JavaScript著名专家撰写

指导读者进入JavaScript框架设计的魔法指南

JavaScript 框架设计

司徒正美 ◎ 编著

人民邮电出版社
POSTS & TELECOM PRESS

JavaScript 框架设计

乔布斯说过：“真正的魔法，是用五千个点子打磨出一个产品。”我想我们很幸运，因为我们正要去做这样一件事：凝聚前人的智慧，实现一个真正的魔法。本书将引导大家了解近10年来大师们打造的优良框架以及其中令人称道的奇思妙想。让我们更深入、更彻底地认识JavaScript，领略jQuery等库的架构之美和设计之美，高屋建瓴地打造适合自己的前端框架！

作者简介

钟钦成 网名司徒正美，著名的JavaScript专家，立志做考古学家的日语系工程师，穿梭于二次元与二进制间的“魔法师”，做过陶艺，写过小说，涉猎Java、Ruby、 JavaScript，3年成就此书！

推荐媒介

博客园 cnblogs.com　51CTO.com 技术成就梦想　ITeYe

封面图介绍

圣家族大教堂，西班牙巴塞罗那最负盛名的观光胜地和地标性建筑，也是世界上唯一一座尚未完成便被列入世界文化遗产的建筑。整体设计以大自然诸如洞穴、山脉、花草、动物为灵感，是一个充满韵律动感的神奇建筑。

人民邮电出版社－信息技术分社
http://weibo.com/ptpitbooks

ISBN 978-7-115-34358-1

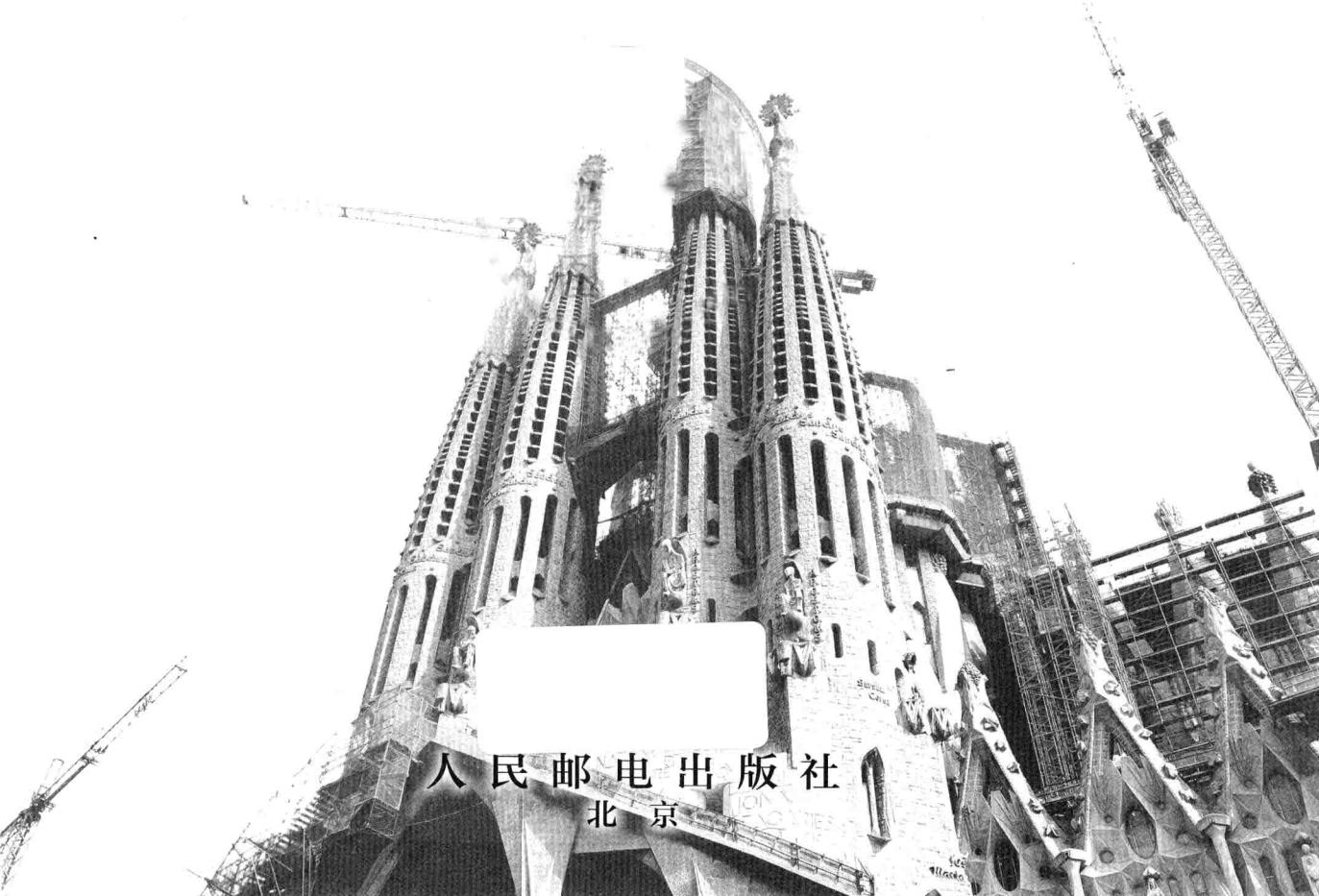
分类建议：计算机／程序设计／JavaScript
人民邮电出版社网址：www.ptpress.com.cn

封面设计：任文杰

ISBN 978-7-115-34358-1
定价：89.00 元

JavaScript 框架设计

司徒正美 ◎ 编著



人民邮电出版社
北京

图书在版编目（CIP）数据

JavaScript框架设计 / 司徒正美编著. -- 北京 : 人民邮电出版社, 2014.4
ISBN 978-7-115-34358-1

Ⅰ. ①J… Ⅱ. ①司… Ⅲ. ①JAVA语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第316783号

内 容 提 要

本书是一本全面讲解 JavaScript 框架设计的图书，详细地讲解了设计框架需要具备的知识，主要包括的内容为：框架与库、JavaScript 框架分类、JavaScript 框架的主要功能、种子模块、模块加载系统、语言模块、浏览器嗅探与特征侦测、样式的支持侦测、类工厂、JavaScript 对类的支撑、选择器引擎、浏览器内置的寻找元素的方法、节点模块、一些有趣的元素节点、数据缓存系统、样式模块、个别样式的特殊处理、属性模块、jQuery 的属性系统、事件系统、异步处理、JavaScript 异步处理的前景、数据交互模块、一个完整的 Ajax 实现、动画引擎、API 的设计、插件化、当前主流 MVVM 框架介绍、监控数组与子模板等。

本书适合前端设计人员、JavaScript 开发者、移动 UI 设计者、程序员和项目经理阅读，也可作为大中专院校相关专业的师生学习用书和培训学校的教材。

◆ 编　　著　司徒正美
责任编辑　张　涛
责任印制　程彦红　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京天宇星印刷厂印刷

◆ 开本：800×1000　1/16
印张：28.75
字数：701 千字　　2014 年 4 月第 1 版
印数：1－3 500 册　　2014 年 4 月北京第 1 次印刷

定价：89.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

框架则是一个半成品的应用，直接给出一个骨架，还例如盖房子，照着图纸砌砖、铺地板与涂漆就行了。在后端 Java 的三大框架中，程序员基本上就是与 XML 打交道，用配置就可以处理 80% 的编程问题。

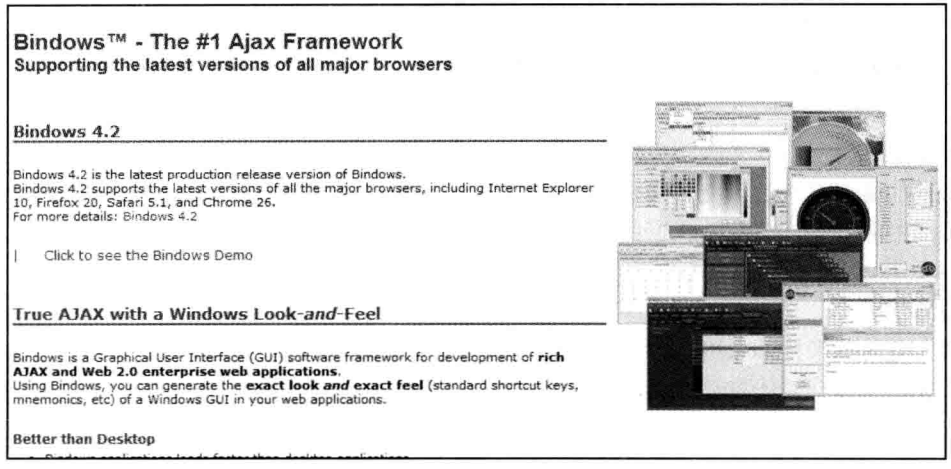
从上面描述来看，框架带来的便利性无疑比库好许多。但前端 JavaScript 由于存在一个远程加载的问题，对 JavaScript 文件的体积限制很大，因此，框架在几年前都不怎么吃香。现在网速快多了，设计师在网页制造的地位（UED）也不比昔日，因此，集成程度更高的 MVC、MVVM 框架也相继面世。

不过，无论是框架还是库，只要在浏览器中运行，就要与 DOM 打交道。如果不用 jQuery，就要发明一个半成品 jQuery 或一个半成品 Prototype。对想提升自己能力的人来说，答案其实很明显，写框架还能提升自己的架构能力。

2. JavaScript 发展历程

第 1 时期，洪荒时代。从 1995 年到 2005 年，就是从 JavaScript 发明到 Ajax 概念[①]的提出。其间打了第一场浏览器战争，IE VS Netscape。这两者的 DOM API 出入很大，前端开发人员被迫改进技术，为了不想兼容某一个浏览器，发明 UA（navigator.userAgent）嗅探技术。

这个时期的杰出代表是 Bindows[②]，2003 年发布，它提供了一个完整的 Windows 桌面系统，支持能在 EXT 看到的各种控件，如菜单、树、表格、滑动条、切换卡、弹出层、测量仪表（使用 VML 实现，现在又支持 SVG）。现在版本号是 4.x，如下图所示。



其他比较著名的还有 Dojo（2004 年）、Sarissa（2003 年）、JavaScript Remote Scripting（2000 年）。

Dojo 有 IBM 做后台，有庞大的开发团队在做，质量有保证，被广泛整合到各大 Java 框架内（struct2、Tapestry、Eclipse ATF、MyFaces）。特点是功能无所不包，主要分为 Core、Dijit、DojoX 3 大块。Core 提供 Ajax、events、packaging、CSS-based querying、animations、JSON 等相关操作 API。

① http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications
② http://www.bindows.net/

Dijit 是一个可更换皮肤、基于模板的 Web UI 控件库。DojoX 包括一些新颖的代码和控件，如 DateGrid、charts、离线应用和跨浏览器矢量绘图等，如下图所示。

JavaScript Remote Scripting 是较经典的远程脚本访问组件，支持将客户端数据通过服务器做代理进行远程的数据/操作交互。

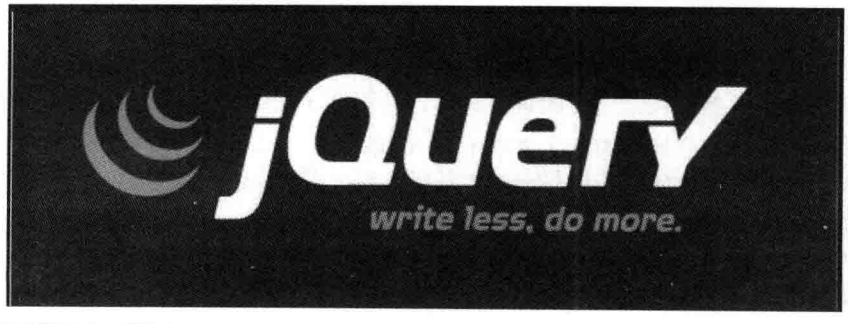
Sarissa 封装了在浏览器端独立调用 XML 的功能。

第 2 时期，Prototype“王朝”，2005 年～2008 年。其间打了第 2 次浏览器战争，交战双方是 IE6、IE7、IE8 VS Firefox 1、Firefox 2、Firefox 3，最后 Firefox 3 大胜。浏览器战争中，Prototype 积极进行升级，加入诸多 DOM 功能，因此，Jser（JavaScript 程序员）比之前好过多了。加之有 rails、script.aculo.us（一流的特效库）、Rico 等助阵，迅速占领了市场。

Prototype 时期，面向对象技术发展到极致，许多组件成套推出。DOM 特征发掘也有序进行，再也不依靠浏览器嗅探去刻意屏蔽某一个浏览器了。无侵入式 JavaScript 开发得到推崇，所有 JavaScript 代码都抽离到 JavaScript 文件，不在标签内“兴风作浪”了。

Prototype 的发展无可限量，直到 1.5 版本对 DOM 进行处理，这是一个错误[①]。如它一个很好用的 API-getElementsByClassName，由于 W3C 的标准化，Prototype 升级慢了，它对 DOM 的扩展成为了它的“地雷”。

第 3 时期，jQuery 纪元，2008 年到现在（如下图所示）。



① 详见 Prototype 核心成员的反思：http://perfectionkills.com/whats-wrong-with-extending-the-dom/

jQuery 则以包裹方式来处理 DOM，而且配合它的选择器引擎，若一下子选到 N 个元素，那么就处理 N 个元素，是集化操作，与主流的方式完全不一样。此外，它的方法名都起得很特别，人们一时很难接受。

> 2007 年 7 月 1 日，jQuery 发布了 1.1.3 版本，它的宣传是。
> （1）速度改良：DOM 的遍历比 1.1.2 版本快了大概 **800%**。
> （2）重写了事件系统：对键盘事件用更优雅的方法进行了处理。
> （3）重写了 effects 系统：提高了处理速度。

停滞不前的 Prototype 已经跟不上时代的节奏，jQuery 在 1.3x 版本时更换 Sizzle，更纯净的 CSS 选择器引擎，易用性与性能大大提高，被程序员一致看好的 mouseenter、mouseleave 及事件代理也被整合进去，jQuery 就占据了市场。

3. JavaScript 框架分类

如果是从内部架构与理念划分，目前 JavaScript 框架可以划分为 5 类。

第 1 种出现的是以命名空间为导向的类库或框架，如创建一个数组用 new Array()，生成一个对象用 new Object()，完全的 Java 风格，因此我们就可以以某一对象为根，不断为它添加对象属性或二级对象属性来组织代码，金字塔般地垒叠起来。代表作如早期的 YUI 与 EXT。

第 2 种出现的是以类工厂为导向的框架，如著名的 Prototype，还有 mootools、Base2、Ten。它们基本上除了最基本的命名空间，其他模块都是一个由类工厂衍生出来的类对象。尤其是 mootoos 1.3 把所有类型都封装成 Type 类型。

第 3 种就是以 jQuery 为代表的以选择器为导向的框架，整个框架或库主体是一个特殊类数组对象，方便集化操作——因为选择器通常是一下子选择到 N 个元素节点，于是便一并处理了。JQuery 包含了几样了不起的东西："无 new 实例化"技术，$(expr)就是返回一个实例，不需要显式地 new 出来；get first set all 访问规则；数据缓存系统。这样就可以复制节点的事件了。此外，IIFE（Immediately-Invoked Function Expression）也被发掘出来。

第 4 种就是以加载器串联起来的框架，它们都有复数个 JavaScript 文件，每个 JavaScript 文件都以固定规则编写。其中最著名的莫过于 AMD。模块化是 JavaScript 走向工业化的标志。《Unix 编程艺术》列举的众多"金科玉律"的第一条就是模块，里面有言——"要编写复杂软件又不至于一败涂地的唯一方法，就是用定义清晰的接口把若干简单模块组合起来，如此一来，多数问题只会出现在局部，那么还有希望对局部进行改进或优化，而不至于牵动全身"。许多企业内部框架都基本采取这种架构，如 Dojo、YUI、kissy、qwrap 和 mass 等。

第 5 种就是具有明确分层构架的 MV*框架。首先是 JavaScript MVC（现在叫 CanJS）、backbonejs 和 spinejs，然后更符合前端实际的 MVVM 框架，如 knockout、ember、angular、avalon、winjs。在 MVVM 框架中，原有 DOM 操作被声明式绑定取代了，由框架自行处理，用户只需专注于业务代码。

4. JavaScript 框架的主要功能

下面先看看主流框架有什么功能。这里面包含 jQuery 这个自称为库的东西，但它接近 9000 行，功能比 Prototype 还齐备。这些框架类库的模块划分主要依据它们在 github 中的源代码，基本上都是一个模块一个 JavaScript 文件。

jQuery

jQuery 强在它专注于 DOM 操作的思路一开始就是对的，以后就是不断在兼容性、性能上进行改进。

jQuery 经过多年的发展，拥有庞大的插件与完善的 Bug 提交渠道，因此，可以通过社区的力量不断完善自身。

Prototype.js

早期的王者，它分为 4 大部分。

- 语言扩展。
- DOM 扩展。
- Ajax 部分。
- 废弃部分（新版本使用其他方法实现原有功能）。

Prototype.js 的语言扩展覆盖面非常广，包括所有基本数据类型及从语言借鉴过来的“类”。其中 Enumerable 只是一个普通的方法包，ObjectRange、PeriodicalExecuter、Templat 则是用 Class 类工厂生产出来的。Class 类工厂来自社区贡献。

mootools

它由于 API 设计得非常优雅，其官方网站上有许多优质插件，因此才没有在原型扩展的反对浪潮中没落。

RightJS

又一个在原型上进行扩展的框架。

MochiKit

一个 Python 风格的框架，以前能进世界前十名的。

Ten

日本著名博客社区 Hatena 的 JavaScript 框架，由 amachang 开发，受 Prototype.js 影响，是最早以命名空间为导向的框架的典范。

mass Framework

它是一个模块化，以大模块开发为目标，jQuery 式的框架。

经过细节比较，我们很容易得出以下框架特征的结论。

- 对基本数据类型的操作是基础，如 jQuery 就提供了 trim、camelCase、each、map 等方法，Prototype.js 等侵入式框架则在原型上添加 camelize 等方法。
- 类型的判定必不可少，常见形式是 isXXX 系列。
- 选择器、domReady、Ajax 是现代框架的标配。
- DOM 操作是重中之重，节点的遍历、样式操作、属性操作也属于它的范畴，是否细分就看

框架的规模了。

- brower sniff 已过时，feature detect 正被应用。不过特性侦测还是有局限性，如果针对于某个浏览器版本的渲染 Bug、安全策略或某些 Bug 的修正，还是要用到浏览器嗅探。但它应该独立成一个模块或插件，移出框架的核心。
- 现在主流的事件系统都支持事件代理。
- 数据的缓存与处理，目前浏览器也提供 data-*属性进行这面的工作，但不太好用，需要框架的进一步封装。
- 动画引擎，除非你的框架像 Prototype.js 那样拥有像 script.aculo.us 这样顶级的动画框架做后盾，最好也加上。
- 插件的易开发和扩展性。
- 提供诸如 Deferred 这样处理异步的解决方案。
- 即使不专门提供一个类工厂，也应该存在一个名为 extend 或 mixin 的方法对对象进行扩展。jQuery 虽然没有类工厂，但在 jQuery UI 中也不得不增加一个，可见其重要性。
- 自从 jQuery 出来一个名为 noConflict 的方法，新兴的框架都带此方法，以求狭缝中生存。
- 许多框架非常重视 Cookie 操作。

最后感谢一下业内一些朋友的帮忙，要不是他们，书不会这么顺利地写出来。以下排名不分先后：玉伯、汤姆大叔、弹窗教主、貘大、linxz 、正则帝 abcd。这些都是专家级人物，在业界早已闻名遐迩。由于本人水平有限，书中难免存有不妥之处，请读者批评指正，源程序和答疑网址：https://github.com/RubyLouvre/jsbook/issues。编辑联系邮箱：zhangtao@ptpress.com.cn。

目　录

第1章 种子模块

种子模块也叫核心模块，是框架的最先执行的部分。即便像 jQuery 那样的单文件函数库，它的内部也分许多模块，必然有一些模块冲在前面立即执行，有一些模块只有用到才执行，也有一些模块可有可无，存在感比较弱，只在特定浏览器下才运行。

种子模块就是其中的急先锋，它里面的方法不一定要求个个神通广大，设计优良，但一定极具扩展性，常用，稳定。扩展性是指通过它们能将其他模块的方法包进来，让种子像大树一样成长；常用是指绝大多数模块都用到它们，防止做重复工作；稳定是指不能轻易在以后版本就给去掉，要信守承诺。

参照许多框架与库的实现，我认为种子模块应该包含如下功能：对象扩展，数组化，类型判定，简单的事件绑定与卸载，无冲突处理，模块加载与 domReady。本章的讲解内容以 mass Framework 的种子模块为范本，可以到以下地址下载：

https://github.com/RubyLouvre/mass-Framework/blob/1.4/mass.js

1.1 命名空间

种子模块作为一个框架的最开始部分，除了负责辅建全局用的基础设施外，你有没有想到给读者一个震撼的开场呢？俗话说，好的开头是成功的一半。

时下“霸主”jQuery 就有一个很好的开头——IIFE（立即调用函数表达式），一下子吸引住读者，让读者吃一颗定心丸——既然作者的水平如此高超，还怕什么啊，直接拿来用。

IIFE 是现代 JavaScript 框架最主要的基础设施，它像细胞膜一样包裹自身，防止变量污染。但我们总得在 Windows 里设置一个立足点，这个就是命名空间。基本上我们可以把命名空间等同于框架的名字，不过对于某些框架，它们是没有统一的命名空间，如 Prototype.js , mootools。它们就是不想让你感觉到框架的存在，它的意义深透到 JavaScript、DOM、COM 等整个执行环境的每个角落，对原生对象的原型进行扩展。由于道格拉斯（JSON 作者）的极力反对，新兴的框架都在命名空间上构建了。

命名空间是干什么用呢？这里就不多说了。我们看怎么在 JavaScript 模拟命名空间。JavaScript 一切基于对象，但只有复合类型的对象才符合这要求，比如 function、regexp、object……不过最常

用的还是 object 与 function。我们往一个对象上添加一个属性，而这个属性又是一个对象，这个对象我们又可以为它添加一个对象，通过这种方法，我们就可以有条不紊地构建我们的框架。用户想调用某个方法，它就以 XXX.YYY.ZZZ()的形式调用。

```
if (typeof(Ten) === "undefined") {
    Ten = {};
    Ten.Function = {/*略*/}
    Ten.Array = {/*略*/}
    Ten.Class = {/*略*/}
    Ten.JSONP = new Ten.Class(/*略*/)
    Ten.XHR = new Ten.Class(/*略*/)
}
```

纵观各大类库的实现，一开始基本都是定义一个全局变量作为命名空间，然后对它进行扩展，如 Base2 的 Base、Ext 的 Ext、jQuery 的 jQuery、YUI 的 YUI、dojo 的 dojo、MochiKit 的 MochiKit 等。从全局变量的污染程度来看，分为两大类。

Prototype、mootools 与 Base2 归为一类。Prototype 的哲学是对 JavaScript 原生对象进行扩展。早些年，Ptototype 差点成为事实的标准，因此基本没有考虑到与其他库的共存问题。基于 Prototype，也发展出诸如 script.aculo.us、rico、Plotr、ProtoChart、Scripty 2 等非常优秀的类库及一大堆收费插件，非 jQuery 那一大堆垃圾插件所能比拟的。而且，有点渊源的插件几乎都与 Prototype 有关，如著名的 lightbox。mootools 是 Prototype 的升级版，更加 OO，全面复制其 API。Base 则是想修复 IE 的 Bug，让 IE 拥有标准浏览器的 API，因此也把所有原生对象污染一遍。

第二类是 jQuery、YUI、EXT 这些框架。YUI 与 EXT 就是像上面给出的代码那样，以叠罗汉方式构建的。jQuery 则另辟蹊径，它是以选择器为导向的，因此它的命名空间是一个函数，方便用户把 CSS 表达器字符串传进来，然后通过选择器引擎进行查找，最后返回一个 jQuery 实例。另外，像 jQuery 最初也是非常弱小的，它想让人家试用自己的框架，但也想像 Prototype 那样使用美元符号作为它的命名空间。因此它特意实现了多库共存机制，在$、jQuery 与用户指定的新命名空间中任意切换。

jQuery 的多库共存原理很简单，因此后来也成为许多小库的标配。首先把命名空间保存到一个临时变量中，注意这时这个对象并不是自己框架的东西，可能是 Prototype.js 等巨头的，然后再搞个 noConflict 放回去。

```
//jQuery1.2
var _jQuery = window.jQuery, _$ = window.$;//先把可能存在的同名变量保存起来

jQuery.extend({
    noConflict: function(deep) {
        window.$ = _$;//这时再放回去
        if (deep)
            window.jQuery = _jQuery;
        returnjQuery;
    }
})
```

但 jQuery 的 noConflict 只对单文件的类库框架有用，像 EXT 就不能复制了。因此把命名空间改名后，将 EXT 置为 null，然后又通过动态加载方式引入新的 JavaScript 文件，该文件再以 EXT 调用，将会导致报错。

mass Framework 对 jQuery 的多库共存进行改进，它与 jQuery 一样拥有两个命名空间，一个是美元符号的短命名空间，另一个是根据 URL 动态生成的长命名空间（jQuery 就是 jQuery!）。

```
namespace = DOC.URL.replace( /(#.+|\W)/g,'');
```

短的命名空间随便用户改名，长的命名空间则是加载新的模块时用的，虽然用户在模块中使用$做命名空间，但当 JavaScript 文件加载下来时，我们会对里面的内容再包一层，将$指向正确的对象，具体实现见 define 方法。

1.2 对象扩展

我们需要一种机制，将新功能添加到我们的命名空间上。这方法在 JavaScript 通常被称做 extend 或 mixin。JavaScript 对象在属性描述符（Property Descriptor）没有诞生之前，是可以随意添加、更改、删除其成员的，因此扩展一个对象非常便捷。一个简单的扩展方法实现是这样。

```
function extend(destination, source) {
   for (var property in source)
      destination[property] = source[property];
   return destination;
}
```

不过，旧版本 IE 在这里有个问题，它认为像 Object 的原型方法就是不应该被遍历出来，因此 for in 循环是无法遍历名为 valueOf、toString 的属性名。这导致，后来人们模拟 Object.keys 方法实现时也遇到了这个问题。

```
Object.keys = Object.keys || function(obj){
   var a = [];
   for(a[a.length] in obj);
   return a ;
}
```

在不同的框架，这个方法还有不同的实现，如 EXT 分为 apply 与 applyIf 两个方法，前者会覆盖目标对象的同名属性，而后者不会。dojo 允许多个对象合并在一起。jQuery 还支持深拷贝。下面是 mass Framework 的 mix 方法，支持多对象合并与选择是否覆写。

```
function mix(target, source) {            //如果最后参数是布尔，判定是否覆写同名属性
   var args = [].slice.call(arguments), i = 1, key,
         ride = typeof args[args.length - 1] == "boolean" ? args.pop() : true;
   if (args.length === 1) {               //处理$.mix(hash)的情形
      target = !this.window ? this : {};
      i = 0;
   }
   while ((source = args[i++])) {
      for (key in source) {               //允许对象糅杂，用户保证都是对象
```

```
            if (ride || !(key in target)) {
                target[ key ] = source[ key ];
            }
        }
    }
    return target;
}
```

1.3 数组化

浏览器下存在许多类数组对象，如 function 内的 arguments，通过 document.forms、form. elements、doucment.links、select.options、document.getElementsByName、document.getElementsBy TagName、childNodes、children 等方式获取的节点集合（HTMLCollection、NodeList），或依照某些特殊写法的自定义对象。

```
var arrayLike = {
    0: "a",
    1: "1",
    2: "2",
    length: 3
}
```

类数组对象是一个很好的存储结构，不过功能太弱了，为了享受纯数组的那些便捷方法，我们在处理它们前都会做一下转换。

通常来说，只要[].slice.call 就能转换了，但旧版本 IE 下的 HTMLCollection、NodeList 不是 Object 的子类，采用如上方法将导致 IE 执行异常。我们看一下各大库怎么处理的。

```
//jQuery 的 makeArray
var makeArray = function(array) {
    var ret = [];
    if (array != null) {
        var i = array.length;
        // The window, strings (and functions) also have 'length'
        if (i == null || typeof array === "string" || jQuery.isFunction(array) ||
array.setInterval)
            ret[0] = array;
        else
            while (i)
                ret[--i] = array[i];
    }
    return ret;
}
```

jQuery 对象是用来储存与处理 dom 元素的，它主要依赖于 setArray 方法来设置和维护长度与索引，而 setArray 的参数要求是一个数组，因此 makeArray 的地位非常重要。这方法保证就算没有参数也要返回一个空数组。

Prototype.js 的$A 方法：

```
function $A(iterable) {
```

```
    if (!iterable)
        return [];
    if (iterable.toArray)
        return iterable.toArray();
    var length = iterable.length || 0, results = new Array(length);
    while (length--)
        results[length] = iterable[length];
    return results;
};
```

mootools 的$A 方法：

```
function $A(iterable) {
    if (iterable.item) {
        var l = iterable.length, array = new Array(l);
        while (l--)
            array[l] = iterable[l];
        return array;
    }
    return Array.prototype.slice.call(iterable);
};
```

Ext 的 toArray 方法：

```
var toArray = function() {
    returnisIE ?
            function(a, i, j, res) {
                res = [];
                Ext.each(a, function(v) {
                    res.push(v);
                });
                return res.slice(i || 0, j || res.length);
            } :
            function(a, i, j) {
                return Array.prototype.slice.call(a, i || 0, j || a.length);
            }
}()
```

Ext 的设计比较巧妙，功能也比较强大。它一开始就自动执行自身，以后就不用判定浏览器了。它还有两个可选参数，对生成的纯数组进行操作。

dojo 的_toArray 和 Ext 一样，后面两个参数是可选的，只不过第二个是偏移量，最后一个是已有的数组，用于把新生的新组元素合并过去。

```
(function() {
  var efficient = function(obj, offset, startWith) {
      return (startWith || []).concat(Array.prototype.slice.call(obj, offset || 0));
  };

  var slow = function(obj, offset, startWith) {
      var arr = startWith || [];
      for (var x = offset || 0; x > obj.length; x++) {
          arr.push(obj[x]);
      }
      returnarr;
```

```
    };

    dojo._toArray =
          dojo.isIE ? function(obj) {
       return ((obj.item) ? slow : efficient).apply(this, arguments);
    } :
          efficient;

})();
```

最后是 mass 的实现，与 dojo 一样，一开始就进行区分，W3C 方直接[].slice.call，IE 自己手动实现一个 slice 方法。

```
$.slice = window.dispatchEvent ? function(nodes, start, end) {
    return [].slice.call(nodes, start, end);
} : function(nodes, start, end) {
    var ret = [],
          n = nodes.length;
    if (end === void 0 || typeof end === "number" && isFinite(end)) {
       start = parseInt(start, 10) || 0;
       end = end == void 0 ? n : parseInt(end, 10);
       if (start < 0) {
          start += n;
       }
       if (end > n) {
          end = n;
       }
       if (end < 0) {
          end += n;
       }
       for (var i = start; i < end; ++i) {
          ret[i - start] = nodes[i];
       }
    }
    return ret;
}
```

1.4 类型的判定

JavaScript 存在两套类型系统，一套是基本数据类型，另一套是对象类型系统。基本数据类型包括 6 种，分别是 undefined、string、null、boolean、function、object。基本数据类型是通过 typeof 来检测的。对象类型系统是以基础类型系统为基础的，通过 instanceof 来检测。然而，JavaScript 自带的这两套识别机制非常不靠谱，于是催生了 isXXX 系列。就拿 typeof 来说，它只能粗略识别出 string、number、boolean、function、undefined、object 这 6 种数据类型，无法识别 Null、RegExpAragument 等细分对象类型。

让我们看一下这里面究竟有多少陷阱。

```
typeof null// "object"
typeof document.childNodes //safari "function"
typeof document.createElement('embed')//ff3-10 "function"
```

```
typeof document.createElement('object')//ff3-10 "function"
typeof document.createElement('applet')//ff3-10 "function"
typeof /\d/i //在实现了 ecma262v4 的浏览器返回 "function"
typeof window.alert //IE678 "object""

var iframe = document.createElement('iframe');
document.body.appendChild(iframe);
xArray = window.frames[window.frames.length - 1].Array;
var arr = new xArray(1, 2, 3); // [1,2,3]
arr instanceof Array; // false
arr.constructor === Array; // false

window.onload = function() {
    alert(window.constructor);// IE67 undefined
    alert(document.constructor);// IE67 undefined
    alert(document.body.constructor);// IE67 undefined
    alert((new ActiveXObject('Microsoft.XMLHTTP')).constructor);// IE6789 undefined
}
isNaN("aaa") //true
```

上面分 4 组，第一组是 typeof 的坑。第二组是 instanceof 的陷阱，只要原型上存在此对象的构造器它就返回 true，但如果跨文档比较，iframe 里面的数组实例就不是父窗口的 Array 的实例。第三组有关 constructor 的陷阱，在旧版本 IE 下 DOM 与 BOM 对象的 constructor 属性是没有暴露出来的。最后有关 NaN，NaN 对象与 null、undefined 一样，在序列化时是原样输出的，但 isNaN 这方法非常不靠谱，把字符串、对象放进去也返回 true，这对我们序列化非常不利。

另外，在 IE 下 typeof 还会返回 unknow 的情况。

```
if (typeof window.ActiveXObject != "undefined") {
    var xhr = new ActiveXObject("Msxml2.XMLHTTP");
    alert(typeof xhr.abort);
}
```

基于这 IE 的特性，我们可以用它来判定某个 VBscript 方法是否存在。

```
<script type="text/VBScript">
       function VBMethod(a,b)
       VBMethod = a + b
       end  function
</script>

<script>
       if(typeof VBMethod === "unknown"){//看这个
            alert(VBMethod(10,34))
       }
</script>
```

另外，以前人们总是以 document.all[①]是否存在来判定 IE，这其实是很危险的。因为用 document.all 来取得页面中的所有元素是不错的主意，这个方法 Firefox、Chrome 觊觎好久了，不

① 事实上是非 IE 浏览器均实现了叫做“伪装为 undefined”特性，当采用逻辑运算或类型判断时，会特意输出 undefined 值。http://fremycompany.com/BG/2013/Internet-Explorer-11-9385-new-features-771/

过人们都这样判定，于是有了在 Chrome 下的这出闹剧。

```
typeof document.all // undefined
document.all // HTMLAllCollection[728] (728 为元素总数)
```

在判定 undefined、null、string、number、boolean、function 这 6 个还算简单，前面两个可以分别与 void（0）、null 比较，后面 4 个直接 typeof 也可满足 90%的情形。这样说是因为 string、number、boolean 可以包装成“伪对象”，typeof 无法按照我们的意愿工作了，虽然它严格执行了 Ecmascript 的标准。

```
typeof new Boolean(1);//"object"
typeof new Number(1);//"object"
typeof new String("aa");//"object"
```

这些还是最简单的，难点在于 RegExp 与 Array。判定 RegExp 类型的情形很少，不多讲了，Array 则不一样。有关 isArray 的实现不下二十种，都是因为 JavaScript 的鸭子类型[①]被攻破了。直到 Prototype.js 把 Object.prototype.toString 发掘出来，此方法是直接输出对象内部的[[Class]]，绝对精准。有了它，可以跳过 95%的陷阱了。

isArray 早些年的探索：

```
function isArray(arr) {
    return arr instanceof Array;
}

function isArray(arr) {
    return !!arr && arr.constructor == Array;
}

function isArray(arr) {//Prototype.js1.6.0.3
    return arr != null && typeof arr === "object" &&
           'splice' in arr && 'join' in arr;
}

function isArray(arr) {//Douglas Crockford
    return typeof arr.sort == 'function'
}

function isArray(array) {//kriszyp
    var result = false;
    try {
        new array.constructor(Math.pow(2, 32))
    } catch (e) {
        result = /Array/.test(e.message)
    }
    return result;
};
```

① 在程序设计中，鸭子类型（英语：duck typing）是动态类型的一种风格。在这种风格中，一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由当前方法和属性的集合决定。这个概念的名字来源于由 James Whitcomb Riley 提出的鸭子测试，“鸭子测试”可以这样表述：“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”在鸭子类型中，关注的不是对象的类型本身，而是它如何被使用的。

```
function isArray(o) {// kangax
    try {
        Array.prototype.toString.call(o);
        return true;
    } catch (e) {
         }
    return false;
};

function isArray(o) {//kangax
    if (o && typeof o == 'object' && typeof o.length == 'number' && isFinite(o.length))
{
        var _origLength = o.length;
        o[o.length] = '__test__';
        var _newLength = o.length;
        o.length = _origLength;
        return _newLength == _origLength + 1;
    }
    return false;
}
```

至于 null、undefined、NaN 直接这样：

```
function isNaN(obj) {
    return obj !== obj
}
function isNull(obj) {
    return obj === null;
}
function isUndefined(obj) {
    return obj === void 0;
}
```

最后要判定的对象是 window，由于 ECMA 是不规范 Host 对象，window 对象属于 Host ，所以也没有被约定，就算 Object.prototype.toString 也对它无可奈何。

- [object Object]IE6
- [object Object]IE7
- [object Object]IE8
- [object Window]IE9
- [object Window]firefox3.6
- [object Window]opera10
- [object DOMWindow]safai4.04
- [object global]chrome5.0.3.22

不过根据 window.window 和 window.setInterval 去判定更加不够谱，用一个技巧我们可以完美识别 IE6、IE7、IE8 的 window 对象，其他还是用 toString，这个神奇的 hack（技巧）就是，window 与 document 互相比较，如果顺序不一样，其结果是不一样的！

```
window == document // IE678 true;
document == window // IE678 false;
```

当然，如果细数起来，JavaScript 匪夷所思的事比比都是。

存在 a !== a 的情况；

存在 a == b && b != a 的情况；

存在 a == !a 的情况；

存在 a === a+100 的情况；

1 < 2 < 3 为 true，　3 > 2 > 1 为 false；

0/0 为 NaN；

……

好了，至此，所有重要的 isXXX 问题都解决了，剩下的就把它们表达出来。经典做法就是直接罗列。

在 Prototype.js 中，拥有 isElement、isArray、isHash、isFunction、isString、isNumber、isDate、isUndefined 方法。

mootools 搞了个 typeOf 判定基本类型，instanceOf 判定自定义“类”。

RightJS 有 isFunction 、isHash、isString、isNumber、isArray 、isElement 、isNode。

EXT 有 isEmpty、isArray、isDate、isObject、isSimpleObject、isPrimitive、isPrimitive、isFunction、isNumber、isNumeric、isString、isBoolean、isElement、isTextNode、isDefined、isIterable，应有尽有。最后，还有 typeOf 判定基本类型。

Underscore.js 有 isElement、isEmpty、isArray、isArguments、isObject、isFunction、isString、isNumber、isFinite、isNaN、isBoolean、isDate、isRegExp、isNull、isUndefined。

isXXX 系列就像恶性肿瘤一样不断膨胀，其实你多弄几个 isXXX 也不能满足用户的全部需求。就像 isDate、isRegExp 会用到的机率有多高呢？

jQuery 就不与其他框架一样了，在 jQuery 1.4 中只有 isFunction、isArray、isPlainObject、isEmptyObject。IsFunction、isArray 肯定是用户用得最多，isPlainObject 则是用来判定是否为纯净的 JavaScript 对象，既不是 DOM、BOM 对象，也不是自定义“类”的实例对象，制造它的最初目的是用于深拷贝，避开像 window 那样自己引用自己的对象。isEmptyObject 是用于数据缓存系统，当此对象为空时，就可以删除它。

```
//jquery2.0
jQuery.isPlainObject = function(obj) {
    //首先排除基础类型不为 Object 的类型，然后是 DOM 节点与 window 对象
    if (jQuery.type(obj) !== "object" || obj.nodeType || jQuery.isWindow(obj)) {
        return false;
    }
    //然后回溯它的最近的原型对象是否有 isPrototypeOf,
    //旧版本 IE 的一些原生对象没有暴露 constructor、prototype，因此会在这里过滤
    try {
        if (obj.constructor &&
                !hasOwn.call(obj.constructor.prototype, "isPrototypeOf")) {
            return false;
```

```
            }
        } catch (e) {
            return false;
        }
        return true;
    }
```

在 avalon.mobile 中有一个更精简的版本，由于它只支持 IE10 等非常新的浏览器，就没有干扰因素了，可以大胆使用 ecma262v5 的新 API。

```
avalon.isPlainObject = function(obj) {
    return obj && typeof obj === "object" && Object.getPrototypeOf(obj) === Object.prototype
}
```

isArrayLike 也是一个常用的方法，但判定一个类数组太难了，唯一的辨识手段是它应该有一个大于或等于零的整型 length 属性。此外还有一些“共识”，如 window 与函数和元素节点（如 form 元素）不算类数组，虽然它们都满足前面的条件。因此至今 jQuery 没有把它暴露出来。

```
//jquery2.0
function isArraylike(obj) {
    var length = obj.length, type = jQuery.type(obj);
    if (jQuery.isWindow(obj)) {
        return false;
    }
    if (obj.nodeType === 1 && length) {
        return true;
    }
    return type === "array" || type !== "function" &&
           (length === 0 ||
                  typeof length === "number" && length > 0 && (length - 1) in obj);
}
//avalon 0.9
function isArrayLike(obj) {
    if (obj && typeof obj === "object") {
        var n = obj.length
        if (+n === n && !(n % 1) && n >= 0) { //检测 length 属性是否为非负整数
            try {//像 Argument、Array、NodeList 等原生对象的 length 属性是不可遍历的
                if ({}.propertyIsEnumerable.call(obj, 'length') === false) {
                    return Array.isArray(obj) || /^\s?function/.test(obj.item || obj.callee)
                }
                return true;
            } catch (e) { //IE 的 NodeList 直接抛错
                return true
            }
        }
    }
    return false
}

//avalon.mobile 更倚重 Object.prototoype.toString 来判定
function isArrayLike(obj) {
    if (obj && typeof obj === "object") {
        var n = obj.length,
```

```
                str = Object.prototype.toString.call(obj)
            if (/Array|NodeList|Arguments|CSSRuleList/.test(str)) {
                return true
            } else if (str === "[object Object]" && (+n === n && !(n % 1) && n >= 0)) {
                return true //由于 ecma262v5 能修改对象属性的 enumerable，因此不能用 propertyIs
                            //Enumerable 来判定了
            }
        }
        return false
    }
```

补充一句，1.3 版本中，Prototype.js 的研究成果（Object.prototype.toString.call）就应用于 jQuery 了。在 1.2 版本中，jQuery 判定一个变量是否为函数非常复杂。

```
isFunction: function( fn ) {
return !!fn&&typeoffn != "string" && !fn.nodeName&&
fn.constructor != Array && /^[\s[]?function/.test( fn + "" );
 }
```

jQuery1.43 引入 isWindow 来处理 makeArray 中对 window 的判定，引入 isNaN 用于确保样式赋值的安全。同时引入 type 代替 typeof 关键字，用于获取数据的基本类型。

```
class2type = {}
jQuery.each("Boolean Number String Function Array Date RegExpObject".split(" "),
function(i, name) {
    class2type[ "[object " + name + "]" ] = name.toLowerCase();
});
jQuery.type = function(obj) {
    return obj == null ?
           String(obj) :
           class2type[toString.call(obj) ] || "object";
})
```

jQuery1.7 中添加 isNumeric 代替 isNaN。这是个不同于其他框架的 isNumber,它可以是字符串，只要外观上像数字就行了。但 jQuery1.7 还做了一件违背之前提到稳定性的事情，贸然去掉 jQuery.isNaN ，因此导致基于旧版本 jQuery 的一大批插件失效。

```
//jquery1.43~1.64
jQuery.isNaN = function(obj) {
    return obj == null || !rdigit.test(obj) || isNaN(obj);
})
//jquery1.7 就是 isNaN 的取反版
jQuery.isNumeric = function(obj) {
    return obj != null && rdigit.test(obj) && !isNaN(obj);
})
//jquery1.71~1.72
jQuery.isNumeric = function(obj) {
    return !isNaN(parseFloat(obj)) && isFinite(obj);
}
//jquery2.1
jQuery.isNumeric = function(obj) {
```

```
    return obj - parseFloat(obj) >= 0;
} .
```

mass Framework 的思路与 jQuery 一致，尽量减少 isXXX 系列的数量，把 isWindow、isNaN、nodeName 等方法都整进去了。这是个野心勃勃的方法，代码比较长，它既可以获取类型，也可以传入第二参数进行类型比较。

```
var class2type = {
    "[objectHTMLDocument]": "Document",
    "[objectHTMLCollection]": "NodeList",
    "[objectStaticNodeList]": "NodeList",
    "[objectIXMLDOMNodeList]": "NodeList",
    "[objectDOMWindow]": "Window",
    "[object global]": "Window",
    "null": "Null",
    "NaN": "NaN",
    "undefined": "Undefined"
},
toString = class2type.toString;
"Boolean,Number,String,Function,Array,Date,RegExp,Window,Document,Arguments,NodeList"
        .replace($.rword, function(name) {
    class2type[ "[object " + name + "]" ] = name;
});

//class2type 这个映射几乎把所有常用判定对象“一网打尽”了
mass.type = function(obj, str) {
    var result = class2type[ (obj == null || obj !== obj) ? obj : toString.call(obj) ]
            || obj.nodeName || "#";
    if (result.charAt(0) === "#") {  //兼容旧版本浏览器与处理个别情况，如 window.opera
    //利用 IE6、IE7、IE8 window == document 为 true,document == window 竟然为 false 的神奇特性
        if (obj == obj.document && obj.document != obj) {
            result = 'Window';          //返回构造器名字
        } else if (obj.nodeType === 9) {
            result = 'Document';        //返回构造器名字
        } else if (obj.callee) {
            result = 'Arguments';       //返回构造器名字
        } else if (isFinite(obj.length) && obj.item) {
            result = 'NodeList';        //处理节点集合
        } else {
            result = toString.call(obj).slice(8, -1);
        }
    }
    if (str) {
        return str === result;
    }
    return result;
}
```

然后 type 方法就轻松了，用 toString.call（obj）得出的值作键，直接从映射中取。只有在 IE6、IE7、IE8 中，我们才费一些周折处理 window、document、arguments、nodeList 等对象。当然，这只是在种子模块的情形，在语言模块，mass Framework 还是会添加 isArray、isFunction 这两个著名

API，此外还有 isPlainObject、isNative、isEmptyObject、isArrayLike 这 4 个方法，在选择器模块，还追加 isXML 方法。

基于实用主义，我们有时不得不妥协。百度的 tangram 就是典型， 与 EXT 一样，能想到的都写上，而且判定非常严谨。

```
baidu.isDate = function(o) {
    return {}.toString.call(o) === "[object Date]" && o.toString() !== 'Invalid Date'
&& !isNaN(o);
}
baidu.isNumber = function(o) {
    return '[object Number]' == {}.toString.call(o) && isFinite(o);
}
```

1.5 主流框架引入的机制——domReady

domReady 其实是一种名为“DOMContentLoaded”事件的别称，不过由于框架的需要，它与真正的 DOMContentLoaded 有一点区别。在许多旧的 JavaScript 书籍中，它们都会教导我们把 JavaScript 逻辑写在 window.onload 回调中，以防 DOM 树还没有建完就开始对节点进行操作,导致出错。而对于框架来说，越早介入对 DOM 的干涉就越好，如要进行什么特征侦测之类的。domReady 还可以满足用户提前绑定事件的需求，因为有时页面图片等资源过多，window.onload 就迟迟不能触发，这时若还没有绑定事件，用户点哪个按钮都没反应（除了跳转外）。因此主流框架都引入 domReady 机制，并且费了很大劲兼容所有浏览器，具体策略如下。

（1）对于支持 DOMContentLoaded 事件的使用 DOMContentLoaded 事件。

（2）旧版本 IE 使用 Diego Perini 发现的著名 hack!

```
//http://javascript.nwbox.com/IEContentLoaded/
//by Diego Perini 2007.10.5
function IEContentLoaded(w, fn) {
    var d = w.document, done = false,
            init = function() {
        if (!done) {//只执行一次
            done = true;
            fn();
        }
    };
    (function() {
        try {//在 DOM 未建完之前调用元素 doScroll 抛出错误
            d.documentElement.doScroll('left');
        } catch (e) {//延迟再试
            setTimeout(arguments.callee, 50);
            return;
        }
        init();//没有错误则执行用户回调
    })();
    // 如果用户是在 domReady 之后绑定这个函数呢? 立即执行它
    d.onreadystatechange = function() {
        if (d.readyState == 'complete') {
```

```
            d.onreadystatechange = null;
            init();
        }
    };
}
```

此外，IE 还可以通过 script defer hack 进行判定。

```
//http://webreflection.blogspot.com/search?q=onContent
//by Andrea Giammarchi 2006.9.24
document.write("<script id=__ie_onload defer src=//0><\/scr" + "ipt>");
script = document.getElementById("__ie_onload");
script.onreadystatechange = function() {//IE 即使是死链也能触发事件
    if (this.readyState == "complete"){
        init(); // 指定了 defer 的 script 会在 DOM 树建完才触发
    };
```

不过有个问题是，如果我们的种子模块是动态加载的，在它插入 DOM 树时，DOM 树已经建完呢？这该怎么触发我们的 ready 回调？jQuery 给出的方案是，连 onload 也监听了，但如果连 onload 也没赶上，就判定 document.readyState 是否等于 complete！这样完美了吧，可惜 Firefox3.6 之前没有这属性！看 mass 给出的方案。

```
var readyList = [];
mass.ready = function(fn) {
    if (readyList) {
        fn.push(fn);
    } else {
        fn();
    }
}
var readyFn, ready = W3C ? "DOMContentLoaded" : "readystatechange";
function fireReady() {
    for (var i = 0, fn; fn = readyList[i++]; ) {
        fn();
    }
    readyList = null;
    fireReady = $.noop; //惰性函数，防止 IE9 二次调用_checkDeps
}

function doScrollCheck() {
    try { //IE 下通过 doScrollCheck 检测 DOM 树是否建完
        html.doScroll("left");
        fireReady();
    } catch (e) {
        setTimeout(doScrollCheck);
    }
}

//在 Firefox3.6 之前，不存在 readyState 属性
//http://www.cnblogs.com/rubylouvre/archive/2012/12/18/2822912.html
if (!DOC.readyState) {
    var readyState = DOC.readyState = DOC.body ? "complete" : "loading";
}
```

```
if (DOC.readyState === "complete") {
    fireReady(); //如果在 domReady 之外加载
} else {
    $.bind(DOC, ready, readyFn = function() {
        if (W3C || DOC.readyState === "complete") {
            fireReady();
            if (readyState) { //IE 下不能改写 DOC.readyState
                DOC.readyState = "complete";
            }
        }
    });
    if (html.doScroll) {
        try { //如果跨域会报错，那时肯定证明是存在两个窗口的
            if (self.eval === parent.eval) {
                doScrollCheck();
            }
        } catch (e) {
            doScrollCheck();
        }
    }
}
```

1.6 无冲突处理

无冲突处理也叫多库共存。不得不说，$是最重要的函数名，这么多框架都爱用它做自己的命名空间。在 jQuery 还比较弱小时，如何让人们试用它呢？当时 Prototype 是主流，jQuery 于是发明了 noConflict 函数，下面是源代码：

```
var
        window = this,
        undefined,
        _jQuery = window.jQuery,
        _$ = window.$,
        //把 window 存入闭包中的同名变量，方便内部函数在调用 window 时不用费大力气查找它
        //_jQuery 与_$用于以后重写
        jQuery = window.jQuery = window.$ = function(selector, context) {
    //用于返回一个 jQuery 对象
    return new jQuery.fn.init(selector, context);
}

jQuery.extend({
    noConflict: function(deep) {
        //引入 jQuery 类库后，闭包外面的 window.$与 window.jQuery 都储存着一个函数
        //它是用来生成 jQuery 对象或在 domReady 后执行里面的函数的
        //回顾最上面的代码，在还没有把 function 赋给它们时，_jQuery 与_$已经被赋值了
        //因此它们俩的值必然是 undefined
        //因此这种放弃控制权的技术很简单，就是用 undefined 把 window.$里面的 jQuery 系的函数清除掉
        //这时 Prototype 或 mootools 的$就可以“明媒正娶”了
        window.$ = _$;//相当于 window.$ = undefined
```

```
        //如果连你的程序也有一个叫 jQuery 的东西，jQuery 可以大方地连这个也让渡出去
        //这时就要为 noConflict 添加一个布尔值，为 true
        if (deep)
            //但我们必须用一个东西接纳 jQuery 对象与 jQuery 的入口函数
            //闭包里面的东西除非被 window 等宿主对象引用，否则就是不可见的
            //因此我们把闭包里面的 jQuery return 出去，外面用一个变量接纳就可以
            window.jQuery = _jQuery;//相当 window.jQuery = undefined
        return jQuery;
    }
});
```

使用时，先引入别人的库，然后引入 jQuery，使用调用$.noConflict()进行改名，这样就不影响别人的$运作了。

mass Framework 更进一步，在引入种子模块的 script 标签上定义一个 nick 属性，那么释放出来的命名空间就是你的那个属性值。里面也偷偷实现了 jQuery 那种机制。

```
<script nick="AAA" src="mass.js"></script>
<script>
AAA.log("xxxxx")
</script>
```

如果你不改，默认还是$——我说过了，大家都对它“垂涎三尺”。

第2章 模块加载系统

任何语言一到大规模应用阶段，必然要经历拆分模块的过程，以有利于维护与团队协作。与Java走得最近的dojo率先引入了加载器，早期的加载器都是同步的，使用document.write与同步Ajax请求实现。后来dojo开始以JSONP的方法设计它的每个模块的结构，以script节点为主体加载它的模块，这个就是目前主流的加载器方式。不得不提的是，dojo的加载器与AMD规范的发明者都是James Burke，dojo加载器独立出来就是著名的require加载器。本章将为你深入理解加载器的原理，讲授样本为mass并行加载器。你可以到这里下载：

https://github.com/RubyLouvre/mass-Framework/blob/master/mass.js

2.1 AMD规范

AMD是“Asynchronous Module Definition”的缩写，意为“异步模块定义”。重点有两个。异步——有效避免了采用同步加载方式中导致的页面假死现象。模块定义——每个模块必须按照一定的格式编写。主要接口有两个，define与require。define是模块开发者关注的方法，require是模块使用者关注的方法。

define的参数情况为define(id?, deps?, factory)。第一个为模块ID，第2个为依赖列表，第3个是工厂方法。前两个都是可选，如果不定义ID，则是匿名模块，加载器运用一些“魔术”能让它辩识自己叫什么。通常情况，模块ID约等于模块在工程中的路径（放到线上，表现为URL）。在开发过程，许多情况未确定，一些JavaScript文件会移来移去的，因此匿名模块就大发所长。deps与factory有个约定，deps有多少个元素，factory就有多少个传参，位置一一对应。传参为其他模块的返回值。

```
define("xxx", ["aaa","bbb"], function (aaa,bbb){

});
```

通常情况下define中还有一个amd对象，里面储存着模块的相关信息。

require的参数情况为require(deps, callback)，第一个为依赖列表，第2个为回调。deps有多少个元素，callback就有多少个传参，情况与define方法一致。因此在内部，define方法会调用require来加载依赖模块。一直这样递归下去。

```
require(["aaa","bbb"], function(aaa,bbb){

})
```

接口就是这么简单，但 require 本身还包含许多特性比如使用“!”来引入插件机制，通过 requirejs.config 进行各种配置。模块只是整合的一部分，你要拆得开，也要合得拢，因此合并脚本的地位在加载器中非常重要。但前端 JavaScript 没有这功能，requirejs 利用 node.js 写了个 r.js 帮你进行合并。

2.2 加载器所在路径的探知

要加载一个模块，我们需要一个 URL 作为加载地址，一个 script 作为加载媒介。但用户在 require 时都用 ID，因此我们需要一个将 ID 转换为 URL 的方法。思路很简单，强加个约定，URL 的合成规则为。

```
basePath + 模块 ID + ".js"
```

由于浏览器自上而下分析 DOM，当浏览器在解析我们的 JavaScript 文件（这个 JavaScript 文件是指加载器）时，它就肯定 DOM 树中最后加入的 script 标签。因此，我们有下面这个方法。

```
function getBasePath() {
    var nodes = document.getElementsByTagName("script");
    var node = nodes[nodes.length - 1];
    var src = document.querySelector ? node.src : node.getAttribute("src", 4);
    return src;
}
```

这个能满足 99%的需求。但如果我们不得不动态加载我们的加载器呢？在旧版本下 IE 就会折戟沉沙，这个不奇怪，许多常规方法在 IE6、IE7、IE8 都失效，除了 API 的差异性，还有它本身的各种 Bug。这个我很难指出是什么，总之要解决。如下面的这个 JavaScript 片断。

```
<script>
    document.write('<script src="avalon.js"><\/script>');
    document.write('<script src="mass.js"><\/script>');
    document.write('<script
src="http://common.cnblogs.com/script/jquery.js"><\/script>');
</script>
```

mass.js 为我们的加载器，里面执行 getBasePath 方法，预期得到 http://192.168.1.32/mass.js，但在 IE7 却返回 http://192.168.1.32/jquery.js。

这时就需要用 readyChange 属性，微软在 document、image、xhr、script 等东西都拥有了这个属性，用来查看加载情况。

```
function getBasePath() {
    var nodes = document.getElementsByTagName("script");
    if (window.VBArray) {//如果是 IE
        for (var i = 0, node; node = nodes[i++]; ) {
            if (node.readyState === "interactive") {
                break;
            }
        }
```

```
    } else {
        node = nodes[nodes.length - 1];
    }
    var src = document.querySelector ? node.src : node.getAttribute("src", 4);
    return src;
}
```

这样就搞定了。现在想一下能否优化。访问 DOM 比一般的 JavaScript 代码消耗高许多。这时我们可以利用 Error 对象。

```
function getBasePath() {
    try {
        a.b.c()
    } catch (e) {
        if (e.fileName) {//firefox
            return e.fileName;
        } else if (e.sourceURL) {//safari
            return e.sourceURL;
        }
    }
    var nodes = document.getElementsByTagName("script");
    if (window.VBArray) {//倒序查找更快
        for (var i = nodes.length, node; node = nodes[--i]; ) {
            if (node.readyState === "interactive") {
                break;
            }
        }
    } else {
        node = nodes[nodes.length - 1];
    }
    var src = document.querySelector ? node.src : node.getAttribute("src", 4);
    return src;
}
```

当然 Opera 与 Chrome 也有一些属性可以供我们提取，但比较复杂，这里就略去了。有兴趣可以到这里看一下：

http://www.cnblogs.com/rubylouvre/archive/2010/04/06/1705817.html

为了方便以后使用，我们先将 mass.js 这个去掉吧。在现实中，为了防止缓存，这个后面可能带版本号、时间戳什么的，也要去掉。

```
url = url.replace(/[?#].*/, "").slice(0, url.lastIndexOf("/") + 1);
```

在 mass 并行加载器中，有一个 getCurrentScript 方法，它用于取得正在被解析的 JavaScript 文件的 src，而不局限于加载器的 JavaScript 地址。这个对实现匿名模块非常有用。想象一下，我们有个 require(["a","b","c","d","e"], callback)，这些模块在 define 里面都没有 ID，识别哪一个模块是对应哪个 JavaScript 文件加载非常麻烦。或者使用 iframe，将各个模块的加载独立于一个沙箱环境中，或者使用一些变量做标识（这存在被覆盖的危险），因此 mass 再次发掘 Error 对象的私有属性。

```
function getCurrentScript(base) {//为 true 时相当于 getBasePath
    var stack;
```

```
    try { //Firefox 可以直接 var e = new Error("test"), 但其他浏览器只会生成一个空 Error
        a.b.c(); //强制报错, 以便捕获 e.stack
    } catch (e) { //Safari 的错误对象只有 line、sourceId、sourceURL
        stack = e.stack;
        if (!stack && window.opera) {
            //Opera 9 没有 e.stack,但有 e.Backtrace, 不能直接取得, 需要对 e 对象转字符串进行抽取
            stack = (String(e).match(/of linked script \S+/g) || []).join(" ");
        }
    }
    if (stack) {
        /**e.stack 最后一行在所有支持的浏览器大致如下。
         *Chrome23:
         * at http://113.93.50.63/data.js:4:1
         *Firefox17:
         *@http://113.93.50.63/query.js:4
         *@http://113.93.50.63/data.js:4
         *IE10:
         *  at Global code (http://113.93.50.63/data.js:4:1)
         *  //firefox4+ 可以用 document.currentScript
         */
        stack = stack.split(/[@ ]/g).pop();          //取得最后一行, 最后一个空格或@之后的部分
        stack = stack[0] === "(" ? stack.slice(1, -1) : stack.replace(/\s/, "");
                                                     //去掉换行符
        return stack.replace(/(:\d+)?:\d+$/i, ""); //去掉行号与或许存在的出错字符起始位置
    }
    ///我们在动态加载模块时, 节点都插入 head 中, 因此只在 head 标签中寻找
    var nodes = (base ? document : head).getElementsByTagName("script");
    for (var i = nodes.length, node; node = nodes[--i]; ) {
        if ((base || node.className === moduleClass) && node.readyState === "interactive")
{
            return node.className = node.src;
        }//如果此模块加载过就重写 className
    }
}
```

2.3 require 方法

require 方法的作用是当依赖列表都加载完毕，执行用户回调。因此这里有个加载的过程，整个加载过程分为以下几步。

（1）取得依赖列表的第一个 ID，转换为 URL。无论是通过 basePath+ID+".js"，还是以映射的方式直接得到。

（2）检测此模块有没有加载过，或正在被加载。因此我们需要一个对象来保持所有模块的加载情况。当用户从来没有加载过此节点时，就进入加载流程。

（3）创建 script 节点，绑定 onerror、onload、onreadychange 等事件判定加载成功与否，然后添加 href 并插入 DOM 树，开始加载。

（4）将模块的 URL，依赖列表等构建成一个对象，放到检测队列中，在上面的事件触发时进行检测。

在 mass 的加载器，它支持允许第一个参数为字符串，然后内部按空格或逗号切分为 ID 数组，以及做去重处理，其他都一样。

我们看一下模块 ID 的转换规则：

http://wiki.commonjs.org/wiki/Modules/1.1.1。

mass 在这基础上做了扩展。

（1）模块 ID 本来就是 URL 的简体，因此可以包含斜线（/），并以/划分为多项。

（2）模块 ID 应该是以符合变量的规则的字符串组成，第一个项可以是“.”或“..”。这些都是 URL 的基本的规则，表示当前目录与父目录。

（3）模块 ID 的末尾可以包含“.js”，但如果它是指向一个 CSS 文件，那么必须以 CSS 结尾。

（4）如果以“/”或“./”开头，表示它与加载它的模块在同一目录。

（5）如果以“..”开头，表示它在加载它的模块的上一级目录，如果存在多个“..”，就要向上找。

（6）如果模块 ID 是“mass”，不做转换与加载，这表示 mass 框架的种子模块，也就是加载器的所在模块。

（7）如果模块 ID 是“ready”，不做转换与加载，这用于延迟用户回调到 DOM 树建完后执行。避免出现 domReady 的回调函数与模块的回调函数出现套嵌。

（8）如果情况就直接在前面按上 basePath——加载器所在的目录！

除了这些情况外，我们通常还用到映射，就是允许用户在事前用一个方法，把 ID 与完整的 URL 对应好，这样就直接拿。mass 称之为别名机制。ID 只是给用户用的，框架还是 URL 做加载或其他检测。此外，AMD 还发展一种 shim 技术，shim 就是垫片的意思，目的是让不符合 AMD 定义的 JavaScript 文件也能无缝切入我们的加载器系统。

如这个是普通的别名机制：

```
require.config({
    alias: {
        'lang': 'http://common.cnblogs.com/script/mass/lang.js',
        'css': 'http://common.cnblogs.com/script/mass/css.js'
    }
});
```

而对于 jQuery 或其插件，我们需要 shim 机制：

```
require.config({
    alias: {
        'jquery': {
            src: 'http://common.cnblogs.com/scriptjquery.js',
            exports: "$"
        },
        'jquery.tooltip': {
            src: 'http://common.cnblogs.com/script/ui/tooltip.js',
            exports: "$",
            deps: ["jquery"]
        }
    }
});
```

下面是 require 的源码：

```
window.require = $.require = function(list, factory, parent) {
    // 用于检测它的依赖是否都为 2
    var deps = {},
            // 用于保存依赖模块的返回值
            args = [],
            // 需要安装的模块数
            dn = 0,
            // 已安装完的模块数
            cn = 0,
            id = parent || "callback" + setTimeout("1");
    parent = parent || basepath;//basepath 为加载器的路径
    String(list).replace($.rword, function(el) {
        var url = loadJSCSS(el, parent)
        if (url) {
            dn++;
            if (modules[url] && modules[url].state === 2) {
                cn++;
            }
            if (!deps[url]) {
                args.push(url);
                deps[url] = "司徒正美"; //去重
            }
        }
    });
    modules[id] = {//创建一个对象,记录模块的加载情况与其他信息
        id: id,
        factory: factory,
        deps: deps,
        args: args,
        state: 1
    };
    if (dn === cn) { //如果需要安装的等于已安装好的
        fireFactory(id, args, factory); //安装到框架中
    } else {
        //放到检测队列中,等待 checkDeps 处理
        loadings.unshift(id);
    }
    checkDeps();
};
```

每 require 一次，相当于把当前的用户回调当成一个不用加载的匿名模块，ID 是随机生成，回调是否执行，要待到 deps 对象里面所有值都为 2。

require 里有三个重要方法：loadJSCSS，它用于转换 ID 为 URL，后面再调用 loadJS, loadCSS，或再调用 require 方法；fireFactory，就是执行用户回调，我们的最终目的；checkDeps，检测依赖是否都安装好，安装好就执行 fireFactory。

```
function loadJSCSS(url, parent, ret, shim) {
    //1. 特别处理 mass|ready 标识符
```

```
    if (/^(mass|ready)$/.test(url)) {
        return url;
    }
    //2. 转化为完整路径
    if ($.config.alias[url]) {//别名机制
        ret = $.config.alias[url];
        if (typeof ret === "object") {
            shim = ret;
            ret = ret.src;
        }
    } else {
        if (/^(\w+)(\d)?:.*/.test(url)) { //如果本来就是完整路径
            ret = url;
        } else {
            parent = parent.substr(0, parent.lastIndexOf('/'));
            var tmp = url.charAt(0);
            if (tmp !== "." && tmp !== "/") { //相对于根路径
                ret = basepath + url;
            } else if (url.slice(0, 2) === "./") { //相对于兄弟路径
                ret = parent + url.slice(1);
            } else if (url.slice(0, 2) === "..") { //相对于父路径
                var arr = parent.replace(/\/$/, "").split("/");
                tmp = url.replace(/\.\.\//g, function() {
                    arr.pop();
                    return "";
                });
                ret = arr.join("/") + "/" + tmp;
            } else if (tmp === "/") {
                ret = parent + url;//相对于兄弟路径
            } else {
                $.error("不符合模块标识规则: " + url);
            }
        }
    }
    var src = ret.replace(/[?#].*/, ""),
            ext;
    if (/\.(css|js)$/.test(src)) {
        ext = RegExp.$1;
    }
    if (!ext) { //如果没有后缀名,加上后缀名
        src += ".js";
        ext = "js";
    }
    //3. 开始加载 JS 或 CSS
    if (ext === "js") {
        if (!modules[src]) { //如果之前没有加载过
            modules[src] = {
                id: src,
                parent: parent,
                exports: {}
            };
            if (shim) {//shim 机制
                require(shim.deps || "", function() {
```

```
                loadJS(src, function() {
                    modules[src].state = 2;
                    modules[src].exports = typeof shim.exports === "function" ?
                            shim.exports() : window[shim.exports];
                    checkDeps();
                });
            });
        } else {
            loadJS(src);
        }
    }
    return src;
} else {
    loadCSS(src);
}
}
```

注意，上面的 modules[src]是以完整路径做 ID 的，它对应的对象没有 state 属性，表示其正在加载中。

loadJS 与 loadCSS 方法就比较纯粹了，不过 loadJS 会做个死链检测 checkFail。

```
function loadJS(url, callback) {
    //通过 script 节点加载目标模块
    var node = DOC.createElement("script");
    node.className = moduleClass; //让 getCurrentScript 只处理类名为 moduleClass 的 script 节点
    node[W3C ? "onload" : "onreadystatechange"] = function() {
        if (W3C || /loaded|complete/i.test(node.readyState)) {
            //factorys 里面装着 define 方法的工厂函数(define(id?,deps?, factory))
            var factory = factorys.pop();
            factory && factory.delay(node.src);
            if (callback) {
                callback();
            }
            if (checkFail(node, false, !W3C)) {
                $.log("已成功加载 " + node.src, 7);
            }
        }
    };
    node.onerror = function() {
        checkFail(node, true);
    };
    //插入到 head 的第一个节点前，防止 IE6 下 head 标签没闭合前使用 appendChild 抛错
    node.src = url;
    head.insertBefore(node, head.firstChild);
}
```

checkFail 方法主要是用于开发调试。有 3 个参数。node——script 节点，onError——是否为 onerror 触发，fuckIE——对应旧版本 IE 的 hack。思路是，JavaScript 文件从加载到解析到执行需要一个过程，在 interact 阶段，我们的 JavaScript 代码已经有些部分可以执行了，这时我们将模块对象的 state 改为 1，如果还是 undefined，我们就可识别它为死链。不过，此 hack 对不是以 AMD 定义的 JavaScript

模块无效，因为将 state 改 1 的逻辑是由 define 方法执行的。如果判定是死链，我们就把此节点移除。

```
function checkFail(node, onError, fuckIE) {
    var id = node.src;//检测是否死链
    node.onload = node.onreadystatechange = node.onerror = null;
    if (onError || (fuckIE && !modules[id].state)) {
        setTimeout(function() {
            head.removeChild(node);
        });
        $.log("加载 " + id + " 失败" + onError + " " + (!modules[id].state), 7);
    } else {
        return true;
    }
}
```

checkDeps 方法会在用户加载模块之前及 script.onload 后各执行一次，检测模块的依赖情况，如果模块没有任何依赖或 state 都为 2 了，我们调用 fireFactory 方法。

```
function checkDeps() {
    loop: for (var i = loadings.length, id; id = loadings[--i]; ) {
        var obj = modules[id],  deps = obj.deps;
        for (var key in deps) {
            if (hasOwn.call(deps, key) && modules[key].state !== 2) {
                continue loop;
            }
        }
        //如果 deps 是空对象或者其依赖的模块的状态都是 2
        if (obj.state !== 2) {
            loadings.splice(i, 1);//必须先移除再安装，防止在 IE 下 DOM 树建完后手动刷新页面，会多次执行它
            fireFactory(obj.id, obj.args, obj.factory);
            checkDeps();//如果成功,则再执行一次，以防有些模块就差本模块没有安装好
        }
    }
}
```

历经千辛万苦，我们终于到达 fireFactory 方法。它的工作是从 modules 中收集各模块的返回值，执行 factory，完成模块的安装。

```
function fireFactory(id, deps, factory) {
    for (var i = 0, array = [], d; d = deps[i++]; ) {
        array.push(modules[d].exports);
    }
    var module = Object(modules[id]),
            ret = factory.apply(global, array);
    module.state = 2;
    if (ret !== void 0) {
        modules[id].exports = ret;
    }
    return ret;
}
```

2.4 define 方法

我们再来看 define 方法，打个比方，它与 require 的关系就是内应外合。define 是应，require 是合。require 拥有加载器 90%的调度资源，以围城姿态攻打我们封闭的 JavaScript 模块。JavaScript 模块则由一个“内鬼”define 来看守城门。当 require 发出请求，define 就打开城门，模块被兼并到 mass Framework 的“庞大帝国内”!

define 有 3 个参数，前面两个为可选，事实上这里的 ID 没什么用，就是给开发者看的，它还是用 getCurrentScript 方法得到 script 节点路径做 ID，deps 没有就补上一个空数组。在 mass 中，先根据浏览器的情况对模块进行分级，一些模块是专门用于打补丁的，只对 IE6、IE7、IE8 有用，但对 Chrome 没有用，这个我们在 require 时就自动排除它。但合并后，我们不得不把它们都加上（因为不知道面对的是什么浏览器），mass 设置一个参数，用于忽略执行这个补丁模块的工厂函数，保证框架用浏览器的原生 API 执行。这个参数就设置在 define 中，为一个布尔，如果为 true 就跳过。

此外，define 还要考虑循环依赖的问题。比如说加载 A，要依赖 B 与 C，加载 B，要依赖 A 与 C，这时 A 与 B 就循环依赖了。A 与 B 在判定各自的 deps 中的键值都是 2 时才执行，结果都无法执行了。

define 的源码:

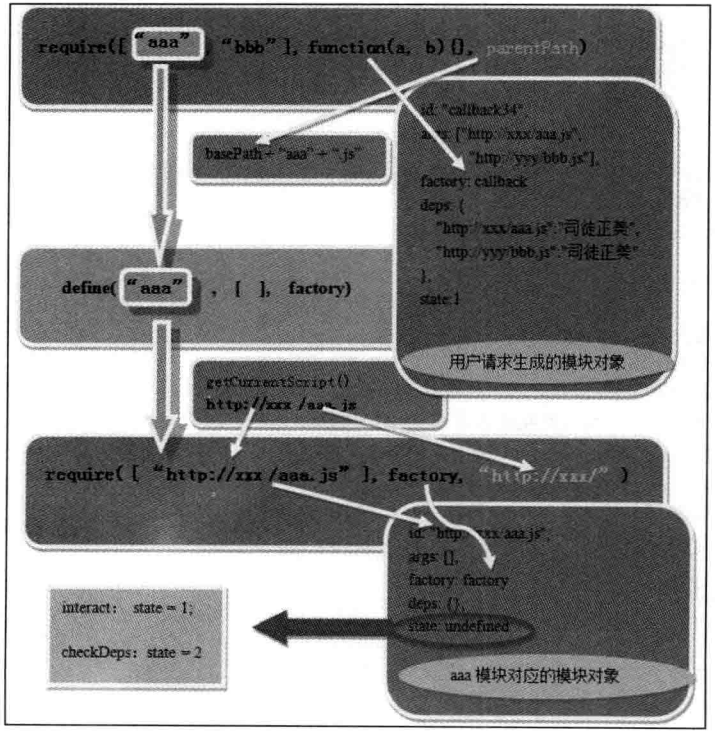
```
window.define = $.define = function(id, deps, factory) {
    var args = $.slice(arguments);
    if (typeof id === "string") {
        var _id = args.shift();
    }
    if (typeof args[0] === "boolean") { //用于文件合并，在标准浏览器中跳过补丁模块
        if (args[0]) {
            return;
        }
        args.shift();
    }
    if (typeof args[0] === "function") {
        args.unshift([]);
    } //上线合并后能直接得到模块 ID，否则寻找当前正在解析中的 script 节点的 src 作为模块 ID
    //现在除了 Safari 外，我们都能直接通过 getCurrentScript 一步到位得到当前执行的 script 节点
    //Safari 可通过 onload+delay 闭包组合解决
    id = modules[id] && modules[id].state >= 1 ? _id : getCurrentScript();
    factory = args[1];
    factory.id = _id; //用于调试
    factory.delay = function(id) {
        args.push(id);
        var isCycle = true;
        try {
            isCycle = checkCycle(modules[id].deps, id);
        } catch (e) {
        }
        if (isCycle) {
```

```
            $.error(id + "模块与之前的某些模块存在循环依赖");
        }
        delete factory.delay; //释放内存
        require.apply(null, args); //0,1,2 --> 1,2,0
    };
    if (id) {
        factory.delay(id, args);
    } else { //先进先出
        factorys.push(factory);
    }
};
```

checkCycle 方法：

```
function checkCycle(deps, nick) {
    //检测是否存在循环依赖
    for (var id in deps) {
        if (deps[id] === "司徒正美" && modules[id].state !== 2 && (id === nick ||
checkCycle(modules[id].deps, nick))) {
            return true;
        }
    }
}
```

define 与 require 互相调用的示意如图 2.1 所示。



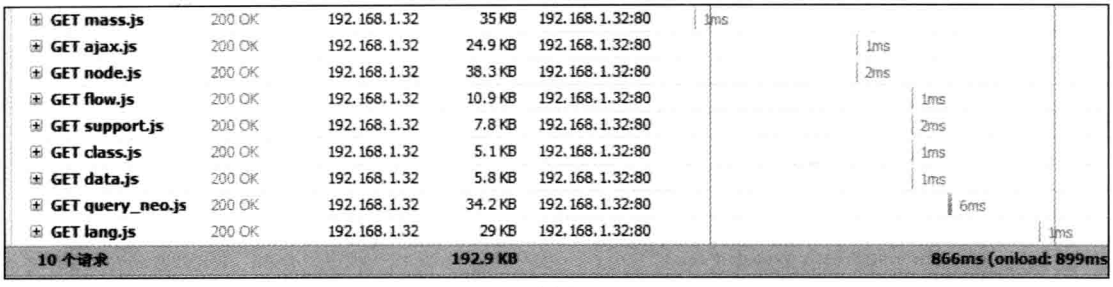
▲图 2.1

至此整个加载器就完成了。现在的难点转为我们如何判定当前模块的加载情况，如何知道它的模块名，如何排查它的对应的链接是否有效。为此 mass 动用一个 modules 对象，两个数组（loadings 与 factory）。小小一个加载器，里面的注释就提及到许多兼容性问题，真正与 DOM 打交道还没有开始呢！

最后看一下 mass 加载器，加载自己框架的 ajax、node 模块一共做了多少事吧！

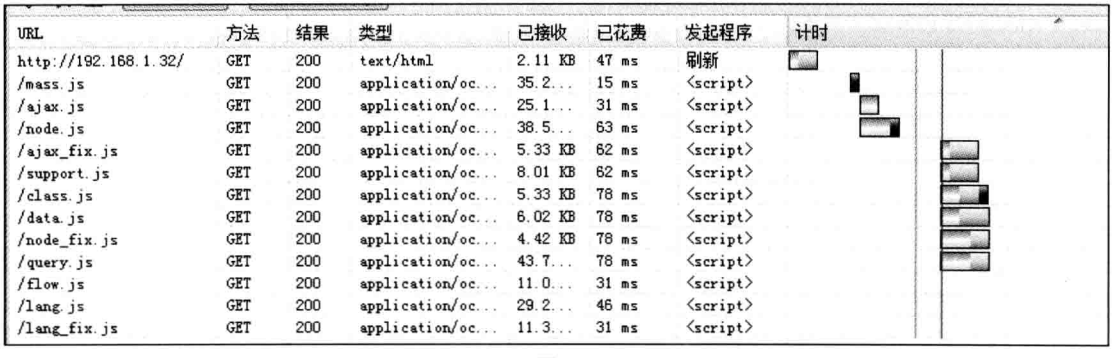
```
require("ajax, node", function($) {
   $.log("加载完成！")
})
```

Firefox20 会加载这么多模块，如图 2.2 所示。



▲图 2.2

IE10 在 IE7 模式下会加载如下模块，如图 2.3 所示。



▲图 2.3

由于使用了模块化，我们就可以分级，对旧版本 IE 使用体积更庞大的 query 模块，其他则使用 query_neo，并且多了许多补丁模块。

此外，mass 加载器会把加载过程全部记录下来，大家可以直接在 Firefox 看到这些消息，如图 2.4 所示。

▲图 2.4

模块加载器会让我们的前端开发变得更为工业化，维护调试都非常方便，强烈建立大家在工作时使用。现在国内主流的几个加载器 seajs、requirejs、mass、oyejs 都是比较好的选择。

第3章 语言模块

1995年，Brendan Eich读完了所有在程序语言设计中曾经出现过的错误，自己又发明了一些更多的错误，然后用它们创造出了LiveScript。之后，为了紧跟Java语言的时髦潮流，它被重新命名为JavaScript。再然后，为了追随一种皮肤病的时髦名字，这语言又被命名为ECMAScript。

上面一段话出自博文《编程语言伪简史》，JavaScript受到最辛辣的嘲讽，可见，它在当时是多少不受欢迎的。抛开偏见，JavaScript的确有许多不足之外，由于互联网的传播性及浏览器大战，JavaScript之父失去对此门语言的掌控权，即便他想修复这些Bug或推出某些新特，也要所有浏览器大厂都点头才行。IE6的市场独占性，打破了他的奢望。这个待到Chrome诞生，才有所改善。

但在IE6时期，浏览器提供的原生API的数量是极其贫乏的，因此各个框架都创造了许多方法以弥补这缺陷。视框架作者原来的语言背景不同，这些方法也是林林总总的。其中最杰出的代表是王者Prototype.js，把ruby语言的那一套方式或范式搬过来，从底层促进了JavaScript的发展。ecma262v6添加那一堆字符串，数组方法，差不多就是改个名字而已。

即便是浏览器的API也不能尽信，尤其是IE6、IE7、IE8？到处是BUG，因此这也列入框架的工作范围。

本章主要是围绕着mass Framework的lang与lang_fix模块展开，可以到这里下载。

https://github.com/RubyLouvre/mass-Framework/blob/1.4/lang.js

https://github.com/RubyLouvre/mass-Framework/blob/1.4/lang_fix.js

3.1 字符串的扩展与修复

我发现脚本语言都对字符串特别关注，有关它的方法特别多。我把这些方法分为三大类。

第一类，与标签无关的实现：charAt、charCodeAt、concat、indexOf、lastIndexOf、localeCompare、match、replace,search、slice、split、substr、substring、toLocaleLowerCase、toLocaleUpperCase、toLowerCase、toUpperCase及从Object继承回来的方法，如toString、valueOf。

第二类，与标签有关的实现，都是对原字符串添加一对标签：anchor、big、blink、bold、fixed、fontcolor、italics、link、small、strike、sub、sup。

第三类是后来添加或未标准化的浏览器方法：trim、quote、toSource、trimLeft、trimRight。其中trim已经标准化，后四个是Firefox的私有实现。

我们再看ecma262v6（2012.6.15）打算要添加的方法：repeat、startsWith、endsWidth、contains。

再看伟大的 Prototype.js 添加的扩展：gsub、sub、scan、truncate、strip、stripTags、stripScripts、extractScripts、evalScripts、escapeHTML、unescapeHTML、parseQuery、toArray、succ、times、camelize、capitalize、underscore、dasherize、inspect、unfilterJSON、isJSON、evalJSON、include、startsWith、endsWith、empty、blank、interpolate。

其中 gsub、sub、scan 与正则相关，直接取自 ruby 的命名。

truncate 是字符串截取，非常有用的方法，许多框架都有它的“微创新”。

strip 即 trim，已标准化。

stripTags 去掉字符串中的标签对，非常有用。

stripScripts 作为 stripTags 的补充，因为单单把 script 标签去掉，里面不该显示出来的 script.text 就暴露出来了。

extractScripts 与 evalScripts 是抽取与执行字符串中的脚本，IE 的 innerHTML 在某种情况下可以这样做，但其他浏览器不行，框架有责任屏蔽此差异性。

escapeHTML 与 unescapeHTML 是对用户的输入输出操作进行转义，非常有用。

parseQuery 基本上用于对 URL 的 search 部分进行操作，转换成对象，非常有用。

toArray 原本也是 ecma262v6 打算要添加的方法，用于转换成数组，不过这个用户也易实现，因此被抛弃了。

succ 是用于 ObjectRange 内部使用的。

times 即 ecma262v6 的 repeat 方法。

Camelize、capitalize、underscore、dasherize 这四个用于转换命名风格，非常有用。

inspect 就是在两端加双引号，用于构建 JSON，相当于 Firefox 的私有实现 quote。

UnfilterJSON、isJSON、evalJSON 与 JSON 相关。

include 就是 contains，与 startsWith、endsWith 成为 ecma262v6 的标准方法。

empty、blank 是对空白进行判定，很简单的方法。

interpolate 用于模板，其他框架通常称之为 format 或 substitute。

Prototype.js 这些有用的扩展会被其他框架抄去，我们查看哪些经常被抄，就知道哪些方法最有价值了。

Rirght.js 的字符串扩展：include、blank、camelize、capitalize、dasherize、empty、endsWith、evalScripts、extractScripts、includes、on、startsWith、stripScripts、stripTags、toFloat、toInt、trim、underscored。

Mootools 的字符串扩展（只取原型扩展）：test、contains、trim、clean、camelCase、hyphenate、capitalize、escapeRegExp、toInt、toFloat、hexToRgb、rgbToHex、substitute、stripScripts。

dojo 的字符串扩展：rep、pad、substitute、trim。rep 就是 repeat 方法.

