　}  
 5 　　var fn2 =function(){  
 6 　　//code  
 7 　　}  
 8 　　addEvent(element,'click',fn1)；  
 9 　　addEvent(element,'click',fn2)  
10   
11　　 //而时间处理函数就办不到  
12 　　element.onclick = fn1;  
13 　　element.onclick = fn2;

　　观察者模式是开发基于行为的应用程序的有力手段，前端程序员可做的就是借助一个事件监听器替你处理各种行为，从而降低内存消耗和提高互动性能。

**命令（Command）模式：将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可取消的操作。**

**命令对象是一个操作和用来调用这个操作的对象的结合体，所有的命名对象都有一个执行操作，其用途就是调用命令对象所绑定的操作。示例：**

1 　　car Calculator={  
 2 　　add:function(x,y){  
 3 　　 return x+y;  
 4 　　},  
 5 　　substract:function(x,y){  
 6 　　 return x-y;  
 7 　　},  
 8 　　multiply:function(x,y){  
 9 　　 return x\*y;  
10 　　},  
11 　　divide:function(x,y){  
12 　　 return x/y;  
13 　　}  
14 　　}  
15 　　Calculator.calc =function(command){  
16 　　return Calculator[command.type](command.op1,command.opd2)  
17 　　};  
18 　　Calculator.calc({type:'add',op1:1,op2:1});  
19 　　Calculator.calc({type:'substract',op1:5,op2:2});  
20 　　Calculator.calc({type:'multiply',op1:5,op2:2});  
21 　　Calculator.calc({type:'divide',op1:8,op2:4});

**命名模式的主要用途是把调用对象（用户界面，API和代理等）与实现操作的对象隔离开，也就是说使对象间的互动方式需要更高的模块化时都可以用到这种模式。**

**职责链（Chain Of Responsibility）模式：为解除请求的发送者和接收者之间耦合，而使多个对象都有机会处理这个请求。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它。**

**职责链由多个不同类型的对象组成：发送者是发出请求的对象，而接收者则是接收请求并且对其进行处理或传递的对象，请求本身有时也是一个对象，它封装着与操作有关的所有数据。**

**典型的流程大致是：**

　　1.发送者知道链中第一个接收者，它向这个接收者发出请求。

　　2.每一个接收者都对请求进行分析，然后要么处理它，要么将其往下传。

　　3.每一个接收者知道的其他对象只有一个，即它在链中的下家。

　　4.如果没有任何接收者处理请求，那么请求将从链上离开，不同的实现对此也有不同的反应，一般会抛出一个错误。

**职责链模式的适用范围:**1.有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定；2.想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求；3.可处理一个请求的对象集合需要被动态指定。

**确实对这种模式不了解，相关资料也较少，所以代码先不上了。看看大家对这个模式有木有什么好的理解或者能较好表达这种模式的代码，谢谢了。**

**结束语：**

　　1.**每种模式都有自己的优缺点**，所以每种模式的正确使用还得看开发人员本身的功力；

　　2.就算不使用JavaScript设计模式一样可以写出 **复杂的可使用** 的代码，可是如果你想真正了解JavaScript面向对象能力，学习提高代码的模块化程度﹑可维护性﹑可靠性和效率，那么合理的运用JavaScript设计模式将会是一个优秀 **前端开发攻城湿** 必备的能力。

　　终于在8月31日发了这篇博客，字很多，辛苦大家了...

　　最后，祝大家工作顺利。