EXT 的字符串扩展：capitalize、ellipsis、escape、escapeRegex、format、htmlDecode、htmlEncode、leftPad、parseQueryString、trim、urlAppend。

qooxdoo 的字符串扩展：format、hyphenate、pad、repeat、startsWidth、stripScripts、stripTags、toArray、trim、trimLeft、trimRight。

Tangram 的字符串扩展：decodeHTML、encodeHTML、escapeReg、filterFormat、format、

formatColor、stripTags、toCamelCase、toHalfWidth、trim、wbr。

通过以上竞争对手分析，我在 mass Framework 为字符串添加如下扩展，各位写框架的朋友可以视自己的情况进行增减：contains、startsWith、endsWith、repeat、camelize、underscored、capitalize、stripTags、stripScripts、escapeHTML、unescapeHTML、escapeRegExp、truncate，wbr、pad。其中前四个 ecma262v6 的标准方法，接着九个发端于 Prototype.js 广受欢迎的工具方法，wbr 是来自 Tangram，用于软换行，这出于汉语排版的需要。pad 也是一个很常用的操作，被收纳。

下面是各种具体实现。

contains 方法：判定一个字符串是否包含另一个字符串。常规思维，使用正则，但每次都要用 new RegExp 来构造，性能太差，转而使用原生字符串方法，如，ndexOf、lastIndexOf、 search。

```
function contains(target, it) {
     return target.indexOf(it) != -1; //indexOf 改成 search，lastIndexOf 也行得通
}
```

在 mootools 的版本中，我看到它支持更多参数，估计目的是判定一个元素的 className 是否包含某个特定的 class。众所周知，元素可以添加多个 class，中间以空格隔开，使用 mootools 的 contains 就很方便检测包含关系了。

```
function contains(target, str, separator) {
   return separator ?
          (separator + target + separator).indexOf(separator + str + separator) > -1 :
          target.indexOf(str) > -1;
}
```

注，本章的所有工具函数都是以静态方法，将它们变成原型方法，我在结束这章时给一个函数变换方法。

startsWith 方法：判定目标字符串是否位于原字符串的开始之处，可以说是 contains 方法的变种。

```
//最后一参数是忽略大小写
function startsWith(target, str, ignorecase) {
   var start_str = target.substr(0, str.length);
   return ignorecase ? start_str.toLowerCase() === str.toLowerCase() :
          start_str === str;
}
```

endsWith 方法：与 startsWith 相反。

```
//最后一参数是忽略大小写
function endsWith(target, str, ignorecase) {
   var end_str = target.substring(target.length - str.length);
   return ignorecase ? end_str.toLowerCase() === str.toLowerCase() :
          end_str === str;
}
```

repeat 方法：将一个字符串重复自身 N 次，如 repeat（"ruby", 2）得到 rubyruby。

版本 1：利用空数组的 join 方法。

```
function repeat(target, n) {
    return (new Array(n + 1)).join(target);
}
```

版本 2：版本 1 的改良版，创建一个对象，拥有 length 属性，然后利用 call 方法去调用数组原型的 join 方法，省去创建数组这一步，性能大为提高。重复次数越多，两者对比越明显。另，之所以要创建一个带 length 属性的对象，是因为要调用数组的原型方法，需要指定 call 的第一个参数为类数组对象。而类数组对象的必要条件是其 length 属性的值为非负整数。

```
function repeat(target, n) {
    return Array.prototype.join.call({
        length: n + 1
    }, target);
}
```

版本 3：版本 2 的改良版，利用闭包将类数组对象与数组原型的 jion 方法缓存起来，省得每次都重复创建与寻找方法。

```
var repeat = (function() {
    var join = Array.prototype.join, obj = {};
    return function(target, n) {
        obj.length = n + 1;
        return join.call(obj, target);
    }
})();
```

版本 4：从算法上着手，使用二分法，比如我们将 ruby 重复 5 次，其实我们在第二次已得 rubyruby，那么 3 次直接用 rubyruby 进行操作，而不是用 ruby。

```
function repeat(target, n) {
    var s = target, total = [];
    while (n > 0) {
        if (n % 2 == 1)
            total[total.length] = s;//如果是奇数
        if (n == 1)
            break;
        s += s;
        n = n >> 1;//相当于将 n 除以 2 取其商,或说开 2 二次方
    }
    return total.join('');
}
```

版本 5：版本 4 的变种，免去创建数组与使用 jion 方法。它的悲剧之处在于它在循环中创建的字符串比要求的还长，需要回减一下。

```
function repeat(target, n) {
    var s = target, c = s.length * n
    do {
        s += s;
    } while (n = n >> 1);
    s = s.substring(0, c);
    return s;
}
```

版本 6：版本 4 的改良版。

```
function repeat(target, n) {
    var s = target, total = "";
    while (n > 0) {
        if (n % 2 == 1)
            total += s;
        if (n == 1)
            break;
        s += s;
        n = n >> 1;
    }
    return total;
}
```

版本 7：与版本 6 相近，不过递归在浏览器下好像都做了优化（包括 IE6），与其他版本相比，属于上乘方案之一。

```
function repeat(target, n) {
    if (n == 1) {
        return target;
    }
    var s = repeat(target, Math.floor(n / 2));
    s += s;
    if (n % 2) {
        s += target;
    }
    return s;
}
```

版本 8：可以说是一个反例，很慢，不过实际上它还是可行的，因此实际上没有人将 n 设成上百成千。

```
function repeat(target, n) {
    return (n <= 0) ? "" : target.concat(repeat(target, --n));
}
```

经测试，版本 6 在各浏览器的得分是最高的。

byteLen 方法：取得一个字符串所有字节的长度。这是一个后端过来的方法，如果将一个英文字符插入数据库 char、varchar、text 类型的字段时占用一个字节，而一个中文字符插入时占用两个字节，为了避免插入溢出，就需要事先判断字符串的字节长度。在前端，如果我们要用户填空的文本，需要字节上的长短限制，比如发短信，也要用到此方法。随着浏览器普及对二进制的操作，这方法也越来越常用。

版本 1：假设字符串每个字符的 Unicode 编码均小于等于 255，byteLength 为字符串长度；再遍历字符串，遇到 Unicode 编码大于 255 时，为 byteLength 补加 1。

```
function byteLen(target) {
    var byteLength = target.length, i = 0;
    for (; i < target.length; i++) {
        if (target.charCodeAt(i) > 255) {
            byteLength++;
```

```
        }
    }
    return byteLength;
}
```

版本 2：使用正则，并支持制定汉字的存储字节数。比如 mysql 存储汉字时，是用 3 个字节数的。

```
function byteLen(target, fix) {
    fix = fix ? fix : 2;
    var str = new Array(fix + 1).join("-")
    return target.replace(/[^\x00-\xff]/g, str).length;
}
```

truncate 方法：用于对字符串进行截断处理，当超过限定长度，默认添加三个点号或其他什么的。

```
function truncate(target, length, truncation) {
    length = length || 30;
    truncation = truncation === void(0) ? '...' : truncation;
    return target.length > length ?
           target.slice(0, length - truncation.length) + truncation : String(target);
}
```

camelize 方法：转换为驼峰风格。

```
function camelize(target) {
    if (target.indexOf('-') < 0 && target.indexOf('_') < 0) {
        return target;//提前判断，提高 getStyle 等的效率
    }
    return target.replace(/[-_][^-_]/g, function(match) {
        return match.charAt(1).toUpperCase();
    });
}
```

underscored 方法：转换为下划线风格。

```
function underscored(target) {
    return target.replace(/([a-z\d])([A-Z])/g, '$1_$2').
           replace(/\-/g, '_').toLowerCase();
}
```

dasherize 方法：转换为连字符风格，亦即 CSS 变量的风格。

```
function dasherize(target) {
    return underscored(target).replace(/_/g, '-');
}
```

capitalize 方法：首字母大写。

```
function capitalize(target) {
    return target.charAt(0).toUpperCase() + target.substring(1).toLowerCase();
}
```

stripTags 方法：移除字符串中的 html 标签，但这方法有缺陷，如里面有 script 标签，会把这些

不该显示出来的脚本也显示出来。在 Prototype.js 中，它与 strip、stripScripts 是一组方法。

```
function stripTags(target) {
   return String(target || "").replace(/<[^>]+>/g, '');
}
```

stripScripts 方法：移除字符串中所有的 script 标签。弥补 stripTags 方法的缺陷。此方法应在 stripTags 之前调用。

```
function stripScripts(target) {
   return String(target || "").replace(/<script[^>]*>([\S\s]*?)<\/script>/img, '')
}
```

escapeHTML 方法：将字符串经过 html 转义得到适合在页面中显示的内容，如将<替换为<。

```
function escapeHTML(target) {
   return target.replace(/&/g, '&amp;')
          .replace(/</g, '&lt;')
          .replace(/>/g, '&gt;')
          .replace(/"/g, "&quot;")
          .replace(/'/g, "&#39;");
}
```

unescapeHTML：将字符串中的 html 实体字符还原为对应字符。

```
function unescapeHTML(target) {
   return  target.replace(/&quot;/g, '"')
          .replace(/&lt;/g, '<')
          .replace(/&gt;/g, '>')
          .replace(/&amp;/g, "&") //处理转义的中文和实体字符
          .replace(/&#([\d]+);/g, function($0, $1) {
       return String.fromCharCode(parseInt($1, 10));
   });
}
```

escapeRegExp 方法：将字符串安全格式化为正则表达式的源码。

```
function escapeRegExp(target) {
   return target.replace(/([-.*+?^${}()|[\]\/\\])/g, '\\$1');
}
```

pad 方法：与 trim 相反，pad 可以为字符串的某一端添加字符串。常见的用法如日历在月份前补零，因此也被称之为 fillZero。我在博客上收集许多版本的实现，在这里转换静态方法一并放出。

版本 1：数组法，创建数组来放置填充物，然后再在右边起截取。

```
function pad(target, n) {
   var zero = new Array(n).join('0');
   var str = zero + target;
   var result = str.substr(-n);
   return result;
}
```

版本 2：版本 1 的变种。

```
function pad(target, n) {
    return Array((n + 1) - target.toString().split('').length).join('0') + target;
}
```

版本 3：二进制法。前半部分是创建一个含有 n 个零的大数，如（1<<5）.toString（2），生成 100000，（1<<8）.toString（2）生成 100000000，然后再截短。

```
function pad(target, n) {
    return (Math.pow(10, n) + "" + target).slice(-n);
}
```

版本 4：Math.pow 法，思路同版本 3。

```
function pad(target, n) {
    return ((1 << n).toString(2) + target).slice(-n);
}
```

版本 5：toFixed 法，思路与版本 3 差不多，创建一个拥有 *n* 个零的小数，然后再截短。

```
function pad(target, n) {
    return (0..toFixed(n) + target).slice(-n);
}
```

版本 6：创建一个超大数，在常规情况下是截不完的。

```
function pad(target, n) {
    return (1e20 + "" + target).slice(-n);
}
```

版本 7：质朴长存法，就是先求得长度，然后一个个地往左边补零，加到长度为 n 为止。

```
function pad(target, n) {
    var len = target.toString().length;
    while (len < n) {
        target = "0" + target;
        len++;
    }
    return target;
}
```

版本 8：也就是现在 mass Framework 使用的版本，支持更多参数，允许从左或从右填充，以及使用什么内容进行填充。

```
function pad(target, n, filling, right, radix) {
    var num = target.toString(radix || 10);
    filling = filling || "0";
    while (num.length < n) {
        if (!right) {
            num = filling + num;
        } else {
            num += filling;
        }
```

```
    }
    return num;
}
```

wbr 方法：为目标字符串添加 wbr 软换行。不过需要注意的是，它并不是在每个字符之后都插入<wbr>字样，而是相当于在组成文本节点的部分中的每个字符后插入<wbr>字样。如 aabbcc，返回 a<wbr>a<wbr>b<wbr>b<wbr>c<wbr>c<wbr>。另外，在 Opera 下，浏览器默认 css 不会为 wbr 加上样式，导致没有换行效果，可以在 css 中加上 wbr：after { content: "\00200B" } 解决此问题。

```
function wbr(target) {
    return String(target)
           .replace(/(?:<[^>]+>)|(?:&#?[0-9a-z]{2,6};)|(.{1})/gi, '$&<wbr>')
           .replace(/><wbr>/g, '>');
}
```

format 方法：在 C 语言中，有一个叫 printf 的方法，我们可以在后面添加不同的类型的参数嵌入到将要输出的字符串中。这是非常有用的方法，因为在 JavaScript 涉及大量这样的字符串拼接工作。如果涉及逻辑，我们可以用模板，如果轻量点，我们可以用这个方法。它在不同框架名字是不同的，Prototype.js 叫 interpolate, Base2 叫 format，mootools 叫 substitute。

```
function format(str, object) {
    var array = Array.prototype.slice.call(arguments, 1);
    return str.replace(/\\?\#{([^{}]+)\}/gm, function(match, name) {
        if (match.charAt(0) == '\\')
            return match.slice(1);
        var index = Number(name)
        if (index >= 0)
            return array[index];
        if (object && object[name] !== void 0)
            return  object[name];
        return  '';
    });
}
```

它支持两种传参方法，如果字符串的占位符为 0、1、2 这样的非零整数形式，要求传入两个或两个以上的参数，否则就传入一个对象，键名为占位符。

```
var a = format("Result is #{0},#{1}", 22, 33);
alert(a);//"Result is 22,33"
var b = format("#{name} is a #{sex}", {
    name: "Jhon",
    sex: "man"
});
alert(b);//"Jhon is a man"
```

quote 方法：在字符串两端添加双引号，然后内部需要转义的地方都要转义，用于接装 JSON 的键名或模析系统中。

```
//http://code.google.com/p/jquery-json/
var escapeable = /["\\\x00-\x1f\x7f-\x9f]/g,
```

```
        meta = {
    '\b': '\\b',
    '\t': '\\t',
    '\n': '\\n',
    '\f': '\\f',
    '\r': '\\r',
    '"': '\\"',
    '\\': '\\\\'
};
function quote(target) {
    if (target.match(escapeable)) {
        return '"' + target.replace(escapeable, function(a) {
            var c = meta[a];
            if (typeof c === 'string') {
                return c;
            }
            return '\\u' + ('0000' + c.charCodeAt(0).toString(16)).slice(-4)
        }) + '"';
    }
    return '"' + target + '"';
}
```

当然，如果浏览器已经支持原生 JSON，我们直接用 JSON.stringify 就行了，另，FF 在 JSON 发明之前，就支持 String.prototype.quote 与 String.quote 方法，我们在使用 quote 之前判定浏览器是否内置这些方法。

字符串好像没有大的浏览器兼容问题，有的话是 IE6、IE7 不支持用数组中括号取它的每一个字符，需要用 charAt 来取；IE6、IE7、IE8 不支持垂直分表符,因此有如下 hack。

```
var isIE678= !+"\v1" ;
```

好了，我们来修复旧版本 IE 中的 trim 函数。这是一个很常用的操作，通常用于表单验证，我们需要把两端的空白去掉，清除“杂质”后，或转换数值进行范围验证，或进行空白验证，或字数验证……由于太常用，相应的实现也非常多。我们可以一起看看，顺便学习一下正则。

版本 1：看起来不怎么样，动用了两次正则替换，实际速度非常惊人，主要得益于浏览器的内部优化。base2 类库使用这种实现。在 Chrome 刚出来的年代，这实现是异常快的，但 Chrome 对字符串方法的疯狂优化，引起了其他浏览器的跟风。于是正则的实现再也比不了字符串方法了。一个著名的例子字符串拼接，直接相加比用 Array 做成的 StringBuffer 还快，而 StringBuffer 技术在早些年备受推崇！

```
function trim(str) {
    return str.replace(/^\s\s*/, '').replace(/\s\s*$/, '');
}
```

版本 2：和版本 1 很相似，但稍慢一点，主要原因是它最先是假设至少存在一个空白符。Prototype.js 使用这种实现，不过其名字为 strip，因为 Prototype 的方法都是力求与 Ruby 同名。

```
function trim(str) {
    return str.replace(/^\s+/, '').replace(/\s+$/, '');
}
```

版本 3：截取方式取得空白部分（当然允许中间存在空白符），总共调用了四个原生方法。设计得非常巧妙，substring 以两个数字作为参数。Math.max 以两个数字作参数，search 则返回一个数字。速度比上面两个慢一点，但基本比 10 之前的版本快！

```
function trim(str) {
   return str.substring(Math.max(str.search(/\S/), 0),
          str.search(/\S\s*$/) + 1);
}
```

版本 4：这个可以称得上版本 2 的简化版，就是利用候选操作符连接两个正则。但这样做就失去了浏览器优化的机会，比不上版本三。由于看来很优雅，许多类库都使用它，如 jQuery 与 mootools。

```
function trim (str) {
   return str.replace(/^\s+|\s+$/g, '');
}
```

版本 5：match 如果能匹配到东西会返回一个类数组对象，原字符匹配部分与分组将成为它的元素。为了防止字符串中间的空白符被排除，我们需要动用到非捕获性分组（?:exp）。由于数组可能为空，我们在后面还要做进一步的判定。好像浏览器在处理分组上比较无力，一个字慢。所以不要迷信正则，虽然它基本上是万能的。

```
function trim(str) {
   str = str.match(/\S+(?:\s+\S+)*/);
   return str ? str[0] : '';
}
```

版本 6：把符合要求的部分提供出来，放到一个空字符串中。不过效率很差，尤其是在 IE6 中。

```
function trim(str) {
   return str.replace(/^\s*(\S*(\s+\S+)*)\s*$/, '$1');
}
```

版本 7：与版本 6 很相似，但用了非捕获分组进行了优点，性能效之有一点点提升。

```
function trim(str) {
   return str.replace(/^\s*(\S*(?:\s+\S+)*)\s*$/, '$1');
}
```

版本 8：沿着上面两个的思路进行改进，动用了非捕获分组与字符集合，用?顶替了*，效果非常惊人。尤其在 IE6 中，可以用疯狂来形容这次性能的提升，直接秒杀 FF3。

```
function trim(str) {
   return str.replace(/^\s*((?:[\S\s]*\S)?)\s*$/, '$1');
}
```

版本 9：这次是用懒惰匹配顶替非捕获分组，在火狐中得到改善，IE 没有上次那么疯狂。

```
function trim(str) {
   return str.replace(/^\s*([\S\s]*?)\s*$/, '$1');
}
```

版本 10：我只想说，搞出这个的人已经不是用厉害来形容，已是专家级别了。它先是把可能的空白符全部列出来，在第一次遍历中砍掉前面的空白，第二次砍掉后面的空白。全过程只用了 indexOf 与 substring 这个专门为处理字符串而生的原生方法，没有使用到正则。速度快得惊人，估计直逼内部的二进制实现，并且在 IE 与火狐（其他浏览器当然也毫无疑问）都有良好的表现。速度都是零毫秒级别的。PHP.js 就收纳了这个方法。

```
function trim(str) {
    var whitespace = ' \n\r\t\f\x0b\xa0\u2000\u2001\u2002\u2003\n\
  \u2004\u2005\u2006\u2007\u2008\u2009\u200a\u200b\u2028\u2029\u3000';
    for (var i = 0; i < str.length; i++) {
        if (whitespace.indexOf(str.charAt(i)) === -1) {
            str = str.substring(i);
            break;
        }
    }
    for (i = str.length - 1; i >= 0; i--) {
        if (whitespace.indexOf(str.charAt(i)) === -1) {
            str = str.substring(0, i + 1);
            break;
        }
    }
    return whitespace.indexOf(str.charAt(0)) === -1 ? str : '';
}
```

版本 11：实现 10 的字数压缩版，前面部分的空白由正则替换负责砍掉，后面用原生方法处理，效果不逊于原版，但速度都非常逆天。

```
function trim(str) {
    str = str.replace(/^\s+/, '');
    for (var i = str.length - 1; i >= 0; i--) {
        if (/\S/.test(str.charAt(i))) {
            str = str.substring(0, i + 1);
            break;
        }
    }
    return str;
}
```

版本 12：版本 10 更好的改进版，注意说的不是性能速度，而是易记与使用方面。

```
function trim(str) {
    var str = str.replace(/^\s\s*/, ""),
            ws = /\s/,
            i = str.length;
    while (ws.test(str.charAt(--i)))
        return str.slice(0, i + 1);
}
```

版本 13：原作者@ialeafs 称它为 trimChunge，通过字符的 charCodeAt 值来判定是否为空白，速度也非常逆天，它仅次于版本 10，快于版本 11、12，不过此版本能处理的空白很有限。

```
function trim(str) {
    var m = str.length;
    for (var i = -1; str.charCodeAt(++i) <= 32; )
    for (var j = m - 1; j > i && str.charCodeAt(j) <= 32; j--)
    return str.slice(i, j + 1);
}
```

但这还没有完。如果你经常翻看 jQuery 的实现，你就会发现 jQuery1.4 之后的 trim 实现，多出了一个对 xA0 的特别处理。这是 Prototype.js 的核心成员 •kangax 的发现，IE 或早期的标准浏览器在字符串的处理上都有 BUG，把许多本属于空白的字符没有列为\s，jQuery 在 1.42 中也不过把常见的不断行空白 xA0 修复掉，并不完整，因此最佳方案还是版本 10。

3.2 数组的扩展与修复

得益于 Prototype.js 的 ruby 式数组方法的侵略，让 Jser()前端工程师大开眼界，原来对数组的操作也如此丰富多彩的。原来 JavaScript 的数组方法基于上就是栈与队列的那一套，像 splice 还是很晚加入的。让我们回顾一下它们的用法。

pop 方法：出栈操作，删除并返回数组的最后一个元素。

push 方法：入栈操作，向数组的末尾添加一个或更多元素，并返回新的长度。

shift 方法：出队操作，删除并返回数组的第一个元素。

unshift 方法：入队操作，向数组的开头添加一个或更多元素，并返回新的长度。

slice 方法：切片操作，从数组中分离出一个子数组，功能类似于字符串的 substring、slice、substr 这三兄弟。此方法也常用于转换类数组对象为真正的数组。

sort 方法：对数组的元素进行排序，有一个可选参数，为比较函数。

reverse 方法：颠倒数组中元素的顺序。

splice 方法：可以同时用于原数组进行增删操作，数组的 remove 方法就是基于它写成的。

concat 方法：用于把原数组与参数合并成一个新数组，如果参数为数组，那么它会把其第一维的元素放入新数组中。因此我们可以利用它实现数组的平坦化操作与克隆操作。

join 方法：把数组的所有元素放入一个字符串。元素通过指定的分隔符进行分隔。你可以想象成字符串的 split 的反操作。

在 ecma262v5 中，它把标准浏览器早已实现的几个方法进行了入户处理，从此我们可以安心使用 forEach 等方法，不用担心它们忽然被废弃掉了。

indexOf 方法：定位操作，返回数组中第一个等于给定参数的元素的索引值。

lastIndexOf 方法：定位操作，同上，不过是从后遍历。索引操作可以说是字符串的同名方法的翻版，存在就返回非负整数，不存在就返回–1。

forEach 方法：迭代操作，将数组的元素依次传入一个函数中执行。Ptototype.js 的对应名字为 each。

map 方法：收集操作，将数组的元素依次传入一个函数中执行，然后把它们的返回值组成一个新数组返回。Ptototype.js 的对应名字为 collect。

filter 方法：过滤操作，将数组的元素依次传入一个函数中执行，然后把返回值为 true 的那个元素放入新数组返回。在 Prototype.js 中，它有三个名字，select、filter、findAll。

some 方法：只要数组中有一个元素满足条件（放进给定函数返回 true），那么它就返回 true。Ptototype.js 的对应名字为 any。

every 方法：只有数组中的元素都满足条件（放进给定函数返回 true），它才返回 true。Ptototype.js 的对应名字为 all。

reduce 方法：归化操作。将数组中的元素中归化为一个简单的数值。Ptototype.js 的对应名字为 inject。

reduceRight 方法：归化操作，同上，不过是从后遍历。

由于许多扩展也基于这些新的标准化方法，因此我先给出 IE6、IE7、IE8 的兼容方案，全部在数组原型上修复它们。

```
Array.prototype.indexOf = function(item, index) {
    var n = this.length, i = ~~index;
    if (i < 0)
        i += n;
    for (; i < n; i++)
        if (this[i] === item)
            return i;
    return -1;
}

Array.prototype.lastIndexOf = function(item, index) {
    var n = this.length,
           i = index == null ? n - 1 : index;
    if (i < 0)
        i = Math.max(0, n + i);
    for (; i >= 0; i--)
        if (this[i] === item)
            return i;
    return -1;
}
```

像 forEach、map、filter、some、every 这几个方法，在结构上非常相似，我们可以这样生成它们。

```
function iterator(vars, body, ret) {
    var fun = 'for(var ' + vars + 'i=0,n = this.length;i < n;i++){' +
           body.replace('_', '((i in this) && fn.call(scope,this[i],i,this))')
           + '}' + ret
    return Function("fn,scope", fun);
}

Array.prototype.forEach = iterator('', '_', '');

Array.prototype.filter = iterator('r=[],j=0,', 'if(_)r[j++]=this[i]', 'return r');

Array.prototype.map = iterator('r=[],', 'r[i]=_', 'return r');

Array.prototype.some = iterator('', 'if(_)return true', 'return false');

Array.prototype.every = iterator('', 'if(!_)return false', 'return true');
```

造轮子的同学要注意一下，数组中的空元素是不会在上述方法中遍历出来的。

```
[1, 2, , 4].forEach(function(e) {
    console.log(e)
});
//依次打印出 1，2，4，忽略第二、第三个逗号间的空元素
```

reduce 与 reduceRight 是一组，我们可以利用 reduce 方法创建 reduceRight 方法。

```
Array.prototype.reduce = function(fn, lastResult, scope) {
    if (this.length == 0)
        return lastResult;
    var i = lastResult !== undefined ? 0 : 1;
    var result = lastResult !== undefined ? lastResult : this[0];
    for (var n = this.length; i < n; i++)
        result = fn.call(scope, result, this[i], i, this);
    return result;
}

Array.prototype.reduceRight = function(fn, lastResult, scope) {
    var array = this.concat().reverse();
    return array.reduce(fn, lastResult, scope);
}
```

接着下来，我们看看主流库为数组增加了那些扩展吧，除去上述那些。

Prototype.js 的数组扩展：eachSlice、detect、grep、include、inGroupsOf、invoke、max、 min、partition、pluck、reject、sortBy、zip、size、clear、first、last、compact、flatten、without、 uniq、intersect、clone、inspect。

Rightjs 的数组扩展：include、clean、clone、compact、empty、first、flatten、includes、last、max、merge、min、random、reject、shuffle、size、sortBy、sum、uniq、walk、without。

mootools 的数组扩展：clean、invoke、associate、link、contains、append、getLast、getRandom、include、combine、erase、empty、flatten、pick、hexToRgb、rgbToHex。

EXT 的数组扩展：contains、pluck、clean、unique、from、remove、include、clone、merge、intersect、difference、flatten、min、max、mean、sum、erase、insert。

Underscore.js 的数组扩展：detect、reject、invoke、pluck、sortBy、groupBy、sortedIndex、first、last、compact、flatten、without、union、intersection、difference、uniq、zip。

qooxdoo 的数组扩展：insertAfter、insertAt、insertBefore、max、min、remove、removeAll、removeAt、sum、unique。

Tangram 的数组扩展：contains、empty、find、remove、removeAt、unique。

我们可以发现，Prototype.js 那一套方法影响深远，许多库都有它的影子，全面而细节地囊括了各种操作，大家可以根据自己的需要与框架宗旨制定自己的数组扩展。我在这方面的考量如下，至少要包含平坦化、去重、乱序、移除这几个操作，其次是两个集合间的操作，如取并集、差集、交集。

下面是各种具体实现。

contains 方法：判定数组是否包含指定目标。

```
function contains(target, item) {
    return target.indexOf(item) > -1
}
```

removeAt 方法：移除数组中指定位置的元素，返回布尔表示成功与否。

```
function removeAt(target, index) {
    return !!target.splice(index, 1).length
}
```

remove 方法：移除数组中第一个匹配传参的那个元素，返回布尔表示成功与否。

```
function remove(target, item) {
    var index = target.indexOf(item);
    if (~index)
        return removeAt(target, index);
    return false;
}
```

shuffle 方法：对数组进行洗牌。若不想影响原数组，可以先拷贝一份出来操作。有关洗牌算法的情况，可以见这篇博文：http://bost.ocks.org/mike/shuffle/。

```
function shuffle(target) {
    var j, x, i = target.length;
    for (; i > 0; j = parseInt(Math.random() * i),
           x = target[--i], target[i] = target[j], target[j] = x) {
    }
    return target;
}
```

random 方法：从数组中随机抽选一个元素出来。

```
function random(target) {
    return  target[Math.floor(Math.random() * target.length)];
}
```

flatten 方法：对数组进行平坦化处理，返回一个一维的新数组。

```
function flatten(target) {
    var result = [];
    target.forEach(function(item) {
        if (Array.isArray(item)) {
            result = result.concat(flatten(item));
        } else {
            result.push(item);
        }
    });
    return result;
}
```

unique 方法：对数组进行去重操作，返回一个没有重复元素的新数组。

```
function unique(target) {
    var result = [];
    loop:  for (var i = 0, n = target.length; i < n; i++) {
        for (var x = i + 1; x < n; x++) {
```

```
            if (target[x] === target[i])
                continue loop;
        }
        result.push(target[i]);
    }
    return result;
}
```

compact 方法：过滤数组中的 null 与 undefined，但不影响原数组。

```
function compact(target) {
    return target.filter(function(el) {
        return el != null;
    });
}
```

pluck 方法：取得对象数组的每个元素的指定属性，组成数组返回。

```
function pluck(target, name) {
    var result = [], prop;
    target.forEach(function(item) {
        prop = item[name];
        if (prop != null)
            result.push(prop);
    });
    return result;
}
```

groupBy 方法：根据指定条件（如回调或对象的某个属性）进行分组，构成对象返回。

```
function groupBy(target, val) {
    var result = {};
    var iterator = $.isFunction(val) ? val : function(obj) {
        return obj[val];
    };
    target.forEach(function(value, index) {
        var key = iterator(value, index);
        (result[key] || (result[key] = [])).push(value);
    });
    return result;
}
```

sortBy 方法：根据指定条件进行排序，通常用于对象数组。

```
function sortBy(target, fn, scope) {
    var array = target.map(function(item, index) {
        return {
            el: item,
            re: fn.call(scope, item, index)
        };
    }).sort(function(left, right) {
        var a = left.re, b = right.re;
        return a < b ? -1 : a > b ? 1 : 0;
    });
    return pluck(array, 'el');
}
```

union 方法：对两个数组取并集。

```
function union(target, array) {
    return unique(target.concat(array));
}
```

intersect 方法：对两个数组取交集。

```
function intersect(target, array) {
    return target.filter(function(n) {
        return ~array.indexOf(n);
    });
}
```

diff 方法：对两个数组取差集（补集）。

```
function diff(target, array) {
    var result = target.slice();
    for (var i = 0; i < result.length; i++) {
        for (var j = 0; j < array.length; j++) {
            if (result[i] === array[j]) {
                result.splice(i, 1);
                i--;
                break;
            }
        }
    }
    return result;
}
```

min 方法：返回数组中的最小值，用于数字数组。

```
function min(target) {
    return Math.min.apply(0, target);
}
```

max 方法：返回数组中的最大值，用于数字数组。

```
function max(target) {
    return Math.max.apply(0, target);
}
```

基于上这么多了，如果你想实现 sum 方法，可以使用 reduce 方法。我们再来抹平 Array 原生方法在各浏览器的差异，一个是 IE6、IE7 下 unshift 不返回数组长度的问题，一个 splice 的参数问题。unshift BUG 很容易修复的，使用函数劫持。

```
if ([].unshift(1) !== 1) {
    var _unshift = Array.prototype.unshift;
    Array.prototype.unshift = function() {
        _unshift.apply(this, arguments);
        return this.length; //返回新数组的长度
    }
}
```

splice 在一个参数的情况下，IE6、IE7、IE8 默认第二个参数为零，其他浏览器为数组的长度，当然我们要以标准浏览器为准！最简单的修复如下。

```
if ([1, 2, 3].splice(1).length == 0) {//如果是 IE6、IE7、IE8，则一个元素也没有删除
    var _splice = Array.prototype.splice;
    Array.prototype.splice = function(a) {
        if (arguments.length == 1) {
            return _splice.call(this, a, this.length)
        } else {
            return _splice.apply(this, arguments)
        }
    }
}
```

另一种方法是使用 slice 进行实现，因此 slice 对待第二个参数的方式与标准的 splice 一致！

```
Array.prototype.splice = function(x, y) {
    var a = arguments, s = a.length - 2 - y, r = this.slice(x, x + y);
    if (s > 0) {
        for (var i = this.length - 1, j = x + y; i >= j; --i)
            this[i + s] = this[i];
    }
    else if (s < 0) {
        for (var i = x + y, j = this.length; i < j; ++i)
            this[i + s] = this[i];
        this.length += s;
    }
    for (var i = 2, j = a.length; i < j; ++i)
        this[i - 2 + x] = a[i];
    return r;
}
```

或者干脆自己实现一个，不利用任何原生方法：

```
Array.prototype.splice = function(s, d) {
    var max = Math.max, min = Math.min,
            a = [], i = max(arguments.length - 2, 0),
            k = 0, l = this.length, e, n, v, x;
    s = s || 0;
    if (s < 0) {
        s += l;
    }
    s = max(min(s, l), 0);
    d = max(min(isNumber(d) ? d : l, l - s), 0);
    v = i - d;
    n = l + v;
    while (k < d) {
        e = this[s + k];
        if (e !== void 0) {
            a[k] = e;
        }
        k += 1;
    }
    x = l - s - d;
```

```
    if (v < 0) {
        k = s + i;
        while (x) {
            this[k] = this[k - v];
            k += 1;
            x -= 1;
        }
        this.length = n;
    } else if (v > 0) {
        k = 1;
        while (x) {
            this[n - k] = this[l - k];
            k += 1;
            x -= 1;
        }
    }
    for (k = 0; k < i; ++k) {
        this[s + k] = arguments[k + 2];
    }
    return a;
}
```

一旦有了 splice 方法，我们也可以自行实现自己的 pop、push、shift、unshift 方法，因此你明白为什么这几个方法是直接修改原数组了吧？浏览器商的思路与我们一样，大概也是用 splice 方法来实现它们！

```
var _slice = Array.prototype.slice;
Array.prototype.pop = function() {
    return this.splice(this.length - 1, 1)[0];
}

Array.prototype.push = function() {
    this.splice.apply(this,
           [this.length, 0].concat(_slice.call(arguments)));
    return this.length;
}

Array.prototype.shift = function() {
    return this.splice(0, 1)[0];
}

Array.prototype.unshift = function() {
    this.splice.apply(this,
           [0, 0].concat(_slice.call(arguments)));
    return this.length;
}
```

3.3 数值的扩展与修复

数值没有什么好扩展的，而且 JavaScript 的数值精度问题一向臭名昭著，要修复它们可不是一两行代码了事。先看扩展，我们只把目光集中于 Prototype.js 与 mootools 就行了。

Prototype.js 为它添加 8 个原型方法。Succ 是加 1，times 是将回调重复执行指定次数 toPaddingString 与上面提到字符串扩展方法 pad 作用一样，toColorPart 是转十六进制，abs、ceil、floor、abs 是从 Math 中偷来的。

mootools 的情况：limit 是从数值限定在一个闭开间中，如果大于或小于其边界，则等于其最大值或最小值，times 与 Prototype.js 的用法相似， round 是 Math.round 的增强版，添加了精度控制，toFloat、toInt 是从 window 中偷来的，其他的则是从 Math 中偷来的。

在 es5_shim.js 这个库，它实现了 ECMA262v5 提到的一个内部方法 `toInteger`。

```
// http://es5.github.com/#x9.4
// http://jsperf.com/to-integer
var toInteger = function(n) {
    n = +n;
    if (n !== n) { // isNaN
        n = 0;
    } else if (n !== 0 && n !== (1 / 0) && n !== -(1 / 0)) {
        n = (n > 0 || -1) * Math.floor(Math.abs(n));
    }
    return n;
};
```

但依我看来都没什么意义，数值往往来自用户输入，我们一个正则就能判定它是不是一个“数”，是则直接 Number（n）！

基于同样的理由，mass Framework 对数字的扩展也是很少的，三个独立的扩展。

limit 方法：确保数值在［n1，n2］闭区间之内,如果超出限界，则置换为离它最近的最大值或最小值。

```
function limit(target, n1, n2) {
    var a = [n1, n2].sort();
    if (target < a[0])
        target = a[0];
    if (target > a[1])
        target = a[1];
    return target;
}
```

nearer 方法：求出距离指定数值最近的那个数。

```
function nearer(target, n1, n2) {
    var diff1 = Math.abs(target - n1),
            diff2 = Math.abs(target - n2);
    return diff1 < diff2 ? n1 : n2
}
```

Number 下唯一需要修复的方法是 toFixed，它是用于校正精确度，最后的那个数会做四舍五入操作，但在一些浏览器中并没有这样干。

想简单修复可以这样处理：

```
if (0.9.toFixed(0) !== '1') {
    Number.prototype.toFixed = function(n) {
```

```
        var power = Math.pow(10, n);
        var fixed = (Math.round(this * power) / power).toString();
        if (n == 0)
            return fixed;
        if (fixed.indexOf('.') < 0)
            fixed += '.';
        var padding = n + 1 - (fixed.length - fixed.indexOf('.'));
        for (var i = 0; i < padding; i++)
            fixed += '0';
        return fixed;
    };
}
```

追求完美的话，还存在这样一个版本，把里面的加、减、乘、除都重新实现了一遍。

```
if (!Number.prototype.toFixed || (0.00008).toFixed(3) !== '0.000' ||
        (0.9).toFixed(0) === '0' || (1.255).toFixed(2) !== '1.25' ||
        (1000000000000000128).toFixed(0) !== "1000000000000000128") {
    // 一些内部方法与变量,防止全局污染
    (function() {
        var base, size, data, i;

        base = 1e7;
        size = 6;
        data = [0, 0, 0, 0, 0, 0];

        function multiply(n, c) {
            var i = -1;
            while (++i < size) {
                c += n * data[i];
                data[i] = c % base;
                c = Math.floor(c / base);
            }
        }

        function divide(n) {
            var i = size, c = 0;
            while (--i >= 0) {
                c += data[i];
                data[i] = Math.floor(c / n);
                c = (c % n) * base;
            }
        }

        function toString() {
            var i = size;
            var s = '';
            while (--i >= 0) {
                if (s !== '' || i === 0 || data[i] !== 0) {
                    var t = String(data[i]);
                    if (s === '') {
                        s = t;
                    } else {
                        s += '0000000'.slice(0, 7 - t.length) + t;
                    }
```

```
        }
    }
    return s;
}

function pow(x, n, acc) {
    return (n === 0 ? acc : (n % 2 === 1 ? pow(x, n - 1, acc * x)
            : pow(x * x, n / 2, acc)));
}

function log(x) {
    var n = 0;
    while (x >= 4096) {
        n += 12;
        x /= 4096;
    }
    while (x >= 2) {
        n += 1;
        x /= 2;
    }
    return n;
}

Number.prototype.toFixed = function(fractionDigits) {
    var f, x, s, m, e, z, j, k;

    // Test for NaN and round fractionDigits down
    f = Number(fractionDigits);
    f = f !== f ? 0 : Math.floor(f);

    if (f < 0 || f > 20) {
        throw new RangeError("Number.toFixed called with invalid number of decimals");
    }

    x = Number(this);

    // Test for NaN
    if (x !== x) {
        return "NaN";
    }

    // If it is too big or small, return the string value of the number
    if (x <= -1e21 || x >= 1e21) {
        return String(x);
    }

    s = "";

    if (x < 0) {
        s = "-";
        x = -x;
    }

    m = "0";
```

```
            if (x > 1e-21) {
                // 1e-21 < x < 1e21
                // -70 < log2(x) < 70
                e = log(x * pow(2, 69, 1)) - 69;
                z = (e < 0 ? x * pow(2, -e, 1) : x / pow(2, e, 1));
                z *= 0x10000000000000; // Math.pow(2, 52);
                e = 52 - e;

                // -18 < e < 122
                // x = z / 2 ^ e
                if (e > 0) {
                    multiply(0, z);
                    j = f;

                    while (j >= 7) {
                        multiply(1e7, 0);
                        j -= 7;
                    }

                    multiply(pow(10, j, 1), 0);
                    j = e - 1;

                    while (j >= 23) {
                        divide(1 << 23);
                        j -= 23;
                    }

                    divide(1 << j);
                    multiply(1, 1);
                    divide(2);
                    m = toString();
                } else {
                    multiply(0, z);
                    multiply(1 << (-e), 0);
                    m = toString() + '0.00000000000000000000'.slice(2, 2 + f);
                }
            }

            if (f > 0) {
                k = m.length;

                if (k <= f) {
                    m = s + '0.00000000000000000000'.slice(0, f - k + 2) + m;
                } else {
                    m = s + m.slice(0, k - f) + '.' + m.slice(k - f);
                }
            } else {
                m = s + m;
            }

            return m;
        }
    }());
}
```

toFixed 方法实现得如此艰难其实也不能怪浏览器，计算机所理解的数字与我们是不一样的。众所周知，计算机的世界是 2 进制，数字也不例外。为了储存更复杂的结构，需要用到更高维的进制。而进制间的换算是存在误差的。虽然计算机在一定程度上反映了现实世界，但它提供的顶多只是一个“幻影”，经常与我们的常识产生偏差。例如，将 1 除以 3，然后再乘以 3，最后得到的值竟然不是 1；10 个 0.1 相加也不等于 1；交换相加的几个数的顺序，却得到了不同的和[①]。

```
console.log(0.1 + 0.2)
console.log(Math.pow(2, 53) === Math.pow(2, 53) + 1) //true
console.log(Infinity > 100) //true
console.log(JSON.stringify(25001509088465005)) //25001509088465004
console.log(0.10000000000000000000000001) //0.1
console.log(0.10000000000000000000000001) //0.1
console.log(0.10000000000000000000000456) //0.1
console.log(0.099999999999999999999) //0.1
console.log(1 / 3) //0.3333333333333333
console.log(23.53 + 5.88 + 17.64)// 47.05
console.log(23.53 + 17.64 + 5.88)// 47.050000000000004
```

这些其实不是 BUG，而是我们无法接受这事实。在 JavaScript 中，数值有三种保存方式。

- 字符串形式的数值内容。
- IEEE754 标准双精度浮点数，它最多支持小数点后带 15～17 位小数，由于存在 2 进制和 10 进制的转换问题，具体的位数会发生变化。
- 一种类似于 C 语言的 init 类型的 32 位整数，它由 4 个 8 bit 的字节构成，可以保存较小的整数。

当 JavaScript 遇到一个数值时，它会首先尝试按照整数来处理该数值，如行得通，则把数值保存为 31 bit 的整数；如果该数值不能视为整数，或超出 31 bit 的范围，则把数值保存为 64 位的 IEEE 754 浮点数。

聪明的读者一定想到了这样一个问题：什么时候规规矩矩的整数会突然变成捉摸不定的双精度浮点数？答案是：当它们的值变得非常庞大时，或者进入 1 和 0 之间时。因此，1 和 0 是首先必须注意的两个数值。

接下来，最大的 Unicode 值是 1114111（7 位数字，相当于（/x41777777），而最大的 RGB 颜色值是 16777215（8 位数字，相当于#FFFFFF）。最大的 32 bit 带符号整数是 2147483647（10 位数字，即 Math.pow（2，31）-1），-2147483648 最小，所以 JavaScript 内部会以整数的形式保存所有 Unicode 值和 RGB 颜色。2147483647 是第三个必须注意的数值，任何大于该值的数据将保存为双精度格式。

9007199254740992（16 位数字，即 Math.pow（2,53））是最大的浮点数，输出时类似整数，所有 Date 对象（按毫秒计算）都小于该值，因此总是模拟整数的格式输出。它是第四个必须注意的数值。

最后，最大的双精度数值是 1.7976931348623157e+308，超出这个范围就要算作无穷大了。

因此，我们就看出缘由了，大数相加出问题是由于精度的不足，小数相加出问题是进制转算时产生误差。第一个好理解。第二个，主要是我们常用的 10 进制转换为 2 进制时，变成循环小数及

① http://www.html-js.com/article/1630

无理数等有无限多小数位的数，计算机要用位数有限的浮点数来表示是无法实现的，只能从某一位进行截短。而且，因为内部表示是 2 进制，10 进制看起来是能除尽的数，往往在 2 进制是循环小数。比如用 2 进制来表示 10 进制的 0.1，就得写成 2 的幂（因为小于 1，所以幂是负数）相加的形式。若一直持续下去，0.1 就成了 0.00011001100110011001...这种循环小数。在有效数字的范围内进行舍入，就会产生误差。

综上所述，我们就尽量避免小数操作与大数操作，或者转交后台去处理，实在避免不了就引入专业的库来处理。

3.4 函数的扩展与修复

ecma262v5 对函数唯一的扩展就是 bind 函数，众所周知，这是来自 Prototype.js，此外，其他重要的函数都来自 Prototype.js。

Prototype.js 的函数扩展如下。

argumentNames：取得函数的形参，以字符串数组形式返回，这只要用于其类工厂的方法链设计。

bind：不用多言，劫持作用域，并预先添加更多参数。

bindAsEventListener：如 bind 相似，但强制返回函数的第一个参数为事件对象，这是用于修复 IE 的多投事件 API 与标准 API 的差异。

curry：函数柯里化，用于一个操作分成多步进行，并可以改变原函数的行为。

wrap：AOP 的实现。

delay：setTimeout 的偷懒写法。

defer：强制延迟 0.01 秒才执行原函数。

methodize：将一个函数变成其调用对象的方法，这也是为其类工厂的方法链服务。

我们先看 bind 方法，它用到了著名的闭包。所谓闭包，就是一个引用着外部变量的内部函数。比如下面这个函数。

```
var observable = function(val) {
    var cur = val;//一个内部变量
    function field(neo) {
        if (arguments.length) {//setter
            if (cur !== neo) {
                cur = neo;
            }
        } else {//getter
            return cur;
        }
    }
    field();
    return field;
}
```

它里面的 field 函数将与外部的 cur 构成一个闭包。Prototype.js 中的 bind 方法只要依仗原函数与经过切片化的 args 构成闭包，而让这方法名符其实的是那个 curry，用户最初的那个传参，劫持

到返回函数修正 this 的指向。

```
Function.prototype.bind = function(context) {
    if (arguments.length < 2 && context == void 0)
        return this;
    var __method = this, args = [].slice.call(arguments, 1);
    return function() {
        return __method.apply(context, args.concat.apply(args, arguments));
    }
}
```

正因为有这东西，我们才方便修复 IE 多投事件 API attachEvent 回调中的 this 问题，它总是指向 window 对象，而标准浏览器的 addEventListener 中的 this 则为其调用对象。

```
var addEvent = document.addEventListener ?
       function(el, type, fn, capture) {
           el.addEventListener(type, fn, capture)
       } :
       function(el, type, fn) {
           el.attachEvent("on" + type, fn.bind(el, event))
       }
```

ECMA62 v5 对其认证后，唯一的增强是对调用者进行检测，确保它是一个函数。顺便总结一下这 3 样东西。

call 是 obj.method（）到 method（obj）的变换。

apply 是 obj.method（a,b,c）到 method（obj, [a,b,c]）的变幻，它要求第二个参数必须存在，一定是数组或 Arguments 这样的类数组，NodeList 这样具有争议性的东西就不要乱传进去了。因此 jQuery 对两个数组或类数组的合并是使用 jQuery.merge，放弃使用 Array.prototype.push.apply。

bind 就是 apply 的变种，保证返回值是一个函数。

这 3 个方法是非常有用，我们可以设法将它们“偷”出来。

```
var bind = function(bind) {
   return{
       bind: bind.bind(bind),
       call: bind.bind(bind.call),
       apply: bind.bind(bind.apply)
   }
}(Function.prototype.bind)
```

那怎么用它们呢？比如我们想合并两个数组，直接调用 concat 方法如下。

```
var concat = bind.apply([].concat);
var a = [1, [2, 3], 4];
var b = [1, 3];
```

使用 bind.bind 方法可以将它们的结果进一步平坦化。

```
var concat = bind.apply([].concat);
console.log(concat(b, a))//[1,3,1,2,3,4]
```

又如切片化操作，它经常用于转换类数组对象为纯数组的。

```
var slice = bind([].slice)
var array = slice({
    0: "aaa",
    1: "bbb",
    2: "ccc",
    length: 3
});
console.log(array)//[ "aaa", "bbb", "ccc"]
```

更常用的操作是转换 arguments 对象，目的是为了使用数组的一系列方法。

```
function test() {
    var args = slice(arguments)
    console.log(args)//[1,2,3,4,5]
}
test(1, 2, 3, 4, 5)
```

我们可以将 hasOwnProperty 提取出来，判定对象是否在本地就拥有某属性。

```
var hasOwn = bind.call(Object.prototype.hasOwnProperty);
hasOwn([],"xx") // false
//使用 bind.bind 就需要多执行一次
var hasOwn2 = bind.bind(Object.prototype.hasOwnProperty);
hasOwn2([],"xx")() // false
```

上面 bind.bind 的行为其实就是一种 curry，它给了你再一次传参的机会，这样你就可以在内部判定参数的个数，决定是否继续返回函数还是结果。这在设计计算器的连续运算上非常有用。从此角度来看，我们可以得到一信息，bind 着重在于作用域的劫持，curry 在于参数的不断补充。

我们可以编写如下一个 curry，当所有步骤输入的参数个数等于最初定义时的函数的形参个数时，就执行它。

```
function curry(fn) {
    function inner(len, arg) {
        if (len == 0)
            return fn.apply(null, arg);
        return function(x) {
            return inner(len - 1, arg.concat(x));
        };
    }
    return inner(fn.length, []);
}

function sum(x, y, z, w) {
    return x + y + z + w;
}
curry(sum)('a')('b')('c')('d'); // => 'abcd'
```

不过这里我们假定了用户每次都只传入一个参数，我们可以改进一下。

```
function curry2(fn) {
    function inner(len, arg) {
        if (len <= 0)
            return fn.apply(null, arg);
        return function() {
```

```
            return inner(len - arguments.length,
                  arg.concat(Array.apply([], arguments)));
        };
    }
    return inner(fn.length, []);
}
```

这样就可以在中途传递多个参数，或不传递参数。

```
curry2(sum)('a')('b', 'c')('d'); // => 'abcd'
curry2(sum)('a')()('b', 'c')()('d'); // => 'abcd'
```

不过，上面的函数形式有个更帅气的名称，叫 self-curry 或 recurry。它强调的是递归调用自身来补全参数。

与 curry 相似的 partial。curry 的不足的参数总是通过 push 的方式来补全，而 partial 则是在定义时所有参数已经都有了，但某些位置上的参数只是个占位符，我们在接着下来的传参只是替换掉它们。博客上有专文《Partial Application in JavaScript》介绍这个内容。

```
Function.prototype.partial = function() {
    var fn = this, args = Array.prototype.slice.call(arguments);
    return function() {
        var arg = 0;
        for (var i = 0; i < args.length && arg < arguments.length; i++)
            if (args[i] === undefined)
                args[i] = arguments[arg++];
        return fn.apply(this, args);
    };
}
```

它是使用 undefined 作为占位符的。

```
var delay = setTimeout.partial(undefined, 10);
//接下来的工作就是代替掉第一个参数
delay(function() {
    alert("A call to this function will be temporarily delayed.");
})
```

有关这个占位符，该博客的评论列表中也有大量的讨论，最后确定下来是使用_作为变量名，内部还是指向 undefined。我认为这样做还是比较危险的，框架应该提供一个特殊的对象，如 Prototype 在内部使用$break = {}作为断点的标识。我们可以用一个纯空对象作为 partial 的占位符。

```
var _ = Object.create(null)
```

纯空对象没有原型，没有 toString, valueOf 等继承自 Object 的方法，是一个很特别的东西。在 IE 下我们可以这样模拟它：

```
var _ = (function() {
    var doc = new ActiveXObject('htmlfile')
    doc.write('<script><\/script>')
    doc.close()
    var Obj = doc.parentWindow.Object
    if (!Obj || Obj === Object)
        return
```

```
    var name, names =
          ['constructor', 'hasOwnProperty', 'isPrototypeOf'
                , 'propertyIsEnumerable', 'toLocaleString', 'toString', 'valueOf']
    while (name = names.pop())
       delete Obj.prototype[name]
    return Obj
}())

function partial(fn) {
    var A = [].slice.call(arguments, 1);
    return A.length < 1 ? fn : function() {
       var a = Array.apply([], arguments);
       var c = A.concat();//复制一份
       for (var i = 0; i < c.length; i++) {
          if (c[i] === _) {//替换占位符
             c[i] = a.shift();
          }
       }
       return fn.apply(this, c.concat(a));
    }
}
function test(a, b, c, d) {
    return "a = " + a + " b = " + b + " c = " + c + " d = " + d
}
var fn = partail(test, 1, _, 2, _);
fn(44, 55)// "a = 1 b = 44 c = 2 d = 55"
```

curry、partial 的应用场景在前端世界真心不多，前端讲究的是即时显示，许多 API 都是同步的，后端由于 IO 操作等耗时够长，像 node.js 提供了大量的异步函数来提高性能，防止堵塞。但是过多异步函数也必然带来回调嵌套的问题，因此我们需要通过 curry 等函数变换，将套嵌减少到可以接受的程度。这个我会在 Ajax 的章节讲述它们的使用方法的。

函数的修复。这涉及两个方法 apply 与 call，这两个方法的本质就是生成一个新的函数，将原函数与用户传参放到里面执行而已。在 JavaScript 创建一个函数有很多办法，常见的有函数声明和函数表达式，次之是函数构造器，再次是 eval, setTimeout……

```
Function.prototype.apply || (Function.prototype.apply = function (x, y) {
    x = x || window;
    y = y ||[];
    x.__apply = this;
    if (!x.__apply)
       x.constructor.prototype.__apply = this;
    var r, j = y.length;
    switch (j) {
       case 0: r = x.__apply(); break;
       case 1: r = x.__apply(y[0]); break;
       case 2: r = x.__apply(y[0], y[1]); break;
       case 3: r = x.__apply(y[0], y[1], y[2]); break;
       case 4: r = x.__apply(y[0], y[1], y[2], y[3]); break;
       default:
          var a = [];
          for (var i = 0; i < j; ++i)
             a[i] = "y[" + i + "]";
          r = eval("x.__apply(" + a.join(",") + ")");
```

```
            break;
        }
        try {
            delete x.__apply ? x.__apply : x.constructor.prototype.__apply;
        }
        catch (e) {}
        return r;
    });

    Function.prototype.call || (Function.prototype.call = function () {
        var a = arguments, x = a[0], y = [];
        for (var i = 1, j = a.length; i < j; ++i)
            y[i - 1] = a[i]
        return this.apply(x, y);
    });
```

3.5 日期的扩展与修复

Date 构造器是 JavaScript 中传参形式最丰富的构造器，大致分为四种。

```
new Date();
new Date(value);//传入毫秒数
new Date(dateString);
new Date(year, month, day /*, hour, minute, second, millisecond*/);
```

其中第三种可以玩 N 多花样，个人建议只使用“2009/07/12 12:34:56”，后面的时分秒可省略。这个所有浏览器都支持。此构造器的兼容列表可见此文：

http://dygraphs.com/date-formats.html

若要修正它的传参，这恐怕是个大工程，要整个对象替换掉，并且影响 Object.prototype.toString 的类型判定。因此不建议修正。es5.js 中有相关源码，大家可以看这里：

https://github.com/kriskowal/es5-shim/blob/master/es5-shim.js

JavaScript 的日期是抄自 Java 的 java.util.Date，但是 Date 这个类中的很多方法对时区等支持不够，且不少都是已过时，Java 程序员也推荐使用 calnedar 类代替 Date 类。JavaScript 可选择的余地比较少，只能捏着鼻子继续用。如对属性使用了前后矛盾的偏移量。月份与小时都是基于 0，月份中的天数则是基于 1，而年则是从 1900 开始的。

接着下来，我们为旧版本浏览器添加几个 ecma262 标准化的日期方法吧。

```
if (!Date.now) {
    Date.now = function() {
        return +new Date;
    }
}
if (!Date.prototype.toISOString) {
    void function() {
        function pad(number) {
            var r = String(number);
            if (r.length === 1) {
                r = '0' + r;
            }
```

```
            return r;
        }

        Date.prototype.toJSON = Date.prototype.toISOString = function() {
            return this.getUTCFullYear()
                    + '-' + pad(this.getUTCMonth() + 1)
                    + '-' + pad(this.getUTCDate())
                    + 'T' + pad(this.getUTCHours())
                    + ':' + pad(this.getUTCMinutes())
                    + ':' + pad(this.getUTCSeconds())
                    + '.' + String((this.getUTCMilliseconds() / 1000).toFixed(3)).slice(2, 5)
                    + 'Z';
        };

    }();
}
```

IE6、IE7 中，getYear、setYear 方法存在 BUG，这个修起来比较简单。

```
if ((new Date).getYear() > 1900) {
    Date.prototype.getYear = function() {
        return this.getFullYear() - 1900;
    };
    Date.prototype.setYear = function(year) {
        return this.setFullYear(year); //+ 1900
    };
}
```

至于扩展，由于涉及本地化的原因，外国许多日期库都需要改一改才能用，其中以 dataFormat 这个很有用的方法为最。我先给一些常用的扩展吧。

传入两个 Date 类型的日期，求出它们相隔多少天。

```
var getDatePeriod = function(start, finish) {
    return Math.abs(start * 1 - finish * 1) / 60 / 60 / 1000 / 24;
}
```

传入一个 Date 类型的日期，求出它所在月的第一天。

```
var getFirstDateInMonth = function(date) {
    return new Date(date.getFullYear(), date.getMonth(), 1);
}
```

传入一个 Date 类型的日期，求出它所在月的最后一天。

```
var getLastDateInMonth = function(date) {
    return new Date(date.getFullYear(), date.getMonth() + 1, 0);
}
```

传入一个 Date 类型的日期，求出它所在季度的第一天。

```
var getFirstDateInQuarter = function(date) {
    return new Date(date.getFullYear(), ~~(date.getMonth() / 3) * 3, 1);
}
```

传入一个 Date 类型的日期，求出它所在季度的最后一天。

```
var getFirstDateInQuarter = function(date) {
    return new Date(date.getFullYear(), ~~(date.getMonth() / 3) * 3 + 3, 0);
}
```

判断是否为闰年。

```
Date.prototype.isLeapYear = function() {
    return new Date(this.getFullYear(), 2, 0).getDate() == 29;
}
```

取得当前月份的天数。

```
function getDaysInMonth1(date) {
    switch (date.getMonth()) {
        case 0:
        case 2:
        case 4:
        case 6:
        case 7:
        case 9:
        case 11:
            return 31;
        case 1:
            var y = date.getFullYear();
            return y % 4 == 0 && y % 100 != 0 || y % 400 == 0 ? 29 : 28;
        default:
            return 30;
    }
}

function getDaysInMonth2(date) {
    switch (date.getMonth()) {
        case 0:
        case 2:
        case 4:
        case 6:
        case 7:
        case 9:
        case 11:
            return 31;
        case 1:
            var y = date.getFullYear();
            return y % 4 == 0 && y % 100 != 0 || y % 400 == 0 ? 29 : 28;
        default:
            return 30;
    }
}

function getDaysInMonth3(date) {
    return new Date(date.getFullYear(), date.getMonth() + 1, 0).getDate();
}
```

最后是 dateForamt 方法，由于太长了，我也不便贴出来，可到以下链接查看：
https://github.com/RubyLouvre/mass-Framework/blob/1.41/avalon.js

第4章 浏览器嗅探与特征侦测

虽然浏览器嗅探现在已经不推荐了，但在某些场合我们还是需要的，比如一些统计脚本。而特征侦测是 Prototype 时期发展起来的一个技术，具体是判定某个原生对象有没有此方法或属性，有时严格一些，则会执行这个方法，看它有没有达到预期的值。在标准浏览器里，它们提供了 document.implementation.hasfeature 方法，可惜有 Bug，不准确。在本书快成形时，W3C 又推出了 CSS.supports 方法，显然大家都对这块非常重视。

4.1 判定浏览器

主流浏览器有五个：IE、Firefox、Opera、Chrome、Safari。早期所有框架都是通过 navigator.userAgent 进行判定，与浏览器商斗智斗谋。显然外国的浏览器商还是有良心的，换是国内的，基本判定不出。

jQuery 给出的解决方案，现在已移出 jQuery 本体，成为一个插件。

```
(function(jQuery, window, undefined) {
   "use strict";
   var matched, browser;

   jQuery.uaMatch = function(ua) {
      ua = ua.toLowerCase();

      var match = /(chrome)[ \/]([\w.]+)/.exec(ua) ||
            /(webkit)[ \/]([\w.]+)/.exec(ua) ||
            /(opera)(?:.*version|)[ \/]([\w.]+)/.exec(ua) ||
            /(msie) ([\w.]+)/.exec(ua) ||
            ua.indexOf("compatible") < 0 && /(mozilla)(?:.*? rv:([\w.]+)|)/.exec(ua) ||
            [];

      var platform_match = /(ipad)/.exec(ua) ||
            /(iphone)/.exec(ua) ||
            /(android)/.exec(ua) ||
            [];

      return {
         browser: match[ 1 ] || "",
         version: match[ 2 ] || "0",
         platform: platform_match[0] || ""
      };
```

```
    };

    matched = jQuery.uaMatch(window.navigator.userAgent);
    browser = {};

    if (matched.browser) {
        browser[ matched.browser ] = true;
        browser.version = matched.version;
    }

    if (matched.platform) {
        browser[ matched.platform ] = true
    }

// Chrome is Webkit, but Webkit is also Safari.
    if (browser.chrome) {
        browser.webkit = true;
    } else if (browser.webkit) {
        browser.safari = true;
    }

    jQuery.browser = browser;

})(jQuery, window);
```

mass Framework 给出的解决方案如下。

```
//https://github.com/RubyLouvre/mass-Framework/blob/1.4/more/brower.js
define("brower", function( ){
    var w = window,ver = w.opera ? (opera.version().replace(/\d$/, "") - 0)
    : parseFloat((/(?:IE |fox\/|ome\/|ion\/)(\d+\.\d)/.
        exec(navigator.userAgent) || [,0])[1]);
    return {
        //测试是否为 IE 或内核为 trident，是则取得其版本号
        ie: !!w.VBArray && Math.max(document.documentMode||0, ver),//内核 trident
        //测试是否为 Firefox，是则取得其版本号
        firefox: !!w.netscape && ver,//内核 Gecko
        //测试是否为 Opera，是则取得其版本号
        opera:  !!w.opera && ver,//内核 Presto 9.5 为 Kestrel 10 为 Carakan
        //测试是否为 Chrome，是则取得其版本号
        chrome: !! w.chrome &&  ver ,//内核 V8
        //测试是否为 Safari，是则取得其版本号
        safari: /apple/i.test(navigator.vendor) && ver// 内核 WebCore
    }
});
```

mass 成形较晚，是使用特征侦测实现的。

特征侦测的好处是浏览器不会随意去掉某一个功能，不过要注意不要使用标准属性与方法做判定依据。每个浏览器都有自己的私有实现，我们用它们做判定好了。下面是我收集的其他一些判定方法。

```
ie = !!document.recalc
ie = !!window.VBArray
ie = !!window.ActiveXObject
```

```
ie = !!window.createPopup;
ie = /*@cc_on!@*/!1;
ie = document.expando;//document.all 在 opera  firefox 的古老版本也存在
ie = (function() {//IE10 中失效
    var v = 3, div = document.createElement('div');
    while (div.innerHTML = '<!--[if gt IE ' + (++v) + ']><br><![endif]-->', div.innerHTML )
        ;
    return v > 4 ? v : !v;
}());

ie678 = !+"\v1";
ie678 = !-[1, ];
ie678 = '\v' == 'v';
ie678 = ('a~b'.split(/(~)/))[1] == "b"
ie678 = 0.9.toFixed(0) == "0"
ie678 = /\w/.test('\u0130') //由群里的 abcd 友情提供
ie8 = window.toStaticHTML
ie9 = window.msPerformance

ie678 = 0//@cc_on+1;

ie67 = !"1"[0] //利用 IE6 或 IE5 的字符串不能使用数组下标的特征
ie8 = !!window.XDomainRequest;
ie9 = document.documentMode && document.documentMode === 9;

//基于条件编译的嗅探脚本，IE 会返回其 JS 引擎的版本号，非 IE 返回 0
var ieVersion = eval("''+/*@cc_on" + " @_jscript_version@*/-0") * 1
ie9 = ieVersion === 5.9
ie8 = ieVersion === 5.8
ie7 = ieVersion === 5.7
ie6 = ieVersion === 5.6
ie5 = ieVersion === 5.5
ie10 = window.navigator.msPointerEnabled
ie11 = '-ms-scroll-limit' in document.documentElement.style

opera = !!window.opera;

//https://developer.mozilla.org/En/Windows_Media_in_Netscape
firefox = !!window.GeckoActiveXObject
firefox = !!window.netscape //包括 firefox
firefox = !!window.Components
firefox = !!window.updateCommands
safari = !!(navigator.vendor && navigator.vendor.match(/Apple/))
safari = window.openDatabase && !window.chrome;
chrome = !!(window.chrome && window.google)
```

移动设备的相关判定。这个建议看 jquery mobile 与 zepto 的源代码。

```
isIPhone = /iPhone/i.test(navigator.userAgent);
isIPhone4 = window.devicePixelRatio >= 2//在网页中,pixel 与 point 比值称为 device-pixel-ratio,
普通设备都是 1，iPhone 4 是 2，有些 Android 机型是 1.5
//http://blog.webcreativepark.net/2011/01/25-173502.html
isIPad = /iPad/i.test(navigator.userAgent);
isAndroid = /android/i.test(navigator.userAgent);
isIOS = isIPhone  || isIPad ;
```

国内的浏览器判定可以看 Tangram 或 qwrap。它们基本是 IE，webkit 内核或 blink 内核。

4.2 事件的支持侦测

Prototype 的核心成员 kangax 写了一篇叫《Detecting event support without browser sniffing》文章[①]，来判定浏览器对某种事件的支持。里面给出的实现如下。

```
var isEventSupported = (function() {
    var TAGNAMES = {
        'select': 'input', 'change': 'input',
        'submit': 'form', 'reset': 'form',
        'error': 'img', 'load': 'img', 'abort': 'img'
    }
    function isEventSupported(eventName) {
        var el = document.createElement(TAGNAMES[eventName] || 'div');
        eventName = 'on' + eventName;
        var isSupported = (eventName in el);
        if (!isSupported) {
            el.setAttribute(eventName, 'return;');
            isSupported = typeof el[eventName] == 'function';
        }
        el = null;
        return isSupported;
    }
    return isEventSupported;
})();
```

现在 jQuery 与 mass 使用的脚本都是其简化版，其中 mass 对 IE 内存泄漏做了优化。

```
$.eventSupport = function(eventName, el) {
    el = el || document.documentElement
    eventName = "on" + eventName;
    var ret = eventName in el;
    if (el.setAttribute && !ret) {
        el.setAttribute(eventName, "");
        ret = typeof el[eventName] === "function";
        el.removeAttribute(eventName);
    }
    el = null;
    return ret;
};
```

不过哪一个也好，这种检测只对 DOM0 事件凑效，像 DOMMouseScroll、DOMContentLoaded、DOMFocusIn、DOMFocusOut、DOMSubtreeModified、DOMNodeInserted、DOMNodeRemoved、DOMNodeRemovedFromDocument、DOMNodeInsertedIntoDocument、DOMAttrModified、DOMCharacterDataModified 这些以 DOM 开头的事件就无难为力了。

这些事件中有的非常有用，如：DOMMouseScroll，Firefox 一直不支持 mousewheel，只能用它做替代品；DOMContentLoaded 是实现 domReady 的重要事件；DOMNodeRemoved 是判定元素是

① http://perfectionkills.com/detecting-event-support-without-browser-sniffing/

否从其父节点移除，父节点可能是其他元素节点或文档碎片；DOMNodeRemovedFromDocument 是移离 DOM 树；DOMAttrModified 以前经常用于模拟 IE 的 onpropertychange；DOMCharacterData-Modified 用于监听 contenteditable 为 true 的元素内容变动。

在 mass Framework 中，是使用以下方法判定的。

```
//https://github.com/RubyLouvre/mass-Framework/blob/1.4/event.js
try {
    //如果浏览器支持创建 MouseScrollEvents 事件对象，那么就用 DOMMouseScroll
    document.createEvent("MouseScrollEvents");
    eventHooks.mousewheel = {
        bindType: "DOMMouseScroll",
        delegateType: "DOMMouseScroll"
    };
    //如果某一天,Firefox 回心转意支持 mousewheel，那么我们就不需要这个钩子
    if ($.eventSupport("mousewheel")) {
        delete eventHooks.mousewheel;
    }
} catch (e) {
}
```

此外，mass 还对 focusin 进行识别。focusin 与 focusout 是一对，判定当中一个就明白另一个情况。这两个事件也很重要，用于实现 focus 与 blur 的事件代理，因为 focus 与 blur 不支持冒泡，需要用它们的冒泡版实现（假若不支持 focusin 与 focusout，jQuery 也找到办法了，不过有原生的就用原生的）。

```
//https://github.com/RubyLouvre/mass-Framework/blob/1.4/support.js#L108
//首先判定它是否是 W3C 阵营，IE 肯定支持
$.support.focusin = !!window.attachEvent;
$(function() {
    var div = document.createElement("div");
    document.body.appendChild(div);
    div.innerHTML = "<a href='#'></a>";
    if (!support.focusin) {
        a = div.firstChild;
        a.addEventListener('focusin', function() {
            $.support.focusin = true;
        }, false);
        a.focus();
    }
});
```

CSS3 添加两种动画，一种是 transition 动画，第一种是 keyframe 补间动画，它们在结束时都有相应的事件回调。但在标准化过程中，浏览器给它们起的名字相当没规则。这个也需预先侦测出来。

下面是 bootstrap 的实现，听说来源于 modernizr，比较粗糙。比如说你现在用的 Opera 已经支持不带前缀的标准事件名，它还是返回 oTransitionEnd。

```
$.support.transition = (function() {
    var transitionEnd = (function() {
        var el = document.createElement('bootstrap'),
               transEndEventNames = {
            'WebkitTransition': 'webkitTransitionEnd',
            'MozTransition': 'transitionend',
```

```
            'OTransition': 'oTransitionEnd otransitionend',
            'transition': 'transitionend'
        };
        for (var name in transEndEventNames) {
            if (el.style[name] !== undefined) {
                return transEndEventNames[name]
            }
        }
    }())
    return transitionEnd && {
        end: transitionEnd
    }
})()
```

keyframe 补间动画的检测来自 mass 的 fx_neo 模块，以后会说到的。

```
var eventName = {
    AnimationEvent: 'animationend',
    WebKitAnimationEvent: 'webkitAnimationEnd'
}, animationend;
for (var name in eventName) {
    if (/object|function/.test(typeof window[name])) {
        animationend = eventName[name]
        break
    }
}
```

4.3 样式的支持侦测

CSS3 带来许多好用的样式，但麻烦的是每个浏览器都有自己的私有前缀。在 mass Framework 提供了一个 cssName 方法处理它们。有就返回可用的驼峰风格样式名，没有就 null。

```
var prefixes = ['', '-webkit-', '-o-', '-moz-', '-ms-'];
var cssMap = {
    "float": $.support.cssFloat ? 'cssFloat' : 'styleFloat',
    background: "backgroundColor"
};
function cssName(name, host, camelCase) {
    if (cssMap[name]) {
        return cssMap[name];
    }
    host = host || document.documentElement
    for (var i = 0, n = prefixes.length; i < n; i++) {
        camelCase = $.String.camelize(prefixes[i] + name);
        if (camelCase in host) {
            return (cssMap[name] = camelCase);
        }
    }
    return null;
}
```

一个样式名可以对应 N 种样式值，如 display 的值就有 N 多种取值。如果要探知浏览器是否支持某一种，会吐血的。为此，浏览器做了一个善举，最近给出一个叫 CSS.supports 的 API。如果不

支持，则尝试一下这个开源项目，显然它还有很多探不出来的。

https://github.com/termi/CSS.supports

4.4 jQuery 一些常用特征的含义

jQuery 在 support 模块列举了一些常用的 DOM 特征的支持情况，不过它们的名字起得很奇怪，这里逐一揭开它们的谜底。由于这些特征在 jQuery 不同版本变动很大，本书以 jQuery1.8 为准。

leadingWhitespace：判定浏览器在进行 innerHTML 赋值时，是否存在 trimLeft 操作。这个原本是 IE 出来搞的，应该遵循 IE 的游戏规则才对。结果其他浏览器一直认为要忠于用户的原始值，最前的空白不能省掉，要变成一个文本节点。最后微软不得不服软，IE6、IE7、IE8 返回 false，其他浏览器返回 true。

tbody：指在用 innerHTML 动态创建元素时，浏览器是否会在 table 内自动补上 tbody。jQuery 是希望浏览器不要自作多情，要补 jQuery 会补！判定浏览器是否智能插入 tbody。在表格布局的年代，这个特征非常受用。因为如果没有 tbody，table 会在浏览器解析到闭合标签时才显示出来，如果起始标签与闭合标签相隔很远，换言之，这个表格很大很长，用户会看什么也看不到。但有了 tbody 分段识别和显示，避免了页面长时间一片空白后突然一下子内容全部出来的局面。看下面实验。

```
var div = document.createElement("div");
div.innerHTML = "<table></table>"
alert(div.innerHTML)
//IE678 返回 "<TABLE><TBODY></TBODY></TABLE>" ,其他返回"<table></table>"
```

htmlSerialize：判定浏览器是否完好支持用 innerHTML 转换一个符合 html 标签规则的字符串为一个元素节点，此过程 jQuery 称为序列化。但 IE 支持不完好，包括 script、link、style、meta 在内的 no-scope 元素[①]都转换失败，需要在它前面添加一些字符，如"x<script src="xxx"><\/script>"，"<script src="xxx"><\/script>" 或者 "<br class='remove'><script src="xxx"><\/script>" 。像 HTML5 的新标签，当然也支持不好。侦测结果同上。

style：这个命名很难懂，不看源码不知道是什么意思。真相是判定 getAttribute 能否返回 style 的用户预设值。IE6、IE7、IE8 没有区分特性属性，返回一个 CSSStyleDeclaration 对象。

hrefNormalized：同样莫名其妙，意为判定 getAttribute 能否返回 href 的用户预设值。IE 会多此一举，补充为完整路径给你。

opacity：判定是否支持 opacity 样式值。IE6、IE7、IE8 不支持，要用透明滤镜。

cssFloat：判定 float 样式值在 DOM 的名字是哪个，W3C 为 cssFloat，IE6、IE7、IE8 为 styleFloat。

checkOn：在大多数浏览器中 checkbox 的 value 默认为 on，唯有 Chrome 返回空字符串。

optSelected：判定能否正确地取得动态添加的 option 元素的 selected。IE6～IE10 与老版本的 Safari 对动态添加的 option 没有设置为 true。解决办法为，在访问 selected 属性前，先访问其父节的 selectedIndex 属性，强制它计算 option 的 selected。

① http://qbaok.blog.163.com/blog/static/101292652013022296590/

```
<select id="optSelected">
</select>
<script type="text/javascript">
    var select = document.getElementById('optSelected');
    var option = document.createElement('option');
    select.appendChild(option);
    alert(option.selected);
    select.selectedIndex;
    alert(option.selected);
</script>
```

optDisabled：判定 select 元素的 disabled 属性是否影响到子元素的 disabled 取值。在 Safari 中，一旦 select 元素被 disabled，它的孩子也被 disabled，导致一个值也取不到。

checkClone：是指，如果一个 checkbox 元素，如果设置了 checked=true，且在多次克隆后，它的复制品能否保持为 true。这个只有在 Safari4 中返回 false，其他为 true。

inlineBlockNeedsLayout：判定是否使用 hasLayout 方法让 display:inline-block 生效。这个只有 IE6、IE7、IE8 为 true。

getSetAttribute：判定是否区分特性属性。只有 IE6、IE7、IE8 为 false。

noCloneEvent：判定在克隆元素时是否克隆 attachEvent 绑定的事件，只有旧版本 IE 及其兼容模式返回 false。

enctype：判定浏览器是否支持 encoding 属性，IE6、IE7 要用 encoding 属性代替。

boxModel：判定浏览器是否在 content-box 盒子模型的渲染模式下。

submitBubbles，changeBubbles，focusinBubbles：判定浏览器是否支持这些事件，一直冒泡到 document。

shrinkWrapBlocks：判定元素是否会被子元素撑开。在 IE6、IE7、IE8 中，非替换元素[①]在设置了大小与 hasLayout 的情况下，会将其父级元素撑大。

html5Clone：判定能否使用 cloneNode 克隆 HTML5 的新标签，旧版本 IE 不支持，需要用到 outerHTML。

deleteExpando：判定能否删除元素节点上的自定义属性，这用于 jQuery 缓存系统。旧版本 IE 不支持，直接置为 undefined 的。

pixelPosition：判定 getComputedStyle 能否转换元素的 top、left、right、bottom 的百分比值。这个 webkit 系出现问题，需要用到 Dean Edwards 大神的 hack。

reliableMarginRight：判定 getComputedStyle 能否正确取得元素的 marginRight。Safari 的早期版本总是取回一个很大的数。

clearCloneStyle：IE9、IE10 出现的奇葩 BUG，当复制一个指定了 background-*样式的元素，对复制品的背景进行清空时，也会清空原来的。

目前就是这么多了，随着浏览器疯狂更新版本，标准浏览器引发的各种 BUG 已超越 IE 了。特征侦测不退反进，越来越重要了。

① http://hi.baidu.com/flowerszhong/item/4b773f0b3522107ebfe97e60

第5章 类工厂

类与继承在 JavaScript 的出现，说明 JavaScript 已经到达大规模开发的门槛了。在此之前的 es4，就试图引入类、模块等东西，但由于过分引入太多特性，搞得 JavaScript 乌烟瘴气，导致被否决，也只不过把类延迟到 es6。到目前为此，JavaScript 还没有真正意义的类，不过我们可以模拟类。曾经一段时间，类工厂是框架的标配。本章将会介绍各种类实现，方便大家在自己框架中选择或现实自己喜欢的那一类风格。

5.1 JavaScript 对类的支撑

在其他语言中，类的实例都要通过构造函数 new 出来。作为一个刻意模仿 Java 的语言，JavaScript 存在 new 操作符，并且它的所有函数都可以作为构造器。构造器与普通的方法没有什么区别。浏览器为了构建它繁花似锦的生态圈，比如 Node，Element、HTMLElement、HTMLParagraphElement，显然使用继承关系方便一些方法或属性的共享，于是 JavaScript 从其他语言借鉴了原型这种机制。prototype 作为一个特殊的对象属性存在于每一个函数上。当一个函数通过 new 操作符“分娩”出其孩子——“实例”，这个名为实例的对象就拥有这个函数的 prototype 对象所有的一切成员，从而实现所有实例对象都共享一组方法或属性。而 JavaScript 所谓的“类”就是通过修改这个 prototype 对象，以区别原生对象及其他自定义“类”。在浏览器中，Node 这个类就是基于 Object 修改而来的，而 Element 则是基于 Node，而 HTMLElement 又基于 Element……相对于我们的工作业务，我们也可以创建自己的类来实现重用与共享。

```
function A() {
}

A.prototype = {
    aa: "aa",
    method: function() {
    }
};
var a = new A;
var b = new A;
console.log(a.aa === b.aa);//true
console.log(a.method === b.method);//true
```

一般地，我们把定义在原型上的方法叫原型方法，它为所有实例所共享。这有好有不好，为了

实现差异化，JavaScript 允许我们直接在构造器内指定其方法，这叫做特权方法。如果是属性，就叫特权属性。它们每一个实例一个副本，各不影响。因此我们通常把共享的用于操作数据的方法放在原型，把私有的数据放在特权属性中。但放于 this 上，还是能让人任意访问到，那就放在函数体内的作用域内吧。这时它就成为名符其实的私有属性。

```
function A() {
    var count = 0
    this.aa = "aa";
    this.method = function() {
      return count
    }
    this.obj = {}
}

A.prototype = {
    aa: "aa",
    method: function() {
    }
};
var a = new A;
var b = new A;
console.log(a.aa === b.aa);//true 由于 aa 的值为基本类型,比较值
console.log(a.obj === b.obj);//false,引用类型,每次进入函数体都重新创建,因此都不一样
console.log(a.method === b.method);//false
```

特权方法或属性只是遮住原型方法或属性，因此只要删掉特权方法，就又能访问到同名的原型方法或属性。

```
delete a.method;
console.log(a.method === A.prototype.method);//true
```

用 Java 的语言来说，原型方法与特权方法都属于实例方法，在 Java 中还有一种叫做类方法与类属性的东西。它们用 JavaScript 来模拟也非常简单，直接定义在函数上就行了。

```
A.method2 = function(){};//类方法
var c = new A;
console.log(c.method2);//undefined
```

接下来，我们看一下继承的实现。上面说过，只要 prototype 有什么东西，它的实例就有什么东西，不论这个属性是后来添加的，还是整个 prototype 都是置换上去的。如果我们将这个 prototype 对象置换为另一个类的原型，那么它就轻而易举得到那个类的所有原型成员。

```
function A(){}
A.prototype = {
    aaa:1
}
function B(){}
B.prototype = A.prototype;
var b= new B;
console.log(b.aaa);//1;
A.prototype.bbb = 2;
console.log(b.bbb);//2;
```

由于是引用着相同的一个对象，这意味着，如果我们修改 A 类的原型，也等同于修改了 B 类的原型。因此我们不能把一个对象赋给两个类。这有两种办法。方法一是，通过 for in 把父类的原型成员逐一赋给子类的原型，方法二是，子类的原型不是直接由父类获得，先将此父类的原型赋给一个函数，然后将这个函数的实例作为子类的原型。

方法一，我们通常要实现 mixin 这样的方法，亦有书称之为拷贝继承，好处是简单直接，坏处是无法通过 instanceof 验证。Prototype.js 的 extend 方法就用来干这事。

```
function extend(destination, source) {
  for(var property in source)
  destination[property] = source[property];
  return destination;
}
```

方法二，就在原型上动脑筋，因此称之为原型继承。下面是个范本。

```
A.prototype = {
  aa: function() {
    alert(1)
  }
}

function bridge() {};
bridge.prototype = A.prototype;

function B() {}
B.prototype = new bridge();
var a = new A;
var b = new B;
//false，说明成功分开它们的原型
console.log(A.prototype == B.prototype);
//true，子类共享父类的原型方法
console.log(a.aa === b.aa);
//为父类动态添加新的原型方法
A.prototype.bb = function() {
  alert(2)
}
//true，孩子总会得到父亲的遗产
console.log(a.bb === b.bb);
B.prototype.cc = function() {
  alert(3)
}
//false，但父亲未必有机会看到孩子的新产业
console.log(a.cc === b.cc);
//并且它能正常通过 JavaScript 自带验证机制——instanceof
console.log(b instanceof A);//true
console.log(b instanceof B);//true
```

并且，方法二能通过 instanceof 验证。现在 es5 就内置了这种方法来实现原型继承，它就是 Object.create，如果不考虑第二个参数，它约等于下面的代码。

```
Object.create = function (o) {
    function F() {}
    F.prototype = o;
    return new F();
}
```

上面方法，要求传入一个父类的原型作为参数，然后返回子类的原型。

不过，这样我们还是遗漏了一点东西——子类不只是继承父类的遗产，还拥有自己的东西。此外，原型继承并没有让子类继承父类的类成员与特权成员。这些我们还得手动添加，如类成员，我们可以通过上面的 extend 方法，特权成员我们可以在子类的构造器中，通过 apply 实现。

```
function inherit(init, Parent, proto){
    function Son(){
        Parent.apply(this,argument);  //先继承父类的特权成员
        init.apply(this,argument);    //再执行自己的构造器
    }
    //由于 Object.create 可能是我们伪造的，因此避免使用第二个参数
    Son.prototype = Object.create(Parent.prototype,{});
    Son.prototype.toString = Parent.prototype.toString;     //处理 IE BUG
    Son.prototype.valueOf = Parent.prototype.valueOf;       //处理 IE BUG
    Son.prototype.constructor = Son; //确保构造器正常指向自身，而不是 Object
    extend(Son.prototype, proto);    //添加子类特有的原型成员
    extend(Son, Parent);             //继承父类的类成员
    return Son;
}
```

下面我们做一组实验，测试一下实例的回溯机制。许多资料都说——但总是语焉不详——当我们访问对象的一个属性，那么它先找其特权成员，如果有同名的就返回，没有就找原型，再没有，找父类的原型……我们尝试把它的原型临时修改一下，看它的属性会变成哪一个！

```
function A() {}
A.prototype = {
  aa: 1
}
var a = new A;
console.log( a.aa);//1
//把它整个原型对象都换掉
A.prototype = {
  aa: 2
}
console.log(a.aa);//1，表示不受影响
//于是我们想到实例都有一个 constructor 方法，指向其构造器，
//而构造器上面正好有我们的原型，JavaScript 引擎是不是通过该路线回溯属性呢
function B(){}
B.prototype = {
  aa: 3
}
a.constructor = B;
console.log( a.aa );//1 表示不受影响
```

因此类的实例肯定通过另一条通道进行回溯，翻看 ecma 规范可知每一个对象都有一个内部属性 [[Prototype]]，它保存着当我们 new 它时构造器所引用的 prototype 对象。在标准浏览器与 IE11 里，它们暴露了一个叫__proto__属性来访问它。因此只要不动__proto__，上面的代码怎么动，a.aa 始终坚定不移地返回 1。

我们再来看一下 new 操作时发生了什么事。

（1）创建一个空对象 instance。

（2）instance.__proto__ = instanceClass.prototype。

（3）将构造器函数里面的 this = instance。

（4）执行构造器里面的代码。

（5）判定有没有返回值，没有返回值默认为 undefined，如果返回值为复合数据类型，则直接返回，否则返回 this。

于是有了下面结果。

```
function A() {
  console.log(this.__proto__.aa); //1
  this.aa = 2
}
A.prototype = {
  aa: 1
}
var a = new A;
console.log(a.aa); //2
a.__proto__ = {
  aa: 3
}
delete a.aa; //删掉特权属性，暴露原型链上的同名属性
console.log(a.aa); //3
```

有了__proto__，我们可以将原型继承设计得更简洁。我们还是拿上面的例子改一下来进行实验。

```
function A() {}
A.prototype = {
  aa: 1
}

function bridge() {};
bridge.prototype = A.prototype;

function B() {}
B.prototype = new bridge();
B.prototype.constructor = B;
var b = new B;
B.prototype.cc = function() {
  alert(3)
}
console.log(b.__proto__ == B.prototype);
//true 这个大家应该都没有疑问
console.log(b.__proto__.__proto__ === A.prototype);
//true  得到父类的原型对象
```

为什么呢？因为 b.__proto__.constructor 为 B，而 B 的原型是从 bridge 中得来的，而 bridge.prototype = A.prototype。反过来，我们在定义时，让 B.prototype.__proto__ = A.prototype，就能轻松实现两个类的继承。

目前，__proto__属性已列入 es6，因此可以通过 if 分支大胆使用它。

5.2 各种类工厂的实现

在第 1 节我们演示各种继承方式的实现，但都很凌乱。我们希望提供一个专门的方法，只要用户传入相应的参数或按一定的简单格式就能创建一个类，特别是子类。

由于主流框架的类工厂实现太依赖于它们庞杂的工具函数，而一个精巧的类工厂也不过百行左右，因此本章就不打算罗列 Prototype.js、Mootools 等的代码了，介绍另外一些不太出名但相当有水准的小库吧。

5.2.1 相当精巧的库——P.js

它的 github 地址为 https://github.com/jayferd/pjs。

这是一个相当于精巧的库，尤其在调用父类的同名方法时，它直接把父类的原型抛在你眼前，连_super 也省了。

它的源码解读如下。

```
var P = (function(prototype, ownProperty, undefined) {

    function isObject(o) {
        return typeof o === 'object';
    }

    function isFunction(f) {
        return typeof f === 'function';
    }

    function BareConstructor() {};

    function P(_superclass /* = Object */ , definition) {
        //如果只传一个参数，没有指定父类
        if(definition === undefined) {
            definition = _superclass;
            _superclass = Object;
        }

        //C 为我们要返回的子类，definition 中的 init 为用户构造器

        function C() {
            var self = new Bare;
            console.log(self.init)
            if(isFunction(self.init)) self.init.apply(self, arguments);
            return self;
        }
```

```
        function Bare() { //这个构造器是为了让C不用new就能返回实例而设的
        }
        C.Bare = Bare;
        //为了防止改动子类影响到父类，我们将父类的原型赋给一个中介者BareConstructor
        //然后再将这中介者的实例作为子类的原型
        var _super = BareConstructor[prototype] = _superclass[prototype];
        var proto = Bare[prototype] = C[prototype] = new BareConstructor; //
        //然后C与Bare都共享同一个原型
        //最后修正子类的构造器指向自身
        proto.constructor = C;
        //类方法mixin，不过def对象里面的属性与方法糅杂到原型里面去
        C.mixin = function(def) {
            Bare[prototype] = C[prototype] = P(C, def)[prototype]; //Bare[prototype] =
            return C;
        }
        //definition最后延迟到这里才起作用
        return(C.open = function(def) {
            var extensions = {};
            //definition有两种形态
            //如果是函数，那么子类原型、父类原型、子类构造器、父类构造传进去，
            //如果是对象则直接置为extensions
            if(isFunction(def)) {
                extensions = def.call(C, proto, _super, C, _superclass);
            } else if(isObject(def)) {
                extensions = def;
            }
            //最后混入子类的原型中
            if(isObject(extensions)) {
                for(var ext in extensions) {
                    if(ownProperty.call(extensions, ext)) {
                        proto[ext] = extensions[ext];
                    }
                }
            }
            //确保init为一个函数
            if(!isFunction(proto.init)) {
                proto.init = _superclass;
            }

            return C;
        })(definition);

        //这里为一个自动执行函数表达式，相当于
        //C.open = function(){/*....*/}
        //C.open(definition)
        //return C;
        //换言之，返回的子类存在3个类成员，Base, mixin, open

    }

    return P; //暴露到全局
})('prototype', ({}).hasOwnProperty);
```

我们尝试创建一个类。

```
var Animal = P(function(proto, superProro) {
  proto.init = function(name) { //构造函数
    this.name = name;
  };
  proto.move = function(meters) { //原型方法
    console.log(this.name + " moved " + meters + "m.");
  }
});
var a = new Animal("aaa")
var b = Animal("bbb");//无 "new" 实例化

a.move(1)
b.move(2)
```

当然在现在的情景下，我们可以使用更简洁的定义方式。

```
var Snake = P(Animal, function(snake, animal) {
  snake.init = function(name, eyes) {
    animal.init.call(this, arguments); //调用父类构造器
    this.eyes = 2;
  }
  snake.move = function() {
    console.log("Slithering...");
    animal.move.call(this, 5); //调用父类的同名方法
  };
});
var s = new Snake("snake", 1);
s.move();
console.log(s.name);
console.log(s.eyes);
```

下面是私有属性的演示，由于放在函数体内集中定义，因此安全可靠！

```
var Cobra = P(Snake, function(cobra) {
  var age = 1; //私有属性
  //这里还可以编写私有方法
  cobra.glow = function() { //长大
    return age++;
  }
});
var c = new Cobra("cobra");
console.log(c.glow()); //1
console.log(c.glow()); //2 又长一岁
console.log(c.glow()); //3 又长一岁
```

此外，它还提供了两个类方法，mixin 用于再次添加新的原型成员，open 的作用同 mixin，但显然它适合于重写父类的方法（在子类方法内部重用父类方法），同时，也可以添加新的私有属性。open 这个命名显然是受 ruby 影响，意为重新打开类，修改其原型。

5.2.2　JS.Class

项目地址:http://code.google.com/p/jsclassextend/

从它的设计来看,它是师承 Dean Edwards 的 Base2，相似的类工厂实现还有 mootools，第 5.2.3 小节的 **simple-inheritance**。它是通过父类构造器的 extend 方法来产生自己的子类，里面存在一个开关，防止在生成类时无意执行 construct 方法。

如 Base2 的 base2.__prototyping，mootools 的 klass.$prototyping。它创建子类时，也不通过中间的函数来断开双方的原型链，而是使用父类的实例来做子类的原型，这点实现得非常精巧。

源码解读如下。

```
var JS = {
    VERSION: '2.2.1'
};

JS.Class = function(classDefinition) {

    //返回目标类的真正构造器
    function getClassBase() {
        return function() {
            //它在里面执行用户传入的构造器 construct
            //preventJSBaseConstructorCall 是为了防止在 createClassDefinition 辅助方法中执行父
            //类的 construct
            if (typeof this['construct'] === 'function' && preventJSBaseConstructorCall ===
false) {
                this.construct.apply(this, arguments);
            }
        };
    }
    //为目标类添加类成员与原型成员
    function createClassDefinition(classDefinition) {
        //此对象用于保存父类的同名方法
        var parent = this.prototype["parent"] || (this.prototype["parent"] = {});
        for (var prop in classDefinition) {
            if (prop === 'statics') {
                for (var sprop in classDefinition.statics) {
                    this[sprop] = classDefinition.statics[sprop];
                }
            } else {
    //为目标类添加原型成员，如果是函数，那么检测它还没有同名的超类方法，如果有
                if (typeof this.prototype[prop] === 'function') {
                    var parentMethod = this.prototype[prop];
                    parent[prop] = parentMethod;
                }
                this.prototype[prop] = classDefinition[prop];
            }
        }
    }

    var preventJSBaseConstructorCall = true;
    var Base = getClassBase();
    preventJSBaseConstructorCall = false;
```

```
    createClassDefinition.call(Base, classDefinition);

    //用于创建当前类的子类
    Base.extend = function(classDefinition) {

        preventJSBaseConstructorCall = true;
        var SonClass = getClassBase();
        SonClass.prototype = new this();//将一个父类的实例当作子类的原型
        preventJSBaseConstructorCall = false;

        createClassDefinition.call(SonClass, classDefinition);
        SonClass.extend = this.extend;

        return SonClass;
    };
    return Base;
};
```

创建一个 Animal 类与一个 Dog 子类。

```
var Animal = JS.Class({
  construct: function(name) {
    this.name = name;
  },
  shout: function(s) {
    console.log(s);
  }
});

var animal = new Animal();
animal.shout('animal'); // animal
var Dog = Animal.extend({
  construct: function(name, age) {
    //调用父类构造器
    this.parent.construct.apply(this, arguments);
    this.age = age;
  },
  run: function(s) {
    console.log(s);
  }
});
var dog = new Dog("dog", 4);
console.log(dog.name);
dog.shout("dog"); // dog
dog.run("run"); // run
```

演示静态成员的实现如下。

```
var Shepherd = Dog.extend({
  statics: { //静态成员
    TYPE: "Shepherd"
  },
  run: function() { //方法链，调用超类同名方法
```

```
    this.parent.run.call(this, "fast");
  }
});
console.log(Shepherd.TYPE); //Shepherd
var shepherd = new Shepherd("shepherd", 5);
shepherd.run(); //fast
```

JS.Class 虽然功能稍微薄弱些，但简洁得惊人。我们可以在它的基础上学习 Base2，mootools 的实现。

5.2.3　simple-inheritance

作者为大名鼎鼎的 John Resig，项目直接放在他的一篇博文中：

http://ejohn.org/blog/simple-javascript-inheritance/

特点是方法链的实现非常优雅，节俭！

源码解读如下。

```
(function() {
    // /xyz/.test(function(){xyz;})是用于判定函数的 toString 是否能暴露里面的实现
    // 因为 Function.prototype.toString 没有做出强制规定如何显示自身，根据浏览器实现而定
    // 如果里面能显示内部内容，那么我们就使用 /\b_super\b/来检测函数里面有没有.super 语句
    // 当然这个也不很充分，只是够用的程度；否则就返回一个怎么也返回 true 的正则
    // 比如一些古老版本的 Safari、Mobile Opera 与 Blackberry 浏览器，无法显示函数体的内容
    // 就需要用到后面的正则
    var initializing = false, fnTest = /xyz/.test(function() {
        xyz;
    }) ? /\b_super\b/ : /.*/;

    // 所有人工类的基类
    this.Class = function() {
    };

//这是用于生成目标类的子类
    Class.extend = function(prop) {
        var _super = this.prototype;//保存父类的原型

        //阻止 init 被触发
        initializing = true;
        var prototype = new this();//创建子类的原型
        //重新打开，方便真实用户可以调用 init
        initializing = false;
        //将 prop 里的东西逐个复制到 prototype,如果是函数将特殊处理一下
        //因为复制过程中可能掩盖了超类的同名方法，如果这个函数里面存在_super 的字样，就笼统地
        //认为它需要调用父类的同名方法，那么我们需要重写当前函数
        //重写函数运用了闭包，因此 fnTest 正则检测可以减少我们重写方法的个数，
        //因为不是每个同名函数都会向上调用父类方法
        for (var name in prop) {
            prototype[name] = typeof prop[name] === "function" &&
                    typeof _super[name] === "function" && fnTest.test(prop[name]) ?
                    (function(name, fn) {
```

```
                    return function() {
                        var tmp = this._super;//保存到临时变量中
                        //当我们调用时，才匆匆把父类的同方法覆写到_super 里
                        this._super = _super[name];
                        //然后才开始执行当前方法（这时里面的 this._super 已被重写），得到想要的效果
                        var ret = fn.apply(this, arguments);
                        //还原 this._super
                        this._super = tmp;
                        //返回结果
                        return ret;
                    };
                })(name, prop[name]) :
                prop[name];
        }

        // 这是目标类的真实构造器
        function Class() {
            // 为了防止在生成子类的原型（new this()）时触发用户传入的构造器 init
            // 使用 initializing 进行牵制
            if (!initializing && this.init)
                this.init.apply(this, arguments);
        }

        //将修改好的原型赋值
        Class.prototype = prototype;

        // 确保原型上 constructor 正确指向自身
        Class.prototype.constructor = Class;

        //添加 extend 类方法，生于生产它的子类
        Class.extend = arguments.callee;

        return Class;
    };
})();
```

创建一个 Animal 类与一个 Dog 子类。

```
var Animal = Class.extend({
  init: function(name) {
    this.name = name;
  },
  shout: function(s) {
    console.log(s);
  }
});

var animal = new Animal();
animal.shout('animal'); // animal
var Dog = Animal.extend({
  init: function(name, age) {
    //调用父类构造器
    this._super.apply(this, arguments);
```

```
    this.age = age;
  },
  run: function(s) {
    console.log(s);
  }
});
var dog = new Dog("dog", 4);
console.log(dog.name); //dog
dog.shout("xxx"); // xxx
dog.run("run"); // run
console.log(dog instanceof Dog && dog instanceof Animal);
```

顺便一提，simple-inheritance 的老师有两个，Base2 的继承系统与 Prototype.js 的方法链系统。Prototype.js 与 simple-inheritance 都是对函数的 toString 进行反编译，看里面有没有_super 或$super 的字眼才决定重写。不同的是 Prototype.js 只检测方法的参数列表，因此杂质更少，更加可靠！

下面是 Prototype.js 方法链演示。

```
var Person = Class.create();
Person.prototype = {
  initialize: function(name) {
    this.name = name;
  },
  say: function(message) {
    return this.name + ': ' + message;
  }
};

var guy = new Person('Miro');
console.log(guy.say('hi')); // "Miro: hi"
//创建子类
var Pirate = Class.create(Person, {
  say: function($super, message) { //注意这里的传参，$super 为超类的同名方法
    return $super(message) + ", yarr!"; //这需要外科手术般的闭包来实现
  }
});

var john = new Pirate('Long John');
console.log(john.say('good bye')) //Long John: good bye, yarr!
```

当然 Prototype.js 这样做有个缺憾，导致定义时与使用时的参数不一样。对于大多数用户来说，实现并不是他们所关心的，保持简洁优雅的接口才是重点。如果翻看 jQuery UI 的源码，它的 widget.类工厂已经把方法链设计得登峰造顶了。它会在函数的 this 对象添加两个临时方法，_super 相当于 simple-inheritance 的_super，它的参数需要用户逐个转手传递。_superApply 是_super 的强化版，因为如果外围方法是不确定的，那么你也没法为_super 传参，因此它只需用户丢个 arguments 对象进去！

5.2.4 体现 JavaScript 灵活性的库——def.js

如果有什么库最能体现 JavaScript 的灵活性，此库肯定名列前茅。它试图在形式上模拟 Ruby 那种继承，让学过 Ruby 的人一眼就看到哪个是父类，哪个是子类。

下面就是 Ruby 的继承示例。

```
class Child < Father
 #略
end
```

def.js 能做如下这个程度。

```
def("Animal")({
  init: function(name) {
    this.name = name;
  },
  speak: function(text) {
    console.log("this is a " + this.name);
  }
});
var animal = new Animal("Animal");

console.log(animal.name)

def("Dog") < Animal({
  init: function(name, age) {
    this._super(); //魔术般地调用父类
    this.age = age;
  },
  run: function(s) {
    console.log(s)
  }
});
var dog = new Dog("wangwang");
console.log(dog.name); //wangwang

//在命名空间对象上创建子类
var namespace = {}
def(namespace, "Shepherd") < Dog({
  init: function() {
    this._super();
  }
});

var shepherd = new namespace.Shepherd("Shepherd")
console.log(shepherd.name);
```

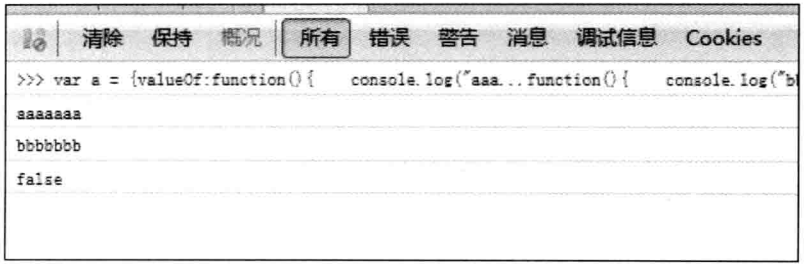
由于涉及的魔术比较多，我逐个分解一下。

第一个是 curry 的运用，在 def（“Animal”）之后它还能直接添加括号，说明它是一个函数。而此时真正的类已经创建出来，可以在当前作用域下+［“Animal”］访问到。

第二个是“<”操作符在这里的作用。其实原项目是用“<<”，不过换成“＋”、“－”也行，但要保证与 Ruby 的拟态，我还是推荐用“<”。“<”操作符目的是强制两边计算自身，从而调用自己的 valueOf 方法。def.js 就是通过重写了父类与子类定义的 valueOf 实现在某个作用域中偷偷地进行原型继承，如图 5.1 所示。

```
var a = {valueOf:function(){
```

```
  console.log("aaaaaaa")
}}, b = {valueOf:function(){
  console.log("bbbbbbb")
}}
a < b
```



▲图 5.1

由于操作符两边都是函数，那么我们能做更多的事！

```
function def(name) {
  console.log("def(" + name + ") called")
  var obj = {
    valueOf: function() {
      console.log(name + " (valueOf)")
    }
  }
  return obj
}
def("Dog") < def("Animal");
```

第三个是 arguments.callee.caller 的运用。大家看一下 Dog 的构造函数，里面只有一句 this._super()，没有传参，但它依然能调用到它的父类构造器，并把 arguments 塞进去。arguments.callee 就是指_super 这个函数，caller 就是 init 这个函数，然后我们访问 caller.arguments，就得到"wangwang"这个传参了。因此它这个_super 比 simple-inheritance 的智能多了，就像 Java 的 super 关键字那样，摆在那里自行干活。同时_super 不但能自动调用父类的构造器，同名超类方法的实现也由它一手打包。

下面是源码解读。

```
//https://github.com/RubyLouvre/def.js
(function(global) {
  //deferred 是整个库中最重要的构件，扮演三个角色
  //1 def("Animal")时就是返回 deferred，此时我们可以直接接括号对原型进行扩展
  //2 在继承父类时 < 触发两者调用 valueOf，此时会执行 deferred.valueOf 里面的逻辑
  //3 在继承父类时，  父类的后面还可以接括号（废话，此时构造器当普通函数使用），当作传送器，
  //  保存着父类与扩展包到_super,_props
  var deferred;

  function extend(source) { //扩展自定义类的原型
    var prop, target = this.prototype;

    for(var key in source)
```

```
    if(source.hasOwnProperty(key)) {
      prop = target[key] = source[key];
      if('function' == typeof prop) {
        //在每个原型方法上添加两个自定义属性，保存其名字与当前类
        prop._name = key;
        prop._class = this;
      }
    }

    return this;
  }
  // 一个中介者，用于切断子类与父类的原型连接
  //它会像 DVD+R 光盘那样被反复擦写

  function Subclass() {}

  function base() {
    // 取得调用 this._super()这个函数本身，如果是在 init 内，那么就是当前类

//http://larryzhao.com/blog/arguments-dot-callee-dot-caller-bug-in-internet-explorer-9/
    var caller = base.caller;
    //执行父类的同名方法，有两种形式，一是用户自己传，二是智能取当前函数的参数
    return caller._class._super.prototype[caller._name].apply(this, arguments.length ?
arguments : caller.arguments);
  }

  function def(context, klassName) {
    klassName || (klassName = context, context = global);
    //偷偷在给定的全局作用域或某对象上创建一个类
    var Klass = context[klassName] = function Klass() {
        if(context != this) { //如果不使用 new 操作符，大多数情况下 context 与 this 都为 window
          return this.init && this.init.apply(this, arguments);
        }
        //实现继承的第二步，让渡自身与扩展包到 deferred
        deferred._super = Klass;
        deferred._props = arguments[0] || {};
      }

      //让所有自定义类都共用同一个 extend 方法
      Klass.extend = extend;

    //实现继承的第一步，重写 deferred，乍一看是刚刚生成的自定义类的扩展函数
    deferred = function(props) {
      return Klass.extend(props);
    };

    // 实现继承的第三步，重写 valueOf，方便在 def("Dog") < Animal({})执行它
    deferred.valueOf = function() {

      var Superclass = deferred._super;

      if(!Superclass) {
        return Klass;
```

```
        }
        // 先将父类的原型赋给中介者，然后再将中介者的实例作为子类的原型
        Subclass.prototype = Superclass.prototype;
        var proto = Klass.prototype = new Subclass;
        // 引用自身与父类
        Klass._class = Klass;
        Klass._super = Superclass;
        //一个小甜点，方便人们知道这个类叫什么名字
        Klass.toString = function() {
          return klassName;
        };
        //强逼原型中的 constructor 指向自身
        proto.constructor = Klass;
        //让所有自定义类都共用这个 base 方法，它是构成方法链的关系
        proto._super = base;
        //最后把父类后来传入的扩展包混入子类的原型中
        deferred(deferred._props);
      };

      return deferred;
    }

    global.def = def;
  }(this));
```

它的实现非常巧妙。这要对 def(“Dog”)<Animal（{}）这一行的代码各个部分的执行顺序有充分的了解。无疑左边的 def 会先执行，重新擦写了 deferred 与 deferred.valueOf，然后是父类 Animal 作为普通函数接受子类的扩展包，扩展包与父类也在这时偷偷附加到 deferred 上。最后是中间的操作符触发 deferred.valueOf，完成继承！

当然也有美中不足的地方，就是利用了 caller 这个被废弃的属性。在 es5 的严格模式下，它是不可用的，导致继承系统瘫痪！这个修改也很简单，直接参考 jQuery UI 的那部分就行了，只是少了一些智能化。

5.3 es5 属性描述符对 OO 库的冲击

es5 最瞩目的升级是为对象引入属性描述符，让我们对属性有了更精细的控制，如这个属性是否可以修改，是否可以在 for in 循环中枚举出来，是否可以删除。这些新增的 API 都集中定义在 Object 下，基本上除了 Object.keys 这个方法外，其他新 API，旧版本 IE 都无法模拟。由于是新 API，没有什么书介绍，我在这里就稍微讲解一下。

Object 下总供添加以下几种新方法。

- Object.keys
- Object.getOwnPropertyNames
- Object.getPrototypeOf
- Object.defineProperty
- Object.defineProperties

- Object.getOwnPropertyDescriptor
- Object.create
- Object.seal
- Object.freeze
- Object.preventExtensions
- Object.isSealed
- Object.isFrozen
- Object.isExtensible

其中除 Object.keys 外，旧版本 IE 都无法模拟这些新 API。旧版式的标准浏览器，可以用__proto__实现 Object.getPrototypeOf，结合__defineGetter__与__defineSetter__来模拟 Object.defineProperty。

Object.keys 用于收集当前对象的可遍历属性（不包括原型链上的），以数组形式返回。这个我在之章的章节已经给出兼容函数。

Object.getOwnPropertyNames 用于收集当前对象不可遍历属性与可遍历属性（不包括原型链上），以数组形式返回。

```
var obj = {
  aa: 1,
  toString: function() {
    return "1"
  }
}
if(Object.defineProperty && Object.seal) {
  Object.defineProperty(obj, "name", {
    value: 2
  })
}
console.log(Object.getOwnPropertyNames(obj));//["aa", "toString", "name"]
console.log(Object.keys(obj));//["aa", "toString"]

function fn(aa, bb) {};
console.log(Object.getOwnPropertyNames(fn));//["prototype","length","name","arguments
","caller"]
console.log(Object.keys(fn));//[]
var reg = /\w{2,}/i

console.log(Object.getOwnPropertyNames(reg));//["lastIndex","source","global","ignore
Case","mnltiline","sticky"]
console.log(Object.keys(reg));//[]
```

Object.getPrototypeOf 返回参数对象的内部属性 [[Prototype]]，它在标准浏览器中一直使用一个私有属性__proto__获取（IE9、IE10、Opera 没有）。需要补充一下，Object 的新 API（除了 Object.create 外）有个统一的规定，要求第一个参数不能为数字、字符串、布尔、null、undefined 这五种的字面量，否则抛出一个 TypeError 异常。

```
console.log(Object.getPrototypeOf(function() {}) == Function.prototype); //true
console.log(Object.getPrototypeOf({}) === Object.prototype); //true
```

Object.defineProperty 暴露了属性描述的接口，之前许多内建属性都是由 JavaScript 引擎在水下操作。如 for in 循环为何不能遍历出函数的 arguments、length、name 等属性名，delete window.a 为何返回 false，这些现象终于有个解释。它一共涉及六个可组合的配置项：是否可重写 writable，当前值 value，读取时内部调用的函数 set，写入时内部调用函数 get，是否可以遍历 enumerable，是否可让人家再次改动这些配置项 configurable。比如我们随便写个对象：

```
var obj = { x : 1 };
```

有了属性描述符，我们就清楚它在底下做的更多细节，它相当于 es5 的这个创建对象的式子：

```
var obj = Object.create(Object.prototype,
  { x : {
    value : 1,
    writable : true,
    enumerable : true,
    configurable : true
  }}
)
```

如果对比 es3 与 es5，就很快明白，曾经的 [[ReadOnly]]、[[DontEnum]]、[[DontDelete]] 改换成 [[Writable]]、[[Enumerable]]、[[Configurable]] 了。因此 configurable 还有兼顾能否删除的职能。

这六个配置项将原有的本地属性拆分为两组，数据属性与访问器属性。我们之前的方法可以像数据属性那样定义。

es3 时代，我们的自定义类的属性可以统统看作是数据属性。

像 DOM 中的元素节点的 innerHTML、innerText、cssText，数组的 length 则可归为访问器属性，对它们赋值时不是单纯的赋值，还会引发元素其他功能的触发，而取值也不一定直接返回我们之前给予的值。

数据属性拥有 1、2、5、6 这四个配置项，访问器属性拥有 3、4、5、6 这四个配置项。如果你设置了 value 与 writable，就不能设置 set、get，反之亦然。如果没有设置，第 2、3、4 项默认为 false，第 1、5、6 项默认为 false。

```
var obj = {};
Object.defineProperty(obj, "a", {
  value: 37,
  writable: true,
  enumerable: true,
  configurable: true
});

console.log(obj.a);//37
obj.a = 40;
console.log(obj.a);//40
var name = "xxx"
for(var i in obj){
    name = i
}
console.log(name);//a
```

```
Object.defineProperty(obj, "a", {
  value: 55,
  writable: false,
  enumerable: false,
  configurable: true
});

console.log(obj.a);//55
obj.a = 50;
console.log(obj.a);//55
name = "b";
for(var i in obj){
   name = i
}
console.log(name);//b


var value = "RubyLouvre";
Object.defineProperty(obj, "b", {
  set: function(a){
    value = a;
  },
  get: function(){
    return value + "!"
  }
});

console.log(obj.b);//RubyLouvre!
obj.b = "bbb";
console.log(obj.b);//bbb!

var obj = Object.defineProperty( {} , 'a', {
  value: "aaa"
});
delete obj.a;//configurable 默认为 false，此属性不能删除
console.log(obj.a);//aaa
```

但这东西各浏览器也有差异，只怪 ecmascript 总爱作事后孔明。

```
var arr = [];
//添加一个属性，但由于是数字字面量，它又会作为数组的第一个元素
Object.defineProperty(arr, '0', {value : "零"});
Object.defineProperty(arr, 'length', {value : 10});
//删除第一个元素，但由于 length 的 writable 在上面被我们设置为 false(不写默认为 false)，因此改不了。
arr.length = 0 ;
alert([arr.length, arr[0]]);//正确应该输出 "1，零"
//IE9、IE10："1,零"
//Firefox4～Firefox19：抛内部错误，说当前不支持定义 length 属性
//Safari5.0.1："0, "，第二值应该是 undefined，说明它忽略了 writable 为 false 的默认设置，让
arr.length 把第一个元素删掉了
//Chrome14-："0,零"，估计后面的 "零" 是作为属性打印出来，chrome24 与标准保持一致
```

此外 defineProperty 的第三个参数配置对象好像没有使用 hasOwnProperty 进行取值，导致一旦 Object.prototype 被污染，就很容易程序崩溃。这情况好像所有现代浏览器都踩坑了。

```
Object.prototype.set = undefined
var obj = {};
Object.defineProperty(obj, "aaa", { value: "OK" });
//TypeError: property descriptor's getter field is neither undefined nor a function
```

或者：

```
Object.prototype.get = function(){};
var obj = {};
Object.defineProperty(obj,  "aaa", { value: "OK" });
//TypeError: property descriptors must not specify a value or be writable when a getter
or setter has been specified
```

如果真的碰巧让你撞上这事，唯有自力更生了。

```
function hasOwn(obj, key) {
  return Object.prototype.hasOwnProperty.call(obj, key);
}
function defineProperty(obj, key, desc) {
//创建一个纯空对象，不继承 Object.prototype，跳过那些粗糙的 for in 遍历 BUG
  var d = Object.create(null);
  d.configurable = hasOwn(desc,"configurable");
  d.enumerable = hasOwn(desc, "enumrable");
  if (hasOwn(desc, "value")) {
    d.writable = hasOwn(desc, "writable")
    d.value = desc.value;
  } else {
    d.get = hasOwn(desc, "get") ? desc.get : undefined;
    d.set = hasOwn(desc, "set") ? desc.set : undefined;
  }
  return Object.defineProperty(obj, key, d);
}
var obj = {};
defineProperty(obj, "aaa", { value: "OK" });//save!
```

在标准浏览器中，如果不支持 Object.defineProperty，我们可以勉强模拟它出来。

```
if(typeof  Object.defineProperty!=='function'){
   Object.defineProperty = function(obj, prop, desc) {
       if ('value' in desc) {
           obj[prop] = desc.value;
       }
       if ('get' in desc) {
           obj.__defineGetter__(prop, desc.get);
       }
       if ('set' in desc) {
           obj.__defineSetter__(prop, desc.set);
       }
       return obj;
   };
}
```

Object.defineProperties 就是 Object.defineProperty 的加强版，它能一下子处理多个属性。因此如果你能模拟 Object.defineProperty，它就不是问题。

```
if(typeof  Object.defineProperties!=='function'){
    Object.defineProperties = function(obj, descs) {
        for (var prop in descs) {
            if (descs.hasOwnProperty(prop)) {
                Object.defineProperty(obj, prop, descs[prop]);
            }
        }
        return obj;
    };
}
```

使用示例：

```
var obj = {};
Object.defineProperties(obj, {
  "value": {
    value: true,
    writable: false
  },
  "name": {
    value: "John",
    writable: false
  }
});
var a = 1;
for(var p in obj) {
  a = p;
}
console.log(a);//1
```

Object.getOwnPropertyDescriptor 用于获得某对象的本地属性的配置对象，其中 configurable,enumerable 肯定包含其中，视情况再包含 value，writable 或 set，get。

```
var obj = {},
  value = 0
  Object.defineProperty(obj, "aaa", {
    set: function(a) {
      value = a;
    },
    get: function() {
      return value
    }
  });
//一个包含 set, get, configurable, enumerable 的对象
console.log(Object.getOwnPropertyDescriptor(obj, "aaa"));
console.log(typeof obj.aaa);//number
console.log(obj.hasOwnProperty("aaa"));//true


(function() {
//一个包含 value, writable, configurable, enumerable 的对象
  console.log(Object.getOwnPropertyDescriptor(arguments, "length"))
})(1, 2, 3);
```

由于属性在现代浏览器划分两阵营了，如果我们想把一个对象的成员赋给另一个对象，原来的 mixin 就会捉襟见肘。这时 getOwnPropertyDescriptor 就大派用场了。

```
function mixin(receiver, supplier) {
    if (Object.getOwnPropertyDescriptor) {
        Object.keys(supplier).forEach(function(property) {
            Object.defineProperty(receiver,  property,  Object.getOwnPropertyDescriptor
(supplier, property));
        });
    } else {
        for (var property in supplier) {
            if (supplier.hasOwnProperty(property)) {
                receiver[property] = supplier[property];
            }
        }
    }
}
```

Object.create 用于创建一个子类的原型，第一个参数为父类的原型，第二个是子类另外要添加的属性的配置对象。如果我们能模拟 Object.defineProperties，它也能模拟得到。

```
if(typeof  Object.create !== 'function') {
  Object.create = function(prototype, descs) {
    function F() {}

    F.prototype = prototype;
    var obj = new F();
    if(descs != null) {
      Object.defineProperties(obj, descs);
    }
    return obj;
  };
}
```

有了它，我们创建子类会更方便些。

```
function Animal(name) {
  this.name = name
}
Animal.prototype.getName = function() {
  return this.name;
}

function Dog(name, age) {
  Animal.call(this, name);
  this.age = age;
}
Dog.prototype = Object.create(Animal.prototype, {
  getAge: {
    value: function() {
      return this.age;
    }
  },
  setAge: {
```

```
    value: function(age) {
      this.age = age;
    }
  }
});

var dog = new Dog("dog", 4);
console.log(dog.name);//dog
dog.setAge(6);
console.log(dog.getAge());//6
```

Object.create(null)还能生成一种叫纯空对象的东西，没有 toString, valueOf，什么也没有，空空荡荡，在 Object.prototype 被污染或极需节省内存的情况下有用。外国还是有人设法在旧版本 IE 中模拟出来。

```
//https://github.com/kriskowal/es5-shim/blob/master/es5-sham.js
var createEmpty;
var supportsProto = Object.prototype.__proto__ === null;
if(supportsProto || typeof document == 'undefined') {
  createEmpty = function() {
    return {
      "__proto__": null
    };
  };
} else {
  // 因为我们无法让一个对象继承自一个不存在的东西，它最后肯定要回溯到
  //Object.prototype，那么我们就从一个新的执行环境中盗取一个 Object.prototype，
  //把它的所有原型属性都砍掉，这样它的实例就既没有特殊属性，也没有什么原型属性
  //（只剩下一个__proto__，值为 null）
  createEmpty = (function() {
    var iframe = document.createElement('iframe');
    var parent = document.body || document.documentElement;
    iframe.style.display = 'none';
    parent.appendChild(iframe);
    iframe.src = 'javascript:';
    var empty = iframe.contentWindow.Object.prototype;
    parent.removeChild(iframe);
    iframe = null;
    delete empty.constructor;
    delete empty.hasOwnProperty;
    delete empty.propertyIsEnumerable;
    delete empty.isPrototypeOf;
    delete empty.toLocaleString;
    delete empty.toString;
    delete empty.valueOf;
    empty.__proto__ = null;

    function Empty() {}
    Empty.prototype = empty;

    return function() {
      return new Empty();
    };
  })();
}
```

但是在 firebug 下还是能区分出它们的不同。

Object.preventExtensions，它是三个封锁对象修改的 API 中程度最轻的那个，就是阻止添加本地属性，不过如果本地属性被删除了，也无法再加回来。以前 JavaScript 对象的属性都是随意添加、删除、修改其值，如果它的原型改动，我们访问它还会有“意外之喜”。

```
var a = {
  aa: "aa"
};
Object.preventExtensions(a)
a.bb = 2;
console.log(a.bb);          //undefined 添加本地属性失败
a.aa = 3;
console.log(a.aa);          //3  允许它修改原有属性
delete a.aa;
console.log(a.aa);          //undefined 但允许它删除已有属性
Object.prototype.ccc = 4;
console.log(a.ccc);         //4  不能阻止它增添原型属性
a.aa = 5;
console.log(a.bb);          //undefined ，不吃回头草，估计里面是以白名单方式实现的
```

Object.seal 比 Object.preventExtensions 更过分,它不准删除已有的本地属性。内部实现就是遍历一下，把每个本地属性的 configurable 改为 false。

```
var a = {
  aa: "aa"
};
Object.seal(a)
a.bb = 2;
console.log(a.bb);          //undefined 添加本地属性失败
a.aa = 3;
console.log(a.aa);          //3   允许它修改已有属性
delete a.aa;
console.log(a.aa);          //3  但不允许它删除已有属性
```

Object.freeze 无疑是最专制的（因此有人说过程式程序很专制，OO 程序则自由些[①]，显然道格拉斯主导的 ecma262v5 想把 JavaScript 引向前者），它连原有本地属性也不让修改了。内部实现就是遍历一下，把每个本地属性的 writable 也改为 false。

```
var a = {
  aa: "aa"
```

① 郑晖著的《冒号课堂——编程范式与 OOP 思想》第 41 页。

```
};
Object.freeze(a)
a.bb = 2;
console.log(a.bb);          //undefined 添加本地属性失败
a.aa = 3;
console.log(a.aa);          //aa   允许它修改已有属性
delete a.aa;
console.log(a.aa);          //aa  但不允许它删除已有属性

Object.isExtensible(object);
Object.isSealed(object);
Object.isFrozen(object);
```

判定一个对象是否被锁定。锁定，意味着无法扩展。如果一个对象被冻结了，它肯定被锁定，也肯定无法扩展新本地属性了。

后面利用这些新 API 开发一个新工厂出来。

```
(function(global) {
 function fixDescriptor(item, definition, prop) {
   // 如果以标准 defineProperty 的第三个参数的形式定义扩展包
   if(isPainObject(item)) {
     if(!('enumerable' in item)) {
       item.enumerable = true;
     }
   } else { //如果是以 es3 那样普通对象定义扩展包
     item = definition[prop] = {
       value: item,
       enumerable: true,
       writable: true
     };
   }
   return item;
 }

 function isPainObject(item) {
   if(typeof item === 'object' && item !== null) {
     var a = Object.getPrototypeOf(item);
     return a === Object.prototype || a === null;
   }
   return false;
 }
 var funNames = Object.getOwnPropertyNames(Function);
 global.Class = {
   create: function(superclass, definition) {
     if(arguments.length === 1) {
       definition = superclass;
       superclass = Object;
     }
     if(typeof superclass !== "function") {
       throw new Error("superclass must be a function");
     }
     var _super = superclass.prototype;
     var statics = definition.statics;
     delete definition.statics;
     //重新调整 definition
```

```
    Object.keys(definition).forEach(function(prop) {
      var item = fixDescriptor(definition[prop], definition, prop);
      if(typeof item.value === "function" && typeof _super[prop] === "function") {
        var __super = function() { //创建方法链
            return _super[prop].apply(this, arguments);
          };
        var __superApply = function(args) {
            return _super[prop].apply(this, args);
          };
        var fn = item.value;
        item.value = function() {
          var t1 = this._super;
          var t2 = this._superApply;
          this._super = __super;
          this._superApply = __superApply;
          var ret = fn.apply(this, arguments);
          this._super = t1;
          this._superApply = t2;
          return ret;
        }
      }
    });
    var Base = function() {
        this.init.apply(this, arguments);
      };
    Base.prototype = Object.create(_super, definition);
    Base.prototype.constructor = Base;
    //确保一定存在 init 方法
    if(typeof Base.prototype.init !== "function") {
      Base.prototype.init = function() {
        superclass.apply(this, arguments);
      };
    }
    if(Object !== superclass) { //继承父类的类成员
      Object.getOwnPropertyNames(superclass).forEach(function(name) {
        if(funNames.indexOf(name) === -1) {
          Object.defineProperty(Base,name,Object.getOwnPropertyDescriptor(superclass, name));
        }
      });
    }
    if(isPainObject(statics)) { //添加自身的类成员
      Object.keys(statics).forEach(function(name) {
        if(funNames.indexOf(name) === -1) {
          Object.defineProperty(Base,name,fixDescriptor(statics[name], statics, name));
        }
      });
    }
    return Object.freeze(Base);
  }
 }

})(this)
```

fixDescriptor 方法用于修正扩展包的值的格式，让大家只描述最重要的部分。

isPainObject 用于判定目标是不是纯净的 JavaScript 对象，且不是其他自定义类的实例。用法与 Prototype.js 的 Class.create 一样，并参照 jQuery UI 提供了完美的方法链与静态成员的继承。

```
var Dog = Class.create(Animal, {
  statics: {
    Name: "Dog",
    type: "shepherd"
  },
  init: function(name, age) {
    this._super(name);
    //或者 this._superApply(arguments)
    this.age = age;
  },
  getName: function() {
    return this._super() + "!"
  },
  getAge: function() {
    return this.age
  },
  setAge: function(age) {
    this.age = age
  }
});
var dog = new Dog("dog", 12)
console.log(dog.getName()); //dog!
console.log(dog.getAge()); //12
console.log(dog instanceof Animal); //true
console.log(Dog.Name); //Dog
```

总结，es5 对 JavaScript 的对象产生深刻影响。Object.create 让原型继承更方便了，但在增添子类的专有原型成员或类成员时，如果它们的属性的 enumerable 为 false，单纯的 for in 循环已经不管用了，我们就要用到 Object.getOwnPropertyNames。另外，访问器属性的复制只有通过 Object.getOwnPropertyDescriptor 与 Object.defineProperty 才能完成。

第6章 选择器引擎

jQuery 凭借选择器风靡全球，从而使各大框架类库争先开发自己的选择器，一时间选择器成为框架的标配。

其实，早期 jQuery 选择器与我们现在看到的大不一样。它最初是使用混杂 xpath 语法的 selector，第二代转为纯 CSS 带自定义伪类（比如从 xpath 借鉴过来位置伪类）的 Sizzle。但 Sizzle 也一直在变，因为它的关系选择器一直存在问题，因此不断重构，在 jQuery1.9 时终于搞定，并最终决定全面支持 CSS3 的结构伪类。

有据可查的早期三大选择器引擎是 2003 年 Simon Willison 的 getElementsBySelector，然后是 2004 年 Dean Edwards 的 cssQuery，jQuery 是 2005 年发布，据 John Resig 在《JavaScript 精粹》说，他本来只想写个选择器引擎，但 cssQuery “光芒太盛”，无法与之争锋，匆忙间作为一个较为完整的 dom 类库面世。

2005 年，Ben Nolan 的 Behaviour.js，内置了早以闻名于世的 getElementsBySelector，是第一个集成事件处理、CSS 风格的选择器引擎与 onload 处理的类库。此外日后霸主 Prototype.js 也在 2005 年诞生。但它勉强称得上是，选择器$与 getElementsByClassName 在 1.2 出现，事件处理在 1.3，因此 Behaviour 还能风光一时。你们可以到这里下载 Behaviour.js：

http://www.ccs.neu.edu/home/dherman/javascript/behavior/

本章介绍如何从头到尾制造一个选择器引擎，在此我们先看看前人的努力吧。

6.1 浏览器内置的寻找元素的方法

请不要追问 2005 年之前开发人员是怎么在这种缺东缺西的环境下干活的，那时浏览器大战打得正酣，程序员们发明了 navigator.userAgent 检测进行“自保”!网景战败，因此有关它的记录不多。但 IE 确确切切留下许多资料，比如取得元素，我们直接可以根据元素的 ID 就取得元素自身[①]，不通过任何 API，自动映射成全局变量。在不关注全局污染时，这是很酷的特性。又如取得所有元素，直接 document.all。取得某一种标签类型的元素，只需做一下分类，如 P 标签，document.all.tags("p")，时至今天，IE4 这个古老 API 还能在 IE10 标准模式下正常运作！

有资料可查的是 getElementById、getElementsByTagName 是 IE5 引入的，那是 1999 年的事，

① 现在所有新浏览器都支持这个特性。http://stackoverflow.com/questions/3434278/ie-chrome-are-dom-tree-elements-global-variables-here

与微软另一个辉煌的产品 Windows98，捆绑在一起。因此，那时的程序的代码都倾向于为 IE 做兼容。我在网上找到一个让 IE4 支持 getElementById 的代码，刻着时代的“烙印”。

```
var ie4=document.all && !document.getElementById;
if(ie4) {
  document.getElementById = new Function('var expr = /^\\w[\\w\\d]*$/,'+
    'elname=arguments[0]; if(!expr.test(elname)) { return null; } '+
    'else if(eval("document.all."+elname)) { return '+
    'eval("document.all."+elname); } else return null;')
}
```

此外还有 getElementsByTagName 的实现。

```
function getElementsByTagName(str) {
   if (str == "*") {
      return document.all
   } else {
      return document.all.tags(str)
   }
}
```

但人们很快就发现问题了，IE 的 getElementById 是不区分表单元素 ID 与 Name，因此如果有一个表单元素只定义 name 并与我们的目标元素 ID 同名，且我们的目标元素在它的后面，那么就会选错元素。这个问题一直延续到 IE7。

IE 的 getElementsByTagName 也有问题。当参数为*号通配符时，它会混入注释节点，并且无法选取 Object 下的元素。

下面是解决方法。

```
//J. Max Wilson
if (/msie/i.test (navigator.userAgent)) {//only override IE
   document.nativeGetElementById = document.getElementById;
   document.getElementById = function(id){
      var elem = document.nativeGetElementById(id);
      if(elem){//IE5
         if(elem.id == id){
            return elem;
         }else{//IE4
            for(var i=1;i<document.all[id].length;i++){
               if(document.all[id][i].id == id){
                  return document.all[id][i];
               }
            }
         }
      }
      return null;
   };
}
//Dean Edwards
function getElementsByTagName(node, tagName) {
  var elements = [], i = 0, anyTag = tagName === "*", next = node.firstChild;
  while ((node = next)) {
    if (anyTag ? node.nodeType === 1 : node.nodeName === tagName) elements[i++] = node;
```

```
        next = node.firstChild || node.nextSibling;
        while (!next && (node = node.parentNode)) next = node.nextSibling;
      }
      return elements;
    };
```

此外 W3C 还提供了一个 getElementsByName 的方法，这个 IE 也有问题，它只能选取表单元素，由于我们后面用不到它，先行略去。

这是 Prototype.js 到来之前，所有可用的原生选择器。因此 Simon Willison 搞出 getElementsBySelector，让世人眼前一亮。

之后的情况大家应该知道了，出现 N 个版本的 getElementsBySelector。不过大多数是在 Simon Willison 的基础上改进的，甚至当时还讨论将它标准化！

http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2005-September/subject.html#4782

虽然这个打算最后搁浅了，但 Simon Willison 的 getElementsBySelector 代表的是历史的前进方向。jQuery 则有点偏向了。Prototype.js 则在 Ajax 热炒浪潮中扶摇直上，1.4 在 document 添加日后成为标准的 getElementsByClassName 与失败了的 getElementsBySelector，此外还有比 jQuery 正更统些的$$。不过，jQuery 最终还是胜利了，Sizzle 的设计很特别，各种优化别出心裁。

浏览器没有闲着，Netscape 借 Firefox 还魂，挑起第二次浏览器战争，其间往 HTML 引入 XML 的 xpath，其 API 为 document.evaluate。但 xpath 又分为 level1、level2、level3，各浏览器在不同版本的支持又不一致，加之语法比较复杂，因此普及不开，更甭论存在什么 BUG。同一时间还有 getElementsByClassName，这个也通常只见于选择器引擎的内部应用，它也存在 BUG，分别位于 Safari 与 Opera（下面章节会介绍）。

微软为了保住占有率，在 IE8 上加入 querySelector 与 querySeletorAll，相当于 getElementsBySelector 的升级版，它还支持前所未有的结构伪类、状态伪类、语言伪类与取反伪类。这时 Chrome 参战，激发标准浏览器的升级热情，IE8 新加的选择器大家都支持了，还支持得更标准。此时还出现了一种类此选择器的匹配器——matchesSelector ，它对我们编写选择器引擎非常有帮助，由于是在版本号竞赛时诞生的，谁也不能担保自己的实现被 W3C 采纳，因此都带有私有前缀。现在 CSS 方面有关 selector 4 的规范还在起草中，querySeletorAll 也暂只支持到 selector 3 部分，但其间的兼容性问题已经很杂乱了。

6.2 getElementsBySelector

我们先来看一下这个最古老的选择器引擎。它规定了今后许多选择器的发展方向。在解读中可能涉及许多概念，但不要紧，后面有更详细的解析。现在只是初步了解一下大概蓝图。

```
/* document.getElementsBySelector(selector)
   New in version 0.4: Support for CSS2 and CSS3 attribute selectors:
   See http://www.w3.org/TR/css3-selectors/#attribute-selectors
       Download by http://www.bvbsoft.com
   Version 0.4 - Simon Willison, March 25th 2003
   -- Works in Phoenix 0.5, Mozilla 1.3, Opera 7, Internet Explorer 6, Internet Explorer
5 on Windows
```

```
    -- Opera 7 fails
*/
 //
function getAllChildren(e) {
  //取得一个元素的所有子孙，兼并容 IE5
  return e.all ? e.all : e.getElementsByTagName('*');
}

document.getElementsBySelector = function(selector) {
  //如果不支持 getElementsByTagName 则直接返回空数组
  if (!document.getElementsByTagName) {
    return new Array();
  }
  //切割 CSS 选择符，分解一个个单元（每个单元可能代表一个或几个选择器，比如 p.aaa 则由标签选择器与类选择器组成）
  var tokens = selector.split(' ');
  var currentContext = new Array(document);
  //从左到右检测每个单元，换言之此引擎是自顶向下选元素
  //我们的结果集如果中间为空，那么就立即中止此循环了
  for (var i = 0; i < tokens.length; i++) {
    //去掉两边的空白（但并不是所有的空白都是没用，
    //两个选择器组之间的空白代表着后代选择器，这要看作者们的各显神通了）
    token = tokens[i].replace(/^\s+/,'').replace(/\s+$/,'');;
    //如果包含 ID 选择器，这里略显粗糙，因为它可能在引号里面
    //此选择器支持到属性选择器，则代表着它可能是属性值的一部分
    if (token.indexOf('#') > -1) {
      // 这里假设这个选择器组以 tag#id 或#id 的形式组成，可能导致 BUG
      //但这暂且不谈，我们还是沿着作者的思路进行下去吧
      var bits = token.split('#');
      var tagName = bits[0];
      var id = bits[1];
      //先用 ID 值取得元素，然后判定元素的 tagName 是否等于上面的 tagName
      //此处有一个不严谨的地方，element 可能为 null，会引发异常
      var element = document.getElementById(id);
      if (tagName && element.nodeName.toLowerCase() != tagName) {
        // 没有直接返回空结果集
        return new Array();
      }
      //置换 currentContext，跳至下一个选择器组
      currentContext = new Array(element);
      continue;
    }
    // 如果包含类选择器，这里也假设它以.class 或 tag.class 的形式
    if (token.indexOf('.') > -1) {

      var bits = token.split('.');
      var tagName = bits[0];
      var className = bits[1];
      if (!tagName) {
        tagName = '*';
      }
      // 从多个父节点出发，取得它们的所有子孙，
```

```
    // 这里的父节点即包含在 currentContext 的元素节点或文档对象
    var found = new Array;//这里是过滤集，通过检测它们的 className 决定去留
    var foundCount = 0;
    for (var h = 0; h < currentContext.length; h++) {
      var elements;
      if (tagName == '*') {
          elements = getAllChildren(currentContext[h]);
      } else {
          elements = currentContext[h].getElementsByTagName(tagName);
      }
      for (var j = 0; j < elements.length; j++) {
        found[foundCount++] = elements[j];
      }
    }
    currentContext = new Array;
    var currentContextIndex = 0;
    for (var k = 0; k < found.length; k++) {
      //found[k].className 可能为空，因此不失为一种优化手段，但 new RegExp 放在//外围更适合
      if (found[k].className && found[k].className.match(new RegExp('\\b'+className+'\\b'))){
        currentContext[currentContextIndex++] = found[k];
      }
    }
    continue;
  }
  //如果是以 tag[attr(~|^$*)=val]或[attr(~|^$*)=val]的形式组合
  if (token.match(/^(\w*)\[(\w+)([=~\|\^\$\*]?)=?"?([^\]"]*)"?\]$/)) {
    var tagName = RegExp.$1;
    var attrName = RegExp.$2;
    var attrOperator = RegExp.$3;
    var attrValue = RegExp.$4;
    if (!tagName) {
      tagName = '*';
    }
    // 这里的逻辑以上面的 class 部分相似，其实应该抽取成一个独立的函数
    var found = new Array;
    var foundCount = 0;
    for (var h = 0; h < currentContext.length; h++) {
      var elements;
      if (tagName == '*') {
          elements = getAllChildren(currentContext[h]);
      } else {
          elements = currentContext[h].getElementsByTagName(tagName);
      }
      for (var j = 0; j < elements.length; j++) {
        found[foundCount++] = elements[j];
      }
    }
    currentContext = new Array;
    var currentContextIndex = 0;
    var checkFunction;
    //根据第二个操作符生成检测函数，后面章节会详解，这里不展开
    switch (attrOperator) {
      case '=': //
```

```
            checkFunction = function(e) { return (e.getAttribute(attrName) == attrValue); };
            break;
          case '~':
            checkFunction = function(e) { return (e.getAttribute(attrName).match(new RegExp
('\\b'+attrValue+'\\b'))); };
            break;
          case '|':
            checkFunction = function(e) { return (e.getAttribute(attrName).match(new RegExp
('^'+attrValue+'-?'))); };
            break;
          case '^':
            checkFunction = function(e) { return (e.getAttribute(attrName).indexOf(attrValue)
== 0); };
            break;
          case '$':
            checkFunction = function(e) { return (e.getAttribute(attrName).lastIndexOf
(attrValue) == e.getAttribute(attrName).length - attrValue.length); };
            break;
          case '*':
            checkFunction = function(e) { return (e.getAttribute(attrName).indexOf(attrValue)
> -1); };
            break;
          default :
            checkFunction = function(e) { return e.getAttribute(attrName); };
        }
        currentContext = new Array;
        var currentContextIndex = 0;
        for (var k = 0; k < found.length; k++) {
          if (checkFunction(found[k])) {
            currentContext[currentContextIndex++] = found[k];
          }
        }
        continue;
      }
      // 如果没有 "#", ".", "[" 这样的特殊字符，我们就当成是 tagName
      tagName = token;
      var found = new Array;
      var foundCount = 0;
      for (var h = 0; h < currentContext.length; h++) {
        var elements = currentContext[h].getElementsByTagName(tagName);
        for (var j = 0; j < elements.length; j++) {
          found[foundCount++] = elements[j];
        }
      }
      currentContext = found;
    }
    return currentContext;//最后返回结果集
}
```

显然受当时的网速限制，页面不会很大，也不可能发展起复杂的交互，因此 JavaScript 还没有到大规模使用的阶段，我们看到那时的库也不怎么重视全局污染。主要 API 直接在 document 上操作，参数只有一个 CSS 表达符。从我们的分析来看，它不支持并联选择器（后面介绍），并且要求每个选择器组不能超出两个，否则报错。换言之，它只对下面这样形式的 CSS 表达式有效：

```
#aa p.bbb [ccc=ddd]
```

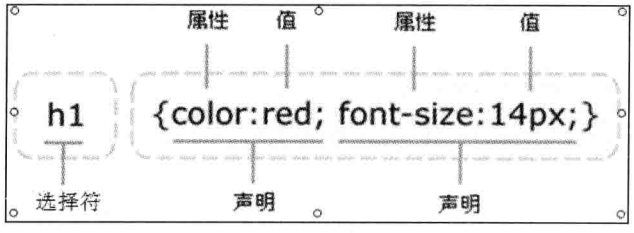
CSS 表达符将以空白分割成多个选择器组，每个选择器不能超过两种选择器类型，并且其中一种为标签选择器。

要求比较严格，文档也没有指明，因此非常糟糕。但对当时的编程环境来说，已经是喜出望外了。作为早期的选择器，它也没有像以后那样对结果集进行去重，把元素逐个按照文档出现的顺序进行排序。我们在第一节指出的 Bug，它也没有规避，这可能受当时 JavaScript 技术交流太少所致。这些都是我们是日后要改进的地方。

6.3 选择器引擎涉及的知识点

这一节我们开始学习一下上节介绍的大量概念。其中，有关选择器引擎实现的概念大多数是我从 Sizzle 中抽取出来的，而 CSS 表达符部分则是 W3C 提供的。那么我们先从 CSS 表达符部分讲起吧。

选择符是指一条 CSS 样式规则的最左边的部分，如图 6.1 所示。



▲图 6.1　CSS 样式规则

上面的只是理想情况，重构人员交给我们的 CSS 文件，里面的选择符可是复杂多了。选择符混杂着大量的标记，可以分割为许多更细的单元。总的来说，分为四大类十七种。此外，还没有包含选择器引擎无法操作的伪元素。

四大类是指并联选择器、简单选择器、关系选择器与伪类。

并联选择器就是“,”，一种不是选择器的选择器，用于合并多个分组的结果。

简单选择器分五种：ID、标签、类、属性、通配符。

关系选择器分四种：亲子、后代、相邻、兄长。

伪类分六种：动作伪类、目标伪类、语言伪类、状态伪类、结构伪类、取反伪类。

简单选择器又称为基本选择器，这是在 Prototype.js 之前的选择器都已经支持的选择器类型。不过 CSS 上，IE7 才支持部分属性选择器。其中，它们设计得非常整齐划一，我们可以通过它的第一个字符决定它们的类型。比如：ID 选择器的第一个字符为“#”；类选择器为“.”；属性选择器为“[”；通配符选择器为“*”；标签选择器为英文字母，你也可以大概解释为没有特殊符号，jQuery 就是使用/isTag = !/\W/.test(part)进行判定的。

在实现上，我们在这里有许多原生 API 可用，如 getElementById、getElementsByTagName、getElementsByClassName、document.all，属性选择器可以用 getAttribute、getAttributeNode、attributes、

hasAttribute， 2003 年曾讨论引入 getElementsByAttribute， 但没有成功，Firefix 上 XUI 的同名 API 就是当时的产物。不过属性选择器的确比较复杂，历史上它是分两步实现的。

CSS2.1 中，属性选择器有以下四种形态。

[att]：选取设置了 att 属性的元素，不管设定的值是什么。

[att=val]：选取所有 att 属性的值完全等于 val 的元素。

[att~=val]：表示一个元素拥有属性 att，并且该属性含有空格分隔的一组值，其中之一为 'val'。这个大家应该能联想到类名，如果浏览器不支持 geElementsByClassName,在过滤阶段，我们可以将.aaa 转换为[class~=aaa]来处理。

[att|=val]：表示一个元素拥有属性 att，并且该属性含 'val' 或以 'val-' 开头。

CSS3 中，属性选择器又增加三种形态。

[att^=val]：选取所有 att 属性的值以 val 开头的元素。

[att$=val]：选取所有 att 属性的值以 val 结尾的元素。

[att*=val]：选择所有 att 属性的值包含 val 字样的元素。以上三者我们都可以通过 indexOf 轻松实现。

此外，大多数选择器引擎，还实现了一种[att!=val]的自定义属性选择器。意思很简单，选取所有 att 属性不等于 val 的元素，这正好与[att=val]相反。这个我们可以通过 CSS3 的取反伪类实现。

我们再看关系选择器。关系选择器是不能单独存在的，它必须在其他两类选择器组合使用，在 CSS 里，它必须夹在它们中间，但选择器引擎可能允许它放在开始。在很长时间内，只存在后代选择器（E　F），就在两个简单选择器 E 与 F 之间的空白。CSS2.1 又添加了两个，亲子选择器（E＞F）与相邻选择器（E＋F），它们也夹在两个简单选择器之间，但允许大于号或加号两边存在空白，这时，空白就不是表示后代选择器。CSS3 又添加了一个，兄长选择器（E～F），规则同上。CSS4 又增加了一个父亲选择器，不过其规则一直在变，这里就不说了。下面是详解。

后代选择器：通常我们在引擎内构建一个 getAll 的函数，要求传入一个文档对象或元素节点取得其子孙。这里要特别注意 IE 下 document.all, getElementsByTagName("*")混入注释节点的问题。

亲子选择器：这个我们如果不打算兼容 XML，那么直接使用 children 就行了。不过 IE5～IE8 它都会混入注释节点。下面是兼容列表。

Chrome	Firefox	IE	Opera	Safari
1+	3.5+	5+	10+	4+

```
function getChildren(el) {
    if (el.childElementCount) {
        return [].slice.call(el.children);
    }
    var ret = [];
    for (var node = el.firstChild; node; node = node.nextSibling) {
        node.nodeType == 1 && ret.push(node);
    }
    return ret;
}
```

相邻选择器：就是取得当前元素向右的一个元素节点，视情况使用 nextSibling 或 nextElement

Sibling。

```
function getNext(el) {
    if ("nextElementSibling" in el) {
        return el.nextElementSibling
    }
    while (el = el.nextSibling) {
        if (el.nodeType === 1) {
            return el
        }
    }
    return null;
}
```

兄长选择器：就是取其右边的所有同级元素节点。

```
function getPrev(el) {
    if ("previousElementSibling" in el) {
        return el.previousElementSibling;
    }
    while (el = el.previousSibling) {
        if (el.nodeType === 1) {
            return el;
        }
    }
    return null;
}
```

上面提到 childElementCount、nextElementSibling 是 2008 年 12 月通过 Element Traversal 规范的，用于遍历元素节点。加上后来补充的 parentElement，我们查找元素就非常方便，如表 6.1 所示。

表 6.1

	遍历所有子节点	遍历所有子元素
第一个	firstChild	firstElementChild
最后一个	lastChild	lastElementChild
前面的	previousSibling	previousElementSibling
后面的	nextSibling	nextElementSibling
父节点	parentNode	parentElement
数量	length	childElementCount

伪类是选择器中最庞大的家族，从 CSS1 开始支持，以字符串开头。在 CSS3，出现了要求传参的结构伪类与取反伪类。

1. 动作伪类

动作伪类又分为链接伪类和用户行为伪类，其中链接伪类由:visited 和:link 组成，用户行为伪类由:hover、:active 和:focus 组成。这里我们基本上只能模拟:link，而在浏览器原生的 querySelectorAll 对它们的支持也存在差异，IE8～IE10 取:link 存在错误，它只能取 A 标签，实际:link 是指代 A、

AREA、LINK 这三种标签，这个其他标签浏览器都正确。另外，除 Opera，Safari 外，其他浏览器取:focus 都正常，除 Opera 外，其他浏览器取:hover 都正确。剩下的:active 与:visited 都为零。下面是测试页面。

```
<!DOCTYPE HTML>
<html>
    <head>
        <title></title>
        <link href="aa" type="text/css" rel="stylesheet" charset="utf-8" />
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <script>
            window.onload = function() {
                document.querySelector("#aaa").onclick = function() {
                    alert(document.querySelectorAll(":focus").length);//1
                }
                document.querySelector("#bbb").onmouseover = function() {
                    //4 html, body, p, a
                    alert(document.querySelectorAll(":hover").length);
                }
            }

            function test() {
                alert(document.querySelectorAll(":link").length);// 6
            }
        </script>
    </head>
    <body>
        <p><a href="javascript:void 0" id="aaa">aaa</a></p>
        <p><a href="javascript:void 0" id="bbb">bbb</a></p>
        <button type="button" onclick="test()">点我</button>
        <img src="planets.jpg" border="0" usemap="#planetmap" alt="Planets" />
        <map name="planetmap" id="planetmap">
            <area shape="circle" coords="180,139,14" href ="venus.html" alt="Venus" />
            <area shape="circle" coords="129,161,10" href ="mercur.html" alt="Mercury" />
            <area shape="rect" coords="0,0,110,260" href ="sun.html" alt="Sun" />
        </map>
    </body>
</html>
```

伪类没有专门的 API 得到结果集，因此我们需要通过上一次得到的结果集进行过滤。在浏览器中，我们可以通过 document.links 得到部分结果，因为它不包含 LINK 标签。因此，最好的方法是判定它的 tagName 是否等于 A、AREA、LINK 的其中一个。

2. 目标伪类

目标伪类即:target 伪类，指其 id 或者 name 属性与 URL 中的 hash 部分（即#之后的部分）匹配上的元素。

譬如文档中有一个元素，其 id 为 section_2，而 URL 中的 hash 部分也是#section_2，那么它就是我们要取的元素。

Sizzle 中的过滤函数如下。

```
"target": function(elem) {
    varhash = window.location && window.location.hash;
    return hash && hash.slice(1) === elem.id;
},
```

3. 语言伪类

语言伪类即:lang 伪类，用来设置使用特殊语言的内容样式，如:lang(de)的内部应该为德语，需要特殊处理。

注意：lang 虽然作为 DOM 元素的一个属性，但:lang 伪类与属性选择器有所不同，具体表现在:lang 伪类具有“继承性”，如下面 HTML 表示的文档。

```
<body lang="de"><p>一个段落</p></body>
```

如果使用[lang=de]则只能选择到 body 元素，因为 p 元素没有 lang 属性。但是使用:lang(de)则可以同时选择到 body 和 p 元素，表现出继承性。

Sizzle 中的过滤函数如下。

```
"lang": markFunction(function(lang) {
    // lang value must be a valid identifider
    if (!ridentifier.test(lang || "")) {
        Sizzle.error("unsupported lang: " + lang);
    }
    lang = lang.replace(runescape, funescape).toLowerCase();
    return function(elem) {
        var elemLang;
        do {
            if ((elemLang = documentIsXML ? elem.getAttribute("xml:lang") || elem.getAttri
bute("lang") : elem.lang)) {

                elemLang = elemLang.toLowerCase();
                return elemLang === lang || elemLang.indexOf(lang + "-") === 0;
            }
        } while ((elem = elem.parentNode) && elem.nodeType === 1);
        return false;
    };
}),
```

对比 mass 的实现如下。

```
lang: { //标准 CSS3 语言伪类
    exec: function(flags, elems, arg) {
        var result = [],
            reg = new RegExp("^" + arg, "i"),
            flag_not = flags.not;
        for (var i = 0, ri = 0, elem; elem = elems[i++];) {
            var tmp = elem;
            while (tmp && !tmp.getAttribute("lang"))
            tmp = tmp.parentNode;
            tmp = !! (tmp && reg.test(tmp.getAttribute("lang")));
            if (tmp ^ flag_not) result[ri++] = elem;
        }
```

```
        return result;
    }
},
```

4. 状态伪类

状态伪类用于标记一个 UI 元素的当前状态，由:checked、:enabled、:disabled 和:indeterminate 这 4 个伪类组成。我们可以分别通过元素的 checked、disabled、indeterminate 属性进行判定。

5. 结构伪类

它又可以分为三种，**根伪类**、**子元素过滤伪类**与**空伪类**。根伪类是由它在文档的位置判定，子元素过滤伪类是根据它在其父亲的所有孩子的位置或标签类型判定，空伪类是根据它孩子的个数判定。

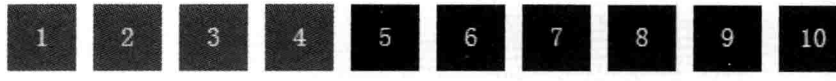
:root 伪类用于选取根元素，在 HTML 文档中通常是 html 元素。

:nth-child 是所有子元素过滤伪类的蓝本，其他 8 种都是由它衍生出来的。它带有参数，可以是纯数字、代数式或单词。如果是纯数字，数字是从 1 计起；如果是代数式，*n* 则从零递增，非常不好理解的规则。下面是示例。

:nth-child(2)选取当前父节点的第 2 个子元素。

:nth-child(n+4)选取大于等于 4 标签，我们可以把 *n* 看成自变量（0 ≤ *n* ≤parent.children.length），此代数式的值为因变量。

:nth-child(–n+4)选取小于等于 4 标签。



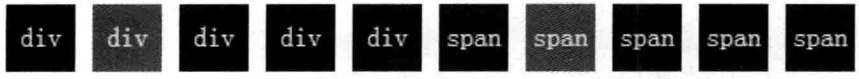
:nth-child(2n)选取偶数标签，2*n* 也可以是 even。

:nth-child(2n–1)选取奇数标签，2*n*–1 可以是 odd。

:nth-child(3n+1)表示每三个为一组，取它的第一个。

:nth-last-child 与:nth-child 差不多，不过是从后面取起。比如:nth-last-child(2)。

:nth-of-type 和:nth-last-of-type 与:nth-child 和:nth-last-child 类似，规则是将当前元素的父节点的所有元素按照其 tagName 分组，只要其参数符合它在那一组的位置就被匹配到。比如：nth-of-type(2)，具体如下。



另一个例子，:nth-of-type(even)，具体如下。

:first-child 用于选择第一个子元素，效果等同于:nth-child(1)。

:last-child 用于选择最后一个子元素，效果等同于:nth-last-child(1)。

:first-of-type 和:last-of-type 等同于:nth-of-type(1)和:nth-last-of-type(1)。

:only-child 用于选择唯一的子元素，当子元素个数超过 1 个时，选择器失效。

:only-of-type 将父节点的子元素按 tagName 分组，如果某一组只有一个元素，那么就选择这些组的元素返回。

:empty 用于选择那些不包含任何元素节点、文本节点、CDATA 节点的元素，但允许里面只存在注释节点。

Sizzle 中的实现如下。

```
"empty": function(elem) {
    for (elem = elem.firstChild; elem; elem = elem.nextSibling) {
        if (elem.nodeName > "@" || elem.nodeType === 3 || elem.nodeType === 4) {
        return false;
        }
    }
    return true;
},
```

mootools 中 Slick.的实现如下。

```
"empty": function(node) {
    var child = node.firstChild;
    return !(child && child.nodeType == 1) && !(node.innerText || node.textContent ||
'').length;
},
```

6. 取反伪类

取反伪类即:not 伪类，其参数为一个或多简单选择器，里面用逗号隔开。在 jQuery 等选择器引擎中允许你传入其他类型的选择器，甚至可以进行多个取反伪类套嵌。

7. 引擎在实现时涉及的概念

种子集：或者叫候选集，如果 CSS 选择符非常复杂，我们要分几步才能得到我们想要的元素。那么第一次得到的元素集合就叫种子集。在 Sizzle 这样基本从右到左，它的种子集中就有一部分为我们最后得到的元素。如果选择器引擎是从左到右选择器，那么它们只是我们继续查它们的孩子或兄弟的“据点”而已。

结果集：选择器引擎最终返回的元素集合，现在约定俗成，它要保持与 querySelectorAll 得到的结果一致，即，没有重复元素，元素要按照它们在 DOM 树上出现的顺序排序。

过滤集：我们选取一组元素后，它之后的每一个步骤要处理的元素集合都可以称为过滤集。比如 p.aaa，如果浏览器不支持 querySelectorAll，若支持 getElementsByClassName，那么我们就用它得到种子集，然后在循环中通过 tagName==="P"进行过滤。若不支持，只能通过 getElementsByTagName 得到种子集，然后通过 className 进行过滤。显然大多数情况下，前者比后者快多了。同理，如果它们之间存在 ID 选择器，由于 ID 在一个文档中不允许重复，因此使用 ID 进行查找更快。在 Sizzle，如果不支持 querySelectorAll，它会智能地以 ID、Class、Tag 的顺序进行查找。

选择器群组：一个选择符被并联选择器“,”划分成的每一个大分组。

选择器组：一个选择器群组被关系选择器划分的第一个小分组。考虑到性能，每一个小分组建议都加上 tagName，因为这样在 IE6 会方便我们使用 getElementsByTagName。比如 p div.aaa 比 p.aaa 快多了。前者是通过两次 getElementsByTagName 寻找，最后用 className 过滤，后者是 getElementsByTagName 得到种子集，然后再取它们的所有子孙，明显这样得到的过滤集比前者的数量多很多。

从实现上说，你可以选择从左到右，也可以像 Sizzle 那样从右到左，但 Sizzle 只能说大体上是这个方向，实际情况复杂多了。

另外，选择器也分为编译型与非编译型，编译型是 EXT 发明的，这个阵营的选择器中有 EXT、QWrap、NWMatchers、JindoJS[①]。非编译型的就更多了，如 Sizzle、Icarus（mass Framework 的选择器引擎）、Slick(mootools 的选择器)、YUI、dojo、uupaa、peppy……

还有一种利用 xpath 实现的选择器，最著名的是 Base2。它先实现了 xpath 那一套，方便 IE 也能使用 document. evaluate，然后将 CSS 选择符翻译成 xpath。其他比较出名的如 casperjs、DOMAssistant。

像 Sizzle、mootools、Icarus 等还支持选择 XML 元素，因为 XML 还是一种重要的数据传输格式，后端通过 XHR 返回我们的可能就是 XML，这样我们通过选择器引擎抽取所需要的数据就简单多了。

① 韩国人写的框架，http://jindo.dev.naver.com

6.4 选择器引擎涉及的通用函数

6.4.1 isXML

最强大的前几名选择器引擎都能操作 XML 文档，但 XML 与 HTML 存在很大的差异，没有 className，getElementById，并且 nodeName 是区分大小写，在旧版本 IE 中还不能直接给 XML 元素添加自定义属性。因此这区分非常有必要。我们看一下各大引擎的实现吧。

Sizzle 的实现如下。

```
var isXML = Sizzle.isXML = function( elem ) {
    var documentElement = elem && (elem.ownerDocument || elem).documentElement;
    return documentElement ? documentElement.nodeName !== "HTML" : false;
};
```

但这样不严谨，因为 XML 的根节点可能是 HTML 标签，比如这样创建一个 XML 文档：

```
try {
    var doc = document.implementation.createDocument(null, 'HTML', null);
    alert(doc.documentElement)
    alert(isXML(doc))
} catch (e) {
    alert("不支持 creatDocument 方法")
}
```

我们来看 mootools 的 slick 的实现。

```
var isXML = function(document) {
    return ( !! document.xmlVersion) || ( !! document.xml) || (toString.call(document) ==
'[object XMLDocument]') || (document.nodeType == 9 && document.documentElement.nodeName !=
'HTML');
};
```

mootools 用到了大量属性来进行判定，从 mootools1.2 到现在还没什么改动，说明还是很可靠的。我们再精简一下。在标准浏览器中，暴露了一个创建 HTML 文档的构造器 HTMLDocument，而 IE 下的 XML 元素又拥有 selectNodes。

```
var isXML = window.HTMLDocument ? function(doc) {
    return !(doc instanceof HTMLDocument)
} : function(doc) {
    return "selectNodes" in doc
}
```

不过这些方法都只是规范，JavaScript 对象可以随意添加，属性法很容易被攻破，最好是使用功能法。无论 XML 或 HTML 文档都支持 createElement 方法，我们判定创建了元素的 nodeName 是否区分大小写。

```
var isXML = function(doc) {
    return doc.createElement("p").nodeName !== doc.createElement("P").nodeName;
};
```

这是我当前能给出的最严谨的函数了。

6.4.2 contains

contains 方法就是判定参数 1 是否包含参数 2。这通常用于优化。比如在早期的 Sizzle，对于#aaa p.class 这个选择符，它会优先用 getElementsByClassName 或 getElementsByTagName 取种子集，然后就不继续往左走了，直接跑到最左的#aaa，取得#aaa 元素，然后通过 contains 方法进行过滤。随着 Sizzle 的体积进行增大，它现在只剩下另一个关于 ID 的用法，即，如果上下文对象非文档对象，那么它会取得其 ownerDocument，这样就可以用 getElementById，然后利用 contains 方法进行验证！

```
//Sizzle 1.10.15
if (context.ownerDocument && (elem = context.ownerDocument.getElementById(m)) &&
contains(context, elem) && elem.id === m) {
    results.push(elem);
    return results;
}
```

我们再看看 contains 的实现。

```
//Sizzle 1.10.15
var rnative = /^[^{]+\{\s*\[native \w/,
hasCompare = rnative.test( docElem.compareDocumentPosition ),
contains = hasCompare || rnative.test(docElem.contains) ?
        function(a, b) {
            var adown = a.nodeType === 9 ? a.documentElement : a,
                bup = b && b.parentNode;
            return a === bup || !!(bup && bup.nodeType === 1 && (
                adown.contains ?
                adown.contains(bup) :
                a.compareDocumentPosition && a.compareDocumentPosition(bup) & 16
                ));
        } :
        function(a, b) {
            if (b) {
                while ((b = b.parentNode)) {
                    if (b === a) {
                        return true;
                    }
                }
            }
            return false;
        };
```

它自己做了预判定，但这时如果传入 XML 元素节点，可能就会出错。因此建议改成实时判定，虽然每次进入都判定一次使用哪个原生 API。下面是 mass Framework 的实现。

```
contains = function(a, b, same) {
    // 第一个节点是否包含第二个节点，same 允许两者相等
    if (a === b) {
        return !!same;
    }
    if(!b.parentNode)
```

```
        return false;
    if (a.contains) {
        return a.contains(b);
    } else if (a.compareDocumentPosition) {
        return !!(a.compareDocumentPosition(b) & 16);
    }
    while ((b = b.parentNode))
      if (a === b) return true;
    return false;
}
```

现在来解释一下 contans 与 compareDocumentPosition 这两个 API。contains 原来是 IE 的私有实现，后来其他浏览器也借鉴这方法，如 Firefox 在 9.0 也装了此方法。它是一个元素节点的方法，如果另一个等于或包含于它的内部，就返回 true。compareDocumentPosition 是 DOM Level 3 specification 定义的方法，Firefox 等标准浏览器都支持，它用于判定两个节点的关系，而不单只是包含关系。这里是从 NodeA.compareDocumentPosition(NodeB) 返回的结果，包含你可以得到的信息，如表 6.2 所示。

表 6.2

Bits		
000000	0	元素一致
000001	1	节点在不同的文档（或者一个在文档之外）
000010	2	节点 B 在节点 A 之前
000100	4	节点 A 在节点 B 之前
001000	8	节点 B 包含节点 A
010000	16	节点 A 包含节点 B
100000	32	浏览器的私有使用

有时候，两个元素的位置关系可能连续满足上表的两种情况，比如 A 包含 B，并且 A 在 B 的前面，那么 compareDocumentPosition 就返回 20。

由于旧版本 IE 不支持 compareDocumentPosition，因此 jQuery 的作者 John Resig 写了个兼容函数，用到 IE 的另一个私有实现 sourceIndex。sourceIndex 会根据元素的位置从上到下，从左到右依次加 1，比如 HTML 标签的 sourceIndex 为 0，HEAD 标签的为 1，BODY 标签为 2，HEAD 的第一个子元素为 3……如果元素不在 DOM 树，那么返回–1。

```
// Compare Position - MIT Licensed, John Resig
function comparePosition(a, b) {
    return a.compareDocumentPosition ? a.compareDocumentPosition(b) :
     a.contains ? (a != b && a.contains(b) && 16)+
      (a != b && b.contains(a) && 8)+
      (a.sourceIndex >= 0 && b.sourceIndex >= 0 ?
          (a.sourceIndex < b.sourceIndex && 4)+
          (a.sourceIndex > b.sourceIndex && 2): 1): 0;
}
```

6.4.3 节点排序与去重

为了让选择器引擎搜到的结果集尽可能接近原生 API 的结果（因为在最新的浏览器中，我们可能只使用 querySelectorAll 实现），我们需要让元素节点按它们在 DOM 树出现的顺序排序。

在 IE 及 Opera 早期的版本，我们可以使用 sourceIndex 进行排序。

标准浏览器可以使用 compareDocumentPosition，上面不是介绍它可以判定两个节点的位置关系吗？我们只要将它们的结果按位与 4 不等于 0 就知道其前后顺序了。

此外，标准浏览器的 Range 对象有一个 compareBoundaryPoints 方法，它也能迅速得到两个元素的前后顺序。

```
var compare = comparerange.compareBoundaryPoints(how, sourceRange);
```

compare：返回 1, 0, –1（0 为相等，1 为 comparerange 在 sourceRange 之后，–1 为 comparerange 在 sourceRange 之前）。

how：比较哪些边界点，为常数。

- Range.START_TO_START——比较两个 Range 节点的开始点。
- Range.END_TO_END——比较两个 Range 节点的结束点。
- Range.START_TO_END——用 sourceRange 的开始点与当前范围的结束点比较。
- Range.END_TO_START——用 sourceRange 的结束点与当前范围的开始点比较。

特别的情况发生于要兼容旧版本标准浏览器与 XML 文档时，这时只有一些很基础的 DOM API，我们需要使用 nextSibling 来判定谁是“哥哥”，谁是“弟弟”。如果它们不是同一个父节点，我们就需要将问题转化为求最近公共祖先，判定谁是“父亲”，谁是“伯父”。到这里，已经是纯算法的问题了。实现的思路有许多，最直观也最笨的做法是，不断向上获取它们的父节点，直到 HTML 元素，连同最初的那个节点，组成两个数组，然后每次取数组最后的元素进行比较，如果相同就去掉，一直去掉到不相同为止，最后用 nextSibling 结束。下面是测试代码，自己找一个 HTML 页面做标本。

```
window.onload = function() {
    function shuffle(a) {
        //洗牌
        var array = a.concat();
        var i = array.length;
        while (i) {
            var j = Math.floor(Math.random() * i);
            var t = array[--i];
            array[i] = array[j];
            array[j] = t;
        }
        return array;
    }
    var log = function(s) {
        //查看调试消息
        window.console && window.console.log(s)
    }
    var sliceNodes = function(arr) {
        //将 NodeList 转换为纯数组
```

```
    var ret = [],
            i = arr.length;
    while (i)
        ret[--i] = arr[i];
    return ret;
}

var sortNodes = function(a, b) {
    //节点排序
    var p = "parentNode",
            ap = a[p],
            bp = b[p];
    if (a === b) {
        return 0
    } else if (ap === bp) {
        while (a = a.nextSibling) {
            if (a === b) {
                return -1
            }
        }
        return 1
    } else if (!ap) {
        return -1
    } else if (!bp) {
        return 1
    }
    var al = [],
            ap = a
    //不断往上取，一直取到 HTML
    while (ap && ap.nodeType === 1) {
        al[al.length] = ap
        ap = ap[p]
    }
    var bl = [],
            bp = b;
    while (bp && bp.nodeType === 1) {
        bl[bl.length] = bp
        bp = bp[p]
    }
    //然后逐一去掉公共祖先
    ap = al.pop();
    bp = bl.pop();
    while (ap === bp) {
        ap = al.pop();
        bp = bl.pop();
    }
    if (ap && bp) {
        while (ap = ap.nextSibling) {
            if (ap === bp) {
                return -1
            }
        }
        return 1
    }
    return ap ? 1 : -1
```

```
    }

    var els = document.getElementsByTagName("*")
    els = sliceNodes(els);          //转换成纯数组
    log(els);
    els = shuffle(els);             //洗牌（模拟选择器引擎最初得到的结果集的情况）
    log(els);
    els = els.sort(sortNodes); //进行节点排序
    log(els);
}
```

此外，我们还有一种方法，就是选择结束后，用 document.getElementsByTagName(“*”)得到所有元素节点，这时它们肯定是排好序的。我们再依次为它们添加一个类似 sourceIndex 的自定义属性，值为它的索引值，接下来怎么做就无需我多言了。

好了，我们暂且放下，看一下各大浏览器的实现。

Mootools 的 Slick 引擎，它的比较函数已经注明是来自 Sizzle 的（准确来说 Sizzle1.6 左右，现在它也被改得面目全非）。

```
features.documentSorter = (root.compareDocumentPosition) ? function(a, b) {
    if (!a.compareDocumentPosition || !b.compareDocumentPosition)
        return 0;
    return a.compareDocumentPosition(b) & 4 ? -1 : a === b ? 0 : 1;
} : ('sourceIndex' in root) ? function(a, b) {
    if (!a.sourceIndex || !b.sourceIndex)
        return 0;
    return a.sourceIndex - b.sourceIndex;
} : (document.createRange) ? function(a, b) {
    if (!a.ownerDocument || !b.ownerDocument)
        return 0;
    var aRange = a.ownerDocument.createRange(),
        bRange = b.ownerDocument.createRange();
    aRange.setStart(a, 0);
    aRange.setEnd(a, 0);
    bRange.setStart(b, 0);
    bRange.setEnd(b, 0);
    return aRange.compareBoundaryPoints(Range.START_TO_END, bRange);
} : null;
```

它没有打算支持 XML 与旧版本标准浏览器，不支持就不排序。

mass Framework 的 Icarus 引擎，它结合了一位编程高手 JK 给出的算法，在排序去重上远胜 Sizzle。突破点在于，无论 Sizzle 或者 Slick，它们都是通过传入比较函数进行排序。而数组的原生 sort 方法，当它传一个比较函数时，不管它内部用哪种排序算法[①]，都需要多次比对，所以非常耗时。如果能设计让排序在不传参的情况进行，那么速度就会提高 N 倍！

① ecma 没有规定 sort 的具体实现，因此浏览器各显神器，比如 Firefox2 使用堆排序，Firefox3 使用归并排序，IE 速度较慢，具体算法不明，可能为冒泡或者插入排序，而 Chrome 则为了最大效率，采用了两种算法：
chrome use quick sort for a large dataset (>22) ,and for a smaller dataset (<=22) chrome use insert sort but modified to use binary search to find insert point ,so traditionally insert sort is stable ,but chrome make it faster and unstable , in conclusion chrome’ s sort is quick and unstable .
资料来自：http://yiminghe.iteye.com/blog/469713

下面是具体思路（当然只能用于 IE 或早期 Opera）。

（1）取出元素节点的 sourceIndex 值，转换成一个 String 对象；

（2）将元素节点附在 String 对象上；

（3）用 String 对象组成数组；

（4）用原生的 sort 进 String 对象数组排序；

（5）在排好序的 String 数组中，按序取出元素节点，即可得到排好序的结果集。

```
function unique(nodes) {
    if (nodes.length < 2) {
        return nodes;
    }
    var result = [],
            array = [],
            uniqResult = {},
            node = nodes[0],
            index, ri = 0,
            sourceIndex = typeof node.sourceIndex === "number",
            compare = typeof node.compareDocumentPosition !== "function";
//如果支持 sourceIndex，我们将使用更为高效的节点排序
//http://www.cnblogs.com/jkisjk/archive/2011/01/28/array_quickly_sortby.html

    if (!sourceIndex && !compare) { //用于旧版本 IE 的 XML
        var all = (node.ownerDocument || node).geElementsByTagName("*");
        for (var index = 0; node = all[index]; index++) {
            node.setAttribute("sourceIndex", index);
        }
        sourceIndex = true;
    }
    if (sourceIndex) { //IE opera
        for (var i = 0, n = nodes.length; i < n; i++) {
            node = nodes[i];
            index = (node.sourceIndex || node.getAttribute("sourceIndex")) + 1e8;
            if (!uniqResult[index]) {      //去重
                (array[ri++] = new String(index))._ = node;
                uniqResult[index] = 1;
            }
        }
        array.sort();                      //排序
        while (ri)
            result[--ri] = array[ri]._;
        return result;
    } else {
        nodes.sort(sortOrder);             //排序
        if (sortOrder.hasDuplicate) {      //去重
            for (i = 1; i < nodes.length; i++) {
                if (nodes[i] === nodes[i - 1]) {
                    nodes.splice(i--, 1);
                }
            }
        }
        sortOrder.hasDuplicate = false;    //还原
```

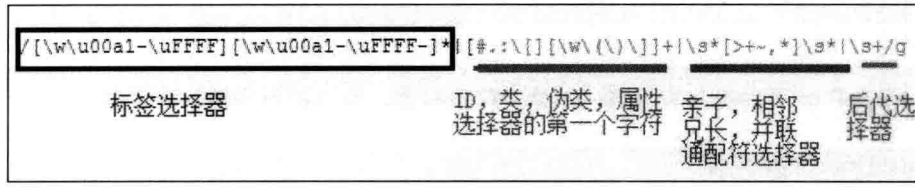
```
        return nodes;
    }
}


function sortOrder(a, b) {
    if (a === b) {
        sortOrder.hasDuplicate = true;
        return 0;
    } //现在标准浏览器的 HTML 与 XML 好像都支持 compareDocumentPosition
    if (!a.compareDocumentPosition || !b.compareDocumentPosition) {
        return a.compareDocumentPosition ? -1 : 1;
    }
    return a.compareDocumentPosition(b) & 4 ? -1 : 1;
}
```

6.4.4 切割器

选择器降低了 JavaScript 的入行门槛，它们在选择元素时都很随意，随心所欲，一级级地往上加 ID 类名，导致选择符非常长。因此如果不支持 querySelectorAll，没有一个原生 API 能承担这工作。因此我们通常使用正常对用户的选择符进行切割。这个步骤有点像编译原理的词法分析，拆分出有用的符号法出来。

这里就拿 Icarus 的切割器来举例，看它是怎么一步步进化，就知道这工作需要多少细致。

最早的版本如图 6.2 所示。



▲图 6.2

比如，对于“.td1,div a,body”，上面的正则可以完美将它分解为如下数组。

```
[".td1",",","div"," ", "*", ",", "body"]
```

然后我们就可以根据这个符号流进行工作。由于没有指定上下文对象，就从 document 开始，发现第一个是类选择器，可以用 getElementsByClassName，如果没有原生的，我们仿造一个也不是难事。然后是并联选择器，将上面得到的元素放进结果集。接着是标签选择器，使用 getElementsByTagName。接着发现是后代选择器，这里可以优化，我们可以预先查看下一个选择器群组是什么，发现是通配符选择器，因此继续使用 getElementsByTagName。接着又是并联选择器，将上面结果放入结果集。最后一个是标签选择器，又使用 getElementsByTagName。最后是去重排序。

显然有了切割好的符号，工作简单多了。

但没有东西一开始就是完美的，比如我们遇到这样一个选择符：":nth-child(2n+1)"。这是一个

单独的子元素过滤伪类，它不应该在这里被分析。后面有专门的正则对它的伪类名与传参进行处理。在切割器里，它能得到的最小词素是选择器！

于是切割器改进如下。

```
//让小括号里面的东西不被切割
var reg = /[\w\u00a1-\uFFFF][\w\u00a1-\uFFFF-]*|[#.:\[](?:[\w\u00a1-
\uFFFF-]|\([^\)]*\)|\])+|(?:\s*)[>+~,*](?:\s*)|\s+/g
```

我们不断增加测试样例，就会发现越来越多问题。又如这样一个选择符：".td1[aa='>111']"，属性选择器被拆碎了！

```
[".td1","[aa",">","111"]
```

正则改进如下。

```
//确保属性选择器作为一个完整的词素
var reg = /[\w\u00a1-\uFFFF][\w\u00a1-\uFFFF-]*|[#.:](?:[\w\u00a1-\uFFFF-
]|\S*\([^\)]*\))+|\[[^\]]*\]|(?:\s*)[>+~,*](?:\s*)|\s+/g
```

对于符择符“td + div span”，如果最后有一大堆空白，会导致解析错误，我们确保后代选择器夹在两个选择器之间。

```
["td", "+", "div"," ","span", " "]
```

最后一个选择器会被我们的引擎认作是后代选择器，需要提前去掉。

```
//缩小后代选择器的范围
var reg = /[\w\u00a1-\uFFFF][\w\u00a1-\uFFFF-]*|[#.:](?:[\w\u00a1-\uFFFF-
]|\S+\([^\)]*\))+|\[[^\]]*\]|(?:\s*)[>+~,*](?:\s*)|\s(?=[\w\u00a1-
\uFFFF*#.[:])/g
```

如果我们也想把最前面的空白去掉，那可能不是单独一个正则能做到的。现在切割器已经被我们搞得相当复杂了，维护性很差。在 mootools 等引擎中，里面的正则表达式更加复杂，可能是用工具生成的。到了这个地步，我们就需要转换思路，将切割器改为一个函数处理。当然，它里面也少不了正则表达式。正则是处理字符串的利器。

```
var reg_split =
/^[\w\u00a1-\uFFFF\-\*]+|[#.:][\w\u00a1-\uFFFF-]+(?:\([^\])*\))?|\[[^\]]*\]|(?:\s*)
[>+~,](?:\s*)|\s(?=[\w\u00a1-\uFFFF*#.[:])|^\s+/;
var slim = /\s+|\s*[>+~,*]\s*$/
function spliter(expr) {
    var flag_break = false;
    var full = [];//这里放置切割单个选择器群组得到的词素，以“,”为界
```

```
    var parts = [];//这里放置切割单个选择器组得到的词素，以关系选择器为界
    do {
        expr = expr.replace(reg_split, function(part) {
            if (part === ",") {//这个切割器只处理到第一个并联选择器
                flag_break = true;
            } else {
                if (part.match(slim)) {//对关系并联。通配符选择器两边的空白进行处理
                    //对 parts 进行反转，因为 div.aaa，反转后先处理.aaa
                    full = full.concat(parts.reverse(), part.replace(/\s/g, ''));
                    parts = [];
                } else {
                    parts[parts.length] = part;
                }
            }
            return "";//去掉已经处理了的部分
        });
        if (flag_break)
            break;
    } while (expr)
    full = full.concat(parts.reverse());
    !full[0] && full.shift();//去掉开头第一个空白
    return full;
}
var expr = "  div  >  div#aaa,span"
console.log(spliter(expr));//["div",">","#aaa", "div"]
```

当然，这个相对于 Sizzle1.8 与 Slick 等引擎的切割器来说，不值一提。写好一个切割器，需要有非常深厚的正则表达式功力。深层的知识包括自动机理论。

6.4.5 属性选择器对于空白字符的匹配策略

已经介绍过属性选择器的七种形态了，但属性选择器并没有这么简单。在 W3C 草案对属性选择器[att~=val]提到一个点，val 不能为空白字符，否则比较值 flag（flag 为 val 与元素实际值的比较结果）总返回 false。如果用 querySelectorAll 测试一下属性其他形态，我们会得到更多类似结果。

```
<!DOCTYPE html>
<html>
    <head>
        <title>属性选择器</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <script>
            window.onload =function(){
                console.log(document.querySelector("#test1[title='']"));
                console.log(document.querySelector("#test1[title~='']"));
                console.log(document.querySelector("#test1[title|='']"));
                console.log(document.querySelector("#test1[title^='']"));
                console.log(document.querySelector("#test1[title$='']"));
                console.log(document.querySelector("#test1[title*='']"));
                console.log("==========================================")
                console.log(document.querySelector("#test2[title='']"));
                console.log(document.querySelector("#test2[title~='']"));
                console.log(document.querySelector("#test2[title|='']"));
```

```
            console.log(document.querySelector("#test2[title^='']"));
            console.log(document.querySelector("#test2[title$='']"));
            console.log(document.querySelector("#test2[title*='']"));
        }
    </script>
</head>
<body>
    <div title="" id="test1"></div>
    <div title="aaa" id="test2"></div>
</body>
</html>
```

运行结果如下。

```
日志: [object HTMLDivElement]
日志:null
日志: [object HTMLDivElement]
日志:null
日志:null
日志:null
日志: ========================================
日志:null
日志:null
日志:null
日志:null
日志:null
日志:null
```

换言之，只要 val 为空，除=或|=外，flag 必为 false，并且，对于非=，!=操作符，如果取得值为空白字符，flag 也必为 false。

```
for (i = 0, ri = 0, elem; elem = elems[i++];) {
    if(!op){
        flag = Icarus.hasAttribute(elem,name,flag_xml);//[title]
    }else if(val ==="" && op > 3){
        flag =false
    }else{
        attr = Icarus.getAttribute(elem,name,flag_xml);
        switch (op) {
            case 1:// = 属性值全等于给出值
                flag = attr === val;
                break;
            case 2://!= 非标准，属性值不等于给出值
                flag = attr !== val;
                break;
            case 3://|= 属性值以"-"分割成两部分，给出值等于其中一部分，或全等于属性值
                flag = attr === val || attr.substring(0, val.length + 1) === val +"-";
                break;
            case 4://~= 属性值为多个单词，给出值为其中一个
                flag = attr !=="" && (" " + attr +" ").indexOf(val) >= 0;
                break;
```

```
            case 5://^= 属性值以给出值开头
                flag = attr !=="" && attr.indexOf(val) === 0 ;
                break;
            case 6://$= 属性值以给出值结尾
                flag = attr !=="" && attr.lastIndexOf(val) + val.length === attr.length;
                break;
            case 7://*= 属性值包含给出值
                flag = attr !=="" && attr.indexOf(val) >= 0;
                break;
        }
    }
    if (flag ^ flag_not)
        tmp[ri++] = elem;
}
```

6.4.6 子元素过滤伪类的分解与匹配

子元素过滤伪类是CSS3新增的一种选择器。比较复杂，这里单独说一下。首先，我们要将它从选择符中分离出来。这个一般由切割器搞定。然后我们用正则将伪类名与它小括号里的传参分解出来。下面介绍一下Icarus的做法。

```
var expr = ":nth-child(2n+1)",
    rsequence = /^([#\.:]|\[\s*]((?:[-\w]|[^\x00-\xa0]|\\.)+)/,
    rpseudo = /^\(\s*("([^"]*)"|'([^']*)'|[^\(\)]*(\([^\(\)]*\))?)\s*\)/,
    rBackslash = /\\/g,
    //这里把伪类从选择符里分解出来
    match = expr.match(rsequence); //[":nth-child",":", "nth-child"]
    expr = RegExp.rightContext;//用它左边的部分重写 expr--> "(2n+1)""
      key = (match[2] || "").replace(rBackslash, "");//去掉换行符 key--> "nth-child"
switch (match[1]) {
    case "#":
        //ID选择器 略
        break;
    case ".":
        //类选择器 略
        break;
    case ":":
        //伪类 略
        tmp = Icarus.pseudoHooks[key];
        //Icarus.pseudoHooks 里面放置我们所有能处理的伪类
        if (match = expr.match(rpseudo)) {
            expr = RegExp.rightContext;//继续取它左边的部分重写 expr
            if ( !! ~key.indexOf("nth")) {//如果是子元素过滤伪类
                args = parseNth[match[1]] || parseNth(match[1]);//分解小括号的传参
            } else {
                args = match[3] || match[2] || match[1]
            }
        }
        break
    default:
        //属性选择器 略
```

```
        break;
    }
```

这里有个小技巧，我们需要不断把处理过的部分从选择符中去掉。一般的选择器引擎是使用 expr = expr.replace(reg, "")进行处理，Icarus 是巧妙地使用正则的 RegExp.rightContext 进行复写。将小括号里面的字符串取得后我们通过 parseNTH 进行加工。将数字 1，4，单词 even, odd, –n + 1 等各种形态转换 an+b 的形态。

```
function parseNth(expr) {
    var orig = expr
    expr = expr.replace(/^\+|\s*/g, ''); //清除无用的空白
    var match = (expr === "even" && "2n" || expr === "odd" && "2n+1" || !/\D/.test(expr)
&& "0n+" + expr || expr).match(/(-?)(\d*)n([-+]?\d*)/);
    return parse_nth[orig] = {
        a: (match[1] + (match[2] || 1)) - 0,
        b: match[3] - 0
    };
}
```

parseNth 是一个缓存函数，这样就能避免重复解析，提高引擎总体性能。

我们再来看一下 an+b 的匹配算法，a 与 b 都是整数，可能是正数、负数、零。最简单的情形是 a 为零，b 随意时。我们得到此元素的父节点下的所有子元素，即 els = el.parentNode.children，那么匹配算法浓缩成!!为 els[b-1]。因为子元素过滤伪类中的传参是以 1 开始，因此需要减 1。如果 a 为 1，b 为 0 的情况，这也很简单，意味着所有孩子都匹配，直接返回 true 就行。如果 b 为 0 时，比如 2n 就是取孩子中的索引值为偶数的个体，假设索引值为 index，那么(index + 1) % 2 == 0 就能匹配出来，由此推断出公式为(index + 1) % a == 0。再看 b 不为 0 的情况，比如:nth-child(3n+1)，一个拥有 10 个孩子的节点，它的索引值中第 0、3、6、9 的孩子被匹配，即(index + 1 – 1) % 3 == 0 时被匹配，因此匹配公式为(index + 1 – b) % a == 0。不过当 b > a 时，又有新状况，比如:nth-child(3n+4)，还是那个拥有十个孩子的节点，这时只有索引值为 3、6、9 的孩子被匹配，第一个孩子 (0 + 1 – 4) % 3 也是等于零，这时我们需要判定商是否大于负数了。由于 index 总是要加 1，那么我们一开始把起点弄成 1，那么就省事多了。总结上面的分析，我们把索引值与 b 的差作为一个变量 diff，那么匹配规则为 diff % a == 0 && diff / a >= 0。

我们再来看 nth-of-type，它其实就是对父元素的 child 进行分类，分类标签是元素的 tagName。然后只要目标元素的索引值（也是从 1 开始）匹配上面的公式就行了。

从性能上考量，并不是每种子元素过滤伪类都需要通过这种比较索引值的方式实现，我们可以变通一下。比如说，:first-child，我们看一下手头上的这个元素是否存在兄长，亦即 previous ElementSibling，存在就返回 false 过滤掉。:fast-child 也是基于相同的思路进行过滤。:only-child，需要判定目标元素是否为其父亲的唯一一个子元素，需要同时判定是否存在 previousElementSibling 与 nextElementSibling。当然，现实是残酷的，为了兼容旧版本 IE，我们一般用 previousSibling、nextSibling 加 nodeType 是否为 1 进行判定。

但对于:nth-child, :nth-last-child, :nth-of-type, :nth-last-of-type 就不行了，我们也不可能每匹配一个元素就把它在兄弟中或 tagName 分组中的索引值计算一遍。因此以空间换时间这万能药再次被

提出来了，我们将缓存仓库设置在父节点上，然后把它所有孩子的索引值或 tagName 分组一次性放在上面，就可以在一次查找过程中反映利用此缓存。至于如何实现可参看 Icarus、Sizzle1.8+、Slice 等引擎，代码比较长，在此不再贴出来。

6.5 Sizzle 引擎

jQuery 最大的特点是其选择器，jQuery1.3 时开始列装其 Sizzle 引擎。Sizzle 引擎与当时主流的引擎大不一样，人们说它是从右到左选择（虽然也不对，只能说大致方向如此），速度远胜当时的选择器。不过那时也没有什么选择器，因此一直是 Sizzle 在自娱自乐。

Sizzle 当时的几大特点如下。

（1）允许以关系选择器开头。

（2）允许取反选择器套取反选择器。

（3）大量的自定义伪类，比如位置伪类(:eq, :first, :even……)，内容伪类（:contains），包含伪类(:has)，标签伪类(:radio, :input, :text, :file……)，可见性伪类（:hidden,:visible）。

（4）对结果进行去重，以元素在 DOM 树的位置进行排序，这样与未来出现的 querySelector 行为一致。

显然，一下子搞出这么多东西，不是一朝半夕的事，说明 John Resig 已经研发了很久。当时 sizzle 的版本号为 0.9.1，代码风格与 jQuery 库大不一样，非常整齐清晰。这个风格一直延续到 jQuery1.7.2，Sizzle 版本号也跟上为 1.7.2。在 jQuery1.8 或者说 Sizzle1.8，它风格大变，首选里面的正则是通过编译得到的，以求更加准确，结构也异常复杂，开始走 EXT 那样的编译函数的路子，以求通过多种缓存手段提高查找速度和匹配速度。

由于 Sizzle 第二阶段的蜕变还没有完成，每星期都在变，tokenize，addMatcher, matcherFrom Tokens，matcherFromGroupMatchers，compile 这些关键的内部函数每天都在改进，不断膨胀。我们还是来看看 Sizzle1.7.2 版吧，这个版本是 John Resig 第一个阶段最完美的思想结晶。

Sizzle 的整体结构如下。

（1）Sizzle 主函数，里面包含选择符的切割，内部循环调用主查找函数，主过滤函数，最后是去重过滤。

（2）其他辅助函数，如 uniqueSort、matches、matchesSelector。

（3）Sizzle.find 主查找函数。

（4）Sizzle.filter 主过滤函数。

（5）Sizzle.selectors 包含各种匹配用的正则、过滤用的正则、分解用过的正则、预处理函数、过滤函数。

（6）根据浏览器的特征设计 makeArray、sortOrder、contains 等方法。

（7）根据浏览器的特征重写 Sizzle.selectors 中的部分查找函数、过滤函数、查找次序。

（8）若浏览器支持 querySelectorAll，那么用它重写 Sizzle，将原来的 Sizzle 作为后备方案包裹在新 Sizzle 里面。

（9）其他辅助函数，如 isXML、posProcess。

下面用源代码的一部分解讲一下，我们可以在下面地址获取 Sizzle1.7.2.js。

```
https://github.com/jquery/sizzle/blob/1.7.2/sizzle.js
var Sizzle = function(selector, context, results, seed) {
//通过短路运算符,设置一些默认值
    results = results || [];
    context = context || document;
    //备份，因为 context 会被改写，如果出现并联选择器，就无法区别当前节点是对应哪一个 content

    var origContext = context;
//上下文对象必须是元素节点或文档对象
    if (context.nodeType !== 1 && context.nodeType !== 9) {
        return [];
    }
//选择符必须是字符串,且不能为空
    if (!selector || typeof selector !== "string") {
        return results;
    }

    var m, set, checkSet, extra, ret, cur, pop, i,
            prune = true,
            contextXML = Sizzle.isXML(context),
            parts = [],
            soFar = selector;
//下面是切割器的实现,每次只处理到并联选择器，extra 留给下次递归自身时作传参
//不过与其他引擎的实现不同的是，它没有一下子切成选择器，而是切成选择器组与关系选择器的集合
//比如 body div > div:not(.aaa),title
//将会得到 parts 数组：["body","div",">", "div:not(.aaa)"]
//后代选择器虽然被忽略了，但在循环这个数组时，它默认每两个选择器组中一定夹着关系选择器
//不存在就放在后代选择器到那个位置上
    do {
        chunker.exec(""); //这一步主要是将 chunker 的 lastIndex 重置，当然直接设置 chunker.
                          //lastIndex 效果也一样
        m = chunker.exec(soFar);
        if (m) {
            soFar = m[3];
            parts.push(m[1]);
            if (m[2]) { //如果存在并联选择器，就中断
                extra = m[3];
                break;
            }
        }
    } while (m);
//略……
}
```

接下来有许多分支，分别是对 ID 与位置伪类进行优化的，我们暂时跳过它们。看另外几个重要概念，查找函数，种子集，映射集。虽然有的之前介绍过，这里是结合 Sizzle 源码讲解的。

查找函数就是 Sizzle.selecters.find 下的几种函数，常规情况下有 ID、TAG、NAME 三个，如果浏览器支持 getElementsByClassName，还会有 Class 函数。正如我前面所介绍的那样，getElement

ById，getElementsByName，getElementsByTagName，getElementsByClassName 不能完全信任它们，即便是标准浏览器都会有 BUG，因此四大查找函数都做了一层薄薄的封装，不支持则返回 undefined，其他则返回数组或 NodeList。

种子集，或叫候选集，就是通过最右边的选择器组得到的元素集合，比如说"div.aaa span.bbb"，最右边的选择器组就是"span.bbb"，这时引擎会根据浏览器的支持情况选择 getElementsByTagName 或 getElementsClassName 得到一组元素，然后再通过 className 或 tagName 进行过滤，这时得到的集合就是种子集。Sizzle 的变量名 seed 就体现了这一点。

映射集，或叫影子集，Sizzle 源码的变量名为 checkSet。这是个怎样的东西呢？当我们取得种子集后，不动种子集，而是将种子集复制一份出来，这就是映射集。种子集是由一个选择器组选出来的，这时选择符不为空，必然往左就是关系选择器。关系选择器会让引擎去选取其兄长或父亲（具体操作见 Sizzle.selectors.relative 下的四大函数），把这些元素置换到候选集对等的位置上。然后到下一个选择器组时，就是纯过滤操作。主过滤函数 Sizzle.filter 会调用 Sizzle.seletors 下 N 个过滤函数对这些元素进行检测，将不符合的元素替换为 false。因此到最后要去重排序时，映射集是一个包含布尔值与元素节点的数组（true 值也在上面步骤中产生）。

让我们继续看源码。种子集是分两步筛选出来的。首先，通过 Sizzle.find 得到一个大体的结果。然后通过 Sizzle.filter，传入最右的那个选择器组剩余的部分作参数，缩少范围。

```
//这是针对最左边的选择器组存在 ID 做出的优化
ret = Sizzle.find(parts.shift(), context, contextXML);
context = ret.expr ? Sizzle.filter(ret.expr, ret.set)[0] : ret.set[0];

ret = seed ? {
    expr: parts.pop(),
    set: makeArray(seed)
    //这里会对~,+进行优化,直接取它的上一级做上下文
    //处理一个上下文对象胜过对付 N 个上下文
} : Sizzle.find(parts.pop(),parts.length === 1 &&
     (parts[0] === "~" || parts[0] === "+") && context.parentNode ?
    context.parentNode : context, contextXML);

set = ret.expr ? Sizzle.filter(ret.expr, ret.set) : ret.set;
```

我们是先取 span 还是取.aaa 呢？这里有个准则，确保我们后面的映射集最小化。直白地说，映射集里面的元素越少，那么调用过滤函数的次数就越少，调用函数的次数越少，说明进入另一个函数作用域所造成的能耗就越少，从而整体提高引擎的选择速度。为了达到此目的，这里做了个优化，原生选择器的调用顺序被放到一个叫 Sizzle.slectors.order 的数组中，对于陈旧的浏览器，其顺序为 ID、NAME、TAG，对于支持 getElementsByClassName 的浏览器，其顺序为 ID、CLASS、NAME、TAG。因为 ID 至多返回一个元素节点，className 与样式息息相关，不是每个元素都有这个类名，name 属性带来的限制可能比 className 更大，但用到的机率比较少，而 tagName 可排除的元素则更少了。那么 Sizzle.find 就会根据上面的数组，取得它的名字依次调用 Sizzle.leftMatch 下对应的正则，从最右的选择器组中切下需要的部分，将换行符处理掉，通过四大查找函数得到一个粗糙的节点集合。如果运气太霉，碰到如"[href=aaa]:visible"这样的选择符，那么只有把文档中的

所有节点作为结果返回。

```
Sizzle.find = function(expr, context, isXML) {
    var set, i, len, match, type, left;

    if (!expr) {
        return [];
    }

    for (i = 0, len = Expr.order.length; i < len; i++) {
        type = Expr.order[i];
        //取得正则,匹想出需要的 ID、CLASS、NAME、TAG
        if ((match = Expr.leftMatch[type].exec(expr))) {
            left = match[1];
            match.splice(1, 1);
            //处理换行符
            if (left.substr(left.length - 1) !== "\\") {
                match[1] = (match[1] || "").replace(rBackslash, "");
                set = Expr.find[type](match, context, isXML);
                //如果不为 undefined,那么去掉选择器组中用过的部分
                if (set != null) {
                    expr = expr.replace(Expr.match[type], "");
                    break;
                }
            }
        }
    }

    if (!set) { //没有,寻找该上下文对象的所有子孙
        set = typeof context.getElementsByTagName !== "undefined" ?
        context.getElementsByTagName("*") : [];
    }

    return {
        set: set,
        expr: expr
    };
};
```

经过主查找函数处理后，我们得到一个初步的结果，这时最右边的选择器组可能还有残余。比如"div span.aaa"可能余下"div span"，"div .aaa.bbb"可能余下"div .bbb"，这个转交主过滤函数 Sizzle.filter 函数处理。它有两种不同的功能，一是不断缩小集合的个数，构成种子集返回。另一种是将原集合中不匹配的元素置换为 false。这个根据它的第三个传参 inplace 而定。

```
Sizzle.filter = function(expr, set, inplace, not) {
    //用于生成种子集或映射集,这视第三个参数而定
    //expr: 选择符
    //set:  元素数组
    //inplace:undefined, null 时进入生成种子集模式,true 时进入映射集模式
    //not:  一个布尔值,来源自取反选择器
    var match, anyFound,
    type, found, item, filter, left,
```

```
i, pass,
old = expr,
    result = [],
    curLoop = set,
    isXMLFilter = set && set[0] && Sizzle.isXML(set[0]);

while (expr && set.length) {
    for (type in Expr.filter) { //ID,TAG,CLASS,TAG,CHILD,POS,PSEUDO
        if ((match = Expr.leftMatch[type].exec(expr)) != null && match[2]) {
            //切割出相应的字符串,作为传参放进 filter 里面
            filter = Expr.filter[type];
            left = match[1];
            //ID -->    ["#aaa","","aaa"]
            //CLASS --> [".aaa","","aaa"]
            //TAG-->    ["div","","div"]
            //ATTR-->   ["[aaa=ggg]", "", "aaa", "^=", undefined, undefined, "ggg"]
            //CHILD--> [":nth-child(even)", "", "nth-child", "", "even"]
            //POS-->    [":eq(2)", "", "eq", "2"]
            //PSEUDO-->[":not(.aaa)", "", "not", "", ".aaa"]
            anyFound = false;
            match.splice(1, 1);

            if (left.substr(left.length - 1) === "\\") {
                continue;
            }

            if (curLoop === result) {
                result = [];
            }

            if (Expr.preFilter[type]) {
                match = Expr.preFilter[type](match, curLoop, inplace, result, not,
                isXMLFilter);
                //这里会对传参进行加工,比如#aaa 得到 aaa, .bbb 得到 bbb,
                //出于优化需要,它会在 Expr.preFilter.CLASS 进行过滤,
                //而不用等于 Expr.filter.ClASS
                //另外,针对 CHILD, POS,它会两入进入这个循环,因为它们同时也能被
                //Expr.leftMatch.PSEUDO 匹配,但它不想被 Expr.filter.PSEUDO 处理
                //于是直接 continue
                if (!match) { //CLASS
                    anyFound = found = true;

                } else if (match === true) { //CHILD, POS
                    continue;
                }
            }

            if (match) {
                //curLoop 为一个映射集,里面包含 false, true
                for (i = 0;
                 (item = curLoop[i]) != null; i++) {
                    if (item) {
                        found = filter(item, match, i, curLoop);
```

```
                        pass = not ^ found;
                        //在映射集模式下,将不匹配的元素置换为 false
                        if (inplace && found != null) {
                            if (pass) {
                                anyFound = true;

                            } else {
                                curLoop[i] = false;
                            }
                            //否则 result 为我们的种子集,把匹配者放进去
                        } else if (pass) {
                            result.push(item);
                            anyFound = true;
                        }
                    }
                }
            }

            if (found !== undefined) {
                if (!inplace) {
                    curLoop = result; //重写种子集为 curLoop
                }
                //削减选择符直到变为空字符串
                expr = expr.replace(Expr.match[type], "");

                if (!anyFound) {
                    return [];
                }

                break;
            }
        }
    }

    // //如果到最后正则表达式也不能改动选择符，说明它有问题
    if (expr === old) {
        if (anyFound == null) {
            Sizzle.error(expr);

        } else {
            break;
        }
    }

    old = expr;
}

return curLoop;
};
```

待到我们把最右边的选择器组的最一个字符都去掉后，种子集宣告完成，然后处理下一个选择器组，并将种子集复制一下，生成映射集。在关系选择器 4 个对应函数——它们位 Sizzle. selectors. relative 命名空间下——只是将映射集里面的元素置换为它们的兄长父亲，个数是不变。因此映射

集与种子集的数量总是相当。另外，这 4 个函数内部也在调用 Sizzle.filter 函数，它的 inplace 参数为 true，走映射集的逻辑。

```
while (parts.length) {
    cur = parts.pop(); //取得关系选择器
    pop = cur;
    if (!Expr.relative[cur]) {
        cur = ""; //如果不是则默认为后代选择器
    } else {
        pop = parts.pop(); //取得后代选择器前面的子选择器群集
    }
    if (pop == null) {
        pop = context;
    }
    Expr.relative[cur](checkSet, pop, contextXML); //根据其他 4 种迭代器改变映射集里面的元素
    //得到诸如[ [object HTMLDivElement] ,false, false, [object HTMLSpanElement] ]的集合
}

    最后一步就是根据映射集甄选候选集。
for (i = 0; checkSet[i] != null; i++) {
    if (checkSet[i] && checkSet[i].nodeType === 1) {
        results.push(set[i]);
    }
}
```

如果存在并联选择器，那就再调用 Sizzle 主函数，把得到的两个结果合并去重。

```
if (extra) {
    Sizzle(extra, origContext, results, seed);
    Sizzle.uniqueSort(results);
}
```

这就是主流程。下面将是根据浏览器的特性进行优化或调整的部分。比如 IE6、IE7 下 getElementById 有 Bug，需要重写 Expr.find.ID 与 Expr.filterID。IE6～IE8 下，Array.prototype.slice.call 无法切割 NodeList，需要重写 makeArray。IE6～IE8，getElementsByTagName(“*”)会混杂注释节点，需要重写 Expr.find.TAG。如果浏览器支持 querySelectorAll，那么需要重写个 Sizzle。

```
if (document.querySelectorAll) {
    (function() {
        var oldSizzle = Sizzle,
            div = document.createElement("div"),
            id = "__sizzle__";

        div.innerHTML = "<p class='TEST'></p>";
        //Safari 在怪异模式下 querySelectorAll 不能工作,终止重写
        if (div.querySelectorAll && div.querySelectorAll(".TEST").length === 0) {
            return;
        }

        Sizzle = function(query, context, extra, seed) {
            context = context || document;
            // querySelectorAll 只能用于 HTML 文档,在标准浏览器的 XML 文档中
```

```
            //虽然也实现了此接口,但不能工作

            if (!seed && !Sizzle.isXML(context)) {
                // See if we find a selector to speed up
                var match = /^(\w+$)|^\.([\w\-]+$)|^#([\w\-]+$)/.exec(query);

                if (match && (context.nodeType === 1 || context.nodeType === 9)) {
                    // 优化只有单个标签选择器的情况
                    if (match[1]) {
                        return makeArray(context.getElementsByTagName(query), extra);
                        // 优化只有单个类选择器的情况

                    } else if (match[2] && Expr.find.CLASS && context.getElementsByClassName) {
                        return makeArray(context.getElementsByClassName(match[2]), extra);
                    }
                }

                if (context.nodeType === 9) {
                    // 优化只有选择符为 body 的情况
                    // 因为文档只有它一个标签,并且有对应属性直接取它
                    if (query === "body" && context.body) {
                        return makeArray([context.body], extra);
                        // 优化只有 ID 选择器的情况
                        // Speed-up: Sizzle("#ID")
                    } else if (match && match[3]) {
                        var elem = context.getElementById(match[3]);
                        //注意,浏览器也会优化过度,它会缓存了上次的结果
                        //即便它现在已经被移出 DOM 树(Blackberry 4.6)
                        if (elem && elem.parentNode) {
                            //IE 与 Opera 会混淆 ID 与 NAME,确保 ID 等于目标值
                            if (elem.id === match[3]) {
                                return makeArray([elem], extra);
                            }

                        } else {
                            return makeArray([], extra);
                        }
                    }

                    try {
                        return makeArray(context.querySelectorAll(query), extra);
                    } catch (qsaError) {}

            //IE8 的 querySelectorAll 实现存在 BUG，它会在包含自己的集合内查找符合自己的元素节点
                    //根据规范，应该是在当前上下文的所有子孙下找
                    //IE8 下如果元素节点为 Object，无法找到元素
                    } else if (context.nodeType === 1 && context.nodeName.toLowerCase() !==
"object") {
                    var oldContext = context,
                        old = context.getAttribute("id"),
                        nid = old || id,
                        hasParent = context.parentNode,
                        relativeHierarchySelector = /^\s*[+~]/.test(query);
```

```
                if (!old) {
                    context.setAttribute("id", nid);
                } else {
                    nid = nid.replace(/'/g, "\\$&");
                }
                if (relativeHierarchySelector && hasParent) {
                    context = context.parentNode;
                }
    //如果存在 ID，则将 ID 取得出来放到这个分组的最前面，比如 div b --> [id=xxx] div b
                //不存在 ID，就创建一个 ID，重复上面的操作，但最后会删掉此 ID
                try {
                    if (!relativeHierarchySelector || hasParent) {
                        return makeArray(context.querySelectorAll("[id='" + nid + "'] "
                        + query), extra);
                    }

                } catch (pseudoError) {} finally {
                    if (!old) {
                        oldContext.removeAttribute("id");
                    }
                }
            }
        }

        return oldSizzle(query, context, extra, seed);
    };
    //将原来的方法重新绑定到新 Sizzle 函数上
    for (var prop in oldSizzle) {
        Sizzle[prop] = oldSizzle[prop];
    }

    // release memory in IE
    div = null;
})();
}
```

从源代码中可以看到，它不单单是重写那么简单，根据不同的情况还有各种提速方案。getElementById 自不用说，速度肯定快，这内部做了缓存，而且 getElementById 最多只返回一个元素节点，而 querySelectorAll 则会返回拥有这个 ID 值的多个元素。这个听起来有点奇怪，但 querySelectorAll 不会理会你的错误行为，机械地执行你的指令。

另外，getElementsByTagName 也是内部使用了缓存，它也比 querySelectorAll 快。GetElemens ByTagName 返回的是一个 NodeList 对象，而 querySelectorAll 返回的是一个 StaticNodeList 对象。一个是动态的，一个是静态的。我们做个实验就可以知道这两者有什么不同。

```
var tag = "getElementsByTagName", sqa = "querySelectorAll";
alert(document[tag]("div")==document[tag]("div")); //true
alert(document[sqa]("div")==document[sqa]("div")); //false
```

true 意味着它们拿到的同是 cache 引用，tatic 每次返回都是不一样的 Object。

网上有数据表明，创建一个动态的 NodeList 对象比一个静态的 StaticNodeListc 对象快上 90%。

querySelectorAll 的问题还不止于此，微软在 IE8 中抢先实现了它，那时针对它的规范还没有完成，因此不明确的地方微软自行发挥了。IE8 下如果对 StaticNodeListc 对象取下标超界，不会返回 undefined，而是抛“Invalid procedure call or argument”异常。测试页面如下。

```
<!DOCTYPE html><html><head></head>
   <body><div></div></body>
<script>
        var els = document.body.querySelectorAll('div');
    alert(els[2]);// 2 > els.length-1
   </script>
</html>
```

因此，对于一些奇特的循环，要适可为止。

最后不得不提，querySelectorAll 这个 API 在不同的浏览器、不同的版本中都存在各式各样的 BUG，不支持的选择器类型多了，我们需要做好充分的测试才能安全使用它。在 Sizzle1.9 中，中枪的伪类就有 focus、:enabled、:disabled、:checked……

其他就没有什么值得好说的，这东西不一定要自己手动实现一下才明白是怎么回事。要想支持的选择器类型越多（基本上是伪类），就需要在结构上设计得有扩展性。但过分添加自己的自定义伪类，意味着未来与 querySelectorAll 过不去。像 zepto.js 就是一个 querySelectorAll 当成自己选择器引擎，而 Sizzle1.9 则长达 1700 行。这个自己权衡。最近 jQuery 也搞了个 selector-native 模块，重新审视未来了。

第7章 节点模块

DOM操作占我们前端工作的很大一部分，其中节点操作又占其50%以上。由于选择器引擎的出现，让繁锁的元素选择简单化，并且一下子返回一大堆元素，这个情景时刻暗示着我们操作元素就像CSS为元素添加样式那样，一操作就要操作一组元素。一些大胆的API设计被提出来。当然我们认为所有时髦新颖的设计其实都是很久以前被忽略的设计或者其他领域的设计。例如，集化操作，这是数据库层面的ROM上就有的。链式操作，JavaScript对象基于原型的构造为它大开方便之门。信息密集型的DSL，有rails这座高峰让大家崇拜。jQuery集众之所长，让节点操作简单到极致。因此这章，我们的设计思路沿着jQuery的展开。不过jQuery对每个方法重载得太厉害了，因此作为教学也不太合适。作一个流行的产品，必须模仿者众，如zepto[①]、kimbo[②]、jqMobi[③]、fluent[④]、xui[⑤]、mass[⑥]、jquip[⑦]等。各人视自己的能力阅读它们的源码。本章主要回绕mass的节点操作模块展开。一是因为自己的库比较熟悉，二是zepto等都是移动框架，与我们大多数人工作时用到的PC框架出入太大，三是，mass的分层做得比较好，node模块相当于jQuery2.0中的manipulation模块与traversing模块，node_fix则是它们兼容IE6～IE8的部分。这里就明显地体现了AMD模块化设计的优势。

我们再来看DOM操作包括哪些。数据库所说的CRUD，C就是创建，在集化操作里，innerHTML可以满足我们一下子创建N个节点的需要。

R，就是读取或查找，如果解释为查找的话，选择器引擎已经为我们做了，如果是读取，那么我们还可以将innerHTML、innerText、outerHTML这些属于元素内容的东西划归它处理。

U，就是更新，更新什么呢？还不是innerHTML、innerText、outerHTML这些老脸孔吗？这就冒出一个问题了，它们需要分开两个API来处理，还是合成一个。不过既然我们要追随jQuery，答案也很明显。但jQuery把读写结合（RU结合）的方式建议只用于框架的底层API，到UI层面还是分开吧。底层这样做，可以保证API总数量少，降低入门成本，尽管重载较多，但它们会因为使用次数较多而减轻这负面因素。高层API，则数量非常庞杂，窗口、面板、跑马灯、拖放等都有一大堆方法，它们每个在一个项目中被调用的次数比底层API少多了，因此人们也不会专门去

① http://zeptojs.com/
② http://kimbojs.com/
③ http://www.jqmobi.com/
④ https://github.com/1000ch/fluent
⑤ http://xuijs.com/
⑥ https://github.com/RubyLouvre/mass-Framework
⑦ https://github.com/mythz/jquip

看，基本上是用到才看文档，那么我们尽量做到语义化，Java 传统的 getXXX、setXXX、addXXX、removeXXX 是我们的首选。

最后是 D，亦即移除。在 jQuery 中，移除分为 3 个 API，remove、detach、empty，各有各的适用范围。此外，别忘了，节点操作是围绕着 DOM 树操作。既然是树，就有插入操作。这个被 jQuery 划分为四个 API，其实我们可以看作是 IE 的 insertAdjacentXXX 的强化版。还有克隆，克隆时允许克隆缓存数据与事件。

本章主要围绕 mass 的 node 与 node_fix 模块，jQuery 的 manipulation 模块展开。大家可以在以下地址查看：

https://github.com/RubyLouvre/mass-Framework/blob/1.4/node.js

https://github.com/RubyLouvre/mass-Framework/blob/1.4/node_fix.js

https://github.com/jquery/jquery/blob/1.8-stable/src/manipulation.js

7.1 节点的创建

浏览器提供了多种手段创建 API。从流行度来看，依次是 document.createElement、innerHTML、insertAdjacentHTML、createContextualFragment。

document.createElement 基本不用说什么，它传入一个标签名，然后返回此类型的元素节点，并且对于浏览器还不支持的标签类型，它也能成功返回，这成了后来 IE6～IE8 支持 HTML5 新标签的救命稻草。在 IE6～IE8 中，它还有一种用法，能允许用户连同属性一起生成，比如 document.createElement(" ")。此法常见于生成带 name 属性的 input 与 iframe 元素，因为 IE6～IE7 下这两种元素的 name 属性是只读的，不能修改。

```
//http://thunderguy.com/semicolon/2005/05/23/setting-the-name-attribute-in-internet-e
xplorer/
function createNamedElement(type, name) {
   var element = null;
   // Try the IE way; this fails on standards-compliant browsers
   try {
      element = document.createElement('<' + type + ' name="' + name + '">');
   } catch (e) {
   }
   if (!element || element.nodeName != type.toUpperCase()) {
      // Non-IE browser; use canonical method to create named element
      element = document.createElement(type);
      element.name = name;
   }
   return element;
}
```

innerHTML 本来是 IE 的私有实现，现在遍地开花了。jQuery1.0 就开始发掘 innerHTML 的潜能，这不但是因为 innerHTML 的创建效率至少比 creatElement 快 2～10 倍不等，还因为 innerHTML 能一下子生成一大堆节点。这与 jQuery 推崇集化操作的宗旨不谋而合。但 innerHTML 发生了兼容性问题。比如 IE 会对用户字符串进行 trimLeft 操作，本意是智能去掉没用的空白，但 Firefox 等则认为要忠于用户输入，对应位置要生成文本节点。

```
window.onload = function() {
    var div = document.createElement("div");
    div.innerHTML = " <b>1</b><b>2</b> "
    alert(div.childNodes.length); //IE6~IE8 弹出 3，其他 4
    alert(div.firstChild.nodeType) //IE6 弹出 1，其他 3
}
```

IE 下有些元素节点的 innerHTML 是只读的，重写 innerHTML 会抛错，这就导致我们在动态插入节点时不能不转求 appendChild、insertBefore 来处理。下面出自 MSDN：

http://msdn.microsoft.com/en-us/library/ms533897(VS.85).aspx

The property is read/write for all objects except the following, for which it is read-only: COL, COLGROUP, FRAMESET, HEAD, HTML, STYLE, TABLE, TBODY, TFOOT, THEAD, TITLE, TR

IE 的 innerHTML 会忽略掉 no-scope element。no-scope element 是 IE 的内部概念，隐藏得很深，仅在 MSDN 中说明注释节点是 no-scope element[①]，或在 social.msdn.microsoft.com 官方论坛中透露一点内容[②]——script 与 style 也是 no-scope elements。经过社区这么多年的发掘，大致确认注释、style、script、link、meta、noscript 等表示功能性的标签为 no-scope element。想要用 innerHTML 生成它们，必须在它们之前加上一些东西，比如文字或其他标签。

```
window.onload = function() { //请在 IE6~IE8 下测试
    var div = document.createElement("div");
    div.innerHTML = '<meta http-equiv="X-UA-Compatible" content="IE=9"/>';
    alert(div.childNodes.length);
    div.innerHTML = 'X<meta http-equiv="X-UA-Compatible" content="IE=9"/>';
    alert(div.childNodes.length);
};
```

另一个众所周知的问题是 innerHTML 不会执行 script 标签里面的脚本。其实也不尽然，如果浏览器支持 script 标签的 defer 属性，它就能执行脚本。这个特性检测比较难做，因此像 jQuery 直接用正则把它里面的内容抽取出来，然后全局 eval 了。mass Framework 则采取另一种策略，反正 innerHTML 赋值后已经将它们转换成节点，那么再将它们取出来用 document.createElement("script") 生成的节点代替就行了。

最后一个问题就是有的标签不能单独作为 div 的子元素，比如 td、th 元素，需要最外面包几层，才能放到 innerHTML 中解释，否则浏览器会当作普通的文本节点生成。这个是 jQuery 团队发现的，现在所有框架都使用此技术生成节点。如果把这些特殊的标签比作是“胚胎”，那么孵化它们出来的那些父元素就是胎盘，在 W3C 规范中，它们都是这样一组组地分成不同的模块，如表 7.1 所示。

表 7.1

胚　胎	胎　盘
area	map
param	object
col	tbody, table,colgroup

① http://msdn.microsoft.com/en-us/library/windows/apps/Hh767342.aspx
② http://social.msdn.microsoft.com/forums/en-US/iewebdevelopment/thread/33fd33f7-e857-4f6f-978e-fd486eba7174/

续表

胚　胎	胎　盘
Legend	fieldset
option,optgroup	select
thead,tfoot, tbody, colgroup, caption	table
tr	table, tbody
td, th	table,tbody,tr

一直以来，人们都是使用完整的闭合标签来包裹这些特殊标签，直到人们发现浏览器会自动补全闭合标签后。

```
window.onload = function() {
    var div = document.createElement("div");
    div.innerHTML = '<table><tbody><tr></tr></tbody></table>';//手动闭合标签
    alert(div.getElementsByTagName("tr").length);//1
    div.innerHTML = '<table><tbody><tr></tr>';//让浏览器自动处理
    alert(div.getElementsByTagName("tr").length);//1
}
```

能自动补完结束标签[①]的元素有 body、colgroup、dd、dt、head、html、li、optgroup、option、p、tbody、td、tfoot、th、thead、tr。浏览器这种机制显然是为了竞争的需要，从而吸引开发者转向自己的阵营。对于浏览器而言，根据上下文补完标签不是什么难事，加之，这能有效减少页面的大小，在网速奇慢的年代是一个优化。同时不写结束标签能避免文本节点出现在元素前后，因此也能减轻页面上节点的总体数量。

但现在已经不推荐这样做，浏览器只会固守规则，少写结束标签，很容易引起错误镶嵌。xhtml 布道者就是抓住这一点死命抨击 html4。但在 JavaScript 框架内部，由于是把标签限制在一个 div 内，由经验丰富的 JSer 来处理，因此还是可以利用的。

InsertAdjacentHTML，这也是 IE 的私有实现，dhtml 的产物。比起其他 API，它具有灵活的插入方式。你可以插入到一个元素的内部的最前面（afterBegin）、内部的最后面（beforeEnd），这个元素的前面（beforeBegin）、后面（afterEnd）。它们一一对应 jQuery 的 prepend、append、before、after。因此用它来构造这几个方法，代码量会大大减少。但不巧的是，insertAdjacentHTML 要我们的字符串同样遵守 HTML 的套嵌规则。在 IE 下，它在 td、th 等元素内部插入新节点还是报错，理由同 innerHTML。不过，若我们能提早判定用户字符串没有这么需要套嵌的元素，没有 no-scope 元素，那么在插入操作中它还是很有用的。jQuery 相关的操作是先经由 append 方法进入 domManip 方法，再到 buildFragment 方法，再到 clean 方法，这么复杂才完成。其间有字符串再加工，script 内容抽取，innerHTML 序列化，文档碎片对象生成，插入 DOM，全局 eval 这么多步骤。在最理想的情况，我们可以用一个 insertAdjacentHTML 搞定。

insertAdjacentHTML 的兼容列表如表 7.2 所示。

① http://www.cnblogs.com/rubylouvre/archive/2012/12/03/2799538.html

表 7.2

浏览器	Chrome	Firefox	IE	Opera	Safari(webkit)
版本	1	8	4	7	4(527)

如果浏览器不支持 insertAdjacentHTML，那么我们可以用下面将介绍到的 createContextual Fragment 来模拟。

```
if (typeof HTMLElement !== "undefined" &&
 !HTMLElement.prototype.insertAdjacentElement) {
   HTMLElement.prototype.insertAdjacentElement = function(where, parsedNode) {
          switch (where.toLowerCase()) {
          case 'beforebegin':
             this.parentNode.insertBefore(parsedNode, this)
             break;
          case 'afterbegin':
             this.insertBefore(parsedNode, this.firstChild);
             break;
          case 'beforeend':
             this.appendChild(parsedNode);
             break;
          case 'afterend':
             if (this.nextSibling)
             this.parentNode.insertBefore(parsedNode, this.nextSibling);
             else this.parentNode.appendChild(parsedNode);
             break;
       }
   }
   HTMLElement.prototype.insertAdjacentHTML = function(where, htmlStr) {
       var r = this.ownerDocument.createRange();
       r.setStartBefore(this);
       var parsedHTML = r.createContextualFragment(htmlStr);
       this.insertAdjacentElement(where, parsedHTML)
   }
   HTMLElement.prototype.insertAdjacentText = function(where, txtStr) {
       var parsedText = document.createTextNode(txtStr)
       this.insertAdjacentElement(where, parsedText)
   }
}
```

createContextualFragment 是 Firefox 推出的私有实现，它是 Range 对象一个实例方法。相对于 insertAdjacentHTML 直接将内容插入到 DOM 树，createContextualFragment 则是允许我们将字符串转换为文档碎片，然后再由你决定插入到哪里。在著名的 emberjs 中，如果支持 Range，那么它的 html、append、prepend、after 等方法都用 createContextualFragment 与 deleteContents 实现。createContextualFragment 与 insertAdjacentHTML 一样，要字符串遵守 HTML 的套嵌规则。

除此以外，我们还可以用 document.write 来创建内容，但我们动态添加节点时多发生在 DOM 树建完之后，因此不太合适，这里就不展开了。

我们来看看 mass Framework 是怎么实现的。它的结构与 jQuery 一样，通过两个构造器与一个原型实现无 new 实例化，这样我们的链式操作就不会被 new 关键字打断。

```
function $(a, b) { //第一个构造器
    return new $.fn.init(a, b); //第二个构造器
}
//将原型对象放到一个名字更短更好记的属性名中
//这是 jQuery 人性化的体现，也方便扩展原型方法
$.fn = $.prototype = {
    init: function(a, b) {
        this.a = a;
        this.b = b;
    }
}
//共用同一个原型
$.fn.init.prototype = $.fn;

var a = $(1, 2);
console.log(a instanceof $);
console.log(a instanceof $.fn.init);
```

上面这个结构非常重要，所有 jQuery 风格的类库框架都沿袭它实现链式操作。

如果不打算支持 IE，包括 IE10，因为我们要用到__proto__，我们能做出更好的无 new 实例化。虽然我们设法搞了个类数组对象出来，但毕竟不是真的数组，要时时刻刻维护 length 属性，也不能安稳地享受 es5 的数组原型方法带来的快感。为此，我发明一种全新的结构，与 jQuery 相仿，但更优于 jQuery，就是兼容性一些。

```
var $ = function(expr, context) {
    //这个 dom 真数组其实通过选择器引擎或 domParser 得到的节点集合
    var dom = [];
    return DomArray(dom, expr, context)
}
//DomArray 为内部函数
function DomArray(dom, expr, context) {
    dom = dom || []
    dom.context = context
    dom.expr = expr;
    dom.__proto__ = DomArray.prototype;//重要
    return dom
}
DomArray.prototype = $.fn = [];//重要，目的是使用数组方法
$.fn.get = function() {//添加原型方法
    alert(this.expr)
}
var a = $("div");
a.push("a", "b", "c")
a.get()// div
alert(a.length);//3
a.forEach(function(i) {
    alert(i);//依次 a、b、c
})
```

下面继续原来的主题。为我们的 jQuery 式类组数对象进行扩展。

```
$.fn.extend({
    init: function(expr, context) {
        // 分支 1: 处理空白字符串、null、undefined 参数
        if (!expr) {
            return this;
        }
        //分支 2:  让$实例与元素节点一样拥有 ownerDocument 属性
        var doc, nodes; //用作节点搜索的起点
        if ($.isArrayLike(context)) { //typeof context === "string"
            return $(context).find(expr);
        }

        if (expr.nodeType) { //分支 3:  处理节点参数
            this.ownerDocument = expr.nodeType === 9 ? expr : expr.ownerDocument;
            return $.Array.merge(this, [expr]);
        }
        this.selector = expr + "";
        if (typeof expr === "string") {//分支 4: 处理 CSS3 选择器
            doc = this.ownerDocument = !context ? document : getDoc(context, context[0]);
            var scope = context || doc;
            expr = expr.trim();
            if (expr.charAt(0) === "<" && expr.charAt(expr.length - 1) === ">" && expr.length
>= 3) {
                nodes = $.parseHTML(expr, doc); //分支 5: 动态生成新节点
                nodes = nodes.childNodes;
            } else if (rtag.test(expr)) { //分支 6: getElementsByTagName
                nodes = scope[TAGS](expr);
            } else { //分支 7: 进入选择器模块
                nodes = $.query(expr, scope);
            }
            return $.Array.merge(this, nodes);
        } else { //分支 8: 处理数组、节点集合、mass 对象或 window 对象
            this.ownerDocument = getDoc(expr[0]);
            $.Array.merge(this, $.isArrayLike(expr) ? expr : [expr]);
            delete this.selector;
        }
    },
    mass: $.mass,
    length: 0,
    valueOf: function() { //转换为纯数组对象
        return Array.prototype.slice.call(this);
    },
    size: function() {
        return this.length;
    },
    toString: function() { //收集它们的 tagName 属性，做成纯数组返回
        var i = this.length,
            ret = [],
            getType = $.type;
        while (i--) {
            ret[i] = getType(this[i]);
        }
        return ret.join(", ");
```

```
},
labor: function(nodes) { //用于构建一个与对象具有相同属性，但里面的节点集不同的 mass 对象
    var neo = new $;
    neo.context = this.context;
    neo.selector = this.selector;
    neo.ownerDocument = this.ownerDocument;
    return $.Array.merge(neo, nodes || []);
},
slice: function(a, b) { //传入起止值，截取原某一部分再组成 mass 对象返回
    return this.labor($.slice(this, a, b));
},
get: function(num) { //取得与索引值相对应的节点，若为负数从后面取起，如果不传，则返回节点集的纯数组
    return !arguments.length ? this.valueOf() : this[num < 0 ? this.length + num : num];
},
eq: function(i) { //取得与索引值相对应的节点，并构成 mass 对象返回
    return i === -1 ? this.slice(i) : this.slice(i, +i + 1);
},
gt: function(i) { //取得原对象中索引值大于传参的节点们，并构成 mass 对象返回
    return this.slice(i + 1, this.length);
},
lt: function(i) { //取得原对象中索引值小于传参的节点们，并构成 mass 对象返回
    return this.slice(0, i);
},
first: function() { //取得原对象中第一个的节点，并构成 mass 对象返回
    return this.slice(0, 1);
},
last: function() { //取得原对象中最后一个的节点，并构成 mass 对象返回
    return this.slice(-1);
},
even: function() { //取得原对象中索引值为偶数的节点，并构成 mass 对象返回
    return this.labor($.filter(this, function(_, i) {
        return i % 2 === 0;
    }));
},
odd: function() { //取得原对象中索引值为奇数的节点，并构成 mass 对象返回
    return this.labor($.filter(this, function(_, i) {
        return i % 2 === 1;
    }));
},
each: function(fn) {
    return $.each(this, fn);
},
map: function(fn) {
    return this.labor($.map(this, fn));
},
clone: function(dataAndEvents, deepDataAndEvents) { //复制原 mass 对象，它里面的节点
                                                    //也一一复制
    //略
},
html: function(item) { //取得或设置节点的 innerHTML 属性
    //略
},
```

```
    text: function(item) { // 取得或设置节点的 text 或 innerText 或 textContent 属性
        //略
    },
    outerHTML: function(item) { // 取得或设置节点的 outerHTML
        //略
    }
});
$.fn.init.prototype = $.fn;
"push,unshift,pop,shift,splice,sort,reverse".replace($.rword, function(method) {
    $.fn[method] = function() {
        Array.prototype[method].apply(this, arguments);
        return this;
    }
});
```

不可否认，这结构有点头重脚轻，但只有这样才模拟 jQuery 那个$方法。在这个难表其义的$方法中，它通过参数个数的不同，参数类型的不同实现方法重载。jQuery 官方文档介绍，它包含 9 种不同的传参方式。

- jQuery(selector [, context])
- jQuery(element)
- jQuery(elementArray)
- jQuery(object)
- jQuery(jQuery object)
- jQuery()
- jQuery(html ,[ownerDocument])
- jQuery(html, attributes)
- jQuery(callback)

假若按功能来分，它大致分为 3 种，选择器、domParser 与 domReady。

由于重载太多了，因此基本上所有号称 jQuery-compatible 的类库框架都没有实现它的所有重载。如果抛开这些细节，我们不难发现，除了最后的 domReady，其他一切目的不过是想获取要操作的节点集合罢了。为了更方便的操作，这些节点与实例通过数字进行关联，构成一个类数组对象，因此你会看到它绑定了 push、unshift、pop、shift、splice、sort、reverse、each、map 等数组方法，让它看起来就是一个数组！

labor 相当于 jQuery 的 pushStack，用于构建下一个类数组对象，比如 map、lt、gt、eq 等方法就是内部调用它来实现。但 jQuery 的 pushStack 没有这么简单，它还有一个 prevObject 属性，保存着上次操作的对象。链式操作得越多，被引用着不能释放的东西就越多，因此对于内存比较拮据的手机环境，不是个好的方案。

再来看 lt、gt、eq、last、first、odd、even，是不是很眼熟？没错，它们就是 jQuery 自定义伪类的仿造器，但却是一个很好的代替品。无论是从性能来说，或者出于未来只使用 querySelectorAll 做选择器的考量，它们都是好东西。工作业务中，只高亮表格的偶数行这一需要也很频繁。因此，做成独立的方法是个明智的决定。

而根据用户传入的字符生成一堆节点的功能则是由 parseHTML 方法实现的。parseHTML 是一个复杂的方法，它对不同浏览器做了分级处理，对于 IE6～IE8，框架还会智能加载 node_fix 模块，里面有 fixParseHTML 方法，为它打补丁。

```
var tagHooks = {
    area: [1, "<map>"],
    param: [1, "<object>"],
    col: [2, "<table><tbody></tbody><colgroup>", "</table>"],
    legend: [1, "<fieldset>"],
    option: [1, "<select multiple='multiple'>"],
    thead: [1, "<table>", "</table>"],
    tr: [2, "<table><tbody>"],
    td: [3, "<table><tbody><tr>"],
    //IE6～IE8 在用 innerHTML 生成节点时，不能直接创建 script、link、meta、style 与 HTML5 的新标签
    _default: $.support.noscope ? [1, "X<div>"] : [0, ""]
},
types = $.oneObject("text/javascript", "text/ecmascript",
    "application/ecmascript", "application/javascript", "text/vbscript"),
    rxhtml = /<(?!area|br|col|embed|hr|img|input|link|meta|param)(([\w:]+)[^>]*)\/>/ig,
    //innerHTML 创建时可能无法生成的标签类型
    rcreate = $.support.noscope ? /(<(?:script|link|style|meta|noscript))/ig : /[^\d\D]/,
    //需要处理套嵌关系的标签类型
    rnest = /<(?:tb|td|tf|th|tr|col|opt|leg|cap|area)/,
    rtagName = /<([\w:]+)/;
tagHooks.optgroup = tagHooks.option;
tagHooks.tbody = tagHooks.tfoot = tagHooks.colgroup = tagHooks.caption = tagHooks.thead;
tagHooks.th = tagHooks.td;
$.fn.parseHTML = function(html, doc) {
    doc = doc || this.nodeType === 9 && this || document;
    html = html.replace(rxhtml, "<$1></$2>").trim();
    //尝试使用 createContextualFragment 获取更高的效率
    //http://www.cnblogs.com/rubylouvre/archive/2011/04/15/2016800.html
    if ($.cachedRange && doc === document && !rcreate.test(html) && !rnest.test(html)) {
        return $.cachedRange.createContextualFragment(html);
    }
    if ($.support.noscope) { //修补 IE，在 link style script 等 no-scope 元素前打个补丁
        html = html.replace(rcreate, "<br class=fix_noscope>$1");丁
    }
    var tag = (rtagName.exec(html) || ["", ""])[1].toLowerCase(),
        //取得其标签名
        wrap = tagHooks[tag] || tagHooks._default,
        fragment = doc.createDocumentFragment(),
        wrapper = doc.createElement("div"),
        firstChild;
    wrapper.innerHTML = wrap[1] + html + (wrap[2] || "");
    var els = wrapper[TAGS]("script");//TAGS = "getElementsByTagName"
    if (els.length) { //使用 innerHTML 生成的 script 节点不会发出请求与执行 text 属性
        var script = doc.createElement("script"),
            neo;
        for (var i = 0, el; el = els[i++];) {
            if (!el.type || types[el.type]) { //如果 script 节点的 MIME 能让其执行脚本
            neo = script.cloneNode(false); //FF 不能省略 cloneNode 的参数
```

```
                for (var j = 0, attr; attr = el.attributes[j++];) {
                    if (attr.specified) { //复制其属性
                        neo[attr.name] = [attr.value];
                    }
                }
                neo.text = el.text; //必须指定,因为无法在 attributes 中遍历出来
                el.parentNode.replaceChild(neo, el); //替换节点
            }
        }
    }
    //移除我们为了符合套嵌关系而添加的标签
    for (i = wrap[0]; i--; wrapper = wrapper.lastChild) {};
    $.fixParseHTML(wrapper, html); //对于 IE6~IE8 进行处理
    while (firstChild = wrapper.firstChild) { // 将 wrapper 上的节点转移到文档碎片上!
        fragment.appendChild(firstChild);
    }
    return fragment;
}
```

函数一开始我们就先尝试使用 createContextualFragment，这时有个$.cachedRange 对象，就是一个 Range 对象，它在 support 模块已经被创建好，以便反复利用。

```
try {
    var range = DOC.createRange();
    range.selectNodeContents(body); //修补 Opera(9.2~11.51) Bug,必须对文档进行选取
    support.fastFragment = !!range.createContextualFragment("<a>");
    $.cachedRange = range;
} catch (e) {
}
```

如果支持 createContextualFragment，我们可以立即返回，因为它本来就是生成一个文档碎片。否则我们需要自己创建一个文档碎片，然后通过 div.innerHTML 生成节点。这时第一个问题是 IE6～IE8 的 no-scope 元素，它需要我们在前面垫些东西。我选择插入一个 br 元素，方便后面用 getElementsByTagName 重新获得它们移除掉。第二个问题是那些需要包裹几层才能生成的标签，这个我用 tagHooks 对象打发掉。第三个问题是 innerHTML 生成 script 标签不会发出请求或执行标签内的脚本，我的策略是动态生成 script 节点，然后逐一替换，把原节点的所有属性赋给新节点。

```
var rtbody = /<tbody[^>]*>/i
$.fixParseHTML = function(wrapper, html) {
    if ($.support.noscope) { //移除所有 br 补丁
        for (els = wrapper["getElementsByTagName"]("br"), i = 0; el = els[i++];) {
            if (el.className && el.className === "fix_noscope") {
                el.parentNode.removeChild(el);
            }
        }
    }
    //当我们在生成 colgroup、thead、tfoot 时，IE 会自作多情地插入 tbody 节点
    if (!$.support.insertTbody) {
        var noTbody = !rtbody.test(html),
            //矛:html 本身就不存在<tbody 字样
```

```
            els = wrapper["getElementsByTagName"]("tbody");
        if (els.length > 0 && noTbody) { //盾：实际上生成的 NodeList 中存在 tbody 节点
            for (var i = 0, el; el = els[i++];) {
                if (!el.childNodes.length) //如果是自动插入的，里面肯定没有内容
                el.parentNode.removeChild(el);
            }
        }
    }
    //IE6、IE7 没有为它们添加 defaultChecked
    if (!$.support.appendChecked) {
        for (els = wrapper["getElementsByTagName"]("input"), i = 0; el = els[i++];) {
            if (el.type === "checkbox" || el.type === "radio") {
                el.defaultChecked = el.checked;
            }
        }
    }
}
```

fixParseHTML 针对 IE6～IE8 又做了些处理，究竟是什么，注释里已经说得很明白，这里不重复了。我们再尝试另一种实现，这个新的 domParser 与刚才给出的新结构一样，只支持最新的浏览器，不过这次仁慈一点，兼容 IE10。没有办法，IE9 的 innerHTML 在某些元素中还是只读的。

由于没有 no-scope 元素，我们只需专心处理套嵌与 script 标签就行了。

```
var TABLE = document.createElement("table");
var TR = document.createElement("tr");
var SELECT = document.createElement("select");
var tagHooks = {
    option: SELECT,
    thead: TABLE,
    tfoot: TABLE,
    tbody: TABLE,
    td: TR,
    th: TR,
    tr: document.createElement("tbody"),
    col: document.createElement("colgroup"),
    legend: document.createElement("fieldset"),
    "*": document.createElement("div")
}
var rparse = /^\s*<(\w+|!)[^>]*>/;
//在 DomArray 的构造器中调用
if (rparse.test(expr)) {
    var html = expr.trim();
    var tag = RegExp.$1;
    dom = $.parseHTML(html, tag)
}

$.parseHTML = function(html, tag) {
    var parent;
    if (tag == null) {
        tag = "*"
    } else if (!(tag in tagHooks)) {
        tag = "*"
    }
```

```
    parent = tagHooks[tag];
    parent.innerHTML = "" + html;
    //这里尝试处理 script 节点
    return [].slice.call(parent.childNodes);
}
```

7.2 节点的插入

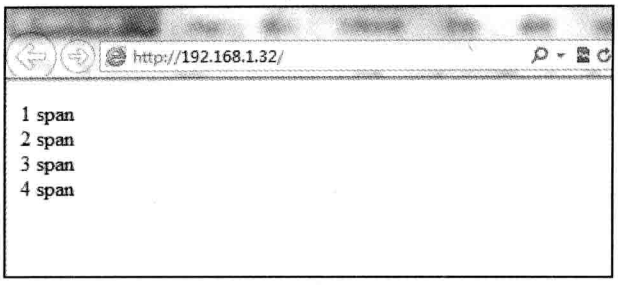
原生 DOM 接口是非常简单的，参数类型确定，不会重载，每次只处理一个元素节点；而 jQuery 式的方法则相反，虽然名字短，但参数类型复杂，过度重载，对于插入这样的写操作，是进行批处理的。

为了简化处理逻辑，jQuery 的做法是统统转换为文档碎片，然后将它复制到与当前 jQuery 对象里面包含的节点集合相同的个数，一个个插入。

```
<!DOCTYPE HTML>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>测试</title>
        <script src="jquery.js"></script>
        <script>
            window.onload = function() {
                var a = document.createElement("span")
                a.innerHTML = "span"
                $("div").append(a)
            }
        </script>
    </head>
    <body>
        <div>
            1
        </div>
        <div>
            2
        </div >
        <div>
            3
        </div>
        <div>
            4
        </div>
    </body>
</html>
```

效果如图 7.1 所示。

为了提高性能，合理利用高级 API，mass Framework 的做法是能用 createContextualFragment 就用 createContextualFragment，能用 insertAdjacentHTML 时就用 insertAdjacentHTML，否则就换转为文档碎片，通过 appendChild、insertBefore 插入。这意味着里面分支很复杂，因此我们需要搞个适配器，让它尽可能简单地分流。



▲图 7.1

至于 API 的命名，将沿袭 jQuery 的那几个名字，append、prepend、before、after 与 replace。值得一提的是，由于这几个方法太受欢迎，W3C 在 DOM4 决定原生支持它们[①]，参数可以是字符串与 DOM 节点。哈哈，连参数的可选类型都这么相近，以后框架实现它们时，就只需薄薄一层包装。

再看 mass Framework，这 5 个都是通过 manipulate 方法实现。由于设计得当，它们的反转方法也实现得轻而易举。

```
"append,prepend,before,after,replace".replace($.rword, function(method) {
    $.fn[method] = function(item) {
        return manipulate(this, method, item, this.ownerDocument);
    };
    $.fn[method + "To"] = function(item) {
        $(item, this.ownerDocument)[method](this);
        return this;
    };
});
```

相比于 jQuery 的实现：

```
append: function() {
    return this.domManip(arguments, true, function(elem) {
        if (this.nodeType === 1 || this.nodeType === 11 || this.nodeType === 9) {
            this.appendChild(elem);
        }
    });
},

prepend: function() {
    return this.domManip(arguments, true, function(elem) {
        if (this.nodeType === 1 || this.nodeType === 11 || this.nodeType === 9) {
            this.insertBefore(elem, this.firstChild);
        }
    });
},

before: function() {
    return this.domManip(arguments, false, function(elem) {
        if (this.parentNode) {
```

① https://dvcs.w3.org/hg/domcore/raw-file/tip/Overview.html#interface-element

```
            this.parentNode.insertBefore(elem, this);
        }
    });
},

after: function() {
    return this.domManip(arguments, false, function(elem) {
        if (this.parentNode) {
            this.parentNode.insertBefore(elem, this.nextSibling);
        }
    });
},

replaceWith: function(value) {
    var isFunction = jQuery.isFunction(value);

    // Make sure that the elements are removed from the DOM before they are inserted
    // this can help fix replacing a parent with child elements
    if (!isFunction && typeof value !== "string") {
        value = jQuery(value).not(this).detach();
    }

    return value !== "" ? this.domManip([value], true, function(elem) {
        var next = this.nextSibling,
            parent = this.parentNode;

        if (parent) {
            jQuery(this).remove();
            parent.insertBefore(elem, next);
        }
    }) : this.remove();
},

jQuery.each({
    appendTo: "append",
    prependTo: "prepend",
    insertBefore: "before",
    insertAfter: "after",
    replaceAll: "replaceWith"
}, function(name, original) {
    jQuery.fn[name] = function(selector) {
        var elems,
        ret = [],
            insert = jQuery(selector),
            last = insert.length - 1,
            i = 0;

        for (; i <= last; i++) {
            elems = i === last ? this : this.clone(true);
            jQuery(insert[i])[original](elems);

            // Support: QtWebKit
            // .get() because core_push.apply(_, arraylike) throws
            core_push.apply(ret, elems.get());
        }
```

```
        return this.pushStack(ret);
    };
});
```

但不管怎么说，双方的思想都一致，接口就是一个空心化的方法，用于提供一个语义化且便捷的名字而已，实现全部转至内部去。mass 内部是由 manipulate 这个大管家负责从适配器寻找最优的方案来处理。

```
function manipulate(nodes, name, item, doc) {
    //我们只允许向元素节点内部插入东西，因此需要转换为纯正的元素节点集合
    var elems = $.filter(nodes, function(el) {
        return el.nodeType === 1;
    }),
        handler = insertHooks[name];
    if (item.nodeType) {
        //如果是传入元素节点、文本节点或文档碎片
        insertAdjacentNode(elems, item, handler);
    } else if (typeof item === "string") {
        //如果传入的是字符串片断
//如果方法名不是 replace，完美支持 insertAdjacentHTML，并且不存在套嵌关系的标签
        var fast = (name !== "replace") && $.support[adjacent] && !rnest.test(item);
        if (!fast) {
            item = $.parseHTML(item, doc);
        }
        insertAdjacentHTML(elems, item, insertHooks[name + "2"], handler);
    } else if (item.length) {
        //如果传入的是 HTMLCollection nodeList mass 实例，将转换为文档碎片
        insertAdjacentFragment(elems, item, doc, handler);
    }
    return nodes;
}

function insertAdjacentNode(elems, item, handler) {
//使用 appendChild、insertBefore 实现，item 为普通节点
    for (var i = 0, el; el = elems[i]; i++) { //第一个不用复制，其他的要复制
        handler(el, i ? cloneNode(item, true, true) : item);
    }
}

function insertAdjacentHTML(elems, item, fastHandler, handler) {
    for (var i = 0, el; el = elems[i++];) { //尝试使用 insertAdjacentHTML
        if (item.nodeType) { //如果是文档碎片
            handler(el, item.cloneNode(true));
        } else {
            fastHandler(el, item);
        }
    }
}

function insertAdjacentFragment(elems, item, doc, handler) {
    var fragment = doc.createDocumentFragment();
    for (var i = 0, el; el = elems[i++];) {
```

```
            handler(el, makeFragment(item, fragment, i > 1));
        }
    }

    function makeFragment(nodes, fragment, bool) {
        //只有非 NodeList 的情况下才为 i 递增
        var ret = fragment.cloneNode(false),
            go = !nodes.item;
        for (var i = 0, node; node = nodes[i]; go && i++) {
            ret.appendChild(bool && cloneNode(node, true, true) || node);
        }
        return ret;
    }
    var insertHooks = {
        prepend: function(el, node) {
            el.insertBefore(node, el.firstChild);
        },
        append: function(el, node) {
            el.appendChild(node);
        },
        before: function(el, node) {
            el.parentNode.insertBefore(node, el);
        },
        after: function(el, node) {
            el.parentNode.insertBefore(node, el.nextSibling);
        },
        replace: function(el, node) {
            el.parentNode.replaceChild(node, el);
        },
        prepend2: function(el, html) {
            el[adjacent]("afterBegin", html);
        },
        append2: function(el, html) {
            el[adjacent]("beforeEnd", html);
        },
        before2: function(el, html) {
            el[adjacent]("beforeBegin", html);
        },
        after2: function(el, html) {
            el[adjacent]("afterEnd", html);
        }
    };
```

这时我们需要留意一下 makeFragment 这个函数，这里涉及两个重要的知识点：NodeList 的循环操作，文档碎片的复制。

NodeList 看起来像数组，但它在插入节点时会立即改变长度。

```
<div id="test">
       <a href="http://www.google.com/">link</a>
</div>

window.onload = function() {
    var els = document.getElementsByTagName("a");
    var div = document.getElementById("test");
```

```
    for (var i = 0; i < els.length; i++) {
        var ele = document.createElement("a");
        ele.setAttribute("href", "http://www.google.com/");
        ele.appendChild(document.createTextNode("new link"));
        div.appendChild(ele); //添加一个新链接
    }
}
```

上面将陷入死循环。我们在循环它时，最好将它的 length 保存到一个变量中，然后比较决定是否中断循环。

第二个是碎片对象的复制问题，我们大可以使用原生的 cloneNode(true)，但在 IE 下 attachEvent 绑定的事件会跟着被复制。由于不是我们框架绑定的事件，那么在移除时就无法找到对应的引用了。

除此之外，jQuery 还提供了 wrap、wrapAll、wrappInner 这三种特殊的插入操作。

wrap 为当前元素提供了一个父节点，此父节点将动态插入原节点的父亲底下。这个我们可以轻松在 IE 下用 neo.applyElement (old, "outside")实现。

wrapAll 则是为一堆元素提供一个共同的父节点，插入到第一个元素的父亲底下，其他元素再统统挪到新节点底下。

wrappInner 是为当前元素插入一个新节点，然后将它之前的孩子挪到新节点底下。这个我们可以轻松在 IE 下用 neo.applyElement (old, "inside")实现。

从上面的描述来看，applyElement 真是很强大可以在标准浏览器扩展一下，让它应用更广！

```
if (!document.documentElement.applyElement && typeof HTMLElement !== "undefined") {
    HTMLElement.prototype.removeNode = function(deep) {
        if (this.parentNode) {
            if (!deep) {
                var fragment;
                var range = this.ownerDocument.createRange();
                range.selectNodeContents(this);
                fragment = range.extractContents();
                range.setStartBefore(this);
                range.insertNode(fragment);
                range.detach();
            }
            return this.parentNode.removeChild(this);
        }
        if (!deep) {
            var range = this.ownerDocument.createRange();
            range.selectNodeContents(this);
            range.deleteContents();
            range.detach();
        }
        return this;
    };
    HTMLElement.prototype.applyElement = function(newNode, where) {
        newNode = newNode.removeNode(false);

        switch ((where || 'outside').toLowerCase()) {

            case 'inside':
                var fragment;
```

```
                var range = this.ownerDocument.createRange();
                range.selectNodeContents(this);
                range.surroundContents(newNode);
                range.detach();
                break;

            case 'outside':
                var range = this.ownerDocument.createRange();
                range.selectNode(this);
                range.surroundContents(newNode);
                range.detach();
                break;
            default:
                throw new Error('DOMException.NOT_SUPPORTED_ERR(9)');
        }

        return newNode;
    };
}
```

7.3 节点的复制

IE 下对元素的复制与 innerHTML 一样，存在许多 BUG，非常著名的就是上节所说的，IE 自作多情地复制 attachEvent 事件。另外，根据测试，标准浏览器的 cloneNode，只会复制元素写在标签内的属性与通过 setAttribute 设置的属性，而 IE6～IE8 还支持通过 node.aaa = "xxx"设置的属性复制。

```
<div id="aaa" data-test="test" title="title">目标节点</div>

window.onload = function() {
    var node = document.getElementById("aaa");
    node.expando = {
        key: 1
    }
    node.setAttribute("attr", "attr")
    var clone = node.cloneNode(false);
    alert(clone.id);//aaa
    alert(clone.getAttribute("data-test"));//test
    alert(clone.getAttribute("title"));//title
    alert(clone.getAttribute("attr"));//attr
    node.expando.key = 2 //修正为 2
    alert(clone.expando.key )//IE6～IE8: 2; 其他: undefined
}
```

如果仅是这样还好办，但 IE 在复制时不但会多复制一些，还会少复制一些，这让程序员不好处理。我们看一下 mass 是怎么处理的。它与 jQuery 一样，支持两个参数，第一个是只复制节点，但不复制数据与事件，默认为 false； 第二个决定如何复制它的子孙，默认是遵循参数一的决定。

```
$.fn.clone = function(dataAndEvents, deepDataAndEvents) {
    dataAndEvents = dataAndEvents == null ? false : dataAndEvents;
    deepDataAndEvents = deepDataAndEvents == null ? dataAndEvents : deepDataAndEvents;
```

```
    return this.map(function() {
        return cloneNode(this, dataAndEvents, deepDataAndEvents);
    });
}
```

可以看出，此方法只对参数进行处理，具体操作由 cloneNode 执行。

```
function cloneNode(node, dataAndEvents, deepDataAndEvents) {
    if (node.nodeType === 1) {
        var neo = $.fixCloneNode(node),//复制元素的 attributes
            src, neos, i;
        if (dataAndEvents) {
            $.mergeData(neo, node);//复制数据与事件
            if (deepDataAndEvents) {//处理子孙的复制
                src = node[TAGS]("*");
                neos = neo[TAGS]("*");
                for (i = 0; src[i]; i++) {
                    $.mergeData(neos[i], src[i]);
                }
            }
        }
        src = neos = null;
        return neo;
    } else {
        return node.cloneNode(true);
    }
}
```

cloneNode 也做了分层设计，如果在标准浏览器，fixCloneNode 只是一个标准的 cloneNode(true)，否则框架会智能加载 node_fix 模块，里面的 fixCloneNode 会做大量打补丁的工作。首先要处理的是 HTML5 的新标签，如果不支持，用户也没有提前使用 document.createElement hack，那么我们唯有使用 outerHTML 进行复制。具体操作是将原节点的 outerHTML 赋给一个动态生成并插入 DOM 树的 div 元素，作为它的 innerHTML，然后再从中提取出来。属性复制则需要一个个元素（包括它们的孩子）进行处理，具体见 fixNode 函数。

```
function fixNode(clone, src) {
    if (src.nodeType == 1) {
        //只处理元素节点
        var nodeName = clone.nodeName.toLowerCase();
        //clearAttributes 方法可以清除元素的所有属性值，如 style 样式，或者 class 属性，与 attachEvent
        //绑定上去的事件
        clone.clearAttributes();
        //复制原对象的属性到克隆体中，但不包含原来的事件、ID、NAME、uniqueNumber
        clone.mergeAttributes(src, false);
        //IE6～IE8 无法复制其内部的元素
        if (nodeName === "object") {
            clone.outerHTML = src.outerHTML;
        } else if (nodeName === "input" && (src.type === "checkbox" || src.type == "radio"))
{
            //IE6～IE8 无法复制 chechbox 的值，在 IE6、IE7 中 defaultChecked 属性也遗漏了
            if (src.checked) {
```

```
                clone.defaultChecked = clone.checked = src.checked;
            }
            // 除 Chrome 外，所有浏览器都会给没有 value 的 checkbox 一个默认的 value 值"on"。
            if (clone.value !== src.value) {
                clone.value = src.value;
            }
        } else if (nodeName === "option") {
                clone.selected = src.defaultSelected; // IE6～IE8 无法保持选中状态
        } else if (nodeName === "input" || nodeName === "textarea") {
            clone.defaultValue = src.defaultValue; // IE6～IE8 无法保持默认值
        } else if (nodeName === "script" && clone.text !== src.text) {
            clone.text = src.text; //IE6～IE8 不能复制 script 的 text 属性
        }

    }
}
var shim = document.createElement("div"); //缓存 parser，防止反复创建

function shimCloneNode(outerHTML, tree) {
    tree.appendChild(shim);
    shim.innerHTML = outerHTML;
    tree.removeChild(shim);
    return shim.firstChild;
}
var unknownTag = "<?XML:NAMESPACE"
$.fixCloneNode = function(node) {
//这个判定必须这么长：判定是否能克隆新标签，判定是否为元素节点，判定是否为新标签
    if (!$.support.cloneHTML5 && node.outerHTML) { //延迟创建检测元素
        var outerHTML = document.createElement(node.nodeName).outerHTML,
            bool = outerHTML.indexOf(unknownTag) // !0 === true;
    }
    //各浏览器 cloneNode 方法的部分实现差异
    //http://www.cnblogs.com/snandy/archive/2012/05/06/2473936.html
    var neo = !bool ? shimCloneNode(node.outerHTML, document.documentElement) : node.
cloneNode(true)
    fixNode(neo, node);
    var src = node[TAGS]("*"),
        neos = neo[TAGS]("*");
    for (var i = 0; src[i]; i++) {
        fixNode(neos[i], src[i]);
    }
}
```

这当中技术含量最高的是 mootools 团队挖掘出来的 mergeAttributes hack。早些年，为了在 IE6～IE8 中不复制 attachEvent 事件，jQuery 被逼动用 outerHTML 来生成新节点。

```
//https://github.com/jquery/jquery/blob/1.4/src/manipulation.js
clone: function( events ) {
       var ret = this.map(function() {
           if ( !jQuery.support.noCloneEvent && !jQuery.isXMLDoc(this) ) {
               var html = this.outerHTML, ownerDocument = this.ownerDocument;
               if ( !html ) {
                   var div = ownerDocument.createElement("div");
```

```
                div.appendChild( this.cloneNode(true) );
                html = div.innerHTML;
            }

            return jQuery.clean([html.replace(rinlinejQuery, "")
                .replace(rleadingWhitespace, "")], ownerDocument)[0];
        } else {
            return this.cloneNode(true);
        }
    });

    //复制事件
    if ( events === true ) {
        cloneCopyEvent( this, ret );
        cloneCopyEvent( this.find("*"), ret.find("*") );
    }
    return ret;
},
```

即便是这样，还是很臃肿，唯一的解决之道就是时间。等时间来淘汰掉 IE6 这些古老浏览器，或者项目主管非常有魄力地只支持新锐浏览器。在 zepto 等手机框架，它们把数据都储存在 data-* 属性中，直接用一个 cloneNode(true)搞定节点的复制！

```
//zepto
clone: function() {
    return this.map(function() {
        return this.cloneNode(true)
    })
}
```

7.4 节点的移除

浏览器提供了多种移出节点的方法，常见的有 removeChild、removeNode，动态创建一个元素节点或文档碎片再 appendChild，创建 Range 对象选中目标节点然后 deleteContents。

removeNode 是 IE 的私有实现，opera 也实现了此方法。它的作用是将目标节点从文档树中删除，返回目标节点。它有一个参数，为布尔值，其默认值为 false，即仅删除目标节点，保留子节点，true 时同 removeChild 的用法。

deleteContents 算是比较偏门的 API，兼容性差，这个以后说。

removeChild 在 IE6～IE7 中有内存泄漏问题[①]，与 IE 的 CG 回收比较失败而引起的。由于这太底层了，这里就不展开了。这里给出 EXT 框架的方案。像 EXT 这样庞大的 UI 库，所有节点都动态生成，因此是非常注重 CG 回收的。

```
var removeNode = IE6 || IE7 ? function() {
    var d; //IE6、IE7 的判定自己写
    return function(node) {
        if (node && node.tagName != 'BODY') {
```

① 具体链接可看 http://social.msdn.microsoft.com/Forums/en-US/iewebdevelopment/thread/c76967f0-dcf8-47d0-8984-8fe1282a94f5
http://www.cnblogs.com/chy1000/archive/2010/08/03/1791372.html

```
            d = d || document.createElement('DIV');
            d.appendChild(node);
            d.innerHTML = '';
        }
    }
}() : function(node) {
    if (node && node.parentNode && node.tagName != 'BODY') {
        node.parentNode.removeChild(node);
    }
}
```

为什么这么写呢？因为在 IE6～IE8 中存在一个叫 DOM 超空间（DOM hyperspace）的概念，当元素移出 DOM 树，又有 JavaScript 关联时元素不会消失，它被保存在一个叫超空间的地方。《PPK 谈 JavaScript》一书指出，可以用是否存在 parentNode 来判定元素是否在超空间。

```
window.onload = function() {
    var div = document.createElement("div");
    alert(div.parentNode); //null
    document.body.removeChild(document.body.appendChild(div));
    alert(div.parentNode); //IE6~IE8 object;其他 null
    if (div.parentNode) {
        alert(div.parentNode.nodeType); //11 文档碎片
    }
}
```

第一个 alert 出 null，这个所有浏览器都一样，因此有时我们误以为可以当作节点是否在 DOM 的基准。但当元素插入 DOM 树再移出时，就有差异了，IE6～IE8 下弹出了一个文档碎片对象。因此可以想象为何 IE 性能这么差了，它自以为这样能重复使用元素，但通常用户移除了就不管，因此久而久之，内存就允许了许多这样的“碎片”，加之其他问题，就很容易造成泄漏。

我们再看 innerHTML 清除元素会怎么样。

```
<body><div id="test"></div></body>
window.onload = function() {
    var div = document.getElementById('test');
    document.body.innerHTML = '';
    alert(div.parentNode);//null
}
```

结果在 IE 下也是 null，但这也不能说明 innerHTML 就比 removeChild 好。我们继续下一个实验。

```
<body>
    <div><div id="test1">test1</div></div>
    <div><div id="test2">test2</div></div>
</body>
window.onload = function() {
    var div1 = document.getElementById('test1');
    div1.parentNode.removeChild(div1);
    alert(div1.id + ":" + div1.innerHTML);//test1:test1
    var div2 = document.getElementById('test2');
    div2.parentNode.innerHTML = "";
    alert(div2.id + ":" + div2.innerHTML);//test2
}
```

这时我们就发现，当用 removeChild 移除节点，原来元素的结构没有发生变化，但在 innerHTML 时，IE6～IE8 下会直接清空其里面的内容，只剩下个空壳，而标准浏览器则与 removeChild 保持一致。打个比喻，在 IE 下，removeChild 就是掰断树枝，但树枝可以再次使用。而 innerHTML 就是把所需要的枝叶给拔下来然后把树枝烧掉。鉴于 IE 下对内存管理的失败，这么干净的清除节点正是我们寻找的方法！因此 EXT 从 1.0 到 4.0，此方法也没大改变。

对于 jQuery 这样的类库框架来说，估计很难走这条路。它已经被自己的数据缓存系统绑架了，移除节点时需要逐个检测元素，从缓存系统中移除对应的缓存体，否则会让浏览器宕机。不过最不好的是 jQuery 通过类数组结构与 preObject 困住节点的方式，这就造成了 jQuery 即便是使用 innerHTML，元素节点在 IE 下还是位于 DOM 超空间中。

jQuery 在性能上没有优势，于是在移除节点的方式上造势。它提供了三种移除节点的方式。remove，移除节点的同时从数据缓存系统上移除对应数据。empty，只清空元素的内部，相当于 IE 下的 removeNode(false)。detach，移除节点但不清除数据。前两种好理解，但为什么要创建第三种方法呢？

从我们的工作业务看，DOM 操作远远不止这些，还有 UI 交互，样式渲染等，但后者都是基于前者上运作。

纯粹的 JavaScript 操作不会带来什么消耗，95%以上的能耗是 DOM 操作引起。出于性能考虑，我们最佳的做法是在设置样式前，将元素移出 DOM 树，处理完再插回来。但绝对大多数操作 DOM 的方法都与数据缓存方法关联在一起，若用 remove 方法，会让它们无法进行数据清理工作，导致内存泄漏，而 detach 就是基于此需要而设的。从设计理念来看，有点像数据库操作的事务。begin 一下（detach），开始一连串 DOM 操作，就算玩得天昏地暗，也不伤及 DOM 树的其他元素，最后 commit 一下（append），将最后的结果显示出来！

下面是它们的实现。

```
"remove,empty,detach".replace($.rword, function(method) {
    $.fn[method] = function() {
        var isRemove = method !== "empty";
        for (var i = 0, node; node = this[i++];) {
            if (node.nodeType === 1) {
                //移除匹配元素
                var array = $.slice(node[TAGS]("*")).concat(isRemove ? node : []);
                if (method !== "detach") {
                    array.forEach(cleanNode);
                }
            }
            if (isRemove) {
                if (node.parentNode) {
                    node.parentNode.removeChild(node);
                }
            } else {
                while (node.firstChild) {
                    node.removeChild(node.firstChild);
                }
            }
        }
```

```
        return this;
    }
});
```

如果我们的框架没有像 jQuery 那样引入一个庞大的数据缓存系统，而是像 zepto.js 那样通过 HTML5 的 data-*来缓存数据，那么许多东西都可以简化了。这也意味着我们不打算兼容 IE6、IE7、IE8，那么我们就可以使用 deleteContents 或 textContent。例如，我们实现一个清空元素内部的 API。

版本一，最传统的方式：

```
function clearChild (node) {//node 可以是元素节点与文档碎片
       while (node.firstChild) {
           node.removeChild(node.firstChild)
       }
       return node
}
```

版本二，使用 deleteContents，创建一个 Range 对象，然后通过 setStartBefore，setEndAfter 选择边界，最后清空它们的节点。

```
var deleteRange = document.createRange()
function clearChild (node) {//node 可以是元素节点与文档碎片
    deleteRange.setStartBefore(node.firstChild)
    deleteRange.setEndAfter(node.lastChild)
    deleteRange.deleteContents()
    return node
}
```

版本三，使用 textContent。textContent 是 W3C 版本的 innerText。这个东西在较新的浏览器中兼容性特别好，并且同时存在于元素节点与文档碎片中。

```
function clearChild (node) {//node 可以是元素节点与文档碎片
     node.textContent = ""
     return node
}
```

7.5 innerHTML、innerText 与 outerHTML 的处理

在开始之前，我们不得不审视一个问题。像 innerHTML，innerText、outerHTML 都是元素节点的一个属性，可读可写。由于我们的对象是一个类数组对象，所有操作都是集化操作，是不是每个方法都来一次 for 循环呢？正常的思路是 get all、set all，类数组对象里有多少个元素节点，就处理多少次。如果是读操作，就返回一个数组，里面包含所有处理后的结果。mootools、YUI、EXT 都采取这种策略。但 jQuery 选择一种奇特的策略，get first, set all。事实证明这个是成功的。如果返回一组结果，我们还不是要二次选取。

此外，jQuery 太多数方法是多态方法，根据参数的情况有多种重载方式。如果每个这样的方法，都需要做这样那样的参数判定，显然很笨拙。因此 jQuery 将它抽象成一个 access 方法。如果细读 $.access 方法，就全部掌握 css、width、height、attr、prop、html、text、data 等多态方法的用法了。

```
$.access = function(elems, callback, directive, args) {
//用于统一配置多态方法的读写访问
    var length = elems.length,
           key = args[0],
           value = args[1]; //读方法
    if (args.length === 0 || args.length === 1 && typeof directive === "string") {
        var first = elems[0]; //由于只有一个回调，我们通过 this == $判定读/写
        return first && first.nodeType === 1 ? callback.call($, first, key) : void 0;
    } else {//写方法
        if (directive === null) {
            callback.call(elems, args);
        } else {
            if (typeof key === "object") {
                for (var k in key) { //为所有元素设置 N 个属性
                    for (var i = 0; i < length; i++) {
                        callback.call(elems, elems[i], k, key[k]);
                    }
                }
            } else {
                for (i = 0; i < length; i++) {
                    callback.call(elems, elems[i], key, value);
                }
            }
        }
    }
    return elems; //返回自身，链式操作
}
```

elems 为要处理的节点集合；callback 为回调，里面有读操作与写操作，由 this 的情况决定进度入哪个分支；directive 为处理指令，由于内部分支很复杂，必须需要额外的 flag 进行分流；args，就是调用$.access 函数的那个函数的参数对象。

有了这个，我们来看看如何实现操作 innerHTML、innerText、outerHTML 的方法。

```
html: function(item) { //取得或设置节点的 innerHTML 属性
    return $.access(this, function(el, value) {
        if (this === $) { //getter
            return "innerHTML" in el ? el.innerHTML : innerHTML(el);
        } else { //setter
            value = item == null ? "" : item + "";
            //如果 item 为 null, undefined 转换为空字符串，其他强制转字符串
            //接着判断 innerHTML 属性是否符合标准,不再区分可读与只读
            //用户传参是否包含了 script style meta 等不能用 innerHTML 直接进行创建的标签
            //及像 col td map legend 等需要满足套嵌关系才能创建的标签，否则会在 IE 与 Safari 下报错
            if ($.support.innerHTML && (!rcreate.test(value) && !rnest.test(value))) {
                try {
                    for (var i = 0; el = this[i++];) {
                        if (el.nodeType === 1) {
                            $.each(el[TAGS]("*"), cleanNode);
                            el.innerHTML = value;
                        }
                    }
                    return;
```

```
                } catch (e) {};
            }
            this.empty().append(value);
        }
    }, null, arguments);
},
text: function(item) { // 取得或设置节点的 text、innerText 或 textContent 属性
    return $.access(this, function(el) {
        if (this === $) { //getter
            if (el.tagName === "SCRIPT") {
                return el.text; //IE6~IE8 下只能用 innerHTML、text 获取内容
            }
            return el.textContent || el.innerText || $.getText([el]);
        } else { //setter
            this.empty().append(this.ownerDocument.createTextNode(item));
        }
    }, null, arguments);
},
outerHTML: function(item) { // 取得或设置节点的 outerHTML
    return $.access(this, function(el) {
        if (this === $) { //getter
            return "outerHTML" in el ? el.outerHTML : outerHTML(el);
        } else { //setter
            this.empty().replace(item);
        }
    }, null, arguments);
}
```

为了兼容 XML，我们又搞了以下几种方法。

```
function outerHTML(el) { //主要是用于 XML
    switch (el.nodeType + "") {
        case "1":
        case "9":
            return "xml" in el ? el.xml : new XMLSerializer().serializeToString(el);
        case "3":
        case "4":
            return el.nodeValue;
        default:
            return "";
    }
}

function innerHTML(el) { //主要是用于 XML
    for (var i = 0, c, ret = []; c = el.childNodes[i++];) {
        ret.push(outerHTML(c));
    }
    return ret.join("");
}

function getText() {
    //获取某个节点的文本，如果此节点为元素节点，则取其 childNodes 的所有文本
    return function getText(nodes) {
        for (var i = 0, ret = "", node; node = nodes[i++];) {
```

```
            // 处理得文本节点与 CDATA 的内容
              if (node.nodeType === 3 || node.nodeType === 4) {
                 ret += node.nodeValue;     //取得元素节点的内容
              } else if (node.nodeType !== 8) {
                 ret += getText(node.childNodes);
          }
          }
          return ret;
      }
  }()
```

可见要实现一个完美的方法多不容易。但它们永远不能算是完美，我们要做出各种权衡，各种妥协，能满足我们的业务需要就行了。

7.6 一些奇葩的元素节点

即使我们的框架设计强大，总有覆盖不到的地方。比如 IE 下 select 标签移到遮罩层的上面来，xml 数据岛提供另一种文档套文档的方式，option 无法通过设置 CSS 让它拥有好看的样式，noscript 取不到里面的 innerHTML……浏览器有太多这样的细节，若无关紧要，框架的核心就尽可能忽略它们，转交插件去处理。

到目前，我们只需重点照顾 3 个元素。

7.6.1 iframe 元素

iframe 是一个古老的标签，IE3 时已经存在了。由于它是用作镶嵌另一个页面到主页面，因此肯定与一般的元素不同，创建起来也不是一般的消耗资源，并消耗连接数。但它却是一个物超所值的东西，有了它，我们可以无障碍地实现无缝刷新，通过保存历史模拟 onhashchange，安全地加载第三方资源与广告，实现富本文编辑器，文件上传，用它搞定 IE6～IE7 的 select BUG，在 iframe 里做特征侦测……HTML5 为它添加了 3 个属性，让它变得更强大。

由于是先出自 iframe，因此免不了兼容性问题。首先是样式相关的。

想要隐藏 iframe 那个很粗的边框时，我们需要用到 frameBorder 属性，例如使用 Dreamweaver 可以生成如下代码。

```
<iframe frameborder=0 src='xxxx' width='xxx' height='xxx'></iframe>
```

但是动态创建时，标准浏览器可以使用 setAttribute 来设置它。这时，作为一个特性，大小写不敏感。但旧版本 IE 不认。

```
var iframe = document.createElement('iframe');
iframe.setAttribute('frameborder',0);//Firefox 下有效，IE 下无效
```

唯有直接赋值时，双方才认。

iframe.frameBorder=0;//Firefox 和 IE 均有效

另外，这个 iframe 相当于 CSS 中的"border:0 none"。

去掉 iframe 中的滚动条：

```
iframe.scrolling = "no";
```

IE 下想设置透明比较麻烦，并且在 IE5.5 才开始支持 iframe 内容透明。要让它透明，必须满足以下两个条件。

- iframe 自身设置 allowTransparency 属性为 true（但设置了 allowTransparency=true 就遮不住 select）。
- iframe 中的文档，background-color 或 body 元素的 bgColor 属性必须设置为 transparent。

具体例子如下。

1. 包含 iframe 页面的代码

```
<body bgColor="#eeeeee"> <iframe allowTransparency="true" src="transparent.htm">
</iframe>
```

2. transparent.htm 页的代码

```
<body bgColor="transparent">
```

获取 iframe 中的 window 对象：

```
function getIframeWindow(node) {
    return node.contentWindow;
}
```

我们也可以通过 frames[iframeName]来取得它的 window 对象，这个所有浏览器都支持，由于 IE，ID 与 Name 不怎么区分，因此它也可以用 frames[iframeID]来取。

获取 iframe 中的文档对象：

```
function getIframeDocument(node) { //w3c || IE
    return node.contentDocument || node.contentWindow.document;
}
```

判定页面是否在 iframe 里面：

```
window.onload = function() {
    alert(window != window.top)
    alert(window.frameElement !== null);
    alert(window.eval !== top.eval)
}
```

判定 iframe 是否加载完毕：

```
if (iframe.addEventListener) {
    iframe.addEventListener("load", callback, false);
} else {
    iframe.attachEvent("onload", callback)
}
```

不过，如果是动态创建 iframe, webkit 系统浏览器可能会出现二次触发 onload 事件的问题。

```
<div id="times">0</div>
<script>
    window.onload = function(){
      var c = document.getElementById("times");
      var iframe = document.createElement("iframe");
      iframe.onload = function(){ c.innerHTML = ++c.innerHTML }
      document.body.appendChild(iframe);
      iframe.src = "http://www.cnblogs.com/rubylouvre"
    }
</script>
```

估计 Safari 和 Chrome 在 appendChild 之后就进行第一次加载，并且在设置 src 之前加载完毕，所以触发了两次。如果在插入 body 之前给 iframe 随便设置一个 src（除了空值），间接加长第一次加载，那么也只触发一次了。不设置或空值的 src 相当于链接到“about:blank”（空白页）。

动态创建 iframe 时，如果想用到 name 属性，用 document.createElement("iframe")创建再设置它的 name 属性，IE6、IE7 是无法辨识此值的。

```
window.onload = function(){
    var iframe = document.createElement("iframe");
    iframe.name = "xxx";
    document.body.appendChild(iframe);
    iframe.src = "http://www.cnblogs.com/rubylouvre/"
    alert(frames["xxx"]);//undefined
     alert(document.getElementsByName("xxx").length)//0
}
```

因此我们需要针对 IE6、IE7，使用 IE 特有的创建元素时连属性一起创建的方式实现。

```
if ("1"[0]) {//IE6、IE7 这里返回 undefined,于是跑到第二个分支
    var iframe = document.createElement("iframe");
    iframe.name = name;
} else {
    iframe = document.createElement('<iframe name="' + name + '">');
}
```

iframe 与父窗口共享 history，基于它我们可以解决 Ajax 时的后退按钮问题。里面的细节非常多，这里就不展开了，github 上有两个著名的项目，可以帮你解决工作的绝大多数问题。

https://github.com/browserstate/history.js

https://github.com/devote/HTML5-History-API

清空 iframe 内容，不保留历史的写法：

```
iframeWindow.location.replace('about:blank');
```

IE6 下 iframe.src="about:blank"在 https 协议下会出现问题，需要用 javascript:false 修正，虽然速度非常慢。详见下面网址的讨论：

http://gemal.dk/blog/2005/01/27/iframe_without_src_attribute_on_https_in_internet_explorer/

iframe 与父窗口间的通信。如果是同源，那么它们之间可以随便操作，如果不同源，就需要 postMessage 与各种 hack!所谓同源，就是指域名、协议、端口相同，如图 7.2 所示。

URL	说明	是否允许通信
http://www.a.com/a.js http://www.a.com/b.js	同一域名下	允许
http://www.a.com/lab/a.js http://www.a.com/script/b.js	同一域名下不同文件夹	允许
http://www.a.com:8000/a.js http://www.a.com/b.js	同一域名，不同端口	不允许
http://www.a.com/a.js https://www.a.com/b.js	同一域名，不同协议	不允许
http://www.a.com/a.js http://70.32.92.74/b.js	域名和域名对应IP	不允许
http://www.a.com/a.js http://script.a.com/b.js	主域相同，子域不同	不允许
http://www.a.com/a.js http://a.com/b.js	同一域名，不同二级域名（同上）	不允许（cookie这种情况下也不允许访问）
http://www.cnblogs.com/a.js http://www.a.com/b.js	不同域名	不允许

▲图 7.2

判定 iframe 与父页面同源：

```
function isSameOrigin(el) {
    var ret = false;
    try {
        ret = !!el.contentWindow.location.href;
    } catch (e) {
    }
    return ret;
}
```

有关 JavaScript 如何跨域的文章网上有一大萝，这里只重点介绍两个，postMessage 与 navigator。

postMessage 是 HTML5 的重要方法之一，估计未来跨域就靠它了。它在 IE8 与稍微新一点的标准浏览器中都支持，并且能跨大域。涉及 postMessage 方法与 message 事件。有关它们的用法可以看 MDN[①]与 MSDN[②]就行，这里直接举例。

首先，我们创建一个主页面，它用于接收消息与回复对方。

```
<!DOCTYPE HTML>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <script>
            window.onmessage = function(e) {
                var event = e || window.event;
                try {
                    console.log(event);
                    console.log(event.data);
                    console.log(event.data.aaa);
                } catch (e) { }
```

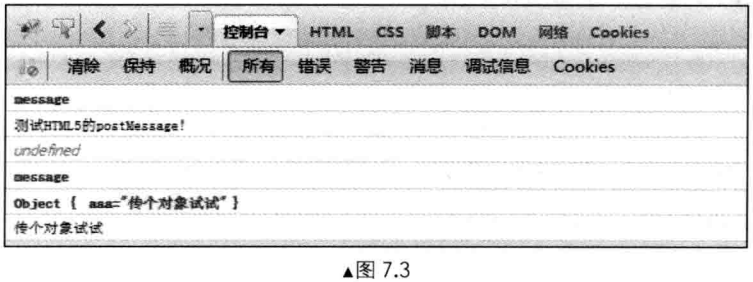
① https://developer.mozilla.org/en-US/docs/DOM/window.postMessage
② http://msdn.microsoft.com/zh-cn/library/ie/cc197015(v=vs.85).aspx

```
            event.source.postMessage("好的,我已经收到你的消息了", event.origin);
        }
    </script>
</head>
<body>
    <p>测试 HTML5 的 postMessage by 司徒正美</p>
    <iframe id="aaa" src="http://www.cnblogs.com/rubylouvre/articles/2956254.html">
    </iframe>
</body>
</html>
```

iframe 中的页面，它首先发出请求和对主页面的消息进行响应。

```
<!DOCTYPE HTML>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <script>
            if (window.postMessage) {
                window.parent.postMessage("测试 HTML5 的 postMessage!", "*")
                window.parent.postMessage({aaa: "传个对象试试"}, "*")
            }
            window.onmessage = function(e) {
                var event = e || window.event;
                console.log(event);
                console.log(event.data)
            }

        </script>
    </head>
   <body>
   </body>
</html>
```

Firebug 下的测试结果如下，如图 7.3 所示。



▲图 7.3

注：在 IE8、IE9 中，postMessage 中能传送字符串，结果可能有不同。

上面实验也可以在这里查看到：

http://rubylouvre.github.com/post.html

第二种就是利用 IE6、IE7 的 navigator 对象的跨大域漏洞，至少没有被封堵，与 postMessage

结合使用应该能满足95%上的跨域通信需求。

主页面如下。

```
<!DOCTYPE HTML>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <p>测试 IE6、IE7 的 navigator hack by 司徒正美</p>
       <iframe id="aaa" src="http://www.cnblogs.com/rubylouvre/articles/2956855.html">
        </iframe>
        <script>
            navigator.log = function(msg) {
            //由于 IE6、IE7 下没有控制台,我们把调试信息打印到页面
                var div = document.createElement("div");
                div.innerHTML = msg;
                document.body.appendChild(div)
            }
            navigator.a = function(msg) {
                navigator.log("这是父页面中的 a 方法: " + msg);
            }
            var iframe = document.getElementById("aaa");
            iframe.attachEvent && iframe.attachEvent("onload", function() {
                setInterval(function() {
                    window.navigator.b('XXXXX');
                }, 3200);
            })
        </script>
    </body>
</html>
```

iframe 页面如下。

```
<!DOCTYPE HTML>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <script>
            navigator.b = function(msg) {
                navigator.log("这是 iframe 中的 b 方法: " + msg);
            }
            setInterval(function() {
                window.navigator.a('YYYYY');
            }, 3300);
        </script>
    </body>
</html>
```

IE6 下的截图，如图 7.4 所示。

▲图 7.4

上面实验也可以在这里查看到：

http://rubylouvre.github.com/navigator.html

现在我们尝试整合它们，让 API 调用尽量接近 postMessage。我们需要一个发送信息的接口，也需要一个绑定监听函数的接口。下面是我的实现，用于 postMessage, JSON 的高级技巧，ondataavailable 回调与 IE6、IE7 的 navigator hack。

```
//跨域通信组件 by 司徒正美
define(["node"], function($) {
    function Messenger(config) {
        var win = config.target
        if (typeof win === "string") {
            //主页面的 target 参数应该为 iframe 元素的 CSS 表达式
            win = $(win).get(0);
            if (win && win.tagName === 'IFRAME') {
                win = win.contentWindow;
            }
        } else {
            //子页面的 target 参数恒为 parent,以防不小心访问它的其他属性时报错
            win = parent;
        }
        this.win = win;
        this._callbacks = [];
        var mode = document.documentMode;
        if (mode === 8 || mode === 9) {//处理 IE8、IE9 postMessage 只支持传字符串的情况
            this.hack = true;
        }
        if (typeof config.onmessage === "function") {
            this.receive(config.onmessage);//添加处理函数
        }
        this.init();//开始进行监听
    }
```

```
    Messenger.prototype = {
        init: function() {
            var me = this;
            me._callback = function(event) {
                if (event.source != me.win)
                    return;//如果不是来源自 win 所指向的窗口,返回
                var data = event.data;
                if (typeof data === "string" && data.indexOf("__hack__") === 0) {
                    data = data.replace("__hack__", "");//还原数据
                    data = JSON.parse(data, function(k, v) {
                        if (v.indexOf && v.indexOf('function') > -1) {
                            return eval("(function(){return " + v + " })()")
                        }
                        return v;
                    });
                }
                for (var i = 0, fn; fn = me._callbacks[i++]; ) {
                    fn.call(me, data);
                }
            };
            $.bind(window, "message", me._callback);
        },
        receive: function(fn) {//添加监听函数,处理来自其他页面的数据
            fn.win = this.win;
            this._callbacks.push(fn);
        },
        send: function(data) {//向其他页面发送数据,为兼容起见,建议只传字符串与没有函数的对象
            var str = data, hack = false;
            //如果对象中存在函数,会抛 DataCloneError: DOM Exception 25 异常,需要序列化补救

//http://stackoverflow.com/questions/7506635/uncaught-error-data-clone-err-dom-except
ion-25-thrown-by-web-worker
            if (!this.hack && typeof data === "object") {
                for (var i in data) {//这是一个简单的检测,只检测它直辖的子属性
                    if (data.hasOwnProperty(i) && typeof data[i] === "function") {
                        hack = true;
                        break;
                    }
                }
            }
            if (hack || (this.hack && typeof str !== "string")) {
                //在 W3C 规范中,data 可以是任意数据类型
                //这时我们就需要用到 JSON 进行序列化与反序列化
                //下面是 data 所有可能的类型
                //JavaScript primitive, such as a string
                //object
                //Array
                //PixelArray object
                //ImageData object
                //Blob
                //File
                //ArrayBuffer
                str = JSON.stringify(str, function(k, v) {
                    if (typeof v === 'function') {
```

```
                return v + '';
            }
            return v;
        });
        data = "__hack__" + str;
    }
    this.win.postMessage(data, '*');//parent
},
destroy: function() {
    // 解除绑定事件
    if (this._callback) {
        $.unbind(this.win, "message", this._callback)
    }
    // 解除绑定事件 IE
    if (document.detachEvent && this._dataavailable) {
        document.detachEvent('ondataavailable', this._dataavailable);
    }
    // 删除实例属性
    for (var p in this) {
        if (this.hasOwnProperty(p)) {
            delete this[p];
        }
    }
    navigator.messages = void 0;
}
};
if (!"1"[0]) {//IE6、IE7
    Messenger.prototype.init = function() {
        var isSameOrigin = false;
        try { //判定是否同源，不同源会无法访问它的属性，并抛出错误
            isSameOrigin = !!this.win.location.href;
        } catch (e) {
        }
        if (isSameOrigin) {
            this.initForSameOrigin();
        } else {
            this.initForCrossDomain();
        }
    };
    Messenger.prototype.initForCrossDomain = function() {
        var fns = navigator.messages = navigator.messages || [];
        var me = this;
        for (var i = 0, fn; fn = this._callbacks[i++]; ) {
            fns.push(fn);
        }
        this.receive = function(fn) {
            fn.win = this.win;
            fns.push(fn);
        };
        this.send = function(data) {
            setTimeout(function() {
                for (var i = 0, fn; fn = fns[i++]; ) {
                    if (fn.win != me.win) {
                        fn.call(me, data);
```

```
                    }
                }
            });
        };
    };
    Messenger.prototype.initForSameOrigin = function() {
        var me = this;
        this.send = function(data) {
            setTimeout(function() {
                var event = me.win.document.createEventObject();
                event.eventType = 'message';
                event.eventSource = window;
                event.eventData = data;
                me.win.document.fireEvent('ondataavailable', event);
            });
        };
        this._dataavailable = function(event) {
            if (event.eventType !== 'message' || event.eventSource != me.win)
                return;
            for (var i = 0, fn; fn = me._callbacks[i++]; ) {
                fn.call(me, event.eventData);
            }
        };
        document.attachEvent('ondataavailable', this._dataavailable);
    };
}

return Messenger;
});
//如果想在旧版本的标准浏览器支持跨域通信，可以使用 window.name
```

用法如下，父页面时刻监听子页面收送信息，并在 iframe 加载完后与子页面通信。

```
require("messenger,event,ready", function(Messenger, $) {
    var messager = new Messenger({
        target: '#iframe',
        onmessage: function(data) {
            console.log(data)
        }
    });
    $("#iframe").on("load", function() {
        messager.send('发给子页面的消息');
    });
})
```

子页面由于是被动加载，因此不等待，直接就可以与父页面通信。

```
require("messenger,event,ready", function(Messenger, $) {
    var messager = new Messenger({
        target: parent,
        onmessage: function(data) {
            console.log(data)
        }
    });
    messager.send('发给父页面的消息');
```

```
    messager.send({
        aaa: "发送对象",
        fn: function() {
            var a = 1
        }
    });
})
```

此实验我们还可以直接浏览以下网址观察到效果：

http://rubylouvre.github.io/messenger.html

7.6.2　object 元素

在 HTML5 的 video 标签出来之前,我们基本上是通过 object、embed、applet 这三个标签来加载视频或其他多媒体资源。

到目前为止，除了内部使用的遗留系统，很少用到 applet。如果只想插入 flash 视频，不进行 JavaScript 交互，embed 标签也足矣。

embed 元素是由 NetScape Navigator 2 引入，用于在 HTML 文档中插入符合网景插件应用程序编程接口（NPAPI）规范的插件。NPAPI 插件是跨平台的，并可以在所有实现了此接口规范的浏览器中使用。目前的主流浏览器中，IE 系列以外的浏览器均支持 NPAPI 插件。

事实上 IE3.0 就支持 NPAPI，但是在 IE5.5 SP2 后微软出于安全考虑停止了对 NPAPI 的支持，转而推荐用户使用其 ActiveX 作为替代。ActiveX 是 Microsoft 对于一系列策略性面向对象程序技术和工具的称呼，其中主要的技术是组件对象模型（COM）。为了加载 COM，微软为 object 标签添加了两个属性：classid 与 codebase。

classid 就是这个 COM 组件在系统中注册的一个 ID 值,有了这个 ID 值系统才能找到这个 COM 组件对应的 DLL 文件，就像普通的软件用其他 DLL 时需要一个路径一样。

codebase 为一个 URL，是用来下载和更新组件用的。比如浏览某个网页时，发现你的机器上没有安装这个组件，IE 就会去 codebase 指示的地址下载组件，有了新的版本也会提示你安装新版本。

微软曾经打算用 object 标签取代 img 与 applet 标签，因为把它设计得非常强大，配合它下面的 param 子元素，能做出更多样化的制定。但由于 IE6 统治时间太长与微软的更新不给力，导致这个本来很先进的东西终于积重难返，被逐渐抛弃了。

别的不说,object标签存在几个陷井,这就让用过它的人感到厌恶。首先,它必须使用innerHTML或 document.write 等手段动态创建。通过 document.createElement 这个标准 API 逐个创建，反而无法加载我们的资料。

另外，创建 param 时必须同时指定 name 属性，否则被忽略掉。

```
var div = document.createElement("div");
div.innerHTML = "<object><param></object>";
alert(div.innerHTML);  // <OBJECT></OBJECT>
div.innerHTML = "<object><param/></object>";
alert(div.innerHTML);  // <OBJECT></OBJECT>
div.innerHTML = "<object><param name=test></object>"
alert(div.innerHTML);  // <OBJECT><PARAM NAME="test" VALUE=""></OBJECT>
```

另一个是旧版本 IE 用 getElementsByTagName(“*”)无法取到 param 元素：

```
<object type="application/x-shockwave-flash" data="/medias/player_flv_maxi.swf"
  width="320" height="240">
     <param name="movie" value="/medias/player_flv_maxi.swf">
     <param name="allowFullScreen" value="true">
     <param  name="FlashVars"  value="flv=/medias/video.flv&amp;width=320&amp;height=240&
amp;showstop=1&amp;showvolume=1&amp;showtime=1&amp;startimage=/medias/startimage_en.jpg
&amp;showfullscreen=1&amp;bgcolor1=189ca8&amp;bgcolor2=085c68&amp;playercolor=085c68">
      <p>Video demo</p>
  </object>

window.onload = function() {
    alert(document.body.getElementsByTagName("*").length) //IE6-8 为 1
    alert(document.body.getElementsByTagName("param").length) //3
}
```

更多的问题出自于播放 flash 视频时，下面是 object 标签与 ember 标签播放 flash 的最简形式。

```
<object type="application/x-shockwave-flash" data="test.swf" width="128" height="128"
><param name="movie" value="test.swf" /></object>

<embed type="application/x-shockwave-flash" src="test.swf" width="128" height="128"
></embed>
```

object 方式显然更臃肿些。

flash 播放器允许我们传入更多参数进行精细的设置，如表 7.3 所示。

表 7.3

参　数	说　明
src	资源名称（如 movieName.swf），仅用于 embed
movie	资源名称（如 movieName.swf），仅用于 object
width	数字或百分数，以像素数或浏览器窗口宽度的百分数形式指定影片的宽度
height	数字或百分数，以像素数或浏览器窗口宽度的百分数形式指定影片的高度
codebase	URL，指向 Flash 播放器的 ActiveX 控件的位置，当浏览器未安装它时，可自动到该位置下载。仅用于 object
pluginspace	URL，指向 Flash 播放器插件的位置，在需要时便于安装。仅用于 embed
swliveconnect	可选。true 或 false. 用于确定在第一次载入 Flash 播放器时是否启动 Java，当该项被省略时取默认值 False。无论什么时候只要在同一页面中包含 JavaScriipt 程序和 Flash，为使“FS Commands”语句起作用，必须运行 Java，但如果页面中的 JavaScript 程序仅用来实现监测浏览器的类型或其他与“FS Commands”语句无关的功能，则可以把 swliveconnect 置为 false 以阻止 Java 的启动，要启动 Java，可显式地将 swliveconnect 置为 true，这将大大增加影片开始播放过程所用的时间。仅用于 embed
play	可选，true 或 false，设置视频加载完后立即播放。默认为 true
loop	可选，true 或 false，设置视频重复播放。默认为 true
quality	可选，可为 low、high、autolow、autohig 或 best 任何一个。决定视频的播放质量。默认为 high
bgcolor	可选，取值：#RRGGBB（16 进制的 RGB 值）。用于指定影片的背景色，该属性可取代 Flash 影片文件中背景色的设定，但不影响影片所在 HTML 页的背景色设定

续表

参　数	说　明
scale	可选，取值：showall、noborder、exactfit。当宽度和高度值以百分数表示时，确定影片被如何放置在浏览器窗口中。 Showall（默认值）在指定尺寸的区域中显示整个影片的内容并保持与原影片相同的长宽比例，影片内容不发生变形。 noborder 在维持影片长宽比例的情况下填充指定区域，影片内容不发生变形，但影片的部分内容可能显示不出来。 exactfit 使整个影片在指定区域可见，因为此时不再维持原有的长宽比例，所以影片有可能变形。该属性被省略时（且宽度和高度值以百分数表示时）将按默认值 showall 执行
align	可选，取值：L、R、T、B，决定视频在浏览器窗口中的位置。 “L”值使影片与浏览器窗口的左边对齐，如果浏览器窗口不足以容纳影片，将调整窗口的上下边和右边。 “R”值使影片与浏览器窗口的右边对齐，如果浏览器窗口不足以容纳影片，将调整窗口的上下边和左边。 “T”值使影片与浏览器窗口的顶边对齐，如果浏览器窗口不足以容纳影片，将调整窗口的左右边和底边。 “B”值使影片与浏览器窗口的底边对齐，如果浏览器窗口不足以容纳影片，将调整窗口的左右边和顶边。 省略时使影片置于浏览器窗口的中央，如果浏览器窗口尺寸比影片所占区域尺寸小，将调整浏览器窗口尺寸，使影片正常显示
salign	可选，取值：L、R、T、B、TL、TR、BL、BR，用于确定经缩放的影片在一指定宽高尺寸的区域中如何放置。各种取值的含义参见 align
base	可选，指定视频相对于 flash 播放器的路径，否则它是相对于当前页面。有载入外部资源的都知道，Flash 相对路径是根据它所在 HTML 而不是 Flash 自己本身，例如网页 http://aaa.org/test.html 有一个 http://aaa.org/swf/test.swf，这个 Flash 以相对路径载入 test.xml，Flash Player 载入 http://aaa.org/test.xml 而不是 http://aaa.org/swf/test.xml。因为这种特性，往往发生很多发布上的问题。开发人员明明在本地测试妥当，所有相关 files 连同 swf 都放在同一个 folder 里面，到交付客户时，却出现问题，因为别人可能喜欢将整个东西放在一个 subfolder 下，在 HTML 嵌入 Flash 时，Flash 和本身 HTML 根本不在同一个 folder 下，结果出现找不到外部资源问题。 base 参数就针对此需求应运而生，只要设定 base="."，就可以跟着 Flash 路径走
menu	可选，取值：true、false。用于指定在浏览器中当对影片所占区域右击鼠标（对 windows）或按住“command”键单击鼠标（对 macintosh）时出现的快捷菜单的类型。true 时将显示整个菜单，允许被演示者对放映过程进行多种控制。false 时只显示包含“About Flash”菜单项的菜单。默认为 true
wmode	可选，取值：window、opaque、transparent。 window 值使得影片在网页中指定的位置播放，这也是几种选项中播放速度最快的一种。 opaque 值将挡住网页上影片后面的内容。 transparent 值使得网页上影片中的透明部分显示网页的内容与背景，有可能降低动画速度。 此参数只对 object 标签有效，默认值为 window
allowScriptAccess	可选，取值：always、never、samedomain，用于决定 Flash 是否可用 getURL、FSCommand 和 ExternalInterace 去调用页面上的 JavaScript 全局函数。 always 允许随时执行脚本操作。 never 禁止所有脚本执行操作。 samedomain 只有在 Flash 与 HTML 页面同域时才允许执行脚本操作
allowFullScreen	可选，取值：true、false。用于决定是否全屏显示
flashvars	可选，以 queryString 的形式传参

由于旧版本 IE 只能通过 innerHTML、outerHTML、document.write 来动态创建 object 标签，因

此为页面添加一个 flash 的基本工作就是拼接字符串。IE 下的 classid("clsid:D27CDB6E- AE6D-11cf-96B8-444553540000")可不是一个容易记的字符串。通常来说，我们可以借助 SWFObject[①]、jQuery.Flash[②]来加载我们的 flash。不过，通过上面的学习，我们完全有能力自己搞一个出来。

```
define(function() {
    var params = function(obj) {
        var ret = [];
        for (var name in obj) {
            if (obj.hasOwnProperty(name)) {
                ret.push('<param name="', name, '" value="', obj[name], '"/>');
            }
        }
        return ret.join("");
    };
    var props = function(obj) {
        var ret = [];
        for (var name in obj) {
            if (obj.hasOwnProperty(name)) {
                ret.push(name + '="' + obj[name] + '" ');
            }
        }
        return ret.join("");
    };
    var query = function(obj) {
        var ret = [];
        for (var name in obj) {
            if (typeof obj[name] !== "function") {
                ret.push(encodeURIComponent(name) + "=" + encodeURIComponent(obj[name]));
            }
        }
        return ret.join("&");
    };
    /**
     * 创建 Flash 对象
     * @param {Element} el 放置 flash 的容器元素
     * @param {Object} obj swf 的相关配置
     * @param {String|Object} vars 可选参数,以 queryString 形式传入
     */
    return function(el, obj, vars) {
        var html, flashvars = typeof vars === "string" ? vars : query(vars);
        // 由于默认的交互参数是 JSON 格式，会有双引号，需要转义掉，以免 HTML 解析出错
        flashvars = flashvars.replace(/"/g, '&quot;');
        // IE 下必须有 id 属性，不然与 JavaScript 交互会报错
        // http://drupal.org/node/319079
        obj.id = obj.id || 'flash' + setTimeout(Date);
        obj.name = obj.name || obj.id;
        obj.width = obj.width || 1;
        obj.height = obj.height || 1;
        obj.flashvars = flashvars;
        if ('classid' in document.createElement('object')) { //旧版本 IE
```

① code.google.com/p/swfobject/ http://www.awflasher.com/flash/articles/swfobj.htm
② http://jquery.lukelutman.com/plugins/flash/

```
            var paramObj = {},
            propObj = {
                id: obj.id,
                name: obj.name,
                width: obj.width,
                height: obj.height,
                "class": obj["class"],
                data: obj.src, //flash 播放器地址
                classid: "clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
            };
            for (var name in obj) {
                if (!(name in propObj)) {
                    paramObj[name] = obj[name];
                }
            }
            paramObj.movie = obj.src; //flash 播放器地址
            html = "<object " + props(propObj) + ">" + params(paramObj) + "</object>";
        } else {
            //style="width:1px;height:1px" 是为了保证 Firefox 下正常工作
            obj.style = "width:" + obj.width + "px;height:" + obj.height + "px;";
            obj.type = "application/x-shockwave-flash";
            html = "<embed " + props(obj) + "/>";
        }
        //注意，el 必须在 DOM 树中， 否则 IE8 下 flash 可能无法正常显示与工作
        el.innerHTML = html;
        return el.firstChild; //返回节点供后续操作
    }
});
```

在 Chrome 中，如果 object 标签前面有一个元素有 background-image 样式，则很有可能该 object 不显示。测试代码，保存为 html 文件，本地 chrome 打开，刷新几次会出现该现象。

```
<div style="background-image: url(xxx.png); width: 270px; height: 129px;"></div>
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
        type="application/x-shockwave-flash"
          codebase=" http://fpdownload.macromedia.com/pub/shockwave/cabs/flash/swflash.
cab#version=7,0,0,0"
          width="600"
           height="360">
        <param name="allowScriptAccess" value="always" />
        <param name="quality" value="high" />
        <param name="wmode" value="transparent" />
        <param name="movie" value=" http://imgc.zol.com.cn/small_flash_channel/donghua/
20060803qrj.swf" />
        <embed wmode="transparent"
              src=" http://imgc.zol.com.cn/small_flash_channel/donghua/20060803qrj.swf"
              quality="high"
              width="600"
              height="360"
              allowscriptaccess="always"
              type="application/x-shockwave-flash"
             pluginspage=" http://www.macromedia.com/go/getflashplayer" />
</object>
```

解决办法，改用 embed 标签加载。

最后提一下，IE 下的 object 标签的 onerror 事件是个废物，即使死链，它还是触发了 onload 回调。

7.6.3 video 标签

video 与 canvas 是 HTML5 中最重要的两个标签，它打破了 Flash 对视频与图形处理的垄断，在苹果公司宣布其 iOS 平台的 iPhone 不支持 Flash，HTML5 得以迅速发展普及。与往常一样，这些新东西一开始是作为私有实现而存在，因此兼容性问题在所难免，像播放器这么复杂的东西，存在各种各样的操作，还有视频解码器的较量，因此 video 标签成了 HTML5 最复杂、差异性最大的标签。

在 HTML5 引入一个视频大概是这样子，不考虑到任何兼容性问题。

```
<video src="myVideo.mp4" controls autoplay width="300" height="210"></video>
```

但这通常只能作梦，第一个问题就是解码器。因为我们的视频存在各种各样的格式，如 avi、rm、rmvb、mp4 等，这就要对应不同的解码器来处理。每个浏览器出于专利或捆绑的考量，自带的解码器不一样，这个 W3C 也无法协调。没有办法，引入 source 标签，用它来链接不同的格式的视频文件，让浏览器自行检测并使用第一个可辨识的格式播放。

```
<video id='video'
  controls preload='none'
  poster="http://media.w3.org/2010/05/sintel/poster.png">
  <source id='mp4'
    src="http://media.w3.org/2010/05/sintel/trailer.mp4"
    type='video/mp4'>
  <source id='webm'
    src="http://media.w3.org/2010/05/sintel/trailer.webm"
    type='video/webm'>
  <source id='ogv'
    src="http://media.w3.org/2010/05/sintel/trailer.ogv"
    type='video/ogg'>
  <p>Your user agent does not support the HTML5 Video element.</p>
</video>
```

运行效果如图 7.5 所示。

▲图 7.5

此外，我们看外国视频时可能非常倚重于字幕的支援，这就需要用到外挂字幕。trace 标签就是这样被搞出来。因此说到 HTML5 视频播放，实际动用的标签不只一个。由于 IE6～IE8 的占有份额很大，我们还需要用到 object 标签加载 flash 播放器。因此一个完整的 video 标签它的结构应

该是这样：

```
<video width="320" height="240" poster="poster.jpg" controls="controls" preload="none">
    <!-- MP4 for Safari, IE9, iPhone, iPad, Android, and Windows Phone 7 -->
    <source type="video/mp4" src="myvideo.mp4" />
    <!-- WebM/VP8 for Firefox4, Opera, and Chrome -->
    <source type="video/webm" src="myvideo.webm" />
    <!-- Ogg/Vorbis for older Firefox and Opera versions -->
    <source type="video/ogg" src="myvideo.ogv" />
    <!-- Optional: Add subtitles for each language -->
    <track kind="subtitles" src="subtitles.srt" srclang="en" />
    <!-- Optional: Add chapters -->
    <track kind="chapters" src="chapters.srt" srclang="en" />
    <!-- Flash fallback for non-HTML5 browsers without JavaScript -->
    <object width="320" height="240" type="application/x-shockwave-flash" data="flashme
diaelement.swf">
        <param name="movie" value="flashmediaelement.swf" />
        <param name="flashvars" value="controls=true&file=myvideo.mp4" />
        <!-- Image as a last resort -->
        <img src="myvideo.jpg" width="320" height="240" title="No video playback capabilities"
/>
    </object>
</video>
```

video 元素的通用 API 如表 7.4 所示。

表 7.4

特　性	属　性	事　件	方　法
src	buffered	abort	play
width	controller	canplay	pause
height	currentSrc	canplaythrough	load
poster	crossorigin	durationchange	canPlayType
autoplay	currentTime	emptied	
loop	duration	ended	
controls	ended	error	
preload	error	loadeddata	
	paused	loadedmetadata	
	mediaGroup	loadstart	
	muted	pause	
	networkState	play	
	playbackRate	playing	
	played	progress	
	readyState	ratechange	
	seekable	seeked	
	seeking	seeking	
	startOffsetTime	vstalled	
	textTracks	suspend	
	videoTracks	timeupdate	
	videoWidth	volumechange	
	videoHeight	waiting	
	volume		

特性（attributes）和属性（properties）的区别是，你不可以在 HTML 内中使用属性，而可以在 HTML 和脚本中使用特性。具体参看属性模块一章。

我们可以通过以下脚本挖掘更多新的属性，每个浏览器都有不同的私有实现。

```
window.onload = function() {
    var el = document.getElementById("video"); //指向页面中一个 video 标签
    var body = document.body
    for (var i in el) {
        if (typeof body[i] === "undefined") {
            console.log(i);
            console.log(el[i])
        }
    }
}
```

它的特性说明如下。

- src：视频文件的路径，使用子元素<source>实现更好。
- width：视频所需的宽度。默认情况下，浏览器会自动检测所提供的视频尺寸。
- height：视频所需高度。
- poster：用于指定一张图片（比如说预览图），在当前视频数据无效时显示。视频数据无效可能是视频正在加载，也可能是视频地址错误等。
- autoplay：布尔值，页面加载完成后自动播放视频。
- loop：则当视频完成播放后再次开始播放。
- controls：布尔值，显示 play/stop 按钮。
- preload：有 3 个可选值。auto，当页面加载后载入整个视频；metadata，当页面加载后只载入元数据（比如说视频的总长度与第一帧画面）；none，当页面加载后不载入视频。若指定了“autoplay”特征，则忽略该特性。

它的属性说明如下。

- buffered：返回表示音频/视频已缓冲部分的 TimeRanges 对象。
- controller：返回表示音频/视频当前媒体控制器的 MediaController 对象
- currentSrc：返回当前视频的 URL（因为我们可能对应多个 source 元素，碰大运般地试验哪个可用）
- crossorigin：如果视频文件是在不同域获得服务，那么 Crossorigin 会被用来进行指示。它适用于所有多媒体的标签，即 video、audio、img，目的是处理 Cross-Origin 资源共享（CORS）的回放问题。

在不同情境中，可以用空字符或是 CORS 设置属性关键词来声明：用户凭证或是匿名。

- currentTime：设置或返回音频/视频中的当前播放位置（以秒计）。
- videoWidth：返回当前视频本来的宽，单位为像素。
- videoHeight：返回当前视频本来的高，单位为像素。
- duration：返回当前视频的长度（以秒计）。
- ended：返回视频的播放是否已结束。

- error：返回表示视频错误状态的 MediaError 对象。
- paused：设置或返回音频/视频是否暂停。
- mediaGroup ：设置或返回音频/视频所属的组合（用于连接多个音频/视频元素）。
- muted：设置或返回音频/视频是否静音。
- networkState：返回音频/视频的当前网络状态。
- playbackRate：设置或返回视频播放的速度。1.0 是“正常速度”，小于 1.0 的值会让媒体播放得比正常速度慢，反之变快。
- played：返回表示视频已播放部分的 TimeRanges 对象。
- readyState：返回视频当前的就绪状态。
- seekable：返回表示视频可寻址部分的 TimeRanges 对象。
- seeking：返回用户是否正在视频中进行查找。
- startOffsetTime：返回表示当前时间偏移的 Date 对象。
- textTracks：返回表示可用文本轨道的 TextTrackList 对象。
- videoTracks：返回表示可用视频轨道的 VideoTrackList 对象。
- volume：设置或返回视频的音量。

它的事件说明如下。

- abort：当视频的加载已放弃时。
- canplay：当浏览器可以播放视频时。
- canplaythrough：当浏览器可在不因缓冲而停顿的情况下进行播放时。
- durationchange：当视频的时长已更改时。
- emptied：当目前的播放列表为空时。
- ended：当目前的播放列表已结束时。
- error：当在视频加载期间发生错误时。
- loadeddata：当浏览器已加载视频的当前帧时。
- loadedmetadata：当浏览器已加载视频的元数据时。
- loadstart：当浏览器开始查找视频时。
- pause：当视频已暂停时。
- play：当视频已开始或不再暂停时。
- playing：当视频在已因缓冲而暂停或停止后已就绪时。
- progress：当浏览器正在下载视频时。
- ratechange：当视频的播放速度已更改时。
- seeked ：当用户已移动/跳跃到视频中的新位置时。
- seeking：当用户开始移动/跳跃到视频中的新位置时。
- vstalled：当浏览器尝试获取媒体数据，但数据不可用时。
- suspend：当浏览器刻意不获取媒体数据时。
- timeupdate：当目前的播放位置已更改时。
- volumechange：当音量已更改时。

- waiting：当视频由于需要缓冲下一帧而停止。

它的方法说明如下。

- canPlayType()：检测浏览器是否能播放指定的视频类型。它并不是返回 boolean 值（True 或者 False）。因为视频格式非常复杂，所以这个方法的返回值有以下几种值。

1．“probably”如果你的浏览器确认可以支持你传入的视频格式。

2．“maybe”如果你的浏览器或许可以支持你传入的视频格式。

3．“”（空的字符串）如果你的浏览器确认不能支持你传入的视频格式。

- load()：重新加载视频元素。
- play()：开始播放视频。
- pause()：暂停当前播放的视频。

我们再来看一下视频格式的支持情况。现在浏览器主要围绕 MP4、WebM 和 Ogg 三种格式混战。每种格式的解码器如下。

1．MP4：使用 H264 视频、AAC 音频。

2．WebM：使用 VP8 视频、Vorbis 音频。

3．Ogg：使用 Theora 视频和 Vorbis 音频。

其中，MP4 容器、H264 视频编解码器以及 ACC 音频编解码器都是 MPEG LA Group 专利的专有格式，不是免费的。免费浏览器厂商如 Mozilla 和 Opera 强烈反对将 H264 列为 HTML5 视频标准。因为这意味着 Mozilla 和 Opera 如果要支持 HTML5 标准，那么他们必须支付相关的授权费。且不说这做法和其信条相去甚远，Mozilla 没有从其开发的浏览器上获得直接收入，反而需要付出相应成本才能自由分发其浏览器产品，Opera 则抱怨说 H264 的授权费太贵。因此两者都认为不可接受[①]。

当前浏览器的支持情况如图 7.6 所示。

浏览器	Ogg Theora	MP4	WebM
Firefox 4.0 +	✓		✓
Firefox 3.5～Firefox3.6	✓		
Safari 3 +		✓	
Chrome 6 +	✓	✓	✓
Chrome 3～Chrome5	✓	✓	
Opera 10.6 +	✓		✓
Opera 10.5	✓		
IE 9 +		✓	

▲图 7.6

如果网站用户量很大，那么建议使用 source 标签都引入这三种格式（因为 iphone、ipad），并且与 object 标签的 flash 播放器一起使用。此标签包含 src、type、media 三个属性。

- src：用于指定视频的地址，和 video 标签的一样。
- media：用于说明视频在何种设备中使用，不设置时默认值为 all。
- type：用于说明 src 属性指定视频的类型，帮助浏览器在获取视频前判断是否支持此类别的

① 当然，其他公司也有自己的想法而不接受某些格式，详见这里 http://www.ruanyifeng.com/blog/2010/05/html5_codec_fight.html 具体如图 7.6 所示。

媒体格式。常见的值有 video/wmv，video/ogg，video/mp4，video/webm，video/m4v。

到这里，有关它的基础知识终于介绍完了，我们再看兼容问题。首当其冲的是全屏问题。打开全屏与取消全屏这两个 API 实现得比较晚，我们需要检测它是否存在才能调用。

```
var elem = document.getElementById("video");
if (elem.requestFullscreen) {
    elem.requestFullscreen();
} else if (elem.mozRequestFullScreen) {
    elem.mozRequestFullScreen();
} else if (elem.webkitRequestFullscreen) {
    elem.webkitRequestFullscreen();
}
```

取消全屏也是这么麻烦。

```
var elem = document.getElementById("video");
if (elem.cancelFullscreen) {
    elem.cancelFullscreen();
} else if (elem.mozCancelFullscreen) {
    elem.mozCancelFullscreen();
} else if (elem.webkitCancelFullscreen) {
    elem.webkitCancelFullscreen();
}
```

上面的代码不能直接用，它必须放到事件回调中才有效，比如点击事件。若忘了，firebug 会提醒你的——“全屏请求被拒绝，因为 Element.mozRequestFullScreen()不是在一个短期运行的由用户引发的事件处理代码段中运行的”。

```
button.onclick = function(){
   //打开全屏与取消全屏的逻辑
}
```

我们还可以通过 document.fullScreen 来判定浏览器是否处于全屏状态，遗憾的是它还没有标准化。

```
function isDocumentInFullScreenMode() {
    // 过去由 F11 触发的那种浏览器全屏模式和 HTML5 中内容的全屏模式是不一样的
    return (document.fullScreenElement && document.fullScreenElement !== null)
    || (!document.mozFullScreen && !document.webkitIsFullScreen);
}
```

在 iPad Safari 一个早期版本中，利用 innerHTML 与 document.createElement 创建 video 元素，无法进行播放，唯有使用 document.write 才有效。

同样在 iPad 上，video 的菜单没有音量控制条，并且你也无法通过 volume 属性来调节它，音量控制只能使用硬件按钮实现。

在可访问性上，IE、Firefox 和 Opera 上能够使用键盘控制 HTML5 视频元素。IE/Firefox 考虑了所有的视频标签，使用空格控制视频的播放/暂停，通过左/右和上/下键控制进度和音量。Opera 中可以使用 tab 键控制所有的控件。

此外就没有更多难解的 Bug 了，只是涉及的属性与方法太多了，变成苦力活了。

第8章 数据缓存系统

数据缓存系统最早应该是jQuery1.2引入的，它是用于关联操作对象和与之相关的数据的一种机制。在DOM中，我们通常操作的数据有3种，元素节点、文档对象与window对象。那时jQuery的事件系统完成照搬DE大神的addEvent.js，而addEvent在实现有个缺憾，它把事件的回调都放到EventTarget之上，这会引发循环引用，如果EventTarget是window对象，又会引发全局污染。有了数据缓存系统，除了规避这两个风险外，我们还可以有效地保存不同方法产生的中间变量，而这些变量会对另一个模块的方法有用，解耦方法间的依赖。对于jQuery来说，它的事件克隆乃至后来的队列实现都离不开缓存系统。

数据缓存系统经过这么多年的发展，总共衍生了4种形态。

（1）属性标记法。

（2）数组索引法。

（3）valueOf重写法。

（4）WeakMap关联法。

8.1 jQuery的第1代缓存系统

jQuery1.2在core模块新增了两个静态方法， data与removeData。data不用说，与jQuery其他方法一样，读写结合。jQuery的缓存系统是把所有数据都放$.cache仓库之上，然后为每个要使用缓存系统的元素节点、文档对象与window对象分配一个UUID。UUID的属性名为一个随机的自定义属性，"jQuery" + (new Date()).getTime()， 值为整数，从零递增。但UUID总要附于一个对象上，如果那个对象是window，岂不是全局污染吗？因此jQuery内部判定它是window对象时，映射为一个叫windowData的空对象，然后UUID加在它之上。有了UUID，我们在首次访问缓存系统时，会在$.cache对象开辟一个空对象（缓存体），用于放置与目标对象有关的东西。这有点像银行开户，UUID的值就是存折。removeData则会删掉不再需要保存数据，如果到最后，数据删光了，它也没有任何键值对，成为空对象，jQuery就会从$.cache中删掉此对象，并从目标对象移除UUID。

```
//jQuery1.2.3
var expando = "jQuery" + (new Date()).getTime(), uuid = 0, windowData = {};
jQuery.extend({
   cache: {},
```

```
    data: function( elem, name, data ) {
        elem = elem == window ? windowData :   elem;//对 window 对象做特别处理
        var id = elem[ expando ];
        if ( !id ) //如果没有 UUID 则新设一个
            id = elem[ expando ] = ++uuid;
        //如果没有在$.cache 中开户，则先开户
        if ( name && !jQuery.cache[ id ] )
            jQuery.cache[ id ] = {};

        // 第三个参数不为 undefined 时，为写操作
        if ( data != undefined )
            jQuery.cache[ id ][ name ] = data;
        //如果只有一个参数，则返回缓存对象，两个参数则返回目标数据
        return name ?  jQuery.cache[ id ][ name ] :   id;
    },

    removeData: function( elem, name ) {
        elem = elem == window ? windowData :   elem;
        var id = elem[ expando ];
        if ( name ) {//移除目标数据
            if ( jQuery.cache[ id ] ) {
                delete jQuery.cache[ id ][ name ];
                name = "";

                for ( name in jQuery.cache[ id ] )
                    break;
                //遍历缓存体，如果不为空，那 name 会被改写，如果没有被改写，则!name 为 true
                //从而引发再次调用此方法，但这次是只传一个参数，移除缓存体
                if ( !name )
                    jQuery.removeData( elem );
            }
        } else {
            //移除 UUID，但 IE 下对元素使用 delete 会抛错
            try {
                delete elem[ expando ];
            } catch(e){
                if ( elem.removeAttribute )
                    elem.removeAttribute( expando );
            }//注销账户
            delete jQuery.cache[ id ];
        }
    }
})
```

jQuery 在 1.2.3 中添加了两个同名的原型方法 data 与 removeData，目的是方便链式操作与集化操作。并在 data 中添加 getData、setData 的自定义事件的触发逻辑。

jQuery1.3 中，数据缓存系统终于独立成一个模块 data.js（内部开发时的划分），并添加了两组方法，命名空间上的 queue 与 dequeue，原型上的 queue 与 dequeue。queue 的目的很明显，就是缓存一组数据，为动画模块服务。dequeue 是从一组数据中删掉一个。

```
//jQuery1.3
```

```
jQuery.extend({
    queue: function( elem, type, data ) {
        if ( elem ){
            type = (type || "fx") + "queue";
            var q = jQuery.data( elem, type );
            if ( !q || jQuery.isArray(data) )//确保储存的是一个数组
                q = jQuery.data( elem, type, jQuery.makeArray(data) );
            else if( data )//然后往这个数据加东西
                q.push( data );
        }
        return q;
    },
    dequeue: function( elem, type ){
        var queue = jQuery.queue( elem, type ),
        fn = queue.shift();//然后删掉一个，早期它是放置动画的回调，删掉它就调用一下
        // 但没有做是否为函数的判定，估计也没有写到文档中，为内部使用
        if( !type || type === "fx" )
            fn = queue[0];
        if( fn !== undefined )
            fn.call(elem);
    }
})
```

fx 模块 animate 方法的调用示例如下。

```
//each 是并行处理多个动画,queue 是一个接一个处理多个动画
this[ optall.queue === false ? "each" : "queue" ](function(){ /*略*/})
```

在元素上添加自定义属性，还会引发一个问题。如果我们对这个元素进行拷贝，就会将此属性也复制过去，导致两个元素都有相同的 UUID 值，出现数据被错误操作的情况。jQuery 早期的复制节点实现非常简单，如果元素的 cloneNode 方法不会复制事件就使用 cloneNode，否则使用元素的 outerHTML，或父节点的 innerHTML，用 clean 方法解析一个新元素出来。但 outerHTML 与 innerHTML 都会将显式属性写在里面，因此需要用正则把它们清除掉。

```
//jQuery1.3.2 core.js clone 方法
var ret = this.map(function(){
    if ( !jQuery.support.noCloneEvent && !jQuery.isXMLDoc(this) ) {
        var html = this.outerHTML;
        if ( !html ) {
            var div = this.ownerDocument.createElement("div");
            div.appendChild( this.cloneNode(true) );
            html = div.innerHTML;
        }

        return  jQuery.clean([html.replace(/  jQuery\d+="(?:\d+|null)"/g,  "").replace
(/^\s*/, "")])[0];
    } else
        return this.cloneNode(true);
});
```

在 jQuery1.4 中，发现它已对 object、embed、applet 这 3 种元素进行特殊处理，缘由是这 3 个

元素是用于加载外部资源的，如 flash、silverlight、media Play、realone Player、windows 自带的日历组件和颜色选择器等。在旧版本 IE 中，元素节点只是 COM 的包装，一旦引入这些资源后，它就会变成那种资源的实例。一旦这资源是由 VB 等语言编写的，由于 VM 有严格的访问控制，不能随便给对象添加新属性与方法，就会遇到抛错的可能。因此 jQuery 做出一个决定，对这 3 种元素，就不为它们缓存数据。jQuery 在内部弄了一个叫 noData 的对象，专门放置它们的 tagName。

```
noData: {
    "embed": true,
    "object": true,
    "applet": true
},
//代码防御
if ( elem.nodeName && jQuery.noData[elem.nodeName.toLowerCase()] ) {
    return;
}
```

jQuery1.4 还对$.data 进行改进，允许第二个参数为对象，方便储存多个数据。UUID 对应的自定义属性 expando 也放进命名空间之下了。queue 与 dequeue 方法被剥离成一个新模块。

jQuery1.43 带来 3 项改进。

首先是添加 changeData 自定义方法。不过这套方法没有什么销量，只是产品经理的自恋吧。

其次，检测元素节点是否支持添加自定义属性的逻辑被独立成一个叫 acceptData 的方法。因为 jQuery 团队发现当 object 标签加载的 flash 资源，它还是可以添加自定义属性的，于是决定对这种情况网开一面。IE 在加载 flash 时，需要对 object 指定一个叫 classId 的属性，值为 clsid:D27CDB6E-AE6D-11cf-96B8-444553540000，因此检测逻辑就变得非常复杂，由于 data、removeData 都要用到，独立出来可有效节省比特。

最后 HTML5 对人们随便添加自定义属性的行为做出回应，新增一种叫"data-*"的缓存机制。当用户设置的属性以"data-"开头，它们会被保存到元素节点的 dataset 对象上。于是 jQuery 团队又有一个主意，允许人们通过设置 data-*来配置 UI 组件，于是他们对 data-*进行如下增强：当用户第一次访问此元素节点，会遍历它所有"data-"开头的自定义属性（为了照顾旧版本 IE，不能直接遍历 dataset），把它们放到 jQuery 的缓存体中。那么当用户取数据时，会先从缓存系统中，没有再使用 setAttribute 访问"data-"自定义属性。但 HTML5 的缓存系统非常弱，只能保存字符串（这当然是出于循环引用的考量），于是 jQuery 会将它们还原为各种数据类型，如"null",, "false", "true"变成 null, false, true, 符合数字格式的字符串会转换成数字，如果它是以"{"开头"}"结尾则尝试转成一个对象。

```
//jQuery1.43 $.fn.data
rbrace = /^(?:\{.*\}|\[.*\])$/;
if ( data === undefined && this.length ) {
    data = jQuery.data( this[0], key );
    if ( data === undefined && this[0].nodeType === 1 ) {
        data = this[0].getAttribute( "data-" + key );

        if ( typeof data === "string" ) {
            try {
                data = data === "true" ? true :
                    data === "false" ? false :
```

```
                data === "null" ? null :
                !jQuery.isNaN( data ) ? parseFloat( data ) :
                rbrace.test( data ) ? jQuery.parseJSON( data ) :
                data;
            } catch( e ) {}

        } else {
            data = undefined;
        }
    }
}
```

jQuery1.5 也带来三项改进。当时 jQuery 已经在 1.42 版打败 Prototype.js，用户量暴增。它的重点改为提升性能，进入 FBUG 阶段（用户多，相当于免费的测试员就越多，测试覆盖面就越大）。

改进 expando，原来是基于时间截，现在是版本号加随机数。因此用户可能在一个页面引入多个版本的 jQuery。

是否有此数据的逻辑被抽出成一个 hasData 方法，处理 HTML5 的"data-*"属性也被抽出成一个私有方法 dataAttr。它们都是为了逻辑显得更清晰。dataAttr 使用 JSON.parse，由于这个 JSON 可能是 JSON2.js 引入的，而 JSON2.js 有个非常糟糕的地方，就是为一系列原生类型添加了 toJSON 方法，导致 for in 循环判定是否为空对象出错。jQuery 被逼搞了个 isEmptyDataObject 方法做处理。

jQuery 的数据缓存系统本来就是为事件系统服务而分化出来的，到后来，它是内部众多模块的基础设施。换言之，它是供框架自己内部使用的，但一旦它公开到文档中，不可避免地，用户会使用 data 方法来保存他们在工作业务中用到的数据。因此这两类数据可能就有互相覆盖的危险，私有数据（框架使用的）与用户数据（用户自己使用的），你不能设置一个优先级来阻止它们的互相覆盖的，因为没有阻止私有数据，可能框架的一些部件就不能运作，比如事件系统在每个元素的缓存体上设置的 events 对象。而你让用户设置的数据莫名其妙不能生效，这也是无法让人接受的。因此早期 jQuery 作出的让步是，框架使用的私有数据的属性名会尽可能的生僻复杂，尽量减少重名的可能，比如__class__、__change__、_submit_attached、_change_attached。

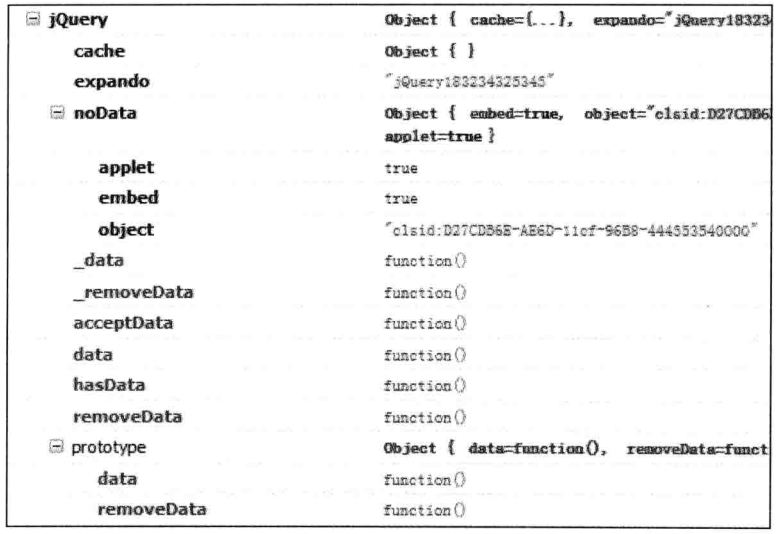
但当 jQuery UI 越来越庞大时，它们对数据缓存的依赖也越发严重。同样的压力也来自普通用户。当 jQuery 成为世界级的著名框架后，用户数据与私有数据发生冲突的几率也就大大增加。jQuery1.5 对缓存体进行改造。原来就是一个对象，什么数据都往里面抛。现在它就这个缓存体内开辟一个子对象，键名为随机的 jQuery.expando 值，如果是私有数据就存到里面去。但 events 私有数据为了向前兼容起见，还是直接放到缓存体之上。至于，如何区分私有数据，非常简单，直接在 data 方法添加第四个参数，真值时为私有数据。removeData 时也相应提供第三个参数，用于删除私有数据。还新设了一个_data 方法，专门用于操作私有数据。下面就是缓存体的结构图。

```
var cache = {
    jQuery14312343254:{/*放置私有数据*/},
    events: {/*放置事件名与它对应的回调列表*/}
    /*这里放置用户数据*/
}
```

jQuery1.7 对缓存体做了改进，系统变量变放置 data 对象中，为此判定缓存体为空也要做相应的改进，现在要跳过 toJSON 与 data。新结构如下。

```
var cache = {
    data:{/*放置用户数据*/}
    /*这里放置私有数据*/
}
```

jQuery1.8 曾添加一个叫 deleteIds 的数组，用于重用 UUID，但昙花一现。UUID 的值从 1.8 起不用 jQuery.uuid 的了，改用 jQuery.guid 递增生成。现在光是缓存系统就是一个庞大家族，如图 8.1 所示。



▲图 8.1

8.2 jQuery 的第 2 代缓存系统

第 2 代缓存系统的作者为 Rick Waldron。依照他的话，要实现以下 6 个目标。

（1）在接口与语义上兼容 1.9.x 分支。

（2）通过简化储存路径为统一的方式来提高维护性。

（3）使用相同的机制来实现“私有”与“用户”数据。

（4）不再把私有数据与用户数据混在一起。

（5）不再在用户对象上添加自定义属性。

（6）方便以后可以平滑地利用 WeakMap 对象进行升级（WeakMap 相关规范大致在 2014 年完成）。

jQuery 第 2 代缓存系统的实现方法是 valueOf 重写！具体原理是，如果目标对象的 valueOf 传入一个特殊的对象，那么它就返回一个 UUID，然后通过 UUID 在 Data 实例的 cache 对象属性上开

辟缓存体。这样一来，我们就不用区分它是 window 对象，使用 windowData 来做替身了；另外，我们也不用顾忌 embed、object、applet 这 3 种在 IE 下可能无法设置私有属性的元素节点，达成第 2、第 5 个目标。

在第 1 代缓存系统中，每个缓存体的结构为一个拥有 data 对象属性的对象，data 对象属性里面放用户数据，而缓存体的其他键值用于对应私有数据。第 2 代则在框架内部添加了一个 Data 类，它的实例有一个 cache 属性，私有数据与用户数据分别由一个 Data 实例来维护，这就达成第 3 个、第 4 个目标。

```
function Data() {
    this.cache = {};
}

Data.uid = 1;

Data.prototype = {
    locker: function(owner) {
        var ovalueOf,
        //owner 为元素节点、文档对象、window 对象
        //首先我们检测一下它们 valueOf 方法有没有被重写，由于浏览器的差异性，
        //我们通过觅得此三类对象的构造器进行原型重写的成本过大，只能对每一个实例的 valueOf 方法进行重写。
        //检测方式为传入 Data 类，如果是返回"object"说明没有被重写，返回"string"则是被重写。
        //这个字符串就是我们上面所说的 UUID，用于在缓存仓库上开辟缓存体。
        unlock = owner.valueOf(Data);
        //这里的重写使用了 Object.defineProperty 方法，因为在这个版本 jQuery 不打算往下兼容 IE6～IE8
        //Object.defineProperty 的第 3 个参数为对象，如果不显示设置 enumerable、writable、
        //configurable，则会默认为 false，这也正如我们所期待的那样，我们不再希望人们来遍历它，重写它，
        //再次动它的配置
        //这个过程被 jQuery 称之为开锁，通过 valueOf 这扇大门，进入到仓库
        if(typeof unlock !== "string") {
            unlock = jQuery.expando + Data.uid++;
            ovalueOf = owner.valueOf;

            Object.defineProperty(owner, "valueOf", {
                value: function(pick) {
                    if(pick === Data) {
                        return unlock;
                    }
                    return ovalueOf.apply(owner);
                }

            });
        }
        //接下来就是开辟缓存体
        if(!this.cache[unlock]) {
            this.cache[unlock] = {};
        }

        return unlock;
    },
    set: function(owner, data, value) {
```

```
        //写方法
        var prop, cache, unlock;
        //得到 UUID 与缓存体
        unlock = this.locker(owner);
        cache = this.cache[unlock];
        //如果传入 3 个参数，第 2 个为字符串，那么直接在缓存体上添加新的键值对
        if(typeof data === "string") {
            cache[data] = value;
            //如果传入 2 个参数，第 2 个为对象
        } else {
            //如果缓存体还没有添加过任何对象，那么直接赋值，否则使用 for in 循环添加新键值对
            if(jQuery.isEmptyObject(cache)) {
                cache = data;
            } else {
                for(prop in data) {
                    cache[prop] = data[prop];
                }
            }
        }
        this.cache[unlock] = cache;

        return this;
    },
    get: function(owner, key) {
        //读方法
        var cache = this.cache[this.locker(owner)];
        //如果只有一个参数，则返回整个缓存体
        return key === undefined ? cache : cache[key];
    },
    access: function(owner, key, value) {
        //决定是读方法或是写方法，然后做相应操作
          if (key === undefined ||
              ((key && typeof key === "string") && value === undefined)) {
           return this.get(owner, key);
        }
        this.set(owner, key, value);
        return value !== undefined ? value : key;
    },
    remove: function(owner, key) {
        //略，与第 1 代差不多
    },
    hasData: function(owner) { //判定此对象是否缓存了数据
        return !jQuery.isEmptyObject(this.cache[this.locker(owner)]);
    },
    discard: function(owner) { //删除它的用户数据与私有数据
        delete this.cache[this.locker(owner)];
    }
};
var data_user, data_priv;

function data_discard(owner) {
    data_user.discard(owner);
    data_priv.discard(owner);
```

```
}

data_user = new Data();
data_priv = new Data();
```

接下来就简单了，暴露给用户调用的方法都只是一个空壳，用来转交给 data_user、date_priv 这两个实例对象处理，并且私有数据的处理再也不通过用户数据的渠道了。

```
jQuery.extend({
    // UUID
    expando: "jQuery" + ( core_version + Math.random() ).replace( /\D/g, "" ),

    //用于向前兼容
    acceptData: function() {
        return true;
    },

    hasData: function( elem ) {//判定是否缓存了数据
        return data_user.hasData( elem ) || data_priv.hasData( elem );
    },

    data: function( elem, name, data ) {//读/写用户数据
        return data_user.access( elem, name, data );
    },

    removeData: function( elem, name ) {//删除用户数据
        return data_user.remove( elem, name );
    },
    _data: function( elem, name, data ) {//读/写私有数据
        return data_priv.access( elem, name, data );
    },

    _removeData: function( elem, name ) {//删除私有数据
        return data_priv.remove( elem, name );
    }
});
```

重写 valueOf 这一招的确漂亮，因此任何非纯空对象都有 valueOf 方法，然后通过闭包保存用于关联缓存仓库的 UUID。但闭包也意味着特吃内存，这是此系统的最大缺憾（本章完成时，jQuery 又在利用 defineProperty 开发下一代缓存系统）。

8.3 mass Framework 的第 1 代数据缓存系统

mass Framework 的一个特色是兼容 jQuery 90%的 API，因此它也必然存在数据缓存系统。可以说，它的缓存系统就是 jQuery 的改良版。其中最大的改进是关联方式的不同。

为了建立目标对象与缓存体的联系，jQuery 选择了目标对象添加一个自定义属性，结果在旧版本 IE 遭遇狙击，被逼奠出 noData。mass Framework 则挖掘出元素节点的 uniqueNumber 属性，这是 IE 的私有实现，不过用于对付旧版本 IE 的 object、applet、embed 标签够了，标准浏览器很好打

发。这个添加自定义属性，取得 UUID 值的操作封装在 mass.js 的 getUid 方法中。

```
//主要用于建立一个从元素到数据的引用，具体用于数据缓存、事件绑定、元素去重
getUid: global.getComputedStyle ? function( obj ){//IE9+,标准浏览器
    return obj.uniqueNumber || ( obj.uniqueNumber = NsVal.uuid++ );
}: function( obj ){
    if(obj.nodeType !== 1){//如果是普通对象、文档对象、window 对象
        return obj.uniqueNumber || ( obj.uniqueNumber = NsVal.uuid++ );
    }//注：旧版本 IE 的 XML 元素不能通过 el.xxx = yyy 设置自定义属性
    var uid = obj.getAttribute("uniqueNumber");
    if ( !uid ){
        uid = NsVal.uuid++;
        obj.setAttribute( "uniqueNumber", uid );
    }
    return +uid;//确保返回数字
},
```

然后其他部分与 jQurery1.8.3 的缓存系统差不多，只是在 acceptData 与转换 HTML5 的"data-*"自定义属性上略有不同。acceptData 不做 noData 检测，因为 IE 下是取原有的属性，不添加自定义属性。只对对象类型进行检测，因为 IE 为文本节点、注释节点添加自定义属性会出错，不为它们缓存数据。转换自定义属性时，统一使用 eval，避开引入 JSON.js 的 BUG。

```
define("data", ["$lang"], function() {
    $.log("已加载 data 模块", 7);
    var remitter = /object|function/, rtype = /[^38]/;
    function innerData(target, name, data, pvt) {//IE6～IE8 不能为文本节点注释节点添加数据
        if ($.acceptData(target)) {
            var id = $.getUid(target), isEl = target.nodeType === 1,
                getOne = typeof name === "string", //取得单个属性
                database = $["@data"],
                table = database[ id] || (database[ id ] = {
                data: {}
            });
            var cache = table;
            //私有数据都是直接放到 table 中，普通数据放到 table.data 中
            if (!pvt) {
                table = table.data;
            }
            if (name && typeof name === "object") {
                $.mix(table, name);//写入一组属性
            } else if (getOne && data !== void 0) {
                table[ name ] = data;//写入单个属性
            }
            if (getOne) {
                if (name in table) {
                    return table[name]
                } else if (isEl && !pvt) {
                    //对于用 HTML5 data-*属性保存的数据， 如<input id="test" data-full-name=
                    "Planet Earth"/>
                    //我们可以通过$("#test").data("full-name")或$("#test").data("fullName")
                    访问到
```

```
                return $.parseData(target, name, cache);
            }
        } else {
            return table
        }
    }
}
function innerRemoveData(target, name, pvt) {
    if ($.acceptData(target)) {
        var id = $.getUid(target);
        if (!id) {
            return;
        }
        var clear = 1, ret = typeof name == "string",
                database = $["@data"],
                table = database[ id ],
                cache = table;
        if (table && ret) {
            if (!pvt) {
                table = table.data
            }
            if (table) {
                ret = table[ name ];
                delete table[ name ];
            }
            loop://判定节点为空
                    for (var key in cache) {
                if (key == "data") {
                    for (var i in cache.data) {
                        clear = 0;
                        break loop;
                    }
                } else {
                    clear = 0;
                    break loop;
                }
            }
        }
        if (clear) {
            try {
                delete database[id];
            } catch (e) {
                database[id] = void 0;
            }
        }
        return ret;
    }
}

$.mix({
    "@data": {},
    acceptData: function(target) {
        return target && remitter.test(typeof target) && rtype.test(target.nodeType);
    },
    hasData: function(target) {
```

```
            var cache = $.data(target),name;
            for (name in cache) {
                if (name === "data" && $.isEmptyObject(cache[name])) {
                    continue;
                }
                if (name !== "toJSON") {
                    return false;
                }
            }
            return true;
        },
        data: function(target, name, data) {  // 读/写用户数据
            return innerData(target, name, data)
        },
        _data: function(target, name, data) {//读/写私有数据
            return innerData(target, name, data, true)
        },
        removeData: function(target, name) {  //删除用户数据
            return innerRemoveData(target, name);
        },
        _removeData: function(target, name) {//删除私有数据
            return innerRemoveData(target, name, true);
        },
        parseData: function(target, name, table, value) {
            //略，处理 HTML5 data-*属性
        },
        mergeData: function(cur, src) {
            //略
        }
    });
});
```

从上面的代码我们可以看到，所有的实现都移到内部，接口只是个空壳。如果我们阅读大量 jQuery 插件，特别是与 UI 相关的，可以见到这种做法是很常见的。无论里面是多么乱，外面都是非常规整统一的接口，方便用户就行了。从实现来看，它与 jQuery 第 1 代缓存系统的出法点都是一样，通过目标上的某个属性来储存 UUID，从来打开进入缓存仓库的大门。

8.4 mass Framework 的第 2 代数据缓存系统

属性标记法的缺点是很明显的，要根据目标对象的不同（jQuery 方式）与浏览器的不同（mass Framework 方法）做出不同的处理，而且清理这些自定义属性又非常麻烦。jQuery 第 2 代缓存系统采取与空间换时间的方式将 UUID 内嵌到目标对象的 valueOf 方法中。mass Framework 第 2 代则是数组索引法，虽然也够占空间，建立两个数组，一个用于装目标对象，另一个在它对应的位置上放它的缓存体，这样就不用对目标对象做任何改动。从而，我们的目标对象可以扩展到任何数据类型，并且即使目标对象作用了 Object.preventExtensions、Object.seal、Object.freeze，我们也能得到 UUID。

下面就是其骨干代码。

```
//https://github.com/RubyLouvre/mass-Framework/blob/1.2/data.js
define("data", ["lang"], function($) {
    var owners = [],
        caches = [];

    function add(owner) {
        var index = owners.push(owner);
        return caches[index - 1] = {
            data: {}
        };
    }

    function innerData(owner, name, data, pvt) {
        var index = owners.indexOf(owner);
        var table = index === -1 ? add(owner) : caches[index];//得到要操作的缓存体
        //略
    }

    function innerRemoveData(owner, name, pvt) {
        var index = owners.indexOf(owner);
        if (index > -1) {
            //略
        }
    }
    $.mix({
        hasData: function(owner) {
            //判定是否关联了数据
            return owners.indexOf(owner) > -1;
        },
        data: function(target, name, data) {  // 读/写用户数据
            return innerData(target, name, data)
        },
        _data: function(target, name, data) {//读/写私有数据
            return innerData(target, name, data, true)
        },
        removeData: function(target, name) {  //删除用户数据
            return innerRemoveData(target, name);
        },
        _removeData: function(target, name) {//删除私有数据
            return innerRemoveData(target, name, true);
        },
        parseData: function(target, name, cache, value) {
            //略 处理 HTML5 data-*属性
        },
        mergeData: function(cur, src) {
            //略
        }
    });
    return $;
});
```

在此方法下，我们判定目标对象是否关联了数据是非常简洁的。

8.5 mass Framework 的第 3 代数据缓存系统

到目前为止，我们都是通过 UUID 的方式来建立目标对象与缓存体之间的连接，作为纽带的 UUID 或通过自定义属性保存，或通过 valueOf 方法闭包引用着，或通过遍历数组计算出来，凡此种种，把这个 UUID 东藏西藏，有没有更好的方式呢？于是 WeakMap 登场了。

WeakMap 是 es6 带来的新集合对象，因此不是每个浏览器都可以使用。Mass Framework 也只是将它实现在 data-neo 这个标识为实验性质的模块内。目前只有 FF6+无障碍地使用它，Chrome 需要手动开启 ECMAScript 6 特性：打开 chrome://flags，勾选“启用实验性 JavaScript”。

简单地介绍一下 WeakMap。平时我们的 JavaScript 对象，键名只能为字符串，键值任意，我们可以通过 for in 循环遍历它的所有键值对（除非此属性的 configurable 为 false）。而 WeakMap 的键名只能为一个非 null 的对象，键值任意，此外它还是一个不透明的对象，我们无法通过 for in 循环遍历它里面的键值对，读写或删除键值对都只能过它暴露的接口进行。到目前为止，它就只有 set、get、has、delete 这 4 个方法。另外，它还有一个最大的好处，它的键名与键值是一种弱引用关系，如果作为键名的对象被删除，那么它对应的缓存体也自动被清除出 WeakMap 对象，从而方便 CG 回收。

```
var map = new WeakMap(), el = document.body
map.set(el, { data:{} });//设置新键值对
var value = map.get(el);//读取目标值
console.log(value);  //{ data:{} }
console.log( map.has(el) );//判定是否存在此键名
map.delete( el ) //删除键值对
```

有了 WeakMap 就好办，因为我们的目标对象也不过是元素节点、文档对象或 window 对象，完全满足作为 WeakMap 的键名的要求。我们直接把缓存仓库改成一个 WeakMap 实例，$["@data"] = new WeakMap;。我们也再不用 UUID 作为桥梁关联两者，直接交给它的 set 方法处理就行了。判定目标对象是否关联着缓存体也变得超简单，直接调用 has 方法，删除缓存体同样如此，有现成 delete 方法使用。WeakMap 真可谓是为缓存系统量身打造的东西。

```
// https://github.com/RubyLouvre/mass-Framework/blob/1.2/data_neo.js
define("data", ["lang"], function($) {
    var caches = new WeakMap;//FF6+
    function innerData(owner, name, data, pvt) {
        var table = caches.get(owner);
        if (!table) {//得到要操作的缓存体
            table = {
                data: {}
            }
            caches.set(table);
        }
        //略
    }
    function innerRemoveData(owner, name, pvt) {
        var table = caches.get(owner);
```

```
        if (!table) {
            return;
        }
        //略
    }

    $.mix({
        hasData: function(target) {
            return caches.has(target);//判定是否关联了数据
        },
        data: function(target, name, data) {  // 读/写用户数据
            return innerData(target, name, data);
        },
        _data: function(target, name, data) {//读/写私有数据
            return innerData(target, name, data, true);
        },
        removeData: function(target, name) {  //移除用户数据
            return innerRemoveData(target, name);
        },
        _removeData: function(target, name) {//移除私有数据
            return innerRemoveData(target, name, true);
        },
        parseData: function(target, name, table, value) {
            //略 处理 HTML5 data-*属性
        },
        mergeData: function(cur, src) {
            //略
        }
    });
});
```

8.6 总结

说到底，数据缓存就是在目标对象与缓存体间建立一对一的关系，然后在缓存体上操作数据，复杂度都集在前者。从软件设计原则上看，这也是最好的结果（吻合 KISS 原则与职责单一则）。

第9章 样式模块

样式模块大致分为两大块，精确获取样式值与设置样式、精确是用于修饰获取的。由于样式分为外部样式、内部样式与行内样式，再加个 important 对选择器的权重的干扰，我们实际很难看到元素是应用了哪些样式规则。因此样式模块，80%的比重在于获取这一块。像什么 offset、滚动条也归入这一块。

大体上，我们在标准浏览器是使用 getComputedStyle，IE6～IE8 下使用 currentStyle 来获取元素的精确样式。不过 getComputedStyle 并不挂在元素上，而是 window 的一个 API，它返回一个对象，可以选择使用 getPropertyValue 方法传入连字符风格的样式名取得其值，或者属性法+驼峰风格的样式名去取值，但考虑到 currentStyle 也是使用属性法+驼峰风格，我们就统一使用后者。

```
var getStyle = function(el, name) {
    if (el.style) {
        name = name.replace(/\-(\w)/g, function(all, letter) {
            return letter.toUpperCase();
        });
        if (window.getComputedStyle) {
            //getComputedStyle 的第二个伪类是用于对付伪类的，如滚动条，placeholder，
            //但 IE9 不支持，因此我们只管元素节点，上面的 el.style 过滤掉了
            return el.ownerDocument.getComputedStyle(el, null)[ name ]
        } else {
            return el.currentStyle[ name ];
        }
    }
}
```

设置样式则更没难度，直接 el.style[name] = value 搞定。

但框架要考虑的东西很多，如兼容性、易用性、扩展性，现列举如下。

（1）样式名要同时支持连字符风格（CSS 的标准风格）与驼峰风格（DOM 的标准风格）。

（2）样式名要进行必要的处理，如 float 样式与 CSS3 带私有前缀的样式。

（3）如果框架是仿 jQuery 风格，要考虑 set all get first。

（4）设置样式时，对于长度宽度可以考虑直接处理数值，由框架智能补上"px"单位。

（5）设置样式时，对于长度宽度可以考虑传入相对值，如"-=20"。

（6）对个别样式的特殊处理，如 IE 下的 z-index, opacity、user-select、background-position、top、left。

（7）基于 setStyle、getStyle 的扩展，如 height、width、offset 等方法。

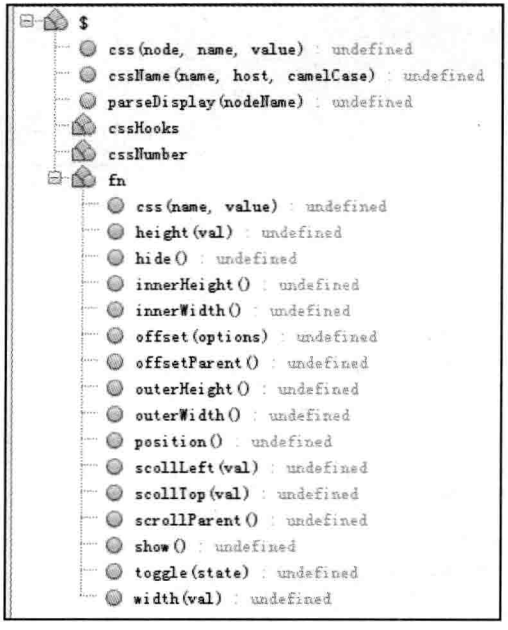
本章将围绕 mass Framework 的 css 模块与 css_fix 模块进行展开。涵盖的内容大致相当于 jQuery 的 css、offset、demensions 模块，或相当于 EXT4 的 Element.style Element.scroll、 Element.position 模块。大家可到这里下载。

https://github.com/RubyLouvre/mass-Framework/blob/master/css.js

https://github.com/RubyLouvre/mass-Framework/blob/master/css_fix.js

9.1 主体结构

mass Framework 是分两个模块来处理样式的，其中 css_fix 用于兼容旧版本 IE。涉及的 API 有 css、cssName、cssNumber、cssHooks、height、width、innerHeight、innerWidth、outerHeight、innerWidth、offset、position、scrollTop、scrollLeft、show、hide、toggle、offsetParent、scrollParent。其中放在$ 上的为静态方法，放在$.fn 上的为原型方法，独立于这两者的是私有方法或对象，如图 9.1 所示。

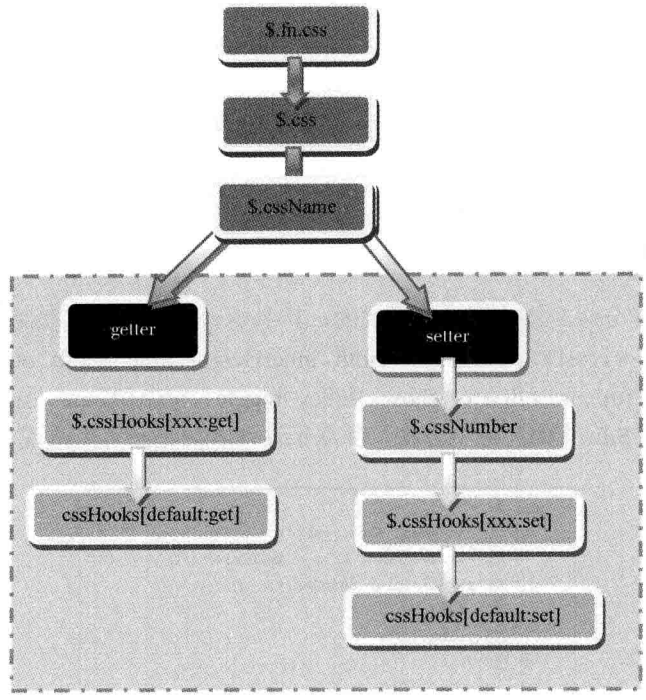


▲图 9.1

其中最重要的几个方法，它们之间的依赖如图 9.2 所示。

其他方法基本上就依赖于$.fn.css、$.fn.css 或$.cssHooks["default:get"]。

在 css 模块中，几乎所有操作都可以追溯到 cssHooks 这个钩子对象上。通过它里面的方法摆平浏览器的各种奇葩样式。这些方法根据传参的不同，分为读方法与写方法，工整地排列在对象的内部。对于不必特意设置或获取的样式值，由交由默认的写方法(cssHooks["defaults:set"])、读方法（cssHooks["defaults:get"]）来处理。默认写方法在各个浏览器是一致的，读方法则分 IE 与 W3C 两个版本。



▲图 9.2

```
var adapter = $.cssHooks = {};
adapter["_default:set"] = function(node, name, value) {
    node.style[name] = value;
};

if (window.getComputedStyle) {
    $.getStyles = function(node) {
        return window.getComputedStyle(node, null);
    };
    adapter["_default:get"] = function(node, name, styles) {
        var ret, width, minWidth, maxWidth
        styles = styles || getStyles(node);
        if (styles) {
            ret = name === "filter" ? styles.getPropertyValue(name) : styles[name]
            var style = node.style;
            //这里只有 Firefox 与 IE10 会智能处理未插入 DOM 树的节点的样式,它会自动找内联样式
            if (ret === "" && !$.contains(node.ownerDocument, node)) {
                ret = style[name]; //其他浏览器需要我们手动取内联样式
            }
            //Dean Edwards 的 hack，用于转换 margin 的百分比值为更有用的像素值
            // webkit 不能转换 top、bottom、left、right、margin、text-indent 的百分比值
            if (/^margin/.test(name) && rnumnonpx.test(ret)) {
                width = style.width;
                minWidth = style.minWidth;
```

```
                maxWidth = style.maxWidth;
                style.minWidth = style.maxWidth = style.width = ret;
                ret = styles.width;
                style.width = width;
                style.minWidth = minWidth;
                style.maxWidth = maxWidth;
            }
        }
        return ret;
    };
} else {
    $.getStyles = function(node) {
        return node.currentStyle;
    };
    var ie8 = !!window.XDomainRequest,
            rfilters = /[\w\:\.]+\([^)]+\)/g,
            rnumnonpx = /^-?(?:\d*\.)?\d+(?!px)[^\d\s]+$/i,
            border = {
        thin: ie8 ? '1px' : '2px',
        medium: ie8 ? '3px' : '4px',
        thick: ie8 ? '5px' : '6px'
    };
    adapter["_default:get"] = function(node, name, styles) {
        //取得精确值，不过它有可能是带 em、pc、mm、pt,%等单位
        var currentStyle = styles || node.currentStyle;
        var ret = currentStyle[name];
        if ((rnumnonpx.test(ret) && !rposition.test(ret))) {
            //①保存原有的 style、left, runtimeStyle.left
            var style = node.style,
                    left = style.left,
                    rsLeft = node.runtimeStyle.left;
            //②由于③处的 style.left = xxx 会影响到 currentStyle.left,
            //因此把它 currentStyle.left 放到 runtimeStyle.left,
            //runtimeStyle.left 拥有最高优先级，不会 style.left 影响
            node.runtimeStyle.left = currentStyle.left;
            //③将精确值赋给到 style.left，然后通过 IE 的另一个私有属性 style.pixelLeft
            //得到单位为 px 的结果，fontSize 的分支见 http://bugs.jquery.com/ticket/760
            style.left = name === 'fontSize' ? '1em' : (ret || 0);
            ret = style.pixelLeft + "px";
            //④还原 style.left, runtimeStyle.left
            style.left = left;
            node.runtimeStyle.left = rsLeft;
        }
        if (ret === "medium") {
            name = name.replace("Width", "Style");
            //border width 默认值为 medium，即使其为"0"
            if (currentStyle[name] === "none") {
                ret = "0px";
            }
        }
        return ret === "" ? "auto" : border[ret] || ret;
    }
}
```

读方法比较复杂。首先我们要知道，当外面的参数跑到 cssHooks["defaults:get"]、cssHooks["defaults:set"]跟前时，样式名已经正确加上必要的私有前缀，转换为驼峰风格，而样式值（如果有的话）也加上必要的单位了。

在标准浏览器下，如果元素节点没有插入 DOM 树，除了 IE9、IE10 与 Firefox 可以智能找到内联样式的相关值，并转换正确的可计算值外，其他需要我们手动找内联样式。因此我们需要利用 $.contains 方法进行判定。此外，对于 webkit 的 Bug，我们还要用到 DE 的技巧。

在旧版本 IE 中，currentStyle 则没有区分是否 DOM 树的烦恼，但想取得长宽这样的度量单位非常麻烦，原因是它们不会进行单位换算。这也依靠 DE 大神的 hacks 度过难关。

$.cssHooks 的外壳就是$.css 方法，它在读分支的处理也是比写分支复杂得多。

首先要判定目标是否为元素节点，然后取得可用的样式，根据参数判定它是写操作还是读操作。最后判定它是否存在对应的钩子函数，有钩子函数就转交给它，没有就一般性处理。细分如下几步：如果是字符串，进行递增递减处理（"-="，"+="）；如果是数字，排除为 NaN 的情况；如果需要加单位，补上单位；如果碰上 IE9、IE10 那个恶心的复制品影响原版的 BUG（见第 5 章），需要特殊处理；最后取钩子函数处理。

```
$.css = function(node, name, value, styles) {
    if (node.style) {
        var prop = /\_/.test(name) ? $.String.camelize(name) : name;
        name = $.cssName(prop) || prop;
        styles = styles || getStyles(node);
        if (value === void 0) { //获取样式
            return(adapter[prop + ":get"] || getter)(node, name, styles);
        } else { //设置样式
            var type = typeof value,
                temp;
            if (type === "string" && (temp = rrelNum.exec(value))) {

                value = (+(temp[1] + 1) * +temp[2]) + parseFloat($.css(node, name, void 0, styles));
                type = "number";

            }
            if (type === "number" && !isFinite(value + "")) {
                return;
            }
            if (type === "number" && !$.cssNumber[prop]) {
                value += "px";
            }
            if (value === "" && !$.support.cloneBackgroundStyle && name.indexOf("background")===0){
                node.style[name] = "inherit";
            }
            var fn = adapter[prop + ":set"] || adapter["_default:set"];
            fn(node, name, value, styles);
        }
    }
};
```

9.2 样式名的修正

不是所有样式名都是直接用正则简单处理一下就行，这里存在三个陷井，float 对应的 JavaScript 属性存在兼容性问题，CSS3 大爆炸时带来的私有前缀，IE 的私有前缀不合流问题。

float 是一个关键字，因此不能直接用，IE 这边给的替换品是 styleFloat，W3C 是 cssFloat。

CSS3 给 Web 开发带来了革命性的影响，以前很多需要 JavaScript 才能实现的复杂效果，现在使用 CSS3 就能简单实现。但，标准制定总是滞后于浏览器商的实现，只要有一个浏览器实现一个很酷的效果，其他浏览器也跟风。浏览器商也有先见之明，不确定自己的实现与 W3C 最后定案的效果是否一致，于是私有前缀便产生了。不过私有前缀是由来以久的东西，并不是 CSS3 这概念被炒热时才出来的，比如-ms-是 IE8 时代就存在，-khtml-就更早了。现在私有前缀在各浏览器定义如表 9.1 所示。

表 9.1

浏览器	IE	Firefox	Chrome	Safari	Opera	Konqueror
前缀	-ms-	-moz-	-webkit-	-webkit-	-o-	-khtml-

2013 年初，Google 嫌 webkit 内核太臃肿，决定自己单干，取名为 blink，并在 Chrome28 起使用此内核。不过为了减轻用户负担，还是使用-webkit-做前缀。目前，使用-webkit-前缀的有 Opera、Safari、Chrome 三家。

上述的这些前缀加上样式名再驼峰化就是真正可用的样式名（对那些试验性样式来说），比如 -ms-transform -> MsTransform, -webkit-transform -> WebkitTransform, -o-transform -> Otransform, -moz-transform -> MozTransform。但试验性样式迟早会退出舞台的，它们会卸掉前缀重新亮相，比如 FF17 就直接可用 transform。但光是这样是不够的，还有第三个问题。IE 下-ms-transform 对应的 JavaScript 属性名为 msTransform。因此这个正则就非常复杂，我们要动用一个函数通过特性侦测手段获取它。在 mass Framework，它叫 cssName，在 jQuery 叫做 vendorPropName。由于特性侦测是 DOM 操作，消耗很大，因此获取后就应缓存起来，避免重复检测，这个对象在 mass Framework 称之为 cssMap。下面是对应的源代码。

```
var prefixes = ['', '-webkit-', '-moz-', '-ms-', '-o-'];
var cssMap = {
    "float": $.support.cssFloat ? 'cssFloat' : 'styleFloat',
    background: "backgroundColor"
};

function cssName(name, host, camelCase) {
    if (cssMap[name]) {
        return cssMap[name];
    }
    host = host || document.documentElement
    for (var i = 0, n = prefixes.length; i < n; i++) {
        camelCase = $.String.camelize(prefixes[i] + name);
        if (camelCase in host) {
```

```
                return (cssMap[name] = camelCase);
            }
        }
        return null;
    }
```

prefixes 的顺序设置得相当有技巧，""表示没有私有前缀，此样式已经标准化，故排在最前。wekbit 是最多浏览器使用的，因此排第 2。“-o-”是最小众的，因此排末尾。其他两个夹中间。

我们通过上面的函数，只需传入一个参数，就可以得到真正可用的样式名了。

9.3 个别样式的特殊处理

现在我们来展示 cssHooks 的价值所在。它是专门用于对付那些有兼容性问题、不按常规出牌的奇葩样式。cssHooks 为一个普通的对象，它每个属性名都以 xxx+":set"或 xxx+":get"命名，值为处理函数。

9.3.1 opacity

在开始之前，我们先了解一下 CSS 是怎么进行的吧，毕竟 JavaScript 设置透明度只是把这过程由写死变成动态。

firefox 和 webkit 系浏览器的古老版本的透明度设置如下。

```
.opacity{
    -moz-opacity: 0.5;
    -khtml-opacity: .5;
}
```

有资料表明 Firefox 在 0.9 声明废弃此样式，反正我在 Firefox3.6.24 与 Firefox16 中测试，-moz-opacity:已经没有效果了。

现在标准浏览器的透明度设置（包括 IE9）如下。

```
.opacity {
    opacity:.5
}
```

opacity 会同时让背景与内容透明，想内容不透明，就要用 rgba 与 hsla。不过它们是一种样式值的格式，并不是样式，不列入我们的讨论范围。

旧版本 IE 的透明度设置，依赖于私有的滤镜 DXImageTransform.Microsoft.Alpha。

```
.opacity { //①
    filter: progid:DXImageTransform.Microsoft.Alpha(opacity=40);
}
```

不过这个太长了，IE 又提供了一个简短的，也是现在主流的在 IE 设置透明度的方式。

```
.opacity {  //②
     filter: alpha(opacity=40)
}
```

在 IE8，开始推广-ms-私有前缀，透明滤镜的写法变成如下。

```
.opacity {  //③
   -ms-filter: "progid:DXImageTransform.Microsoft.Alpha(opacity=40)";
/*IE8 专用，必须用引号括起*/
}
```

对于 IE6 和 IE7 还需要注意一点：为了使得透明设置生效，元素必须是“有布局”。一个元素可以通过使用一些 CSS 属性来使其被布局，有如 width 和 position。关于微软专有的 hasLayout 属性详情，以及如何触发它，可以看下面链接：

http://www.blueidea.com/tech/site/2006/3698.asp

了解这些，我们在框架实现它们就很简单了，由于 opacity 已经被标准浏览器所支持，我们的重心落在旧版本 IE 中。

先实现获取透明度，在 IE 中如果元素设置了滤镜，元素的 filters 属性会存在对应的键值。不过，由于 IE 可以用 3 种不同的滤镜形态来设置它，因此取时也得区分一下：①、③的设置方式通过 node.filter[“DXImageTransform.Microsoft.Alpha”]取到透明度，②可以通过 node.filter.alpha 取到。如果透明度是设置在父元素上，那么旧版本 IE 与标准浏览器保持一致，直接返回 1。但由于样式值在标准中统一为字符串，因此应该返回“1”。

```
adapter[ "opacity:get" ] = function( node){
   //这是最快的获取 IE 透明值的方式，不需要动用正则了！
    var alpha = node.filters.alpha || node.filters[salpha],
      op = alpha ? alpha.opacity: 100;
    return ( op /100 )+"";//确保返回的是字符串
}
```

设置透明度就有点麻烦，首先用户传入的是 0～1 的数值，我们应用于滤镜需要放大 100 倍。在 IE6、IE7 中，我们需要判定元素有没有 hasLayout，没有使用 zoom =1 让其 hasLayout。在 IE7、IE8 中，如果透明度为 100，会让文本模糊不清，需要清掉透明滤镜，这时我们就遇到一个问题了，一个元素上可能设置了多个滤镜，不能简单用 el.style.filter = ""清掉，这要用正则把单个滤镜分割出来，把当中的透明滤镜去掉。如果滤镜在 0～99，我们设置透明度有个窍门，如果已经存在透明滤镜，直接找到其滤镜对象，改其 alpha 值就行了，否则就需要小心翼翼拼字符串！

```
rfilters = /[\w\:\.]+\([^)]+\)/g
adapter["opacity:set"] = function(node, name, value, currentStyle) {
   var style = node.style;
   if (!isFinite(value)) { //"xxx" * 100 = NaN
      return;
   }
   value = (value > 0.999) ? 100 : (value < 0.001) ? 0 : value * 100;
   if (!currentStyle.hasLayout)
      style.zoom = 1; //让元素获得 hasLayout
   var filter = currentStyle.filter || style.filter || "";
   //http://snook.ca/archives/html_and_css/ie-position-fixed-opacity-filter
   //IE7、IE8 的透明滤镜当其值为 100 时会让文本模糊不清
   if (value === 100) { //IE7、IE8 的透明滤镜当其值为 100 时会让文本模糊不清
```

```
        value = style.filter = filter.replace(rfilters, function(a) {
            return /alpha/i.test(a) ? "" : a; //可能存在多个滤镜，只清掉透明部分
        });
        //如果只有一个透明滤镜 就直接去掉
        if (value.trim() === "" && style.removeAttribute) {
            style.removeAttribute("filter");
        }
        return;
    }
    //如果已经设置过透明滤镜，可以使用以下便捷方式
    var alpha = node.filters.alpha || node.filters[salpha];
    if (alpha) {
        alpha.opacity = value;
    } else {
        style.filter = ((filter ? filter + "," : "") + "alpha(opacity=" + value + ")");
    }
};
```

9.3.2　user-select

CSS3 有一个叫 user-select 的样式，用于控制文本内容的可选择性。比如拖动时，会出现文字被选择中的状况，分散用户的注意力，这里就可以尝试使用此属性。在标准浏览器下，由于$.cssName 的存在，一下子就找到可使用的样式名。而在旧版本 IE 中，没有这样的样式，是使用 unselectable 属性代替。不过由于 unselectable 不具继承性，加之子元素是位于父元素的上面，因此单设置当前元素是不行的，要把它与它的所有子孙都设置，具体代码如下。

```
adapter[ "userSelect:set" ] = function(node, name, value) {
    var allow = /none/.test(value) ? "on" : "",
            e, i = 0, els = node.getElementsByTagName('*');
    node.setAttribute('unselectable', allow);
    while ((e = els[ i++ ])) {
        switch (e.tagName.toLowerCase()) {
            case 'iframe' :
            case 'textarea' :
            case 'input' :
            case 'select' :
                break;
            default :
                e.setAttribute('unselectable', allow);
        }
    }
};
```

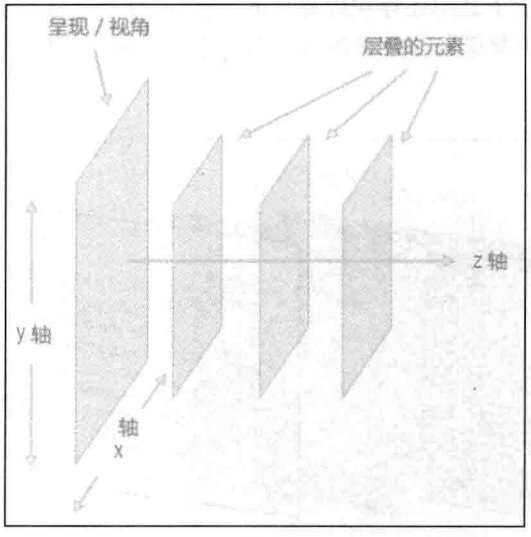
9.3.3　background-position

在旧版本 IE 中，IE 只支持 backgroundPositionX 与 backgroundPositionY，不支持 backgroundPosition，而在 FF 早期的版本中（3.0 之前），FF 只支持 backgroundPosition，不支持 backgroundPositionX 与 backgroundPositionY。它们俩总是对着干的。不过，FF 加入了 Chrome 引发的版本号竞赛，FF3 已经很少人用了，我们还是照顾好 IE6、IE7 就行了。实现很简单，分别取 backgroundPositionX 与 backgroundPositionY，然后把它们合在一起就是 backgroundPosition。

```
adapter[ "backgroundPosition:get" ] = function( node, name, value ) {
        var style = node.currentStyle;
        return style.backgroundPositionX +" "+style.backgroundPositionX
};
```

9.3.4 z-index

z-index 并不是一个难以理解的属性，但它却因错误的假设而使很多初级的开发人员陷入混乱。混乱发生的原因是因为 z-index 只能工作在被明确定义了 absolute、 fixed 或 relative 这三个定位属性的元素中,它会让元素沿 *z* 轴进行排序（*z* 轴的起点为其父节点所在的层，终点为屏幕）。如果为正数，则离用户更近，为负数则表示离用户更远。想象这样一个场合，一个相对定位的父节点，然后里面有 *N* 个绝对定位的子元素，如果没有指定 z-index，它们的显示方式是按照出现的先后顺序排列（nextElementSibling 会在 previousElementSibling 之上），如果有就按 z-index 排列，如果是负数，标准浏览器下会表现为元素排在其父节点的背后。当然，实际上元素的层叠顺序涉及更多东西，比如 stacking context 概念，IE 下的 select 元素的 BUG,z-index 为负时的浏览器差异等，如图 9.3 所示。



▲图 9.3

z-index 在下拉菜单、tooltip、灯箱效果、相册与拖动等中被经常使用。为了让目标控件排在最前，我们需要得知它们的 z-index，然后有目的地改 z-index 或重排元素（将目标元素移出 DOM 树再插入父元素内部的最后一个元素之后）。

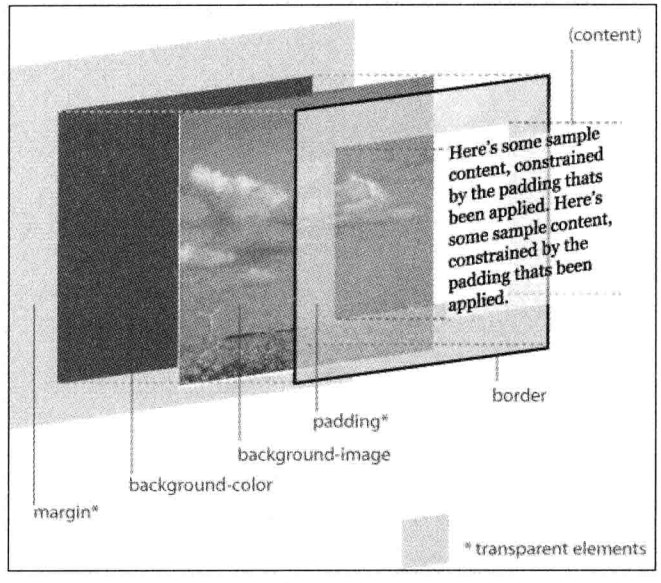
想获取 z-index，这里得应对一个特殊情况， 目标元素没有被定位，需要往上回溯其祖先定位元素。如果找到，就返回定位祖先的 z-index 值。如果最后都没找到，就返回 0。

```
adapter["zIndex:get"] = function(node) {
   while (node.nodeType !== 9) {
   //即使元素定位了，但如果 zindex 设置为"aaa"这样的无效值，浏览器都会返回 auto
      //如果没有指定 zindex 值，IE 会返回数字 0，其他返回 auto
```

```
        var position = getter(node, "position") || "static";
        if (position !== "static") {
            // <div style="z-index: -10;"><div style="z-index: 0;"></div></div>
            var value = parseInt(getter(node, "zIndex"), 10);
            if (!isNaN(value) && value !== 0) {
                return value;
            }
        }
        node = node.parentNode;
    }
    return 0;
};
```

9.3.5　盒子模型

在开始讲元素的尺寸位置时，我们先了解一下 CSS 盒子模型，如果没有这知识储备，后面就进行不下去。我们可以把页面上每一个元素节点看成一个装了东西的盒子，盒子里面的内容到盒子边框之间的距离即填充（padding），盒子本身有边框（border），用来放置子元素或文本的区域叫 content，盒子边框与其他盒子之间还存在边界（margin），为了把盒子装饰得五彩缤纷，而不只是一个死板的矩形，它还有背景颜色与背景图片。为了方便渲染，它们都会于不同的层上，如图 9.4 所示。



▲图 9.4

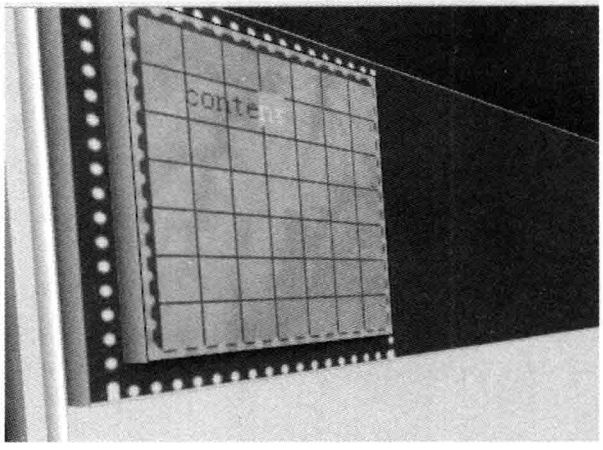
上面显示了各属性在 z 轴的层次关系：边界 → 背景颜色 → 背景图片 → 边框 → 填充 → 内容区。

3D 模型，我们还可以得出以下结论。

- 背景颜色、背景图片和边框之间是无法设置空白的。

- 背景图片在背景颜色之上，也就是说背景图片可以覆盖背景色。
- 元素背景指的是 content 和 padding 区域。

在 CSS3 中，它引进了 border-shodaw，能把元素打扮得更加惊艳，轻易实现纸张卷边效果。如果元素的第一个元素为 inset，那么它就是内阴影，否则为外阴影。此外 CSS2.1 还有一个虚线框——outline。那么盒子模型的立体层次就变成：边界 → 虚线框 → 外阴影 → 背景颜色 → 背景图片 → 内阴影 → 影边框 → 填充 → 内容区，如图 9.5 所示。



▲图 9.5

FF15 下，元素的 3D 图，可以直接看到它们之间的层叠关系。

不过真正影响布局的还是那几个，margin、 border、 padding、content。在早期，存在两种盒子模型，它决定着 width、height 的计算公式。W3C 的盒子模型，亦即后来的 content-box，内容区的宽即为 el.style.width，IE 在怪异模式的盒子模型，亦即后来的 border-box，内容区的宽是未知的，要用 width — 左右边框宽 — 左右填空宽。这就导致 IE 的盒子总比 W3C 的 smat。从设计与计算的角度，尤其是百分比设置宽高，IE 的盒子模型更为合理，符合人们的常识。因此 W3C 后来搞了个 box-sizing 的 CSS3 新属性来搪塞人们的疑问。box-sizing 在 W3C 规范中，拥有三种值，content-box，padding-box，border-box，宽高的计算起点由它们的名字决定。

9.3.6 元素的尺寸

由于元素的高与宽的取法都是一样的，我这里只拿宽来讲解。参照 jQuery 的行为，width 是指内容区的宽。一般情况下，我们可以使用 window.getComputedStyle 精确得到元素的宽，但如果元素的 display 为 none，或元素的祖先的 dispaly 为 none，又或者元素脱离了 DOM 树，window.getComputedStyle 就无能为力了。旧版本 IE 那边也差不多。并且浏览器在非 content-box 模式下，el.currentSyle.width 或 window.getComputedStyle(el, null).width 得到的值是整个盒子的宽，不是内容区的宽（FF 好像是例外），因此我们需要另辟蹊径。

首先要把隐藏元素显示出来。如果元素是隐藏的，它的 offsetWidth 为 0，但这不能作为判定元

素是隐藏的充分条件，因为用户可能直接设置 width：0px。这时，我们需要判定它的 display 值是否为 none。

```
function showHidden(node, array) {
    //http://www.cnblogs.com/rubylouvre/archive/2012/10/27/2742529.html
    if (node && node.nodeType === 1 && node.offsetWidth <= 0) { //opera.offsetWidth 可能小于 0
        if (getter(node, "display") === "none") {
            var obj = {
                node: node
            };
            for (var name in cssShow) {
                obj[name] = node.style[name];
                node.style[name] = cssShow[name] || $.parseDisplay(node.nodeName);
            }
            array.push(obj);
        }
        showHidden(node.parentNode, array);
    }
}
```

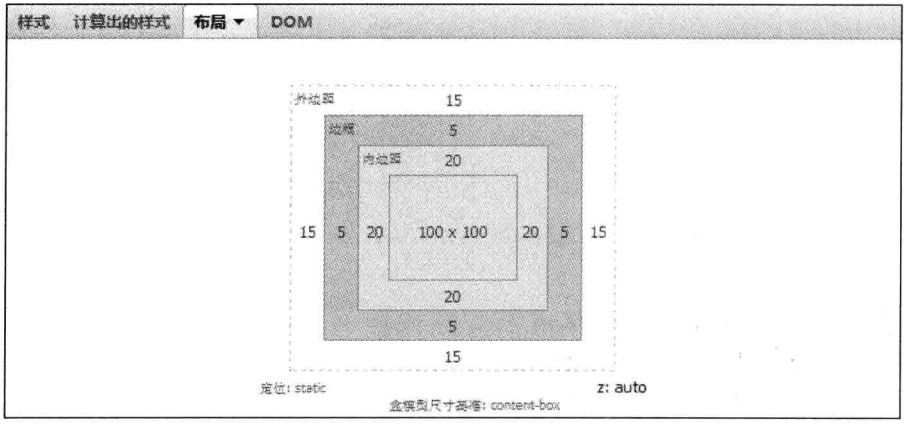
接下来，我们还要判定元素的盒子模型。IE 在怪异模式下，盒子模式为 border-box，如果不支持 box-sizing，那么大家都是 content-box，否则根据 box-sizing 值进入裁剪。裁剪什么呢？裁剪 IE 带来的好东西 offsetWidth，现在它已被标准浏览器良好支持，并列入规范。

看一个实验：

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>contentbox by 司徒正美</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <style type="text/css">
            body,html{
                height:100%;
                background:gray;
            }
            #parent {
                background: red;
                width:300px;
                height:300px;
                border:1px solid greenyellow;
            }
            #son{
                width:100px;
                height:100px;
                background-color:blue;
                padding:20px;
                margin:15px;
                overflow:hidden;
                border:5px solid yellow;
            }
        </style>
        <script>
```

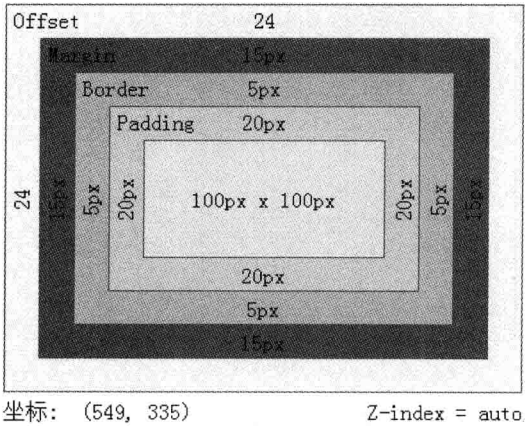
```
            window.onload = function() {
                var el = document.getElementById("son");
                console.log(el.offsetWidth);
                console.log(window.getComputedStyle(el, null).width);
            }
        </script>
    </head>
    <body>
        <div id="parent">
            <div id="son"> </div>
        </div>
    </body>
</html>
```

我们可以在 firebug 中看到 son 元素的盒子模型的具体参数与结构，如图 9.6 所示。



▲图 9.6

我们也可以在 IE8+的开发人员工具中看到类似的信息，如图 9.7 所示。



▲图 9.7

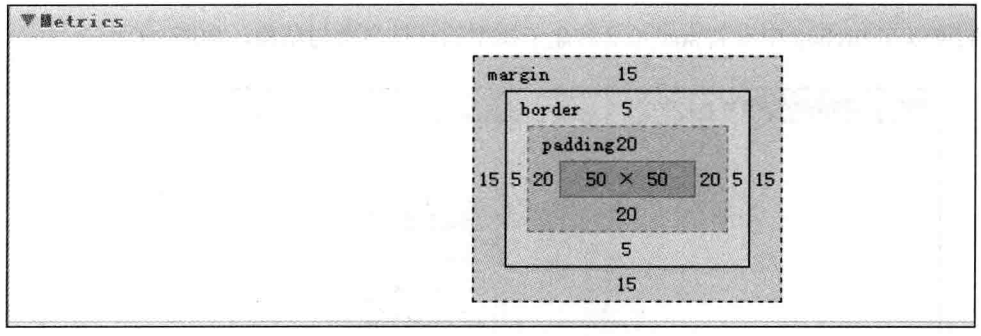
控制台输出为 150px、100px，我们可以从出推导出这样一个公式：

```
offsetWidth = borderWidth + paddingWidth + width
```

如果我们在 div#son 的样式添加多一行样式规则：

```
box-sizing:border-box;
-moz--sizing:border-box;
```

这时，红色方块就会比刚才的显示得小很多（Firefox 在 15 版中依然显示错误）。控制台输出 100， 100px(Firefox15 为 100， 50px)，如图 9.8 所示。

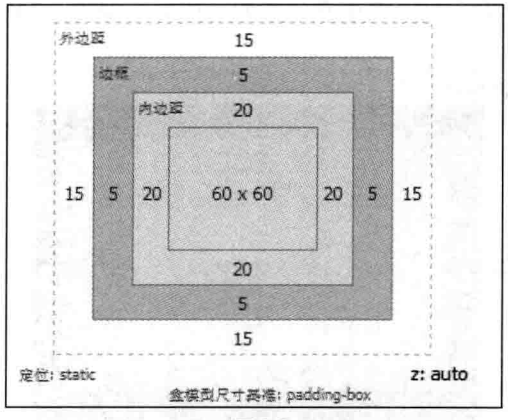


▲图 9.8

图 9.8 为 Chrome 开发者工具的元素信息图。

offsetWidth = borderWidth + paddingWidth + width，这公式依然有效。

在我写这章节时，只有 Firefox 支持 padding-box。修改上面的实验代码，在 firebug 中看到控制台输出为 110px、60px，如图 9.9 所示。



▲图 9.9

offsetWidth = borderWidth + paddingWidth + width，这公式依然有效。

因此取元素的内容宽使用上面公式最合算不过，完全不受盒子模型影响。在 jQuery 的

dimensions 模块中，又提供两个只读方法 innerWidth、outerWidth 用于取得 padding-box、border-box 与 margin-box 的宽。不过，依赖 outerWidth 进行加减，一点难度也没有。

不过设置元素内容区的宽就麻烦了，在非 content-box 中，它完全是一个间接值，一个不存在的属性。如果是 padding-box,我们要将用户传入值加上 padddingWidth，如果是 border-box，再在刚才的基础上加上 borderWidth。

最后，在我们日常工作中，还有两个尺寸非常重要——窗口的大小与页面的大小，它们都是只读。为了方便起见，我们都把它们交由本模块处理。

先看窗口的宽。以前网景浏览器就提供了一个好用的只读属性 innerWidth，标准浏览器一脉相承，都有这东西，IE9 也支持它，W3C 草稿已在 2011 年收录它。因此如果你的框架只支持 IE9+与较新版本的标准浏览器，只用它取窗口宽。旧版本 IE，我们可以要区分一下它是否处于怪异模式。

怪异模式实质上就是 IE6 之前的模式。除了盒子模型的计算范围不一样外，另一个重要区别在于 body 标签的解释。在远古时代，body 元素是最顶层的可视元素，而 HTML 元素保持隐藏。然后，现代浏览器认为 body 只是一个普通的块状元素，HTML 则是包含整个浏览器窗口的可视元素。

IE 又发明一套叫 clientXXX 的属性，用于取得元素的可视区的尺寸，它是不包含滚动条以及被隐藏的部分。因此窗口的宽可以这样取：

```
windowWidth = document.innerWidth || document.documentElement.clientWidth ||
document.body.clientWidth
```

不过，实际上 jQuery 与 mass Framework 等框架都不打算支持怪异模式。怪异模式涉及许多东西，还有许多不可预测的怪异行为，而且殃及页面所有元素的盒子模型，不像 box-sizing 那样可以对单个元素进行设置。于是乎，上面的代码可以简化成这样：

```
windowWidth = document.documentElement.clientWidth
```

如果你的框架是手机框架，可以放胆去用 innerWidth，即：

```
windowWidth = window.innerWidth;
```

再看页面的宽。页面的宽即文档的宽。一旦出现横向滚动条，我们就要考虑加上被隐藏的部分。标准浏览器又有一套叫 outerXXX 的属性，但那是取浏览器的尺寸的，因此不能像 innerWidth 那样照样画葫芦。IE 在此又亲切地为我们奉上另两套叫 scrollXXX、offsetXXX 的属性，标准浏览器继续照抄不误，但又抄得不好，留下兼容性问题。

- offsetWidth

IE、Opera 认为 offsetWidth = clientWidth + 滚动条 + 边框。

NS、FF 认为 offsetWidth 是网页内容实际宽度，可以小于 clientWidth。

- scrollWidth

IE、Opera 认为 scrollWidth 是网页内容实际宽度，可以小于 clientWidth。

NS、FF 认为 scrollWidth 是网页内容宽度，不过最小值是 clientWidth。

你想想世界上有这么浏览器，谁知道它们又是怎么想的。因此直接把这些属性放在一起，取最大值。

```
var pageWidth = Math.max( document.documentElement.scrollWidth,
document.documentElement.offsetWidth, document.documentElement.clientWidth,
document.body.scrollWidth, document.body.offsetWidth);
```

document.body.clientWidth 肯定是最小的，不用比较。

在 Firefox 中还提供了 window.scrollMaxX 这样一个属性，与 window.innerWidth 相加，恰好等于页面的宽度。不过 MDC 警告说它们之和不等于页面宽，可能是之前版本的问题吧。

在 webkit 系浏览器，它们直接在 document 中添加 width/height 属性，很方便。

最后别忘了在操作之前，先判定一下元素的盒子模型。

```
var cssBoxSizing = $.cssName("box-sizing");
adapter["boxSizing:get"] = function(node, name) {
   return cssBoxSizing ? getter(node, name) : document.compatMode === "BackCompat"
   ? "border-box" : "content-box";
};
```

接下来是具体实现。

```
var cssPair = {
   width: ['Left', 'Right'],
   height: ['Top', 'Bottom']
};
var cssShow = {
   position: "absolute",
   visibility: "hidden",
   display: "block"
};
function toNumber(styles, name) {
   return parseFloat(styles[name]) || 0;
}

function showHidden(node, array) {
}

function setWH(node, name, val, extra) {
   var which = cssPair[name],
         styles = getStyles(node);
   which.forEach(function(direction) {
      if (extra < 1)
         val -= toNumber(styles, 'padding' + direction);
      if (extra < 2)
         val -= toNumber(styles, 'border' + direction + 'Width');
      if (extra === 3) {
         val += parseFloat(getter(node, 'margin' + direction, styles)) || 0;
      }
      if (extra === "padding-box") {
         val += toNumber(styles, 'padding' + direction);
      }
      if (extra === "border-box") {
         val += toNumber(styles, 'padding' + direction);
         val += toNumber(styles, 'border' + direction + 'Width');
      }
   });
```

```
    return val;
}

function getWH(node, name, extra) { //注意 name 是首字母大写
    var hidden = [];
    showHidden(node, hidden);
    var val = setWH(node, name, node["offset" + name], extra);
    for (var i = 0, obj; obj = hidden[i++]; ) {
        node = obj.node;
        for (name in obj) {
            if (typeof obj[name] === "string") {
                node.style[name] = obj[name];
            }
        }
    }
    return val;
}

"Height,Width".replace($.rword, function(name) {
    var lower = name.toLowerCase(),
            clientProp = "client" + name,
            scrollProp = "scroll" + name,
            offsetProp = "offset" + name;
    $.cssHooks[lower + ":get"] = function(node) {
        return getWH(node, name, 0) + "px"; //添加相应适配器
    };
    $.cssHooks[lower + ":set"] = function(node, nick, value) {
        var box = $.css(node, "box-sizing"); //nick 防止与外面 name 冲突
        node.style[nick] = box === "content-box" ? value :
                setWH(node, name, parseFloat(value), box) + "px";
    };
    "inner_1,b_0,outer_2".replace($.rmapper, function(a, b, num) {
        var method = b === "b" ? lower : b + name;
        $.fn[method] = function(value) {
            num = b === "outer" && value === true ? 3 : Number(num);
            value = typeof value === "boolean" ? void 0 : value;
            if (value === void 0) {
                var node = this[0]
                if ($.type(node, "Window")) { //取得窗口尺寸,IE9 后可以用 node.innerWidth
                                              // /innerHeight 代替
                    return node["inner" + name] || node.document.documentElement[clientProp];
                }
                if (node.nodeType === 9) { //取得页面尺寸
                    var doc = node.documentElement;
                    //FF chrome    html.scrollHeight< body.scrollHeight
                    //IE 标准模式 : html.scrollHeight> body.scrollHeight
                    //IE 怪异模式 : html.scrollHeight 最大等于可视窗口多一点?
                    return Math.max(node.body[scrollProp], doc[scrollProp],
                    node.body[offsetProp], doc[offsetProp], doc[clientProp]);
                }
                return getWH(node, name, num);
            } else {
                for (var i = 0; node = this[i++]; ) {
                    $.css(node, lower, value);
```

```
                }
            }
            return this;
        };
    });
});
```

showHidden 用于显示隐藏元素并把它们收集到第 2 参数中。setWH 是用于修正用户传参到可用的大小的，或加上 padding，或减去 padding，由第 4 参数决定。getWH 用于取得元素大小。接下来，善用字符串的 replace 方法，一下子生成 width、height、innerWidth、innerHeight、outerWidth、outerHeight 这六种原型方法，免于重复代码。replace 循环生成相近方法这一套路遍布 mass Framework 的各个模块，是个非常好用的东西。

9.3.7　元素的显隐

想让元素在页面上看不见有许多办法，但我这里只讲 display。display 为 none 时它不占有物理空间，附近的元素就可以顺势向它的位置挪过去，比如手风琴效果、下拉效果都依赖于它。因此这种隐藏方式非常有用。在许多情况下，它是没有什么好说的，显示就设置 block，隐藏就是 none，切换就是根据它原来的状态决定显隐。

```
$.fn.show = function() {
    return this.each(function() {
        this.style.display = "";
    });
};
$.fn.hide = function() {
    return this.each(function() {
        this.style.display = "none";
    });
};
$.fn.toggle = function( ) {
    return this.each(function() {
        this.style.display = isHidden(this) ? "" : "none";
    });
};
```

但 jQuery 与 mass Framework 对元素的操作往往是一种集化操作，一下子抓起这么多元素，混有块状元素，内联元素，还有像 thead、tbody、tr 等具有特定默认 display 值的元素，它们一旦设置了 block，表格就立即崩溃。而且 Prototype.js 提倡的用空字符串来重新显示的方法往往是无效的。比如下面这例子：

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>display by 司徒正美</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <style type="text/css">
            body,html{
                height:100%;
                background:gray;
```

```
            }
            #parent {
                background: red;
                width:300px;
                height:300px;
            }
            #son{
                width:100px;
                height:100px;
                background:blue;
                padding:20px;
                margin:15px;
                display:none;
                border:5px solid yellow;
            }
        </style>
        <script>
            window.onload = function(){
                var el = document.getElementById("son")
                el.style.display= ""

            }
        </script>
    </head>
    <body>
        <div id="parent">
            <div id="son"> </div>
        </div>
    </body>
</html>
```

我们必须赋以正确的 display 值才能让它生效。问题是如何界定这个“正确”。单单是块状元素设置 block，内联元素设置 inline 让它们显示是不行的。这还引进了一个问题，如何区分它是块状元素或内联元素。对于不断增长的 HTML5 元素种类，hash 法也不可靠。而且增长的不单是元素种类，还有 display 值的类型。

在老旧的 CSS 1 规范中，display 值仅包括：block、inline、list-item、none。在 CSS 2.1 规范中，它已经扩张到如下这么多值：inline、block、list-item、run-in、inline-block、table、inline-table、table-row-group、table-header-group、table-footer-group、table-row、table-column-group、table-column、table-cell、table-caption、none、inherit。CSS3 引入更强大的布局模型 Flexible Box，对 display 增加 ruby、ruby-base、ruby-text、ruby-base-group、ruby-text-group、flex、grid 等值（注，新添值不太稳定，一切以 W3C 上的最新草稿为准）。

不过有些元素，我们可以肯定它们的默认样式值，对于一些特别的元素就需要实时去取了。为了取得干净的默认 display 属性，我们需要在 iframe 沙箱中去取，不过，如果浏览器支持 getDefaultComputedStyle，就更好办。

```
var cacheDisplay = $.oneObject("a,abbr,b,span,strong,em,font,i,kbd", "inline"),
      blocks = $.oneObject("div,h1,h2,h3,h4,h5,h6,section,p", "block");
$.mix(cacheDisplay, blocks);
$.parseDisplay = function(nodeName) {
```

```
    //用于取得此类标签的默认 display 值
    nodeName = nodeName.toLowerCase();
    if (!cacheDisplay[nodeName]) {
        $.applyShadowDOM(function(win, doc, body, val) {
            var node = doc.createElement(nodeName);
            body.appendChild(node);
            if (win.getComputedStyle) {
                val = win.getComputedStyle(node, null).display;
            } else {
                val = node.currentStyle.display;
            }
            cacheDisplay[nodeName] = val;
        });
    }
    return cacheDisplay[nodeName];
};
//https://developer.mozilla.org/en-US/docs/DOM/window.getDefaultComputedStyle
if (window.getDefaultComputedStyle) {
    $.parseDisplay = function(nodeName) {
        nodeName = nodeName.toLowerCase();
        if (!cacheDisplay[nodeName]) {
            var node = document.createElement(nodeName);
            var val = window.getDefaultComputedStyle(node, null).display;
            cacheDisplay[nodeName] = val;
        }
        return cacheDisplay[nodeName];
    };
}
```

在$.parseDispaly 中依赖一个 applyShadowDOM 方法，ShadowDOM 是 HTML5 的一个新特征。此方法原本是基本它构建的，可惜在出来后不久，Chrome 又改接口，于是统一使用 iframe 实现。

```
var shadowRoot, shadowDoc, shadowWin, reuse;
$.applyShadowDOM = function(callback) {
    //用于提供一个沙箱环境,IE6~IE10、Opera、Safari、Firefox 使用 iframe, Chrome20+(25+不需要
    //开启实验性 JavaScript)使用 Shodow DOM
    if (!shadowRoot) {
        shadowRoot = document.createElement("iframe");
        shadowRoot.style.cssText = "width:0px;height:0px;border:0 none;";
    }
    $.html.appendChild(shadowRoot);
    if (!reuse) { //Firefox、Safari、Chrome 不能重用 shadowDoc、shadowWin
        shadowWin = shadowRoot.contentWindow;
        shadowDoc = shadowWin.document;
        shadowDoc.write("<!doctype html><html><body>");
        shadowDoc.close();
        reuse = window.VBArray || window.opera; //opera9-12, ie6-10 有效
    }
    callback(shadowWin, shadowDoc, shadowDoc.body);
    setTimeout(function() {
        $.html.removeChild(shadowRoot);
    }, 1000);
};
```

接下来，我们界定一下什么是隐藏元素。有了 display 这个大前提，第一条件是其精确值为 none，第二个条件是它是否在 DOM 树上。

```
$.isHidden = function(node) {
       return node.sourceIndex === 0 ||
         getter(node, "display") === "none" ||
         !$.contains(node.ownerDocument, node);
};
```

然后我们创建一个方法，把$.fn.show, $.fn.hide, $.fn.toggle 的功能全部转交给它做。

```
function toggelDisplay(nodes, show) {
   var elem, values = [],
          status = [],
          index = 0,
          length = nodes.length;
   //由于传入的元素们可能存在包含关系，因此分开两个循环来处理，第一个循环用于取得当前值或默认值
   for (; index < length; index++) {
      elem = nodes[index];
      if (!elem.style) {
         continue;
      }
      values[index] = $._data(elem, "olddisplay");
      status[index] = $.isHidden(elem);
      if (!values[index]) {
         values[index] = status[index] ? $.parseDisplay(elem.nodeName) : getter(elem,
"display");
         $._data(elem, "olddisplay", values[index]);
      }
   }
   //第二个循环用于设置样式，-1 为 toggle，1 为 show，0 为 hide
   for (index = 0; index < length; index++) {
      elem = nodes[index];
      if (!elem.style) {
         continue;
      }
      show = show === -1 ? !status[index] : show;
      elem.style.display = show ? values[index] : "none";
   }
   return nodes;
}
```

最后，我们只要稍微在外面一层就能实现与 jQuery 一模一样的 show、hide、toggle。这样做还有个好处，接口与实现相分离，实现隐藏于内部。保持既有功能，不受太多制约，安心优化与升级。

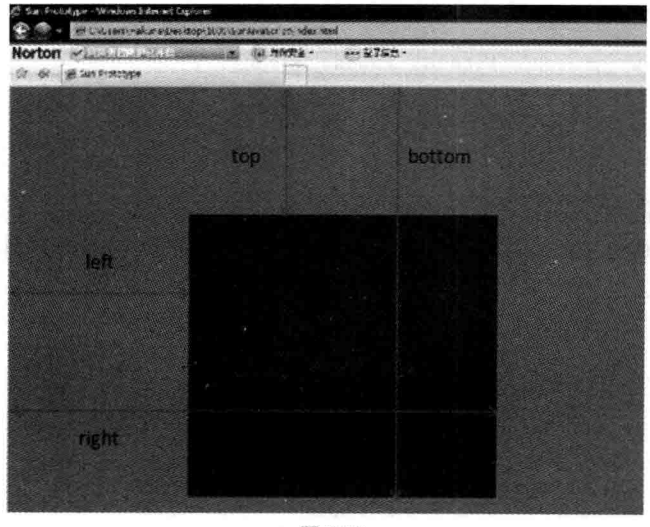
```
$.fn.show = function() {
   return toggelDisplay(this, 1);
};
$.fn.hide = function() {
   return toggelDisplay(this, 0);
};
//state 为 true 时，强制全部显示，为 false，强制全部隐藏
$.fn.toggle = function(state) {
   return toggelDisplay(this, typeof state === "boolean" ? state : -1);
};
```

9.3.8　元素的坐标

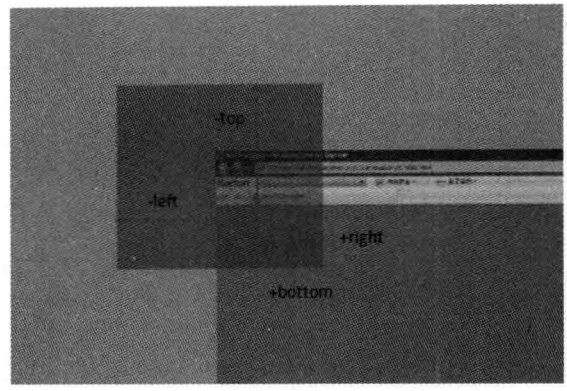
元素的坐标就是指其 top 与 left 值。node.style 恰逢有这两个属性,但它只有被定位了才有效，否则即使在 IE9、FF15、Chrome23、Opera12 等浏览器下都返回 auto。幸好，即使一个元素没有被定位，它的 offsetTop、offsetLeft 也是有效的，它们是相对于 offsetParent 的距离。我们一级级向上累加，就能得到相对页面的坐标，亦有人称之为元素的绝对坐标。

```
function offset(node) {
    var left = node.offsetLeft,
          top = node.offsetTop;
    do {
        left += node.offsetLeft;
        top += node.offsetTop;
    } while (node = node.offsetParent);
    return {
        left: left,
        top: top
    }
}
```

此外，相对于可视区的坐标也很实用，比如让弹出窗口居中对齐。以前实现这个计算量非常巨大，自从 IE 的 getBoundingClientRect 方法被发掘出来后，简直是小菜一碟。更令人高兴的是，它现在也被标准浏览器普遍接受，并列入 W3C 标准，无兼容性之忧。此方法能获取页面中某个元素(border-box)的左、上、右和下分别相对浏览器视窗的位置。它返回一个 Object 对象，该对象肯定有这样 4 个属性：top、left、right、bottom，标准浏览器下可能还多出 width、height 这两个属性。这里的 top、left 和 CSS 中的理解很相似，width、 height 是元素自身的宽高，但是 right、bottom 和 CSS 中的理解有点不一样。right 是指元素右边界距窗口最左边的距离，bottom 是指元素下边界距窗口最上面的距离。具体看示意如图 9.10 和图 9.11 所示。

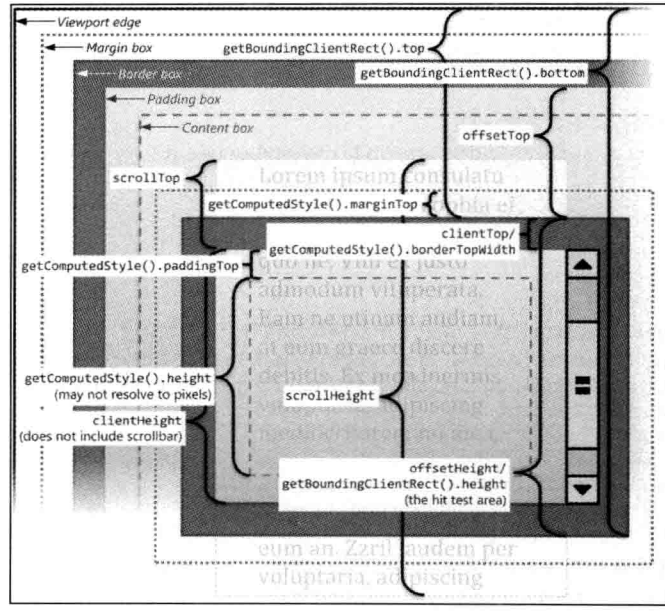


▲图 9.10



▲图 9.11

图 9.12 是微软的 MSDN《Measuring Element Dimension and Location with CSSOM in Internet Explorer 9》的图示。用于演示 CSSOM 各种属性。示例页面中拥有一个相对定位的红色元素。蓝色元素是红色元素的父节点，它是用于演示各种盒子，如 content-box、padding-box、border-box、margin-box 以及 offsetTop（这是属于蓝色元素的，换言之，它是蓝色元素的 offsetParent）。viewport 是个黑色虚框，为 html 标签。蓝色元素还拥有滚动条，方便我们观察 clientTop 与 scrollTop，clientHeight 与 scrollHeight 的差异。getBoundingClientRec 中的 top 与 bottom 也在此图中显示出来。掌握此图对我们构建此模块非常有帮助，如图 9.12 所示。



▲图 9.12

http://msdn.microsoft.com/en-us/library/ms530302%28VS.85%29.aspx

getBoundingClientRect 的支持情况如表 9.2 所示。

表 9.2

浏览器	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari
支持	1.0	3.0 (1.9)	4.0	(Yes)	4.0

因此我们也可以很方便地利用它求出相对于页面的距离，将它相对于可视区的距离加上滚动距离就行了。

```
var left = this.getBoundingClientRect().left+document.documentElement.scrollLeft;
var top =this.getBoundingClientRect().top+document.documentElement.scrollTop;
```

到目前为止，我们只是把所有用到的知识介绍一遍，就像 W3C 文案那样总是那样完美而不实用，我们还要考虑浏览器兼容性呢。有了兼容性，上面简洁的公式就要改得非常臃肿丑鄙才可以用。

那真正的解法是如何的呢？首先，我们判定它是否在 DOM 树上，不在直接返回（0，0）。否则取得元素在可视区的距离加上滚动距离然后减去浏览器的边框。因此，计算可视区距离与滚动距离时都已经包含浏览器的边框。而浏览器的边框即最顶层的可视元素的边框，在标准模式下，顶层可视元素为 html，怪异模式下为 body。计算滚动距离也是这样，需要选好顶层可视元素。由于 Windows8 与 IE10 的发布，加速了旧版本 IE 的淘汰，像 jQuery 与 mass Framework 已经不考虑支持怪异模式。各位可以参考自己公司的客户组成斟酌一下是否支持怪异模式。

```
$.fn.offset = function(options){
    if ( arguments.length ) {//设置匹配元素的 offset
        return (!options || ( !isFinite(options.top) && !isFinite(options.left) ) ) ?  this :
        this.each(function() {
            setOffset( this, options );
        });
    }
    //取得第一个元素的相对于页面的坐标
    var node = this[0], doc = node && node.ownerDocument, pos = {
        left:0,
        top:0
    };
    if ( !doc ) {
        return pos;
    }
    //我们可以通过 getBoundingClientRect 来获得元素相对于 client 的 rect
    //http://msdn.microsoft.com/en-us/library/ms536433.aspx
    var box = node.getBoundingClientRect(),win = getWindow(doc),
    root = doc.documentElement ,
    clientTop  = root.clientTop  || 0,
    clientLeft  = root.clientLeft  || 0,
    scrollTop  = win.pageYOffset ||  root.scrollTop,
    scrollLeft  = win.pageXOffset ||  root.scrollLeft;
    // 把滚动距离加到 left、top 中去
    // IE 一些版本中会自动为 HTML 元素加上 2px 的 border，我们需要去掉它
    // http://msdn.microsoft.com/en-us/library/ms533564(VS.85).aspx
    pos.top  = box.top  + scrollTop  - clientTop,
    pos.left  = box.left  + scrollLeft  - clientLeft;
    return pos;
}
```

我们再来取元素相对于其 offsetParent 的位置，亦有人称之为元素的相对坐标。要取得此值，我们先要确定其 offsetParent。根据 W3C 给出的规律，元素是这样寻找其 offsetParent 的。如果元素被移出 DOM 树，或 display 为 none，或作为 HTML 或 BODY 元素，或其 position 的精确值为 fixed 时，返回 null。否则分两种情况，position 为 absolute、relative 的元素的 offsetParent 总是为其最近的已定位的祖先，没有找最近的 td,th 元素，再没有返回 body；position 为 static 的元素的 offsetParent，则是先找最近的 td、th、table 元素，再没有返回 body。但现实中，Firefox 在 position 为 fixed 返回 body；在 IE6～IE8 下，会增加一条规则，先寻找离元素最近的设置有能激活 hasLayout 的祖先元素。

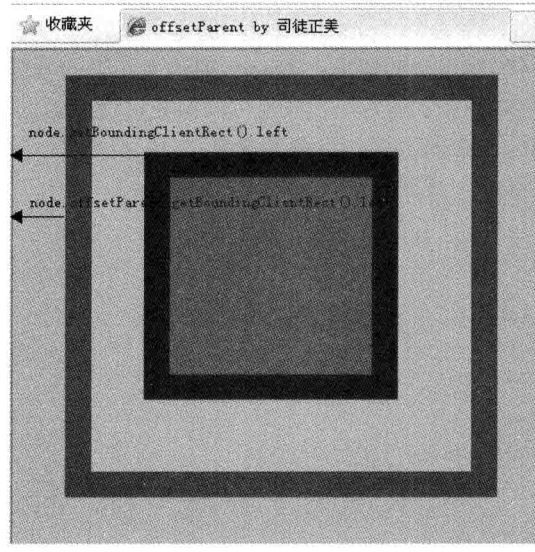
假若依据这个规律，我们在太多情况下会得到 offsetParent 为 null，导致无法计算。我们看 jQuery 是怎么做的。jQuery 也有个 offsetParent 方法，它是将选中元素的所有“offsetParent”收集起来，重新包装为 jQuery 对象返回。而这个“offsetParent”的定义被它修改了。浏览器认为 offsetParent 最高只能取到 body，而且存在为 null 的情况；jQuery 则认为元素的 offsetParent 的 position 必须为 relative 或 absolute，否则继续回上寻找另一个被定位的祖先，没有返回 html。另外，jQuery 认为 position:fixed 的元素也有 offsetParent，就是当前可视区。

为了让大家有个直观的认识，还是建一个 HTML 页面。

```
<!DOCTYPE HTML>
<html id="html">
    <head>
        <title>offsetParent by 司徒正美</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <style type="text/css">
            body,html{
                height:100%;
                background:orange;
            }
            body{
                margin:20px;
            }
            #parent {
                position: relative;
                width:250px;
                height:250px;
                margin:20px;
                background:aqua;
                border:20px solid red;
                padding:20px;
            }
            #son {
                position: absolute;
                width:150px;
                height:150px;
                margin:20px;
                background:fuchsia;
                border:20px solid blue;
                padding:2px;
            }
        </style>
```

```
    </head>
    <body id="body">
        <div id="parent">
            <div id="son"></div>
        </div>
    </body>
</html>
```

从盒子模型的角度来看，相对于 offsetParent 的距离，是指此元素的 margin-box 的左上角到 offsetParent 的 content-box 的左上角。由于 offsetParen、getBoundingClientRect 等都是由 IE 首先提出来的，因此盒子都以 border-box 为计算单元，我们需要减于 offsetParent 的左边框与元素的左边界的宽，如图 9.13 所示。



▲图 9.13

因此 x 轴距离的计算公式如下。

```
X = node[clientLeft] - offsetParent[client_left] -
offsetParent[borderLeftWidth] - node[marginLeftWidth]
```

整个方法实现如下。

```
$.fn.position = function() {//取得元素相对于其 offsetParent 的坐标
    var offset, offsetParent, node = this[0],
            parentOffset = {//默认的 offsetParent 相对于视窗的距离
        top: 0,
        left: 0
    }
    if (!node || node.nodeType !== 1) {
        return
    }
    //fixed 元素是相对于 window
    if (getter(node, "position") === "fixed") {
```

```
        offset = node.getBoundingClientRect();
    } else {
        offset = this.offset();//得到元素相对于视窗的距离（我们只有它的 top 与 left）
        offsetParent = this.offsetParent();
        if (offsetParent[ 0 ].tagName !== "HTML") {
            parentOffset = offsetParent.offset();//得到它的 offsetParent 相对于视窗的距离
        }
        parentOffset.top += parseFloat(getter(offsetParent[ 0 ],
        "borderTopWidth")) || 0;
        parentOffset.left += parseFloat(getter(offsetParent[ 0 ],
        "borderLeftWidth")) || 0;
    }
    return {
        top: offset.top - parentOffset.top - (parseFloat(getter(node, "marginTop"))
|| 0),
        left: offset.left - parentOffset.left - (parseFloat(getter(node,
"marginLeft")) || 0)
    };
}
```

我们再回头看 offset 方法。offset 也是一个 set all get first 方法。想把一个元素挪到页面某个位置有许多实现，比如移动滚动条，修改 margin，修改 top、left。offset 的写方法就是最后一种。不过可能存在被定位的祖先元素，因此不借助框架。这个工作对一般人而言有点吃力，而这正是框架的价值所在。

思路如下，用户传入新的相对于页面的坐标，然后判定当前元素的 position。因为想让 top、left 有效，必须是定位元素。如果不是，我们就设置元素为相对定位。接着求取当前相对页面的坐标与用户传入的坐标间的偏移量，最后加到当前的 top、left 上。

```
function setOffset(node, options) {
    if (node && node.nodeType == 1) {
        var position = $.css(node, "position");
        //强逼定位
        if (position === "static") {
            node.style.position = "relative";
        }
        var curElem = $(node),
                curOffset = curElem.offset(),
                curCSSTop = $.css(node, "top"),
                curCSSLeft = $.css(node, "left"),
                calculatePosition = (position === "absolute" || position === "fixed") &&
                [curCSSTop, curCSSLeft].indexOf("auto") > -1,
                props = {}, curPosition = {}, curTop, curLeft;
        if (calculatePosition) {
            curPosition = curElem.position();
            curTop = curPosition.top;
            curLeft = curPosition.left;
        } else {
            //如果是相对定位，只要用当前 top、left 做基数
            curTop = parseFloat(curCSSTop) || 0;
            curLeft = parseFloat(curCSSLeft) || 0;
        }
```

```
            if (options.top != null) {
                props.top = (options.top - curOffset.top) + curTop;
            }
            if (options.left != null) {
                props.left = (options.left - curOffset.left) + curLeft;
            }
            curElem.css(props);
        }
    }
```

这样我们的 offset 方法就完整了。

9.4 元素的滚动条的坐标

这是浏览器一组非常重要的属性，因此光是浏览器本身就提供了多个方法来修改它们，比如挂在 window 下的 scroll、scrollTo、scrollBy 方法，挂在元素下的 scrollLeft、scrollTop、scrollIntoView。jQuery 在 css 模块就提供了 scrollLeft、scrollTop 来修改或读取元素或窗口的滚动坐标，在 animation 模块，更是允许它以更平滑的方式来挪动它们。EXT 更是用整一个模块来满足用户对滚动条的各种需求。

修改 top、left 挪动元素有一个坏处，可能遮在某些元素之上，而修改 scrollTop、scrollLeft 不会。这里我们还是仿照 jQuery 那样，把这两个方法取名为 scrollTop、scrollLeft。对于一般的元素节点，读写它们没有什么难点，因为元素上就有这两个属性了。我们只需集中精力对付最外面的滚动条，也称浏览器的滚动条，位于最顶层的可视元素之上。设置时，我们要用到 window 中的 scrollTo 方法，里面传入你要滚到的坐标。读取时，我们尝试使用 pageXOffset、pageYOffset 这组属性，标准浏览器从网景时代就支持了，IE 则直接取 html 元素的 scrollLeft, scrollTop 属性，因为我也不打算支持怪异模式。想支持怪异模式的朋友，可以继续尝试从 body 元素中取相关属性。

```
"scrollLeft_pageXOffset,scrollTop_pageYOffset".replace($.rmapper, function(_, method,
prop) {
    $.fn[method] = function(val) {
        var node, win, top = method === "scrollTop";
        if (val === void 0) {
            node = this[0];
            if (!node) {
                return null;
            }
            win = getWindow(node); //获取第一个元素的 scrollTop/scrollLeft
            return win ? (prop in win) ? win[prop] : win.document.documentElement[method] :
            node[method];
        }
        return this.each(function() { //设置匹配元素的 scrollTop/scrollLeft
            win = getWindow(this);
            if (win) {
                win.scrollTo(!top ? val : $(win).scrollLeft(), top ? val : $(win).scrollTop());
            } else {
                this[method] = val;
            }
        });
    };
});
```

第10章 属性模块

通常我们把对象的非函数成员叫做属性。对于元素节点来说，其属性大体分成两大类，固有属性与自定义属性（特性）。固有属性一般遵循驼峰命名风格，拥有默认值，并且无法删除。自定义属性是用户随意添加的键值对，由于元素节点也是一个普通的 JavaScript 对象，没有什么严格的访问操作，因此命名风格林林总总，值的类型也乱七八糟。但是随意添加属性显然不够安全，比如引起循环引用什么的，因此，浏览器提供了一组 API 来供人们操作自定义属性，即 setAttribute、getAttribute、removeAttribute。当然还有其他 API，不过这是标准套装，只有在 IE6、IE7 那样糟糕的环境下，我们才求助于其他 API，一般情况下这 3 个足矣。我们通称它们为 DOM 属性系统。DOM 属性系统对属性名会进行小写化处理，属性值会统一转字符串。

```
var el =  document.createElement("div")
el.setAttribute("xxx", "1")
el.setAttribute("XxX", "2")
el.setAttribute("XXx", "3")
console.log(el.getAttribute("xxx"))
console.log(el.getAttribute("XxX"))
```

IE6、IE7 会返回"1"，其他浏览器返回“3”。在前端的世界，我们真是走到哪都能碰到兼容性问题。这只是冰山一角，IE6、IE7 在处理固有属性时要求进行名字映射，比如 class 变成 className，for 变成 htmlFor。对于布尔属性（一些只返回布尔的属性），浏览器间的差异更大，具体如表 10.1 所示。

```
<input type="radio"  id="aaa">
<input type="radio" checked id="bbb">
<input type="radio" checked="checked" id="ccc">
<input type="radio" checked="true" id="ddd">
<input type="radio" checked="xxx" id="eee">

"aaa,bbb,ccc,ddd,eee".replace(/\w+/g,function( id ){
   var elem =  document.getElementById( id )
   console.log(elem.getAttribute("checked"));
})
```

表 10.1

	#aaa	#bbb	#ccc	#ddd	#eee
IE7	false	true	true	true	true
IE8	""	"checked"	"checked"	"checked"	"checked"

续表

IE9	null	""	"checked"	"true"	"xxx"
FF15	null	""	"checked"	"true"	"xxx"
Chrome23	null	""	"checked"	"true"	"xxx"

因此框架很有必要提供一些 API 来屏蔽这些差异性。但在 IE6 统治时期，这个需求并不明显，因为 IE6、IE7 不区分固有属性与自定属性，setAttribute 与 getAttribute 在当时的人看来只是一个语法糖，用 el.setAttribute("innerHTML","xxxx")与用 el.innerHTML = "xxx"效果一样，而且后者更方便。即使早期应用那么广泛的 Prototype.js，提供属性操作 API 也非常贫乏，只有 identify、readAttribute、writeAttribute、hasAttribute、classNames、hasClassName、addClassName、removeClassName、toggleClassName 与$F 方法。Prototype.js 也察觉到固有属性与自定义属性在 DOM 属性系统的差异，在它的内部，搞了个 Element._attributeTranslations。然而，Prototype.js 这个属性系统内部还是优先使用 el[name] 方式来操作属性，而不是 set/getAttribute（接下来的章节，我会分析它的实现）。

jQuery 早期的 attr 方法，其行为与 Prototype 一模一样。只不过 jQuery1.6 之前，是使用 attr 方法同时实现了读、写、删掉这三种操作。从易用性来说，不区分固有属性与自定义属性，由框架自动内部处理应该比 attr、prop 分家更容易接受。那么是什么逼迫 jQuery 这样做的呢？是选择器引擎。

jQuery 是最早以选择器为向导的类库。它最开始的选择器引擎是 Xpath 式，后来换成 Sizzle，以 CSS 表达式风格来选取元素。在 CSS2.1 引入了属性选择器[aaa=bbb]，IE7 也开始残缺支持。Sizzle 当然毫不含糊地实现了这语法。属性选择器是最早突破类名与 ID 的限制求取元素的。为了显摆它的强大，设计者让它拥有多种形态，满足人们各种匹配需要。比如，它可以只写属性名，[checked]，那么上例中的后四个元素都选中。[checked=true]选中第四个元素。[checked=xxx]选中第四个元素。true 与 xxx 都是用户在标签的预设值。而一字不差地取回这个预设值的工作也只有 setAttribute 才能做到。根据标准，setAttribute 是应该返回用户设置的那个字符串。而 el[xxx]这样的取法就不一定了，比如 el.checked 就返回布尔值，表示两种状态。这种通过状态取元素的方式就不归属性选择器管了，CSS3 又新设了个状态伪类满足人们的需求。

此外，属性还能以[name^=value]、[name*=value]、[name$=value]等更精致的方式来甄选元素，而这一切都建立在获取用户预设值的基础上。因此 jQuery 下了很大决心，把 prop 从 attr 切割出来。虽然为了满足用户的向前兼容需求，又偷偷地让 attr 做了 prop 的事，但以此为契机，jQuery 团队挖掘出更多兼容性问题与相应解决方案。元素内部撑起整个属性系统的 attributes 类数组属性也从幕后走到前台，为世人所知。

浏览器经过这么多年的发展，谁也说不清某个元素节点拥有多少个属性。for..in 循环也不行，因为它对不可遍历的属性无能为力。在 IE6、IE7 中，attributes 会包含上百个特性节点，不管你是用 setAttribute 定义的属性，还是以 el[xxx]=yyy 的定义的属性，还是没有定义的属性。可惜到 IE8 与其他浏览器中，你只看到寥寥可数的几个特性节点。称为显式属性（specified attribute）。

显式属性就是被显式设置的属性，分两种情况，一种是写在标签内的 HTML 属性，一种是通过 setAttribute 动态设置的属性，这些属性不分固有还是自定义，只要设置了，就出现在 attributes 中。在 IE6、IE7 中，我们也可以通过特性节点的 specified 属性判定它是否被显式设置。在 IE8 或

其他浏览器，我们想判定一个属性是否为显式属性，直接用 hasAttribute API 判定。

```
var  isSpecified = !"1"[0] ? function(el, attr){
     return el.hasAttribute(attr)
} : function(el, attr){
     var val = el.attributes[attr]
     return !!val && val.specified
}
```

此外，HTML5 对属性进行了更多的分类，从外包到不同的对象。dataset 对象装载着所有以 data-开头的自定义属性。classList 装载着元素的所有类名，并且提供一套 API 来操作它们。formData 装载着所有要提供到后台的数据，以表单元素的 name 值与 value 值构成的不透明对象。尽管如此，还是有大量属性是没有编制的，它们代表着元素的各种状态以及其他元素的联结。正因为如此，它们的值才五花八门，如 uniqueNumber、tabIndex、colspan、rowspan 为整数，designMode、unselectable、autocomplete 的值不是 on 就是 off，iframe 通过 frameborder 的值是 0 还是 1 决定显示边框，通过 scrolling 的值是 yes 还是 no 决定显示滚动条，表单元素的 form 属性总是指向其外围的表单对象，表单元素的 checked、disabled、readOnly 等属性总是返回布尔值……

面对如此庞杂的属性，主流框架也纷纷建立了对应的模块来整治它。在 jQuery 中有 attributes 模块，YUI3 是 dom-class、dom-attrs 模块，dojo 是 dom-attr、dom-prop、dom-class 模块。从内容来看，类名都被单独提出来处理，在 jQuery 中，表单元素的 value 也被单独提出来处理。Prototype.js 虽然没有划分出来，但对付一般属性有 readAttribute 与 writeAttribute，ID 有 identify，表单元素的 value 有$F，类名更是对应多个方法，这个阵营与 jQuery 的属性模块一模一样。在 1.5 节之前，Prototype.js 的属性模块一直是 jQuery 属性模块的蓝本。

10.1 如何区分固有属性与自定义属性

在 jQuery、mass Framework 中，提供两组方法来处理它们。但用户首先要知道他正在处理的东西是何物。虽然我们可以在以下链接查到每个元素拥有什么方法与属性，但显然不是每个人都那么勤奋。

http://msdn.microsoft.com/library/ms533029%28v=VS.85%29.aspx

我们需要做些实验找出其规律。

```
<!doctype html>
<html lang="en">
 <head>
   <meta charset="utf-8" />
   <meta content="IE=8" http-equiv="X-UA-Compatible"/>
   <title>如何区分属性与特性 by  司徒正美</title>
   <script type="text/javascript">
     var test = function(){
       var a = document.getElementById("test");
       a.setAttribute("title", "title");
       a.setAttribute("title2", "title2");
       alert(a.parentNode.innerHTML);
```

```
    }
  </script>
 </head>
 <body>
  <p><strong id="aaa">司徒正美</strong></p>
  <p><button type="button" onclick="test()">点我，进行测试</button></p>
 </body>
</html>
```

IE8 下打印出：

```
<STRONG id=test title=title title2="title2">司徒正美</STRONG>
```

Firefox15、Chrome23、IE9 下打印出：

```
<strong title2="title2" title="title" id="test">司徒正美</strong>
```

再将 a.setAttribute("title", "title"); a.setAttribute("title2", "title2")这两行改成 a.title = "title";a.title2 = "title2"。

IE8 下打印出：

```
<STRONG id=test title=title title2="title2">司徒正美</STRONG>
```

Firefox15、Chrome23、IE9 下打印出：

```
<strong title2="title2" title="title" id="test">司徒正美</strong>
```

经过观察，旧版本 IE 下固有属性，如 title、id 是不带引号的，自定义属性是带引号的，但在标准浏览器下，我们无法区分，否决此方案。

在元素中有一个 attributes 属性，里面有许多特性节点，这些特性节点在 IE 下有一种名为 expando 的布尔属性，可以判定它是否为自定义属性。但在标准浏览器下没有此属性，否决此方案。

```
function isAttribute(attr, host){ //仅有 IE
  var attrs =  host.attributes;
  return attrs[attr] && attrs.expando == true
}
```

我们再换一个角度来看，如果是固有属性，以 el[xxx] = yyy 的形式赋值，再用 el.getAttribute() 来取值，肯定能取到东西，但自定义属性就不一样。

```
var a = document.getElementById("test");
a.title = 222
console.log(a.getAttribute("title"))       //"222"
console.log(typeof a.getAttribute("title")) //"string"

a.setAttribute("custom", "custom")
console.log(a.custom)         //undefined
console.log(typeof a.custom)   //"undefined"
```

不过要注意 IE6、IE7 下的特例：

```
a.setAttribute("innerHTML","xxx")
console.log(a.innerHTML)          //"xxx"
```

即使如此，我们也可以轻松绕过这个陷井，建一个干净的同类型元素作为测试样本就行了。

```
var a = document.createElement("div")
console.log(a.getAttribute("title"))
console.log(a.getAttribute("innerHTML"))
console.log(a.getAttribute("xxx"))
console.log(a.title)
console.log(a.innerHTML)
console.log(a.xxx)
```

IE6、IE7 下返回""、""、null、""、""、undefined。IE8、IE9、Chrome23、FF15，Opera12 下返回 null、null、null、""、""、undefined。

因此我们可以推导出这样一个方法，回答我们这一节的标题。

```
function isAttribute(attr, host){
   //有些属性是特殊元素才有的，需要用到第二个参数
   host = host || document.createElement("div");
   return host.getAttribute(attr) === null && host[attr] === void 0
}
```

10.2 如何判定浏览器是否区分固有属性与自定义属性

经过社区的努力，现在大家都知道 IE6、IE7 不区分固有属性与自定义属性。这带来的结果是，对某个固有属性进行 setAttribute，我们不需要名字映射就能生效。但如果想通过浏览器嗅探法来识别 IE6、IE7，是远远不够的，因为用户可能使用旧版的标准浏览器上网。另外，我们也不得不考虑国内可恶的加壳 IE 浏览器。因此我们最好是通过特征侦测来判定浏览器是否支持此特性。

mootools 与 jQuery 各自使用了两种截然不同的方法来判定。mootools 以属性法设置一个自定义属性，然后通 getAttribute 去取，看是不是等于预设值，是就证明它不区分固有属性与自定义属性。jQuer 则是先用 setAttribute 去设置 className，然后看它是当作固有属性还是自定义属性，如果是自定义属性，我们用 el.className 是取不到值的，具体如表 10.2 所示。

```
var el = document.createElement("div")
el.random = 'attribute';              //mootools
console.log(el.getAttribute("random") != 'attribute')

el.setAttribute("className", "t");     //jQuery
console.log(el.className !== "t")
```

表 10.2

	IE6	IE7	IE8	FF9	FF15	Chrome23
Mootools	false,	false	false	true	true	true
jQuery	false	false	true	true	true	true

要看哪个更精确，只需要 IE8 浏览器就行了。IE8 大肆重写内核，是区分固有属性与自定义属

性的，因此 el.className 应该返回 undefined，导致结果为 true，因此 jQuery 获胜。jQuery 把这个特性称之为 getSetAttribute，意即 get/SetAttribute 没有 BUG；mass Framework 称之为 attrInnateName，意即不需要名字映射用原名就可以取值。它们都位于 supports 模块中。

10.3 IE 的属性系统的三次演变

微软在 IE4（1997 年）添加 setAttribute、getAttribute API，当时，DOM 标准（1998 年）还没有出来呢！而它的对手 NS6 到 2000 年才难产出来。

早期的 DOM API 于微软来说，只是它已有的一些方法的再包装，这些包装方法无法与它原来的那一套相媲美。document.getElementById("xxx")取元素节点，IE 下，这些带 ID 的元素节点自动就映射成一个个全局变量，直接 xxx 就拿到元素节点了，很便捷，或者使用 document.all[ID]来取，无论哪种都比标准的短；又如 getElementsByTagName，IE 下有 document.all.tags()方法，此方法直到 IE9 还有效。而 setAttribute("xxx","yyy")与 var ret = getAttribute("xxx")只不过是 el.xxx = "yyy"与 var ret = el.xxx 的另一种操作形式罢了。明白这一点我们就立即理解 IE 下这两个 API 的一些奇怪行为了。

el.setAttribute("className","aaa")是可行的，但 el.setAttribute("class","aaa")失败，因为我们可以用 className 修改类名，但不能用 class。

el.setAttribute("onclick", Function("alert(1)")) 是可行的，但 el.setAttribute("onClick", Function("alert(1)"))失败，因为我们是通过 el.onclick 绑定事件，而不是 el.onClick。

el.setAttribute("innerHTML"," test ")是可行的，因为我们可以用 innerHTML 添加内容。

element.setAttribute("style", "background-color: #fff; color: #000;")失败，因此 style 在 IE 下是个对象，"background-color: #fff; color: #000;"只能作为它的 cssText 属性的值。

DOM level 1 隔年就制定出来了。setAttribute/getAttribute 并没有微软想象得那么简单，它早期规定 getAttribute 必须也返回字符串，就算不存在也是空字符串。到后来，setAttribute/getAttribute 会对属性名进行小写化处理。用 getAttribute 去取没有显式设置的固有属性时，返回默认值（多数时候它为 null 或空字符串），对于没有显式设置的自定义属性，则返回 undefined。于是微软傻了眼，第一次改动就是匆匆忙忙支持小写化处理，并在 getAttribute 方法添加第二参数，以实现 DOM1 的效果。

getAttribute 的第二个参数有 4 个预设值：0 是默认，照顾 IE 早期的行为；1 属性名区分大小写；2 取出源代码中的原字符串值(注，IE5～IE7 对动态创建的节点没效，IE5～IE8 对布尔属性无效)；4 用于 href 属性，取得完整路径。

第二次演变是区分固有属性与自定义属性，取类名再也不用 className 了；布尔属性则遵循一个奇怪的规则，只要是显式设置了就返回与属性名同名的字符串，没有则返回""。我相信早期的标准浏览器也是这样做的。但标准浏览器很快就变脸了，统一返回用户的预设值。IE8 变得两边都不讨好，尽管它一心想与标准保持一致，标榜自己才是最标准的。比如它在当时还推出了 Object.defineProperty、querySelector、postMessage 等具有革命意义的新 API，但它们都与 W3C 争吵完的结果有出入。

第三次演变，不再对属性值进行干预，用户设什么返回什么，忠于用户的决定。对于这尘埃落定的方案，IE9 终于与标准吻合了。这也说明 IE 这种慢吞吞的大版本发布方式已经落伍了，虽然大家都对 FF 的版本狂飙有微词，但人家却保住了与 Chrome 叫板的地位。

综观 IE 的属性系统的悲剧，都因为微软总想抢占先机，而又与标准同步太慢所致。

10.4 className 的操作

我们操作一个属性通常只有三个选择：设置、读取、删除。但 className 有点特殊，它的值是可以用空格隔开，分为多个类名。因此对类名的操作变成读取、添加、删减。在上代王者 Prototype.js，就已经把人们想要的类名操作总结出来。

```
//Prototype 1.7
  classNames: function(element) {
    return new Element.ClassNames(element);
  },

  hasClassName: function(element, className) {
    if (!(element = $(element))) return;
    var elementClassName = element.className;
    return (elementClassName.length > 0 && (elementClassName == className ||
      new RegExp("(^|\\s)" + className + "(\\s|$)").test(elementClassName)));
  },

  addClassName: function(element, className) {
    if (!(element = $(element))) return;
    if (!Element.hasClassName(element, className))
      element.className += (element.className ? ' ' : '') + className;
    return element;
  },

  removeClassName: function(element, className) {
    if (!(element = $(element))) return;
    element.className = element.className.replace(
      new RegExp("(^|\\s+)" + className + "(\\s+|$)"), ' ').strip();
    return element;
  },

  toggleClassName: function(element, className) {
    if (!(element = $(element))) return;
    return Element[Element.hasClassName(element, className) ?
      'removeClassName' : 'addClassName'](element, className);
  },
```

除了这些外，还提供了一个 Element.ClassNames 类，用于直接操作类名。这不禁让我想起 HTML5 提供的 classList。

Prototype.js 基本定格了所有类名的操作方式，比如 toggleClassName 这种切换方法，纷纷被其他框架抄去。如果我们把上述方法名的 Name 去掉，就是 jQuery 的那一套方法名，虽然 YUI、EXT、dojo 都是这么叫，如表 10.3 所示。

表 10.3

MochiKit	hasElementClass addElementClass removeElementClass toggleElementClass setElementClass swapElementClass
Ten	hasClassName addClassName　removeClassName
jQuery	hasClass　addClass　removeClass toggleClass
YUI3、mass、kissy	hasClass　addClass　removeClass toggleClass replaceClass
dojo1.8	containsClass　addClass　removeClass toggleClass replaceClass
EXT4	containsClass　addClass　removeClass toggleClass replaceClass
RightJS	hasClass　addClass　removeClass　toggleClass　radioClass setClass getClass

由此可见，这套 API 在业内已被认可，如果我们写框架时，命名时最好不要与它们出入太大。我们亦可以把 Prototype.js 的实现精简一下，变成一些工具函数，在不引入类库时使用。

```
var getClass = function(ele) {
   return ele.className.replace(/\s+/," ").split(" ");
};

var hasClass =function(ele,cls){
   return -1 < (" "+ele.className+" ").indexOf(" "+cls+" ");
}

var addClass = function(ele,cls) {
   if (!this.hasClass(ele,cls))
      ele.className += " "+cls;
}

var removeClass = function(ele,cls) {
   if (hasClass(ele,cls)) {
      var reg = new RegExp('(\\s|^)'+cls+'(\\s|$)');
      ele.className=ele.className.replace(reg," ");
   }
}

var clearClass = function(ele,cls) {
   ele.className = ""
}
```

我们再来看框架里的方法是什么样子的。它们应该保证一致性（它与框架的其他方法的设计一致）、大众化（它与其他框架的相同功能的方法名差不多）。这两个原则就确保了 API 易于上手，吸引用户毫无障碍地从其他框架迁移过来。为此，我规定我的框架处理类名的方法都能同时处理多个元素，同时处理多个类名，与 jQuery 设计保持一致。API 分别有 addClass、removeClass、toggleClass、hasClass、replaceClass，与主流框架的 API 保持一致，这也与最底层的 DOM API 设计相仿（classList

提供了 add、remove、toggle、contains，它们刚好能对应上）。

我们先来实现 addClass 方法，重点在于去重。

jQuery 法：

```
addClass: function( value ) {
        var classNames, i, l, elem,setClass, c, cl;
        if ( jQuery.isFunction( value ) ) {
            return this.each(function( j ) {
                jQuery( this ).addClass( value.call(this, j, this.className) );
            });
        }
        if ( value && typeof value === "string" ) {
            classNames = value.split( /\s+/ );
            for ( i = 0, l = this.length; i < l; i++ ) {
                elem = this[ i ];
                if ( elem.nodeType === 1 ) {
                    if ( !elem.className && classNames.length === 1 ) {
                        elem.className = value;

                    } else {
                        setClass = " " + elem.className + " ";

                        for ( c = 0, cl = classNames.length; c < cl; c++ ) {
                            if ( setClass.indexOf( " " + classNames[ c ] + " " ) < 0 ) {
                                setClass += classNames[ c ] + " ";
                            }
                        }
                        elem.className = jQuery.trim( setClass );
                    }
                }
            }
        }

        return this;
},
```

jQuery 的参数可以为函数，而且通过 indexOf 来回避已有的类名，添加新类名后还需要 trim 操作，因此方法非常长。

正则去重法：

```
addClass: function(item) {
    if (typeof item == "string") {
        for (var i = 0, el; el = this[i++];) {
            if (el.nodeType === 1) {
                if (!el.className) {
                    el.className = item;
                } else {
                    //rclass = /(^|\s)(\S+)(?=\s(?:\S+\s)*\2(?:\s|$))/g,
                    el.className = (el.className + " " + item).replace(rclass, "");
                }
            }
        }
    }
```

```
    return this;
},
```

此方法的最大的优点是非常短，但正则回溯比较厉害，性能比较低下。不过这性能完全可以忽视。它最大的问题是，它是删掉排在前面的重复元素，与其他的算法结果不一致。其他都是遵循如果存在才添加的规则，因此是从后面删起。

hash 去重法：

```
addClass: function(item) {
    if (typeof item == "string") {
        for (var i = 0, el; el = this[i++];) {
            if (el.nodeType === 1) {
                if (!el.className) {
                    el.className = item;
                } else {
                    var obj = {}, set = "";
                    (el.className + " " + cls).replace(/\S+/g, function(w) {
                        if (!obj['@' + w]) {//对付旧版本 IE 的 toString
                            set += w + " ";
                            obj['@' + w] = 1;
                        }
                    });
                    el.className = set.slice(0, set.length - 1);
                }
            }
        }
    }
    return this;
},
```

此方法几乎无可挑剔。众所周知，hash 去重是最快的。但方法还是有点长。

数组去重法：

```
addClass: function(item) {
    if (typeof item == "string") {
        for (var i = 0, el; el = this[i++];) {
            if (el.nodeType === 1) {
                if (!el.className) {
                    el.className = item;
                } else {
                    var a = (el.className + " " + cls).match(/\S+/g);
                    a.sort();
                    for (var i = a.length - 1; i > 0; --i)
                    if (a[i] == a[i - 1]) a.splice(i, 1);
                    el.className = a.join(" ");
                }
            }
        }
    }
    return this;
},
```

这方法就是现在 mass Framework 采取的版本，性能不低，体积也颇为满意。

再来看 removeClass 的方法，我们要求它能同时删除多个类名，或什么也不传，清掉所有类名。前一种情况相当一种差集操作，这方法没什么好说的，直接给出 mass Framework 的实现：

```
removeClass: function( item ) {
    if ( (item && typeof item === "string") || item === void 0 ) {
        var classNames = ( item || "" ).match( rnospaces ), cl = classNames.length;
        for ( var i = 0, node; node = this[ i++ ]; ) {
            if ( node.nodeType === 1 && node.className ) {
                if ( item ) {//rnospaces = /\S+/g
                    var set = " " + node.className.match( rnospaces ).join(" ") + " ";
                    for ( var c = 0; c < cl; c++ ) {
                        set = set.replace(" " + classNames[c] + " ", " ");
                    }
                    node.className = set.slice( 1, set.length - 1 );
                } else {
                    node.className = "";
                }
            }
        }
    }
    return this;
},
```

hasClass 方法。这方法在切换卡组件经常用到。jQuery 只要匹配元素中有一个方法拥有此类名就返回 true，相当于数组的 some 方法的行为。mass Framework 添加了第二个参数，可以指定是 every 还是 some 行为。另外，判定一个元素拥有某个类名，在 HTML5 中，添加了一个新属性 classList，它有 add、toggle、remove、contains 等方法。可以试用一下。

```
//如果第二个参数为 true，要求所有匹配元素都拥有此类名才返回 true
hasClass: function( item, every ) {
    var method = every === true ? "every" : "some",
    rclass = new RegExp('(\\s|^)'+item+'(\\s|$)');//判定多个元素，正则比 indexOf 快点
    return $.slice(this)[ method ](function( el ){//先转换为数组
        return "classList" in el ? el.classList.contains( item ):
          (el.className || "").match(rclass);
     });
   },
```

toggleClass 方法。这方法常用于下拉菜单的展开收起。如果存在（不存在）就删除（添加）指定的类名。JQuery、mass Framework 等框架依傍数据缓存系统，还可以做出更高级的行为，在删除之前，先把它们存起来，那么下次要求把这些类名加上时，就直接从缓存系统中“反刍”出来。另外，根据 jQuery 的设计，toggleClass 方法与样式模块的 toggle 一样，如果用户传入的是布尔，那么 true 肯定表现为 addClass，false 表现为 removeClass。Dojo、YUI、kissy 的 toggleClass 虽然没有与缓存系统联袂，但也支持使用布尔参数强迫表现为 addClass 或 removeClass 的行为。

```
toggleClass: function( value ){
    var type = typeof value, className, i,
    classNames = type === "string" && value.match( /\S+/g ) || [];
    return this.each(function( el ) {
        i = 0;
```

```
        if(el.nodeType === 1){
            var self = $( el );
            if(type == "string" ){
                while ( (className = classNames[ i++ ]) ) {
                    self[ self.hasClass( className ) ? "removeClass" : "addClass" ]( className );
                }
            } else if ( type === "undefined" || type === "boolean" ) {
                if ( el.className ) {
                    $._data( el, "__className__", el.className );
                }
                el.className = el.className || value === false ? "" : $._data( el, "__className
__") || "";
            }
        }
    });
},
```

replaceClass 方法。这个 jQuery 没有提供，但 EXT4、dojo、YUI3 都有它的身影，也是一个非常常用的操作，不能拉下它。replaceClass 要来传入两个类名，A 与 B，如果匹配元素存在类名 A 则将其置换为类名 B。

```
replaceClass: function( old, neo ){
        for ( var i = 0, node; node = this[ i++ ]; ) {
            if ( node.nodeType === 1 && node.className ) {
                var arr = node.className.match( rnospaces ), arr2 = [];
                for ( var j = 0; j < arr.length; j++ ) {
                    arr2.push( arr[j] == old ? neo : arr[j]);
                }
                node.className = arr2.join(" ");
            }
        }
        return this;
    },
```

有了以上这五个方法，就能应付所有关于类名的需求。

10.5 Prototype.js 的属性系统

无论哪个框架类库，早年都是将它们混在一起操作。杰出的代表是 Prototype.js 的 readAttribute、writeAttribute 与 jQuery 的 attr。我先来看 Prototype.js 的伟大遗产吧。

- 名字映射的发明。
- href,src 的 IE 处理。
- getAttributeNode 的发掘。
- 事件钩子的处理。
- 布尔属性的处理。
- style 属性的 IE 处理。

```
// Prototype.js 1.61
readAttribute = function(element, name) {
```

```
    element = $(element);
    if (Prototype.Browser.IE) {
        var t = Element._attributeTranslations.read;
        if (t.values[name])
            return t.values[name](element, name);
        if (t.names[name])
            name = t.names[name];
        if (name.include(':')) {
            return (!element.attributes || !element.attributes[name]) ? null :
                   element.attributes[name].value;
        }
    }
    return element.getAttribute(name);
}
```

Prototype.js 认为总是 IE 在拖后腿，只对 IE 进行调教就行了。处理方式与 IE 的属性系统演变史一样，对属性名的名字映射、属性值进行忠实用户的字符串还原。于是搞出 Element._attribute Translations 对象，它有两大块，read 对象用于 readAttribute，write 对象用于 writeAttribute，然后每部分都有 names 映射列表与 values 函数集。

```
Element._attributeTranslations = (function() {
    var classProp = 'className',
           forProp = 'for',
           el = document.createElement('div');

    el.setAttribute(classProp, 'x');

    if (el.className !== 'x') {
        el.setAttribute('class', 'x');
        if (el.className === 'x') {
            classProp = 'class';
        }
    }
    el = null;

    el = document.createElement('label');
    el.setAttribute(forProp, 'x');
    if (el.htmlFor !== 'x') {
        el.setAttribute('htmlFor', 'x');
        if (el.htmlFor === 'x') {
            forProp = 'htmlFor';
        }
    }
    el = null;

    return {
        read: {
            names: {
                'class': classProp,
                'className': classProp,
                'for': forProp,
                'htmlFor': forProp
            },
            values: {
```

```
            _getAttr: function(element, attribute) {
                return element.getAttribute(attribute);
            },
            _getAttr2: function(element, attribute) {
                return element.getAttribute(attribute, 2);
            },
            _getAttrNode: function(element, attribute) {
                var node = element.getAttributeNode(attribute);
                return node ? node.value : "";
            },
            _getEv: (function() {

                var el = document.createElement('div'), f;
                el.onclick = Prototype.emptyFunction;
                var value = el.getAttribute('onclick');

                if (String(value).indexOf('{') > -1) {
                    f = function(element, attribute) {
                        attribute = element.getAttribute(attribute);
                        if (!attribute)
                            return null;
                        attribute = attribute.toString();
                        attribute = attribute.split('{')[1];
                        attribute = attribute.split('}')[0];
                        return attribute.strip();
                    };
                }
                else if (value === '') {
                    f = function(element, attribute) {
                        attribute = element.getAttribute(attribute);
                        if (!attribute)
                            return null;
                        return attribute.strip();
                    };
                }
                el = null;
                return f;
            })(),
            _flag: function(element, attribute) {
                return $(element).hasAttribute(attribute) ? attribute : null;
            },
            style: function(element) {
                return element.style.cssText.toLowerCase();
            },
            title: function(element) {
                return element.title;
            }
        }
    }
}
})();

Element._attributeTranslations.write = {
    names: Object.extend({
        cellpadding: 'cellPadding',
```

```
        cellspacing: 'cellSpacing'
    }, Element._attributeTranslations.read.names),
    values: {
        checked: function(element, value) {
            element.checked = !!value;
        },
        style: function(element, value) {
            element.style.cssText = value ? value : '';
        }
    }
};

Element._attributeTranslations.has = {};

$w('colSpan rowSpan vAlign dateTime accessKey tabIndex ' +
        'encType maxLength readOnly longDesc frameBorder').each(function(attr) {
    Element._attributeTranslations.write.names[attr.toLowerCase()] = attr;
    Element._attributeTranslations.has[attr.toLowerCase()] = attr;
});

(function(v) {
    Object.extend(v, {
        href: v._getAttr2,
        src: v._getAttr2,
        type: v._getAttr,
        action: v._getAttrNode,
        disabled: v._flag,
        checked: v._flag,
        readonly: v._flag,
        multiple: v._flag,
        onload: v._getEv,
        onunload: v._getEv,
        onclick: v._getEv,
        ondblclick: v._getEv,
        onmousedown: v._getEv,
        onmouseup: v._getEv,
        onmouseover: v._getEv,
        onmousemove: v._getEv,
        onmouseout: v._getEv,
        onfocus: v._getEv,
        onblur: v._getEv,
        onkeypress: v._getEv,
        onkeydown: v._getEv,
        onkeyup: v._getEv,
        onsubmit: v._getEv,
        onreset: v._getEv,
        onselect: v._getEv,
        onchange: v._getEv
    });
})(Element._attributeTranslations.read.values);
```

我很喜欢 Prototype.js 这种自动执行函数，有效隔绝不同模块间的代码污染。对于 for、class 的映射，它是使用特性侦测来实现的。然后对不同的属性，采取不同的方法取值，其实就是一个适配器。整个 Element._attributeTranslations.values 就是个 Bug 发掘器，发现绝大部分的 Bug，以后 jQuery

的工作就是进行改良与深挖。

IE6～IE8 一些表示 URL 的属性会返回补全的改过编码的路径，如 action、background、Base Href、cite、codeBase、data、dynsrc、href、longDesc、lowsrc、pluginspage、profile、src、url 与 vrml。但一般框架只处理 href、src。处理方式很简单，将 getAttriubte 的第二个参数设为 2。

```
<a href="index.html">home</a>
<script>
var link = document.getElementsByTagName('a')[0];
link.getAttribute('href') // "http://www.cnblogs.com/rubylouvre/index.html";
link.getAttribute('href',2) //"index.html";
</script>
```

至于编码的情况就更复杂了。我们可以使用<a href="${链接 1}"作为实验样本，如表 10.4 所示。

表 10.4

	.href	getAttribute("href")	getAttribute("href", 2)
IE6	转绝对，不编码	转绝对，不编码	正常
IE7	转绝对，汉字不编码，特殊符号编码	转绝对，汉字不编码，特殊符号编码	正常
IE8	转绝对，汉字不编码，特殊符号编码	正常	正常
Firefox 3.0+	转绝对，全部编码	正常	正常
Chrome 2.0	转绝对，全部编码	正常	正常
Safari 4.0	转绝对，汉字编码，特殊符号不编码	正常	正常
Opera 10.0beta	转绝对，汉字编码，特殊符号不编码	正常	正常

其中，“正常”的意思是，得到 href 属性里原始链接，不自动转绝对地址，汉字和符号都不编码。

IE6～IE7，对于 form 元素用 getAttribute 取属性值，可能得到它辖下的 ID 值或 name 值相同的表单元素。在 Prototype.js 下，只处理比较常见的 action 属性。下面的例子，在 IE6、IE7 下都是返回元素节点。

```
<form  action="#" >
       <input id="name" >
       <input id="action" >
       <input name="id"  >
       <input name="length"  >
       <input id="xxx" >
       <input id="yyy" >
 </form>
var el = document.getElementsByTagName("form")[0]
alert(el.getAttribute("action"))
alert(el.getAttribute("id"))
alert(el.getAttribute("name"))
alert(el.getAttribute("length"))
alert(el.getAttribute("xxx"))
alert(el.getAttribute("yyy"))
```

Prototype.js 于是发掘到 getAttributeNode 方法。当然也可以用 attriubtes[xxx]的方法，只要得到特性节点就好办。

对于事件钩子，Prototype.js 可谓是费力来修改回调的 toString。不过在此之前，它还判定元素是否支持事件。如果 Prototype.js 的作者再多做些测试，其实 getAttributeNode 与 attriubtes[xxx]方式也能取到正确值。

布尔属性，Prototype.js 是用_flag 内部方法去取。如果 el[xxx]是返回 true 的情况，直接返回与属性同名的字符串，否则返回字符串。这是一个没有办法的办法，因为对于布尔属性，IE6～IE8 都无法取得用户预设值，无论是 getAttribute 的第二个参数方式，getAttributeNode 方法，attributes[xxx]方式，outerHTML 的字符串裁剪方式。后来标准浏览器改变游戏规则了，对于真值的布尔属性不再返回同名字符串，假值不再返回空字符串，忠于用户输入，于是 Prototype.js 这个 readAttribute 方法在不同浏览器中的返回值不同。jQuery 做出的处理是，编写了一个正则，用于匹配布尔属性，统一所有浏览器在真值的情况下返回同名字符串，假值返回 undefined。

```
rboolean = /^(?:autofocus|autoplay|async|checked|controls|defer|disabled|hidden|loop
|multiple|open|readonly|required|scoped|selected)$/i
```

但谁也无法保证，这正则已经囊括所有布尔属性，而且大多数布尔只对个别元素有效，因此这是晕招。不过比起 Prototype.js 已是很大进步，后者只处理表单元素的几个特定属性：disabled、checked、readonly、multiple。

style 属性的 IE 处理，统一转换它的 cssText 属性。这个所有框架都一样。

在写入属性值时，Prototype.js 察觉到属性有两种形态。一种是存在于 HTML 标签内的，不区分大小写，不过现在浏览器都统一将它们小写化。对于存在多个字母个数与排列顺序相同但大小写不同的元素，标准浏览器会做合并处理，只保留第一个属性。DOM 形态是区分大小写，并且对保留字进行回避，如著名的 class、for，有的是两个单词组成，需要转换成驼峰风格，有的则完全是没有规则。正如英语单词的单复数转换，一开始没有规范，大家都照抄网景的，然后又搞些独创的属性，经过这么多的发展，我们一个小小映射能应对 90%的特殊情况就是谢天谢地了。

Prototype.js 列出了 15 个需要映射的情况，而 jQuery 的只有 12 个，其中 encoding 这映射在标准浏览器中还不会出现。这一数字在 mass Framework 飙升了 36 个，算是目前兼容较全面的。

Prototype.js 干这活的对象为 Element._attributeTranslations.write.names，一共处理以下属性：accessKey、cellPadding、cellSpacing、className、colSpan、dateTime、encType、frameBorder、htmlFor、longDesc、maxLength、readOnly、rowSpan、tabIndex、vAlign。

jQuery 干这活的对象为$.propFix，一共处理以下属性：cellPadding、cellSpacing、className、colSpan、colSpan、contentEditable、encoding、frameBorder、htmlFor、rowSpan、rowSpan、useMap。

mass Framework 干这活的对象为$.propMap，一共处理以下属性：acceptCharset、accessKey、allowTransparency、bgColor、cellPadding、cellSpacing、ch、chOff、className、codeBase、codeType、colSpan、contentEditable、dateTime、defaultChecked、defaultSelected、defaultValue、encoding、frameBorder、htmlFor、httpEquiv、isMap、longDesc、marginHeight、marginWidth、maxLength、noHref、noResize、noShade、readOnly、rowSpan、tabIndex、useMap、vAlign、vSpace、valueType。

Prototype.js 的 writeAttribute 方法是在 1.6 中出现的，决定了传入 null 与 false 属性值进行删除操作的游戏规则，被 jQuery1.6 抄去。然而事实上，传入 false 消去当前布尔属性的效果，也只有 removeAttribute 一途。这是聪明的做法。但至于把它应用到固有属性，就需慎重考虑了。反过来想，

正因为 Prototype.js 想用一个 API 同时处理固有属性与自定义属性，才导致删除操作也不得不保持一致。jQuery 追随 Prototype.js 的步伐，于是掉进同一个坑。

```
//Prototype.js1.61
writeAttribute = function(element, name, value) {
    element = $(element);
    var attributes = {}, t = Element._attributeTranslations.write;
    if (typeof name == 'object')
        attributes = name;
    else
        attributes[name] = Object.isUndefined(value) ? true : value;
    for (var attr in attributes) {
        name = t.names[attr] || attr;
        value = attributes[attr];
        if (t.values[attr])
            name = t.values[attr](element, value);
        if (value === false || value === null)
            element.removeAttribute(name);
        else if (value === true)
            element.setAttribute(name, name);
        else
            element.setAttribute(name, value);
    }
    return element;
}
```

10.6 jQuery 的属性系统

早期的 jQuery 针对当时如日中天的 Prototype.js，提出了一个重要的口号：它更小。

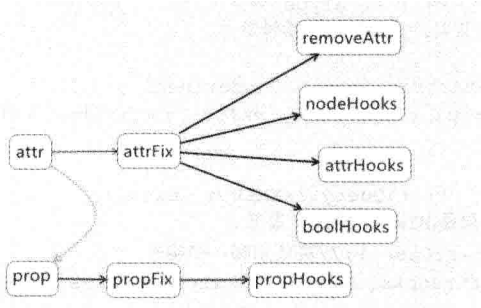
我还是希望能带来一点启迪与勉励。jQuery 的属性系统也是经年累月，量变引发质变的结果。

```
// jQuery1.01
attr: function(elem, name, value){
        var fix = {
            "for": "htmlFor",
            "class": "className",
            "float": "cssFloat",
            innerHTML: "innerHTML",
            className: "className",
            value: "value",
            disabled: "disabled"
        };

        if ( fix[name] ) {
            if ( value != undefined ) elem[fix[name]] = value;
            return elem[fix[name]];
        } else if ( elem.getAttribute ) {
            if ( value != undefined ) elem.setAttribute( name, value );
            return elem.getAttribute( name, 2 );
        } else {
            name = name.replace(/-([a-z])/ig,function(z,b){return b.toUpperCase();});
            if ( value != undefined ) elem[name] = value;
```

```
            return elem[name];
        }
    },
```

这个方法然后不断膨胀，加入从 Prototype.js 发掘出来的特殊属性处理，以及社区提交的补丁。到 jQuery1.5.2，这个 attr 方法已经接近一百行的规模。于是在 1.6 模块，把 1.5 节中 CSS 模块想出的好方法后，cssHooks 适配器机制移值过来，解决了扩展的难题。

在 jQuery1.6 中存在 4 个适配器，从单词直译过来，就是钩子：formHooks、attrHooks、propHooks、valHooks。formHook 是在 attr 方法对付旧版本 IE 的 form 元素用的。到 jQuery1.6.1，增加一个 boolHooks 对付布尔属性。到 jQuery1.6.3，人们发现 IE 大多数情况使用 getAttriubteNode 就能取到正确值，因此对 formHooks 重构一下，更名为 nodeHooks，便形成今天 jQuery 的属性系统，如图 10.1 所示。



▲图 10.1

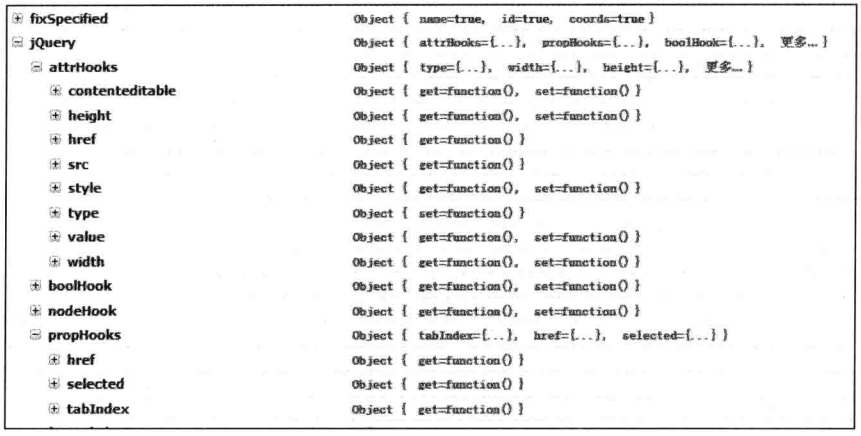
上面就是 jQuery 的属性系统概览图。新生的 prop 方法异常简单，复杂性都移到钩子。古老的 attr 方法则无比复杂，兼任读、写、删除 3 职，由于 IE 的情况不得不动用到 3 个钩子。钩子只处理有问题的属性，不处理一般情况。

```
//jquery1.83
prop: function( elem, name, value ) {
    var ret, hooks, notxml,  nType = elem.nodeType;

    if ( !elem || nType === 3 || nType === 8 || nType === 2 ) {
        return;//跳过注释节点、文本节点、特性节点
    }

    notxml = nType !== 1 || !jQuery.isXMLDoc( elem );
    if ( notxml ) {//如果是 HTML 文档的元素节点
        name = jQuery.propFix[ name ] || name;
        hooks = jQuery.propHooks[ name ];
    }

    if ( value !== undefined ) {//写方法
        if ( hooks && "set" in hooks && (ret = hooks.set( elem, value, name )) !== undefined )
{
            return ret;//处理特殊情况
```

```
        } else {//处理通用情况
            return ( elem[ name ] = value );
        }

    } else {//读方法
        if ( hooks && "get" in hooks && (ret = hooks.get( elem, name )) !== null ) {
            return ret;//处理特殊情况
        } else {//处理通用情况
            return elem[ name ];
        }
    }
},
attr: function( elem, name, value ) {
    var ret, hooks, notxml, nType = elem.nodeType;

    if ( !elem || nType === 3 || nType === 8 || nType === 2 ) {
        return;//跳过注释节点、文本节点、特性节点
    }
    if ( typeof elem.getAttribute === "undefined" ) {
        return jQuery.prop( elem, name, value );//文档与 window，只能使用 prop
    }

    notxml = nType !== 1 || !jQuery.isXMLDoc( elem );
    if ( notxml ) {//如果是 HTML 文档的元素节点
        name = name.toLowerCase();//决定用哪一个钩子
        hooks = jQuery.attrHooks[ name ] || ( rboolean.test( name ) ? boolHook : nodeHook );
    }

    if ( value !== undefined ) {
        if ( value === null ) {//模仿 Prototype.js，移除属性
            jQuery.removeAttr( elem, name );

        } else if ( hooks && notxml && "set" in hooks &&
            (ret = hooks.set( elem, value, name )) !== undefined ) {
            return ret;//处理特殊情况

        } else {//处理通用情况
            elem.setAttribute( name, value + "" );
            return value;
        }
    } else if ( hooks && notxml && "get" in hooks && (ret = hooks.get( elem, name )) !==
null ) {
        return ret;//处理特殊情况

    } else {//处理通用情况
        ret = elem.getAttribute( name );
        return ret === null ? undefined :  ret;
    }
},
```

每个钩子的结构也不一样，propHooks、attrHooks 里面是以属性命名的对象，里面或存在 get 方法、或 set 方法、或两者都有，boolHooks、nodeHooks 则只有 set、get 两个方法，如图 10.2 所示。

钩子在所有特殊情况下都命中的结构图。目前在 Firefox 下命中最少，在 IE6、IE7 命中最多。



▲图 10.2

jQuery 对属性系统的主要贡献是发明更多兼容性问题与解决方法，具体如下。

（1）tabindex 的取值问题。tabindex 默认情况下只对表单元素与链接有效，对于这些元素没有显式设置会返回 0，对于像 DIV 这样普通元素返回–1，但 IE 都是返回 0。jQuery 对此做了统一。

（2）Safari 下，option 元素的 selected 取值问题，必须向上访问一下 select 元素才得到结果。

（3）表单元素的 value 属性的操作。由于表单元素种类够多，存在严重的兼容性问题，jQuery 为此下了很大功夫让我们 write less ,do more！

但 jQuery 的属性系统也存在明显的缺憾，具体如下。

（1）名字映射是一种穷举机制，但 jQuery 也有待完善，让 attrFix、propFix 都没有尽责。

（2）对布尔属性的判定存在硬编码，准确率极低。

（3）它添加了一个与 removeAttr 对称的 removeProp 方法，但里面的实现用到 delete 操作符，在 chrome 真的是把固有属性从原型中删掉，导致下次赋值时出错。

（4）nodeHooks 是使用 getAttributeNode 实现的，虽然能应对所有自定义属性，但判定某些固有属性是否为显式属性时，却不得不使用 fixSpecified 补漏洞。可惜 fixSpecified 对象作为一种穷举机制，在吝惜类库体积的 jQuery 中注定水土不服。

（5）因为旧版本 IE7 不能修改表单元素的 type 属性，就阻止我们在所有浏览器修改 type，这种行为很难服众。比如密码域的 replaceholder 模拟，苹果应用上那些人性化的掩码，就需要我们修改 type。

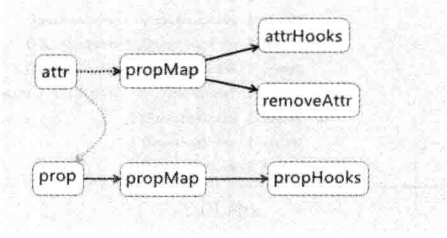
因此接着下来重点介绍 mass Framework 的实现，它在 jQuery 的属性系统上做了大量改进，克服了以上缺点。

10.7 mass Framework 的属性系统

mass Framework 使用的是真正的适配器。jQuery 的之所以叫钩子，它只会特定属性进行回调

处理（同步的回调），最后根据它的结果是否直接返回还是再走默认处理流程。在 prop 方法，默认处理，就是对目标进行数组法取值赋值，在 attr 方法就是直接调用 setAttribute 或 getAttribute。mass Framework 在用户抵达适配器之前，已经做好一切判定了，是写方法还是读方法，是使用特殊处理，还是默认处理，布尔属性也作为一种特殊处理的适配函数。在 prop 方法中，默认处理是"@default:get"、"@default:set"这一组方法，在 attr 方法中，默认处理是"@w3c:get"、"@w3c:set"，"@ie:get"、"@ie:set"这两组方法。我们只要直接返回适配方法的结果就行了，整个流程十分清晰简单。

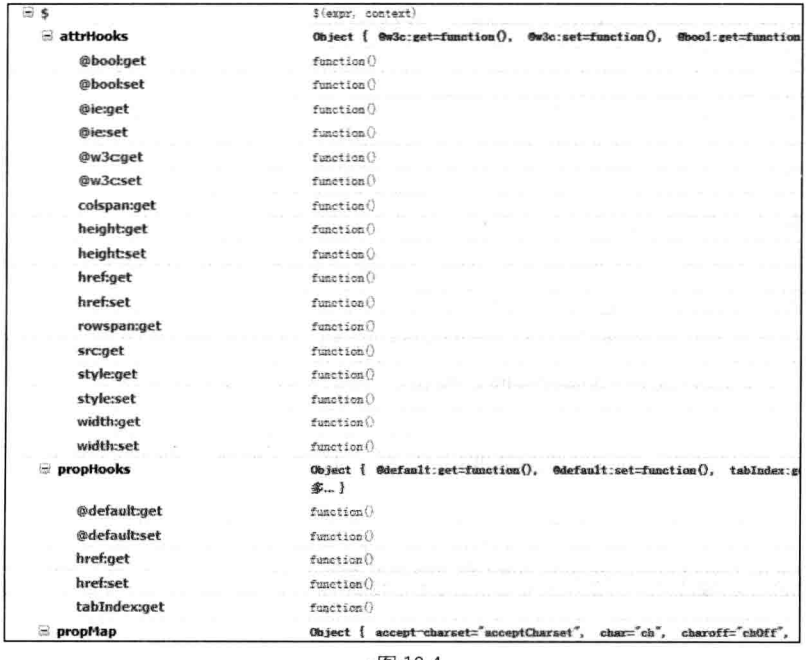
mass Framework 的属性系统概览如图 10.3 所示。



▲图 10.3

attr 只处理元素节点，如果是 XML 元素或其他对象类型，转交 prop 处理。如果 isXML 为 false，那么就尝试在 propMap 取得其 JavaScript 属性名，然后在 attrHooks 或 propHooks 取得钩子函数处理。在 attr 中，如果值为 false 或 null，需要做移除操作。

下面是它的适配器在所有特殊情况下都命名的结构，如图 10.4 所示。



▲图 10.4

```
//https://github.com/RubyLouvre/mass-Framework/blob/1.4/attr.js
prop: function(node, name, value) {
    if ($["@bind"] in node) {
        if (node.nodeType === 1 && !$.isXML(node)) {
            name = $.propMap[name.toLowerCase()] || name;
        }
        var access = value === void 0 ? "get" : "set";
        return($.propHooks[name + ":" + access] ||
                $.propHooks["@default:" + access])(node, name, value);
    }
}
attr: function(node, name, value) {
    if ($["@bind"] in node) {
        if (typeof node.getAttribute === "undefined") {
            return $.prop(node, name, value);
        }
//这里只剩下元素节点
        var noxml = !$.isXML(node),
                type = "@w3c";
        if (noxml) {
            name = name.toLowerCase();
            var prop = $.propMap[name] || name;
            if (!support.attrInnateName) {
                type = "@ie";
            }
            var isBool = typeof node[prop] === "boolean" &&
                typeof defaultProp(node, prop) === "boolean"; //判定是否为布尔属性
        }
//移除操作
        if (noxml) {
            if (value === null || value === false && isBool) {
                return $.removeAttr(node, name);
            }
        } else if (value === null) {
            return node.removeAttribute(name);
        }
//读写操作
        var access = value === void 0 ? "get" : "set";
        if (isBool) {
            type = "@bool";
            name = prop;
        }
        ;
        return(noxml && $.attrHooks[name + ":" + access] ||
                $.attrHooks[type + ":" + access])(node, name, value);
    }
}
```

这里着重说明一下布尔属性，mass Framework 并没有使用 hash 或正则来检测它们，而是使用一种全新的检测手段。首先判定值是否为布尔，然后忽略用户用 el [xxx] = true 的方式添加自定义属性的情况，我们可以参考 10.1 节提到 isAttribute 的实现，新建一个干净的同类型元素节点作为测试样本。如果在第二个条件中它依然为布尔，那它肯定是布尔属性。为了减少创建元素的次数，我

们可以把结果缓存起来，以后碰到同类型标签的同名属性，直接返回结果。

```
var cacheProp = {}
function defaultProp(node, prop){
    var name = node.tagName+":"+prop;
    if(name in cacheProp){
        return cacheProp[name]
    }
    return cacheProp[name] = document.createElement(node.tagName)[prop]
}
```

下面说说 mass Framework 对 jQuery 的改进。

（1）tabIndex 在浏览器中差异表现有两点。一、默认拥有 tabIndex 元素的种类，W3C 与 Netscape6 仅是把 tabIndex 添加到有限的几个元素，a、area、button、input、object、select、textarea 上，也就是所谓的表单元素与链接。IE4 则比它多以下元素：applet、body、div、embed、isindex、marquee、span、table、与 td。到了 IE5，几乎所有能渲染的元素都拥有这属性（像 br 元素就是不能渲染的）。二、没有显式设置时 el.tabIndex 的默认值。在标准浏览器下，表单元素与链接是返回 0，普通元素返回–1，在 IE6～IE8 下没有做区分，都返回 0。

```
"tabIndex:get": function( node ) {
    //http://www.cnblogs.com/rubylouvre/archive/2009/12/07/1618182.html
    var ret = node.tabIndex;
    if( ret === 0 ){//在标准浏览器下，不显式设置时，表单元素与链接默认为 0，普通元素为-1
        ret = rtabindex.test(node.nodeName) ? 0 : -1
    }
    return ret;
}
```

（2）在 IE6～IE8 下，href 属性多出一个适配方法。缘由是，如果一个 A 标签，它里面包含@字符，并且没任何元素节点，那么它里面的文本会变成链接值。修复方法就是插入一个隐藏的元素节点。

```
if ( !support.attrInnateValue ) {//IE6~IE8
    // http://gabriel.nagmay.com/2008/11/javascript-href-bug-in-ie/
    $.propHooks[ "href:set" ] = $.propHooks[ "href:set" ] = function( node, name, value ) {
        var b
        if(node.tagName == "A" && node.innerText.indexOf("@") > 0
            && !node.children.length){
            b = node.ownerDocument.createElement('b');
            b.style.display = 'none';
            node.appendChild(b);
        }
        node.setAttribute(name, value+"");
        if (b) {
            node.removeChild(b);
        }
    }
}
```

（3）在旧版本 IE 下取得用户预设值。jQuery 主要依赖 getAttributeNode API 与一个不怎么可靠

的 fixSpecified。mass Framework 则直接取其 outerHTML、innerHTML，进行字符串切割，得到用户的预设值。这样就轻松避开 form 元素与 button 元素的陷阱了。

```
rattrs = /\s+([\w-]+)(?:=("[^"]*"|'[^']*'|[^\s>]+))?/g,
rquote = /^['"]/
//.....
"@ie:get": function( node, name ){
    var str = node.outerHTML.replace(node.innerHTML, ""), obj = {}, k, v;
    while (k = rattrs.exec(str)) { //属性值只有双引号与无引号的情况
        v = k[2]
        obj[ k[1].toLowerCase() ] = v ? rquote.test( v ) ? v.slice(1, -1) : v : ""
    }
    return obj[ name ];
},
```

（4）对固有属性的移除。jQuery 是直接 undefined+delete，但这样在 Chrome 中会出事。mass Framework 是分开处理，对于文档对象与 window 使用 undefined，对于元素节点还原为默认值。默认值可以通过上面的 defaultProp 得到。

```
removeProp: function( node, name ) {
    if(node.nodeType === 1){
        if(!support.attrInnateName){
            name = $.propMap[ name.toLowerCase() ] ||  name;
        }
        node[name] = defaultProp(node, name)
    }else{
        node[name] = void 0;
    }
},
```

10.8 value 的操作

之所以把它独立出来，是因为它非常重要，涉及与后端的交互。而 value 值，一般而言，只有表单元素的 value 才对我们有用。问题是表单元素的种类非常多，每一个取法与赋值各有不同，因此我们最好在内部使用一个适配器来实现它。有关表单元素取 value 值的大部分浏览器兼容性问题被 jQuery 发现并消灭了，mass Framework 只是在字节上做压缩处理。

下面分别描述每个表单元素的情况。

select 元素，它的 value 值就是其被选中的 option 孩子的 value 值。不过，select 有两种形态，一种 type 为 select-one，另一种为 select-multiple，就是当用户显式设置了 multiple 属性。在多选形态下，我们可以在 Windows 下按住 Ctrl 键进行多选，Mac 下按住 command 键进行多选。

option 元素，它的 value 值可以是 value 属性的值，亦可以是其中间的文本，换言之是 innerText。当用户没有显式设置 value 属性时，它就取 innerText，不过这个 innerText 要用 text 属性来取，就像 script 标签那样。可能有人会问，为什么不用 innerHTML 呢？因为这个 option 元素的 text 属性比 innerHTML 多做了一个 trim 操作，去掉两边的空白（旧版本 IE 的 innerHTML 会做 trim 操作，标准浏览器不会）。那么如何判定它是显示设置了 value 属性呢？这在 10.1 节已提到，元素节点有个

attribute 属性，它是个类数组对象，里面是一个个对象，每个对象拥有 value、name、specified、ownerElement 等一大堆属性，我们判定 specified 是否为 true 就行了。IE8 与其他浏览器，我们还可以使用 hasAttribute 方法来判定。

button 元素，它的取值情况与 option 元素有点类似但又不尽然。在 IE6、IE7 中，它是取元素的 innerText，到 IE8 时它才与其他浏览器保持一致，取 value 属性的值。不过在标准浏览器下，button 标签只有当其为提交按钮，并且点击它时，才会提交其自身的 value 值。我们应该统一返回其 value 值。

checkbox、radio 在设置 value 时，应该考虑对 checked 属性的修改。

因此在 val 方法对应的适配器内，应该有六组方法，对 select、option 的特殊处理，对 button 的兼容性处理，对 checkbox、radio 的 checked 值处理，最后是默认处理。

```
var valHooks = {
    "option:get": function(node) {
        var val = node.attributes.value;
        //黑莓手机 4.7 下 val 会返回 undefined,但我们依然可用 node.value 取值
        return !val || val.specified ? node.value : node.text;
    },
    "select:get": function(node, value, getter) {
        var option, options = node.options,
                index = node.selectedIndex,
                one = node.type === "select-one" || index < 0,
                values = one ? null : [],
                max = one ? index + 1 : options.length,
                i = index < 0 ? max : one ? index : 0;
        for (; i < max; i++) {
            option = options[i];
            //旧版本 IE 在 reset 后不会改变 selected，需要改用 i === index 判定
            //我们过滤所有 disabled 的 option 元素，但在 Safari5 下，如果设置 select 为 disable,
            //那么其所有孩子都 disable
            //因此当一个元素为 disable，需要检测其是否显式设置了 disable 及其父节点的 disable 情况
            if ((option.selected || i === index) && !(support.optDisabled ? option.disabled :
/ disabled=/.test(option.outerHTML.replace(option.innerHTML, "")))) {
                value = getter(option);
                if (one) {
                    return value;
                }
                //收集所有 selected 值组成数组返回
                values.push(value);
            }
        }
        return values;
    },
    "select:set": function(node, name, values, getter) {
        values = [].concat(values); //强制转换为数组
        for (var i = 0, el; el = node.options[i++]; ) {
            el.selected = !!~values.indexOf(getter(el));
        }
        if (!values.length) {
            node.selectedIndex = -1;
```

```
        }
    }
}

//checkbox 的 value 默认为 on，唯有 Chrome 返回空字符串
if (!support.checkOn) {
    valHooks["checked:get"] = function(node) {
        return node.getAttribute("value") === null ? "on" : node.value;
    };
}
//处理单选框、复选框在设值后 checked 的值
valHooks["checked:set"] = function(node, name, value) {
    if (Array.isArray(value)) {
        return node.checked = !!~value.indexOf(node.value);
    }
}
```

至于应该分配到哪个适配方法，则是由 getValType 方法决定的。

```
function getValType(el) {
    var ret = el.tagName.toLowerCase();
    return ret === "input" && /checkbox|radio/.test(el.type) ? "checked" : ret;
}
```

最后 val 方法只是个代理，它唯一要做的是，将参数转换为字符串或字符串数组（针对 select 元素），然后让钩子函数干活就行了。

```
val = function(item) {
    var getter = valHooks["option:get"];
    if (arguments.length) {
        if (Array.isArray(item)) {
            item = item.map(function(item) {
                return item == null ? "" : item + "";
            });
        } else if (isFinite(item)) {
            item += "";
        } else {
            item = item || "";
            //我们确保传参为字符串数组或字符串，null/undefined 强制转换为"",number 变为字符串
        }
        return $.access(this, function(el) {
            if (this === $) { //getter
                var ret = (valHooks[getValType(el) + ":get"] ||
                        $.propHooks["@default:get"])(el, "value", getter);
                return typeof ret === "string" ? ret.replace(rreturn, "") : ret == null ? "" : ret;
            } else { //setter
                if (el.nodeType === 1) {
                    (valHooks[getValType(el) + ":set"] ||
                            $.propHooks["@default:set"])(el, "value", item, getter);
                }
            }
        }, 0, arguments);
    }
}
```

第11章 事件系统

事件系统是一个框架非常重要的部分，用于响应用户的各种行为。

浏览器提供了3种层次的API。

最原始的是写在元素标签内。

再次是在脚本中，以el.onXXX= function绑定的方式，通称为DOM0事件系统。

最后是多投事件系统，一个元素的同一类型事件可以绑定多个回调，通称为DOM2事件系统。由于浏览器大战，现存在两套API。

- IE与Opera方

绑定事件：el.attachEvent(“on”+type，callback)

卸载事件：el.detachEvent(“on”+type，callback)

创建事件：document.createEventObject()

派发事件：el.fireEvent(type, event)

- W3C方

绑定事件：el.addEventListener(type, callback, [phase])

卸载事件：el.removeEventListener(type, callback, [phase])

创建事件：el.createEvent(types)

初始化事件：event.initEvent()

派发事件：el.dispatchEvent(event)

从API的数量与形式来看，W3C提供的复杂很多，相对应也强大很多，下面我会逐一详述。

首先呈上我几个简单的实现。如果是简单的页面，就用简单的方式打发，没有必要动用框架。不过，事实上，整个事件系统就建立在它们的基础上。

```
function addEvent(el, type, callback, useCapture ){
    if(el.dispatchEvent){//W3C 方式优先
        el.addEventListener( type, callback, !!useCapture );
    }else {
        el.attachEvent( "on"+type, callback );
    }
    return callback;//返回 callback 方便卸载时用
}

function removeEvent(el, type, callback, useCapture ){
    if(el.dispatchEvent){//W3C 方式优先
```

```
        el.removeEventListener( type, callback, !!useCapture  );
    }else {
        el.detachEvent( "on"+type, callback );
    }
}

function fireEvent(el, type, args, event){
    args = args || {}
    if(el.dispatchEvent){//W3C 方式优先
        event = document.createEvent("HTMLEvents");
        event.initEvent(type, true, true );
    }else {
        event = document.createEventObject();
    }
    for(var i in args)  if(args.hasOwnProperty(i)){
        event[i] = args[i]
    }
    if(el.dispatchEvent){
        el.dispatchEvent(event);
    }else{
        el.fireEvent('on'+type,event)
    }
}
```

11.1 onXXX 绑定方式的缺陷

onXXX 既可以写在 HTML 标签内，也可以独立出来，作为元素节点的一个特殊属性来处理。不过作为一种古老的绑定方式，它很难预测到后来人对这方面的扩展。

总结起来有以下不足。

（1）onXXX 对 DOM3 新增事件或 FF 某些私有实现无法支持，主要有以下事件：

DOMActivate
DOMAttrModified
DOMAttributeNameChanged
DOMCharacterDataModified
DOMContentLoaded
DOMElementNameChanged
DOMFocusIn
DOMFocusOut
DOMMouseScroll
DOMNodeInserted
DOMNodeInsertedIntoDocument
DOMNodeRemoved
DOMNodeRemovedFromDocument
DOMSubtreeModified
MozMousePixelScroll

不过稍安勿躁，上面的事件就算是框架，也只用到 DOMContentLoaded 与 DOMMouseScroll。DOMContentLoaded 用于检测 DomReady，DOMMouseScroll 用于在 FF 模拟其他浏览器的 mousewheel 事件。

（2）onXXX 只允许元素每次绑定一个回调，重复绑定在冲掉之前的绑定。

（3）onXXX 在 IE 下回调没有参数，在其他浏览器下回调的第一个参数是事件对象。

（4）onXXX 只能在冒泡阶段可用。

11.2 attachEvent 的缺陷

attachEvent 是微软在 IE5 添加的 API，Opera 也支持，也对于 onXXX 方式，它可以允许同一元素同种事件绑定多个回调，也就是所谓的多投事件机制。但它带来的麻烦只多不少，存在以下几点缺陷。

（1）IE 下只支持微软系的事件，DOM3 事件一概不能用。

（2）IE 下 attchEvent 回调中的 this 不是指向被绑定元素，而是 window!

（3）IE 下同种事件绑定多个回调时，回调并不是按照绑定时的顺序依次触发的！

（4）IE 下 event 事件对象与 W3C 的存在太多差异了，有的无法对上号，比如 currentTarget。

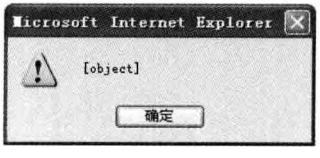
（5）IE 还是只支持冒泡阶段。

不过 IE 还是不断进步的。

```
<!DOCTYPE html>
<html>
    <head>
        <title></title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <script>
            window.attachEvent("onload", function(e) {
                alert(e);
            });
        </script>
    </head>
    <body>
        <div>TODO write content</div>
    </body>
</html>
```

IE6、IE7（见图 11.1）。

IE8（见图 11.2）。



▲图 11.1

▲图 11.2

IE9（见图 11.3）。

▲图 11.3

从一开始不知弹出什么东西，到现在明确这是一个事件对象。不过，W3C 的事件系统已经是大势所趋，微软也 hold 不住，从 IE9 起支持 W3C 那一套 API，这对我们实现事件代理非常有帮助！

11.3 addEventListener 的缺陷

W3C 这一套 API 也不是至善至美，毕竟标准总是滞后于实现，剩下的那 4 个标准浏览器各有自己的算盘，它们之间亦有不一致的地方。

（1）新事件非常不稳定，可能还没有普及开就被废弃。在早期的 Sizzle 选择器引擎中，有这么几句：

```
document.addEventListener("DOMAttrModified", invalidate, false);
document.addEventListener("DOMNodeInserted", invalidate, false);
document.addEventListener("DOMNodeRemoved", invalidate, false);
```

现在这 3 个事件被废弃了（准确来说，所有变动事件都完蛋了），FF14 与 Chrome18 开始用 MutationObserver 代替它。

（2）Firefox 既不支持 focusin、focus 事件，也不支持 DOMFocusIn、DOMFocusOut，直接现在也不愿意用 mousewheel 代替 DOMMouseScroll。Chrome 不支持 mouseenter 与 mouseleave。

因此不要以为标准浏览器就肯定会实现 W3C 钦定的标准事件，它们也有抗旨的时候，特征侦测必不可少。最恶心的是国内一些浏览器套用 webkit 内核，为了“超越”本版浏览器的 HTML5 跑分（HTML5test.com），竟然实现了一些无用的空接口来骗过特征侦测，因此必要时，我们还得使用非常麻烦的功能侦测来检测浏览器是否支持此事件。

（3）CSS3 给私有实现添加自定前缀标识的坏习惯也漫延到一些与样式息息相关的事件名上。比如 transitionend 事件名，这个后缀名与大小写混合成 5 种形态，相当棘手。

（4）第三个、第四个、第五个参数。

第三个参数 useCapture 只有非常新的浏览器中才是可选项，比如 FF6 或之前是必选的，为安全起见，请确保第三个参数为布尔。

第四个参数听说是 FF 专有实现，允许跨文档监听事件。

第五个参数，的确存在第五参数，不过它只存在于 Flash 语言的同名方法中。现在前端工程师还是要求助于 Flash，就作为一个知识点收下吧。有的面试官跨界考这种东西。在 Flash 下，addEventListener 的第四个参数用于设置该回调执行时的顺序，数字大的优先执行，第五个参数用来指定对侦听器函数的引用是弱引用还是正常引用。

（5）事件对象成员的不稳定。

W3C 那套是从浏览器商抄来的，人家都用了这么久，难免与标准不一致。

FF 下 event.timeStamp 返回 0 的问题，这个 BUG,2004 年就有人提交了，直到 2011 年才被修复。https://bugzilla.mozilla.org/show_bug.cgi?id=238041

Safari 下 event.target 可能是返回文本节点。

event.defaultPrevented，event.isTrusted 与 stopImmediatePropagation 的可用性很低，它们属于 DOM3 event 规范。

defaultPrevented 属性是用于确定事件对象有没有调用 preventDefault 方法。之前标准浏览器都统一用 getPreventDefault 方法来干这事，在 jQuery 源码中，你会发现它是用 isDefaultPrevented 方法来做。不过，isDefaultPrevented 的确曾列入 W3C 草稿，可参见这里：

http://www.w3.org/TR/2003/WD-DOM-Level-3-Events-20030331/ecma-script-binding.html

isTrusted 属性用于表示当前事件是否是由用户行为触发(比如说真实的鼠标点击触发一个 click 事件)，还是由一个脚本生成的（使用事件构造方法，比如 event.initEvent)。

stopImmediatePropagation 用于阻止当前事件的冒泡行为并且阻止当前事件所在元素上的所有相同类型事件的事件处理函数的继续执行。

IE9+全部实现（但是，IE9、IE10 的 event.isTrusted 有 BUG。link.click()后返回的也是 true)。

Chrome5～Chrome17 部分实现（event.isTrusted 未支持)。

Safari5 才部分实现（event.isTrusted 未支持)。

Opera10、Opera11 部分实现（stopImmediatePropagation 以及 event.isTrusted 未实现，而仅仅实现了 defaultPrevented)。

Opera12 部分实现（stopImmediatePropagation 仍然未实现，但实现了 e.isTrusted)。

Firefox1.0～Firefox5，stopImmediatePropagation 和 defaultPrevented 未实现，仅仅实现了 event.isTrusted。isTrusted 在成为标准前，是 Firefox 的私有实现。

Firefox6～Firefox10，仅未实现 stopImmediatePropagation。Firefox11，终于实现了 stopImmediate Propagation。

（6）标准浏览器没有办法模拟像 IE6～IE8 的 propertychange 事件。

虽然标准浏览器有 input、DOMAttrModified、MutationObserver，但比起 propertychange 都弱爆了。propertychange 可以监听多种属性变化，而不单单是 value 值，另外它不区分 attribute 和 property，因此你无论是通过 el.xxx = yyy,还是 el.setAttribute(xxx, yyy)都接触过此事件。具体可参阅下面这篇博文。

http://www.cnblogs.com/rubylouvre/archive/2012/05/26/2519263.html

11.4 Dean Edward 的 addEvent.js 源码分析

这是 Prototype 时代早期出现的一个事件系统，jQuery 事件系统的源头，亮点如下。

- 有意识地屏蔽 IE 与 W3C 在阻止默认行为与事件传播的接口差异。
- 处理 IE 执行回调时的顺序问题。
- 处理 IE 的 this 的指向问题。
- 没有平台检测代码，因为是使用最通用原始的 onXXX 构建。
- 完全跨浏览器（包括 IE4 和 NS4）。

```
//http://dean.edwards.name/weblog/2005/10/add-event/
function addEvent(element, type, handler) {
    //回调添加 UUID，方便移除
    if (!handler.$$guid)
        handler.$$guid = addEvent.guid++;
    //元素添加 events，保存所有类型的回调
    if (!element.events)
        element.events = {};
    var handlers = element.events[type];
    if (!handlers) {
        //创建一个子对象，保存当前类型的回调
        handlers = element.events[type] = {};
        //如果元素之前以 onXXX = callback 的方式绑定过事件，则成为当前类别第一个被触发的回调
        //问题是这回调没有 UUID，只能通过 el.onXXX = null 移除
        if (element["on" + type]) {
            handlers[0] = element["on" + type];
        }
    }
    //保存当前的回调
    handlers[handler.$$guid] = handler;
    //所有回调统一交由 handleEvent 触发
    element["on" + type] = handleEvent;
}

addEvent.guid = 1;//UUID
//移除事件，只要从当前类别的储存对象 delete 就行
function removeEvent(element, type, handler) {
    if (element.events && element.events[type]) {
        delete element.events[type][handler.$$guid];
    }
}
function handleEvent(event) {
    var returnValue = true;
    //统一事件对象阻止默认行为与事件传统的接口
    event = event || fixEvent(window.event);
    //根据事件类型，取得要处理回调集合，由于 UUID 是纯数字，因此可以按照绑定时的顺序执行
    var handlers = this.events[event.type];
    for (var i in handlers) {
        this.$$handleEvent = handlers[i];
        //根据返回值判定是否阻止冒泡
        if (this.$$handleEvent(event) === false) {
            returnValue = false;
        }
    }
    return returnValue;
```

```
}
;
//对 IE 的事件对象做简单的修复
function fixEvent(event) {
    event.preventDefault = fixEvent.preventDefault;
    event.stopPropagation = fixEvent.stopPropagation;
    return event;
};
fixEvent.preventDefault = function() {
    this.returnValue = false;
};
fixEvent.stopPropagation = function() {
    this.cancelBubble = true;
};
```

不过在 Dean Edward 对应的博文的评论中就可以看到许多指正与有用的 patch。比如说，既然所有的修正都冲着 IE 去，那么标准浏览器直接使用 addEventListener 就行。有的还提到在 iframe 中点击事件时，事件对象不对的问题，提交以下有用补丁：

```
event = event || fixEvent(((this.ownerDocument || this.document || this).parentWindow ||
window).event);
```

其中第 54 条回复，直接导致了 jQuery 数据缓存的系统的产生。为了避免交错引用产出的内存泄漏，建议元素就分配一个 UUID，所有回调都放到一个对象中储存。

```
function addEvent(element, type, handler) {
    if (!handler.$$guid)
        handler.$$guid = addEvent.guid++;
    //每个元素都分配一个 UUID，用于访问它们的所有回调
    if (!element.$$guid)
        element.$$guid = addEvent.guid++;
    if (!addEvent.handlers[element.$$guid])
        addEvent.handlers[element.$$guid] = {};
    //每个元素的回调都分类储存在不同的 hash 中
    var handlers = addEvent.handlers[element.$$guid][type];
    if (!handlers) {
        handlers = addEvent.handlers[element.$$guid][type] = {};
        if (element["on" + type]) {
            handlers[0] = element["on" + type];
        }
    }
    handlers[handler.$$guid] = handler;
    element["on" + type] = handleEvent;
}
addEvent.guid = 1;
// 放置所有回调的仓库
addEvent.handlers = {};
```

但随着时间的推移，使用者发现 onXXX 在 IE 存在不可消弥的内存泄漏。因此你翻看 jQuery 早期的版本，1.01 是照抄 Dean Edward 的，1.1.3.1 版，开始吸收上面提到的第 54 条回复的建议，元素只分配一个 UUID，回调集中储放的方式了，并使用 attachEvent/ removeEventListener 绑定事

件——每个元素只绑定一次，然后所有回调都在类似 handleEvent 的函数中调用。不过这是后话，无损 Dean Edward 的光辉形象，他的 addEvent 在当时可是有着划时代的意义。无入侵式 JavaScript 只有在这样键壮的事件系统出现后才能得到迅猛发展。

附内存泄漏的报告与测试样本。

http://javascript.crockford.com/memory/leak.html

http://www.sitepoint.com/forums/showthread.php?742336-attachEvent-amp-ie6

11.5 jquery1.8.2 的事件模块概览

jQuery 的事件模块发端于 Dean Edward 的 addEvent，然后它不断吸收社区的插件与补丁，发展成为非常成熟的事件系统。其中不得不提的是其事件代理与事件派发机制。

早在 2007 年，Brandon Aaron 为 jQuery 写了一个划时代的插件叫 livequery，它可以监听后来插入的元素的事件。比如说，一个表格，我们为 tr 元素绑定 mouseover/mouseout 事件时，只有十行，然后我们又动态添加了 20 行，这 20 个 tr 元素也同样能执行 mouseover/mouseout 回调。魔术在于，它并没有把事件侦探器绑定在 tr 元素上，而是绑定最顶层的 document 上，然后通过事件冒泡，取得事件源，判定它是否匹配用户给定的 CSS 表达式，才执行用户回调，具体可以看下面这篇博文。

http://brandonaaron.net/blog/2007/08/19/new-plugin-live-query

如果一个表格有 100 个 tr 元素，每个都要绑定 mouseover/mouseout 事件，改成事件代理的方式，可以节省 99 次绑定，这优化很好，更何况它能监听将来添加的元素，因此立马被吸收到 jQuery1.3 中去，成为它的 live 方法。再把一些明显的 Bug 修复了，jQuery1.32 成为受欢迎的版本，与后来的 jQuery1.42 都是里程碑式！

不过话说回来，live 方法需要对某些不冒泡的事件做些处理，比如一些表单事件，有的只冒泡到 form,有的冒泡到 document，有的压根不冒，如表 11.1 所示。

表 11.1

	IE6	IE8	FF3.6	Opera10	Chrome4	Safari4
submit	form	form	document	document	document	document
reset	form	form	document	document	form	form
change	不冒泡	不冒泡	document	document	不冒泡	不冒泡
click	document	document	document	document	document	document
select	不冒泡	不冒泡	document	document	不冒泡	不冒泡

对于 focus、blur、change、submit、reset、select 等不会冒泡的事件，在标准浏览器中，我们可以设置 addEventListener 的最后一个参数为 true 轻松搞定。IE 就有点麻烦了，要用 focusin 代替 focus，focusout 代替 blur，selectstart 代替 select。Change、submit 与 reset 就复杂了，必须利用其他事件来模拟，还要判断事件源的类型，selectedIndex、keyCode 等相关属性，这课题最后由一个叫 reglib 的库搞定。reglib 的作者还写了一篇很著名的博文《Goodbye mouseover，hello mouseenter》，来推

广微软系的两个事件 mouseenter 与 mouseleave。正如你们今天看到那样，jQuery 全面接纳了它们。相关链接如下。

https://blogs.oracle.com/greimer/entry/mouse_over_out_versus_mouse

live 方法带来的全新体验是空前的，但毕竟要冒泡到最顶层，对 IE 还是有点坎坷，有时还会失灵。最好能指定父节点，一个绑定时已经存在的父节点，这样就不用费力了。当时有三篇博文提出相近的方案，它们给出的接口一篇比一篇接近 John Resig 接纳的方案。

http://danwebb.net/2008/2/8/event-delegation-made-easy-in-jquery

http://raxanpdi.com/blog-jquery-event-delegate.html

http://blog.threedubmedia.com/2008/10/jquerydelegate.html

比如最后那篇博文，它已经是这个样子：

```
$("div").delegate( 'click', 'span', function( event ){
      $( this ).toggleClass('selected');
      return false;
});
```

并提出对应解除代理的 API——undelegate。

而 jQuery1.42 在 2010 年 2 月 19 日推出时，也是这两个接口，前两个参数只调换一下：

```
$("div").delegate( "span","click", function( event ){
      $( this ).toggleClass('selected');
      return false;
});
```

正所谓“众人拾柴火焰高”，jQuery 的强大不无道理。在 jQuery1.8 中，它又吸收 dperini / nwevents 的点子，改进其事件代理，大大提高代理性能。

先看一下其主要接口，如图 11.4 所示。

其中 bind、unbind、one、trigger、toggle、hover、ready 一开始就有。

tirggerHandler 是 jQuery1.23 增加的，内部依赖于 trigger，只对当前匹配元素的第一个有效，不冒泡不触发默认行为。

live 与 die 是 jQuery1.3 增加的，用于事件代理，统一由 document 代理。

delegate 与 undelegate 是 jQuery1.42 增加的，允许指定代理元素的事件代理，它内部是利用 live，die 实现的。

on 与 off 是 jQuery1.7 增加的，目的是统一事件接口。bind、one、live、delegate 直接由 on 衍生，unbind、die、undelegate 直接由 off 衍生。

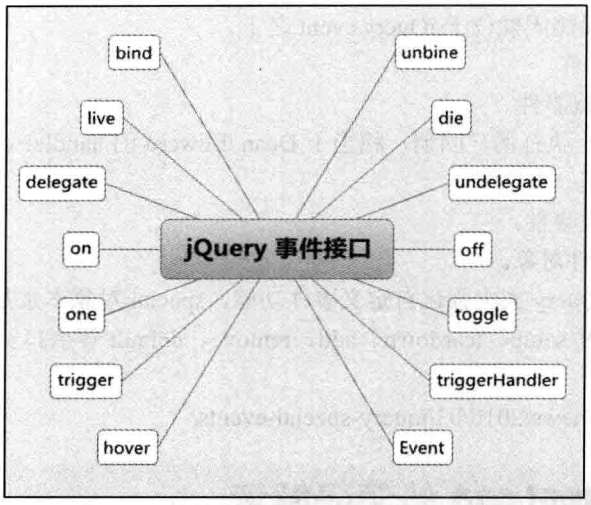
hover 用于模拟 css 的 hover 效果,内部依赖于 mouseenter 与 mouseleave。

ready 可以看作为 load 事件的增强版，获取最早的 DOM 可用时机后立即执行各种 DOM 操作。

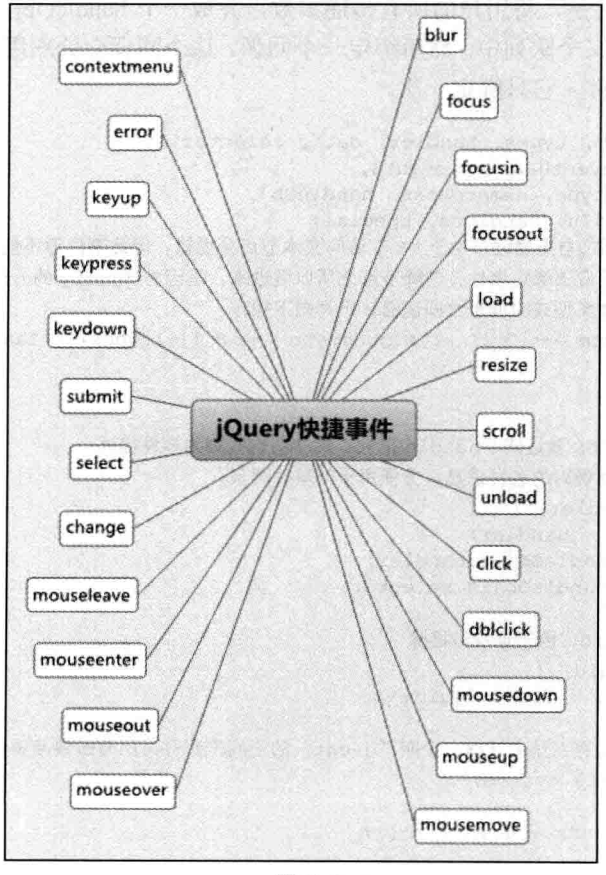
toggle 是 click 的加强版，每次点击都执行不同的回调并切换到下一个。

triggle 与 triggleHandler 是 jQuery 的 fireEvent 实现。

此外，jQuery 还有 25 个以事件类型命名的快捷方法，当传参数个数为 2 时表现为绑定事件，个数为 0 表现为派发事件，如图 11.5 所示。


jQuery 事件接口
bind
unbine
live
die
delegate
undelegate
on
off
one
toggle
trigger
triggerHandler
hover
Event

▲图 11.4


jQuery快捷事件
blur
contextmenu
focus
error
focusin
keyup
focusout
keypress
load
keydown
resize
submit
scroll
select
unload
change
click
mouseleave
dblclick
mouseenter
mousedown
mouseout
mouseup
mouseover
mousemove

▲图 11.5

不过最重要的基础设施无疑位于 jQuery.event 之下。

add 方法用于绑定事件。

remove 方法用于卸载事件。

dispath 方法用于统一执行用户回调，相当于 Dean Edward 的 handleEvent 方法，因此 1.7 节前它的名字一直叫 handle。

trigger 方法用于派发事件。

fix 方法用于修正事件对象。

从 jQuery 1.4 起，jQuery 大大强化自定义事件功能。special 对象本来是用于修正个别事件的，现在它允许这些事件通过 setup、teardown、add、remove、default 等接口实现 DOM 事件的各种行为。详见以下博文：

http://benalman.com/news/2010/03/jquery-special-events/

11.6 jQuery.event.add 的源码解读

add 方法的主要目的是，将用户的所有传递参数，并成一个 handleObj 对象放到元素对应的缓存体中的 events 对象的某个队列中，然后绑定一个回调。这个回调会处理用户的所有回调，因此对于每一个元素每一种事件，它只绑定一次。

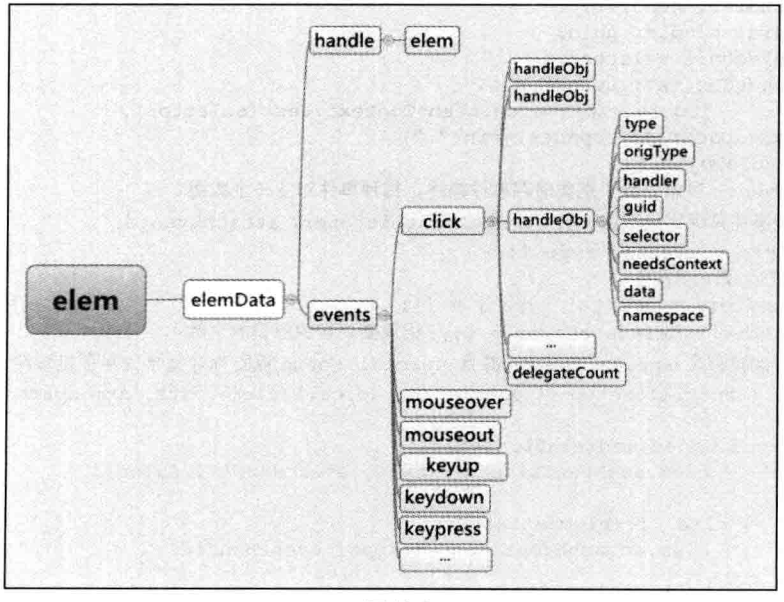
```
add = function(elem, types, handler, data, selector) {
    var elemData, eventHandle, events,
            t, tns, type, namespaces, handleObj,
            handleObjIn, handlers, special;
    //如果 elem 不能添加自定属性，由于 IE 下访问文本节点会抛错，因此事件源不能为文本节点
    //注释节点本来就不应该绑定事件，注释节点之所以混进来，是因为 jQuery 的 html 方法所致
    //如果没有指定事件类型或回调也立即返回，不再向下操作
    if (elem.nodeType === 3 || elem.nodeType === 8 || !types || !handler || !(elemData =
jQuery._data(elem))) {
        return;
    }
    //取得用户回调与 CSS 表达式，handleObjIn 这种结构我称为事件描述
    //记叙用户绑定此回调时的各种信息，方便用于“事件拷贝”
    if (handler.handler) {
        handleObjIn = handler;
        handler = handleObjIn.handler;
        selector = handleObjIn.selector;
    }
    //确保回调拥有 UUID，用于查找与移除
    if (!handler.guid) {
        handler.guid = jQuery.guid++;
    }
    //为此元素在数据缓存系统中开辟一个叫“event”的空间来保存其所有回调与事件处理器
    events = elemData.events;
    if (!events) {
        elemData.events = events = {};
    }
    eventHandle = elemData.handle;//事件处理器
```

```
    if (!eventHandle) {
        elemData.handle = eventHandle = function(e) {
            //用户在事件冒充时，被二次 fire 或者在页面 unload 后触发事件
            return typeof jQuery !== "undefined" && (!e || jQuery.event.triggered !== e.type) ?
                    jQuery.event.dispatch.apply(eventHandle.elem, arguments) :
                    undefined;
        };
        //原注释是说，防止 IE 下非原生事件内存泄漏，不过我觉得直接的影响是明确了 this 的指向
        eventHandle.elem = elem;
    }
    //通过空格隔开同时绑定多个事件，比如 jQuery(...).bind("mouseover mouseout", fn);
    types = jQuery.trim(hoverHack(types)).split(" ");
    for (t = 0; t < types.length; t++) {

        tns = rtypenamespace.exec(types[t]) || [];  //取得命名空间
        type = tns[1];//取得真正的事件
        namespaces = (tns[2] || "").split(".").sort();//修正命名空间
        //并不是所有事件都能直接使用，比如 FF 下没有 mousewheel，需要用 DOMMouseScroll 冒充
        special = jQuery.event.special[ type ] || {};
        //有时候我们只需要在事件代理时进行冒充，比如 FF 下的 focus、blur
        type = (selector ? special.delegateType : special.bindType) || type;

        special = jQuery.event.special[ type ] || {};
        // 构建一个事件描述对象
        handleObj = jQuery.extend({
            type: type,
            origType: tns[1],
            data: data,
            handler: handler,
            guid: handler.guid,
            selector: selector,
            needsContext: selector &&
                    jQuery.expr.match.needsContext.test(selector),
            namespace: namespaces.join(".")
        }, handleObjIn);
        //在 events 对象上分门别类储存事件描述，每种事件对应一个数组
        //每种事件只绑定一次监听器（即 addEventListener，attachEvent）
        handlers = events[ type ];
        if (!handlers) {
            handlers = events[ type ] = [];
            handlers.delegateCount = 0;//记录要处理的回调的个数
            //如果存在 special.setup 并且 special.setup 返回 0 才直接使用多投事件 API
            if (!special.setup || special.setup.call(elem, data, namespaces, eventHandle)
=== false) {
                if (elem.addEventListener) {
                    elem.addEventListener(type, eventHandle, false);

                } else if (elem.attachEvent) {
                    elem.attachEvent("on" + type, eventHandle);
                }
            }
        }
```

```
            if (special.add) {//处理自定义事件
                special.add.call(elem, handleObj);

                if (!handleObj.handler.guid) {
                    handleObj.handler.guid = handler.guid;
                }
            }

            // Add to the element's handler list, delegates in front
            if (selector) {//如果是使用事件代理，那么把此事件描述放到数组的前面
                handlers.splice(handlers.delegateCount++, 0, handleObj);
            } else {
                handlers.push(handleObj);
            }
            //用于 jQuery.event.trigger，如果此事件从来没有绑定过，也没有必要进入 trigger 的真正处理逻辑
            jQuery.event.global[ type ] = true;
        }
        //防止 IE 内存泄漏
        elem = null;
    }
```

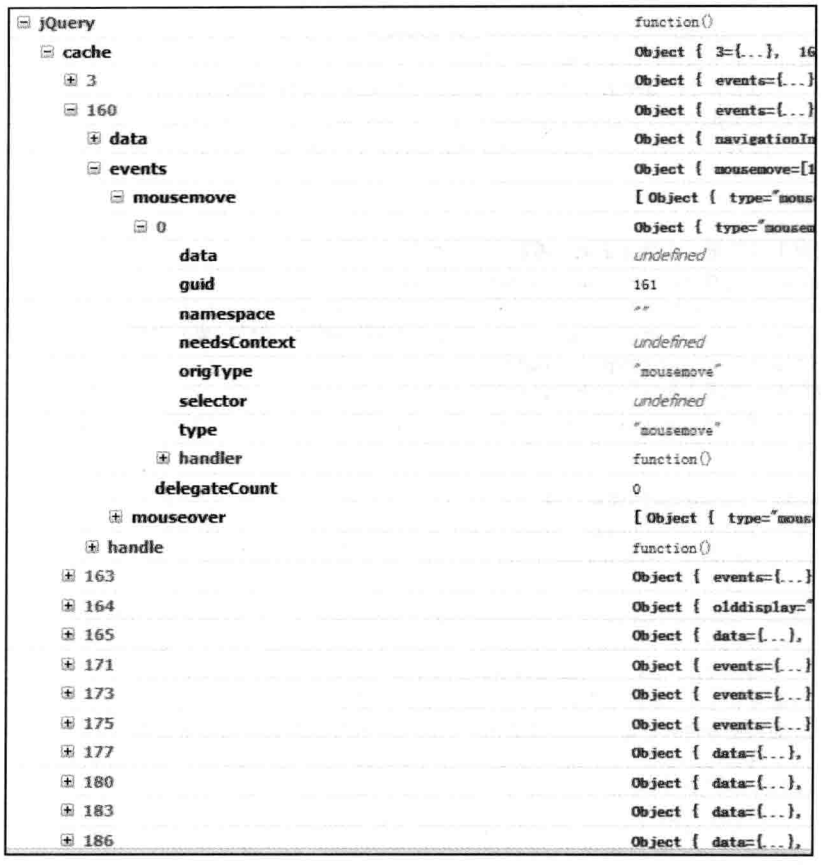
从上面的注释中，我们可以一眼得知 jQuery 的回调不再直接与元素挂钩，而是通过 UUID 访问数据缓存系统，抵达对应的 events 对象，再根据事件类型得到一组事件描述。并事件描述里面没有事件源的记录，因此非常方便到处挪动，为事件克隆大开方便之门。当然，这其中数据缓存系统是关键，接着看下去，你就会发现其事件代理部分对数据缓存依赖得更严重。

下面是元素数据缓存与事件描述之间的结构，如图 11.6 所示。



▲图 11.6

这是在 firebug 下查看到的结构，如图 11.7 所示。



▲图 11.7

11.7 jQuery.event.remove 的源码解读

remove 方法的主要目的是，根据用户传参，找到事件队列，从里面把匹配的 handleObj 对象移除，在参数不足的情况，可能传移除 N 个或所有。当队列的长度为零就移除事件，当 events 为空对象，则清掉 UUID。

```
remove = function( elem, types, handler, selector ) {
    var t, tns, type, origType, namespaces, origCount,
    j, events, special, eventType, handleObj,
    elemData = jQuery.hasData( elem ) && jQuery._data( elem );
    //如果不支持添加自定义属性或没有缓存与事件有关的东西，立即返回
    if ( !elemData || !(events = elemData.events) ) {
        return;
    }
    //hover 转换为 "mouseenter mouseleave"，并且按空格进行切割，方便移除多种事件类型
```

```
    types = jQuery.trim( hoverHack( types || "" ) ).split(" ");
    for ( t = 0; t < types.length; t++ ) {
        tns = rtypenamespace.exec( types[t] ) || [];
        type = origType = tns[1];//取得事件类型
        namespaces = tns[2];//取得命名空间
        if ( !type ) {//如果没有指定事件类型，则移除所有事件类型或移除所有与此命名空间有关的事件类型
            for ( type in events ) {
                jQuery.event.remove( elem, type + types[ t ], handler, selector, true );
            }
            continue;
        }
        //利用事件冒充，取得真正用于绑定的事件类型
        special = jQuery.event.special[ type ] || {};
        type = ( selector? special.delegateType : special.bindType ) || type;
        eventType = events[ type ] || [];//取得装载事件描绘对象的数组
        origCount = eventType.length;
        //取得用于过滤命名空间的正则，没有为 null
        namespaces = namespaces ? new RegExp("(^|\\.)" + namespaces.split(".").sort().
join("\\.(?:.*\\.|)") + "(\\.|$)") : null;

        // 移除符合条件的事件描述对象
        for ( j = 0; j < eventType.length; j++ ) {
            handleObj = eventType[ j ];

            if ( ( origType === handleObj.origType ) &//比较事件类型是否一致
            ( !handler || handler.guid === handleObj.guid ) &&//如果传进了回调,判定 UUID 是否相同
             ( !namespaces || namespaces.test( handleObj.namespace ) ) &&
            //如果 types 含有命名空间,用正则看是否匹配
            //如果是事件代理必有 CSS 表达式，比较与事件描述对象中的是否相等
            ( !selector || selector === handleObj.selector || selector === "**" &&
handleObj.selector ) ) {
                eventType.splice( j--, 1 );//是就移除

                if ( handleObj.selector ) {//同时 delegateCount 减一
                    eventType.delegateCount--;
                }
                if ( special.remove ) {//处理个别事件的移除
                    special.remove.call( elem, handleObj );
                }
            }
        }
        //如果已经移除所有此类型回调，则卸载框架绑定去的 elemData.handle
        //origCount !== eventType.length 是为了防止死循环
        if ( eventType.length === 0 && origCount !== eventType.length ) {
            if ( !special.teardown || special.teardown.call( elem, namespaces, elemData.
handle ) === false ) {
                jQuery.removeEvent( elem, type, elemData.handle );
            }
            delete events[ type ];
        }
    }
   //如果 events 为空，则从 elemData 中删除 events 与 handler
```

```
        if ( jQuery.isEmptyObject( events ) ) {
            delete elemData.handle;
            jQuery.removeData( elem, "events", true );
        }
    }
```

事件卸载部分算是 jQuery 事件系统中最简单的部分，主要逻辑都花在要移除的事件描述对象的匹配条件上。

11.8 jQuery.event.dispatch 的源码解读

这是 jQuery 事件系统的核心。它就是利用这个 dispatch 方法，从缓存体中的 events 对象取得对应队列，然后修复事件对象，逐个传入用户的回调中执行，根据返回值决定是否断开循环（stopImmediatePropagation）、阻止默认行为和事件传播。

```
dispatch = function(event) {
    //创建一个伪事件对象(jQuery.Event 实例)，从真正的事件对象上抽取得相应的属性附于其上，
    //如果是 IE，亦可以将它们转换成对应的 W3C 属性，抹平两大平台的差异
    event = jQuery.event.fix(event || window.event);

    var i, j, cur, ret, selMatch, matched, matches, handleObj, sel, related,
            //取得所有事件描述对象
            handlers = ((jQuery._data(this, "events") || {})[ event.type ] || []),
            delegateCount = handlers.delegateCount,
            args = core_slice.call(arguments),
            run_all = !event.exclusive && !event.namespace,
            special = jQuery.event.special[ event.type ] || {},
            handlerQueue = [];

    //重置第一个参数为 jQuery.Event 实例
    args[0] = event;
    event.delegateTarget = this;//添加一个人为属性，用于事件代理
    //执行 preDispatch 回调,它与后面的 postDispatch 构成一种类似 AOP 的机制
    if (special.preDispatch && special.preDispatch.call(this, event) === false) {
        return;
    }

    //如果是事件代理，并且不是来自于非左键的点击事件
    if (delegateCount && !(event.button && event.type === "click")) {
        //从事件源开始，遍历其所有祖先一直到绑定事件的元素
        for (cur = event.target; cur != this; cur = cur.parentNode || this) {
            //不要触发被 disabled 的元素的点击事件
            if (cur.disabled !== true || event.type !== "click") {
                selMatch = {};//为了节能起见，每种 CSS 表达式只判定一次，通过下面的
                //jQuery( sel, this ).index( cur ) >= 0 或 jQuery.find( sel, this,null,
                //[ cur ] ).length
                matches = [];//用于收集符合条件的事件描述对象
                //使用事件代理的事件描述对象总是排在前面
                for (i = 0; i < delegateCount; i++) {
```

```
                handleObj = handlers[ i ];
                sel = handleObj.selector;

                if (selMatch[ sel ] === undefined) {
                    //有多少个元素匹配就收集多少个事件描述对象
                    selMatch[ sel ] = handleObj.needsContext ?
                            jQuery(sel, this).index(cur) >= 0 :
                            jQuery.find(sel, this, null, [cur]).length;
                }
                if (selMatch[ sel ]) {
                    matches.push(handleObj);
                }
            }
            if (matches.length) {
                handlerQueue.push({elem: cur, matches: matches});
            }
        }
    }
}
//取得其他直接绑定的事件描述对象
if (handlers.length > delegateCount) {
    handlerQueue.push({elem: this, matches: handlers.slice(delegateCount)});
}

//★★★★这个循环是从下到上执行的
for (i = 0; i < handlerQueue.length && !event.isPropagationStopped(); i++) {
    matched = handlerQueue[ i ];
    event.currentTarget = matched.elem;
    //执行此元素的所有与 event.type 同类型的回调，除非用户调用了 stopImmediatePropagation 方法，
    //它会导致 isImmediatePropagationStopped 返回 true，从而中断循环
    for (j = 0; j < matched.matches.length
            && !event.isImmediatePropagationStopped(); j++) {
        handleObj = matched.matches[ j ];
    //最后的过滤条件为事件命名空间，比如著名的 bootstrap 的命名空间为 data-api
        if (run_all || (!event.namespace && !handleObj.namespace) || event.namespace_re &&
                event.namespace_re.test(handleObj.namespace)) {
            event.data = handleObj.data;
            event.handleObj = handleObj;
            //执行用户回调(有时可能还要外包一层,来自 jQuery.event.special[type].handle)
            ret = ((jQuery.event.special[ handleObj.origType ] || {}).handle ||
            handleObj.handler).apply(matched.elem, args);
            //根据结果判定是否阻止事件传播与默认行为
            if (ret !== undefined) {
                event.result = ret;
                if (ret === false) {
                //http://heikezhi.com/yuanyi/jquery-events-stop-misusing-return-false
                    event.preventDefault();
                    event.stopPropagation();
                }
            }
        }
    }
}
//执行 postDispatch 回调
```

```
    if (special.postDispatch) {
        special.postDispatch.call(this, event);
    }
    return event.result;
}
```

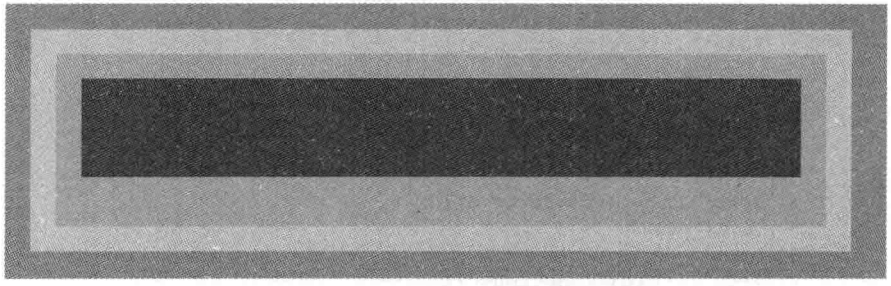
本节的难点在于如何模拟事件传播的机制，jQuery 实际只模拟冒泡那一阶段。

比如下面的例子：

```
<!DOCTYPE HTML>
<html>
    <head>
        <title></title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <script src="jquery1.82.js"> </script>
        <script>
            $(function(){
                var log = function(s){
                    window.console && console.log(s)
                }
                $("#aaa").click(function(e){
                    log("aaa")
                })
                $("#bbb").click(function(e){
                    log("bbb1")
                })
                $("#bbb").click(function(e){
                    log("bbb2")
                })
                $("#ccc").click(function(e){
                    log("ccc")
                })
                $("body").delegate("div","click", function(e){
                    log("delegate");
                })
            })
        </script>
        <style>
            html,body{
                background:#999;
                height:90%;
            }
            body{
                padding:20px;
            }
            #aaa{
                background:#cc9;
                padding:20px;
            }
            #bbb{
                background:#c9c;
                padding:20px;
                height:100px;
            }
            #ccc{
```

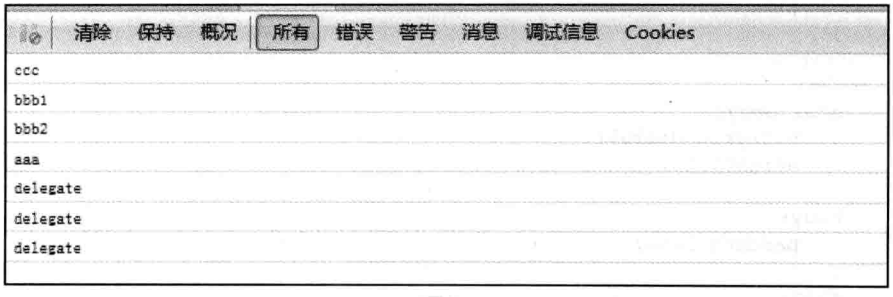
```
            background:#5e850a;
            padding:20px;
            height:40px;
        }
    </style>
</head>
<body>
    <div id="aaa">
        <div id="bbb">
            <div id="ccc">

            </div>
        </div>
    </div>
</body>
</html>
```

具体如图 11.8 所示。



▲图 11.8

当我们点击最里层的 div#ccc 元素时，控制台会依次打印出，如图 11.9 所示。



▲图 11.9

前四个是以直接绑定的方式打印出来，后三个是以事件代理的方式打印出来——恰逢 body 里面有三个 DIV，于是打印三次。打印的次序与绑定时基本无关（除了第二个与第三个），与绑定事件的元素在 DOM 树的顺序有关。事件代理中的绑定元素通常位于 DOM 树的顶部，比如 document、html、body，因此都是比较晚执行的。我们回顾一下在 jQuery.event.add 方法中，有个 delegateCount 变量，在绑定时刻意把对应的事件描述对象放在前面，因此在 dispatch 方法中就轻松多了。

```
if ( selMatch[ sel ] === undefined ) {
    selMatch[ sel ] = handleObj.needsContext ? jQuery( sel, this ).index( cur ) >= 0 :
jQuery.find( sel, this, null, [ cur ] ).length;
}
```

每向上冒泡一层，都会对当前进行 delegateCount 次匹配检测，只要有一个检测通过，就放进 matches 数组中。needsContext 标识来自于 Sizzle 引擎，用于识别关系选择器是否排在前面与 jQuery 自定义的位置伪类。jQuery.find(sel, this, null, [cur])这一句其实是 Sizzle 中的 matchesSelector()

```
Sizzle.matchesSelector = function( elem, expr ) {
    return Sizzle( expr, null, null, [ elem ] ).length > 0;
};
```

看名字就知道它是用来模拟浏览器的 matchesSelector API，因此对于 jQuery 自定义的表达式只能移动取得所有符合条件的元素，然后判定它是否其中之一（第二个分支）。

matches 保证的是回调被遗漏。而 handlerQueue 中的 elem 则是保证 currentTarget 的正确性。

比如说 body 代理 div#aaa，那么回调中的 e.currentTarget 为 div#aaa，而不是 body。这虽然有点违背 currentTarget 的原意——currentTarget 是指绑定事件的元素，node.addEventListener(type, fn, phase),currentTarget 就是那个 node。但 jQuery 基本不想让人知道这绑定是用事件代理实现，$(expr).delegate(expr1, type, fn)的指向匹配 expr1 的元素，而不是匹配 expr 的元素。

更值得让人玩味的是标有★★★★这一段循环，这是双层循环。

外层循环用于向上冒泡，更换节点。不过手动冒泡只用于事件代理，普通绑定的冒泡是依赖 DOM 自身的冒泡机制，换言之，每冒泡一次则换一个 dispatch 回调，事件对象每次也重新生成一次，也重新修正一次。这当中肯定涉及许多棘手之处，要不 jQuery 不会采取如此低效的实现。当用户调用 event.stopPropagation()，会导致 isPropagationStopped 返回 true,从而中断循环。

内层循环是用于执行回调，当用户调用 event.stopImmediatePropagation()，会导致 isImmediatePropagationStopped 返回 true,从而中断循环。由此我们可以看到 stopImmediatePropagation 其实有着 stopPropagation 的效果，在 jQuery 的源码我们会看到：

```
stopImmediatePropagation: function() {
    this.isImmediatePropagationStopped = returnTrue;
        this.stopPropagation();
},
```

因此 stopPropagation 有着与 isImmediatePropagationStopped 相同的开关，一个让整个循环中断的属性，基本不用把 isImmediatePropagationStopped、isPropagationStopped 都写上。在 opera 的 userjs 实现中，就有一个叫 propagationStopped 的属性满足这功能。

http://opera.im/kb/userjs/

jQuery 的事件系统是基于很早期的 W3C 草案，现在的原生事件对象都没有 isImmediatePropagationStopped 与 isPropagationStopped 方法，它们只是 jQuery 在循环中模拟 stopImmediatePropagation 与 stopPropagation 的一个楔子而已。退一步讲，花如此周折来完美模拟事件冒泡到底值不值得？当本书出版时，所有主流浏览器都支持这两个方法。

11.9 jQuery.event.trigger 的源码解读

从来没有一个事件系统如此耐心地重新实现一个 fireEvent 方法，这是 jQuery 追求兼容性的杰出表现。

```
trigger = function(event, data, elem, onlyHandlers) {
    // 必须要指定派发事件的对象，不能是文本节点与元素节点
    if (elem && (elem.nodeType === 3 || elem.nodeType === 8)) {
        return;
    }

    // Event object or event type
    var cache, exclusive, i, cur, old, ontype, special, handle, eventPath, bubbleType,
        type = event.type || event,
        namespaces = [];

    // focus/blur morphs to focusin/out; ensure we're not firing them right now
    if (rfocusMorph.test(type + jQuery.event.triggered)) {
        return;
    }

    if (type.indexOf("!") >= 0) {
        // Exclusive events trigger only for the exact event (no namespaces)
        type = type.slice(0, -1);
        exclusive = true;
    }
    //如果事件类型带点号就分解出命名空间
    if (type.indexOf(".") >= 0) {
        namespaces = type.split(".");
        type = namespaces.shift();
        namespaces.sort();
    }
    //customEvent 与 global 用于优化，既然从来没有绑定过这种事件，就不用继续往下进行了！
    if ((!elem || jQuery.event.customEvent[ type ]) && !jQuery.event.global[ type ]) {
        return;
    }
    //将用户传入的第一个参数都转换为 jQuery.Event 实例
    event = typeof event === "object" ?
        //如果是 jQuery.Event 实例
        event[ jQuery.expando ] ? event :
        //如果是原生事件对象
        new jQuery.Event(type, event) :
        //如果是事件类型
        new jQuery.Event(type);

    event.type = type;
    event.isTrigger = true;
    event.exclusive = exclusive;
    event.namespace = namespaces.join(".");
    event.namespace_re = event.namespace ? new RegExp("(^|\\.)" +
        namespaces.join("\\.(?:.*\\.|)") + "(\\.|$)") : null;
    ontype = type.indexOf(":") < 0 ? "on" + type : "";
```

```
// Handle a global trigger
//如果没有指明触发者，只有将整个缓存系统寻找一遍了
if (!elem) {
    // TODO: Stop taunting the data cache; remove global events and always attach to
    //document
    cache = jQuery.cache;
    for (i in cache) {
        if (cache[ i ].events && cache[ i ].events[ type ]) {
            jQuery.event.trigger(event, data, cache[ i ].handle.elem, true);
        }
    }
    return;
}
//清掉 result，方便重复使用
event.result = undefined;
if (!event.target) {
    event.target = elem;//但事件源是保证不变的
}
//data 用于放置派发事件时的额外参数，为了方便 apply 必须整成数组，并把 event 放在第一位
data = data != null ? jQuery.makeArray(data) : [];
data.unshift(event);
//如果此事件类型指定了它的 trigger 方法，就使用它的
special = jQuery.event.special[ type ] || {};
if (special.trigger && special.trigger.apply(elem, data) === false) {
    return;
}

//预先决定冒泡的路径，一直冒泡到 window
eventPath = [[elem, special.bindType || type]];
if (!onlyHandlers && !special.noBubble && !jQuery.isWindow(elem)) {

    bubbleType = special.delegateType || type;
    cur = rfocusMorph.test(bubbleType + type) ? elem : elem.parentNode;
    for (old = elem; cur; cur = cur.parentNode) {
        eventPath.push([cur, bubbleType]);
        old = cur;
    }
    if (old === (elem.ownerDocument || document)) {
        eventPath.push([old.defaultView || old.parentWindow || window, bubbleType]);
    }
}

//沿着规划好的路径把经过的元素节点的指定事件类型的回调逐一触发
for (i = 0; i < eventPath.length && !event.isPropagationStopped(); i++) {

    cur = eventPath[i][0];
    event.type = eventPath[i][1];

    handle = (jQuery._data(cur, "events") || {})[ event.type ] && jQuery._data(cur, "handle");
    //handle 其实就是调用 dispatch 函数，因此 trigger 是把整个冒泡过程都人工实现
    if (handle) {
        handle.apply(cur, data);
```

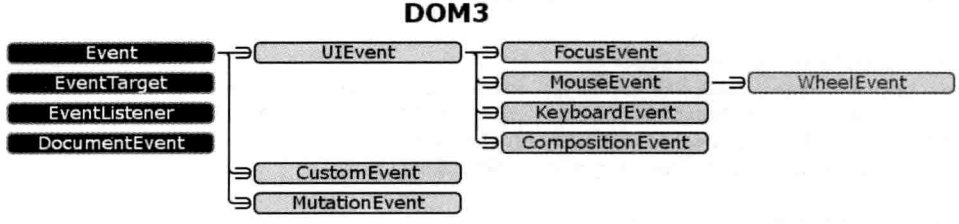
```
        }
        //处理以 onXXX 绑定的回调，无论是写在 HTML 标签内还是以无侵入方式
        handle = ontype && cur[ ontype ];
        if (handle && jQuery.acceptData(cur) && handle.apply && handle.apply(cur, data)
=== false) {
            event.preventDefault();//如果返回 true 就中断循环
        }
    }
    event.type = type;
    //如果用户没有调用 preventDefault 或 return false，就模拟默认行为
    //具体是指执行 submit、blur、focus、select、reset、scroll 等方法
    //不过其实它并没有模拟所有默认行为
    //比如点击链接时会跳转
    //又比如点击复选框单选框，元素的 checked 会改变
    if (!onlyHandlers && !event.isDefaultPrevented()) {
        //如果用户指定了默认行为，则只执行它的默认行为，并跳过链接的点击事件
        if ((!special._default || special._default.apply(elem.ownerDocument, data) ===
false) &&
                !(type === "click" && jQuery.nodeName(elem, "a")) && jQuery.acceptData(elem)) {
            //如果元素同时存在 el["on"+type]回调与 el[type]方法，则表示它有默认行为
            //对于 el[type]属性的检测，jQuery 不使用 isFunction 方法，因为它的 typeof 在 IE6～IE8
            //返回 object
            //jQuery 也不打算触发隐藏元素的 focus 或 blur 默认行为，IE6～IE8 下会抛出
            //“由于该控件目前不可见、未启用或其类型不允许，因此无法将焦点移向它”错误
            //jQuery 也不打算触发 window 的默认行为，防止触发了 window.scroll 方法
     //scroll()方法在 IE 与标准浏览器存在差异，IE 会默认 scroll()为 scroll(0,0)
            if (ontype && elem[ type ] && ((type !== "focus" && type !== "blur") ||

                    event.target.offsetWidth !== 0) && !jQuery.isWindow(elem)) {

                // onXXX 回调已经在$.event.dispatch 方法执行过了，不用再次触发
                old = elem[ ontype ];

                if (old) {
                    elem[ ontype ] = null;
                }
                //标识正在触发此事件类型，防止下面 elem[type]()重复执行 dispatch
                jQuery.event.triggered = type;
                elem[ type ]();//执行默认行为
                jQuery.event.triggered = undefined;//还原
                if (old) {//还原
                    elem[ ontype ] = old;
                }
            }
        }
    }
    //与 dispatch 一样，返回 event.result
    return event.result;
}
```

笼统来说，trigger 就是 dispatch 的加强版。dispatch 只触发当前元素与其底下元素(通过事件代

理的方式)的回调，trigger 则是模拟整个冒泡过程，除了它自身，还触发其祖先节点与 window 的同类型的回调。不过，从 trigger 的代码来看，它比 dispatch 多做的事就是触发事件的默认行为。这涉及太多判定，如果再把 dispatch 的代码写在一块就不好维护了。

但是我觉得，trigger 其实用不着写得这么复杂。trigger 要做的事就是在某一元素触发一个回调（dispatch），生产一个事件对象，然后让它顺势冒泡，触发其他回调（dispatch）就行了。浏览器提供了原生派发机制。IE 下叫 fireEvent，标准浏览器为 dispatchEvent。IE 下的问题是，有许多事件不能冒泡或冒泡不到顶层，如果能把 IE 的代码独立出来，标准浏览器用一套，IE 用一套（不过也是旧版本 IE 罢了，IE9 已经可能在标准模式中支持 dispatchEvent），这样性能会大大提高，以后抛开包袱什么的也很容易。zepto.js 就是这样干的。

在标准浏览器，要创建一个事件对象，需要知道它属于什么类型。我们需要重新开始认识一下事件的继承体系，如图 11.10 所示。



▲图 11.10

对于 document.createEvent 的传参，我们只需要在上图给出的事件名加个“s”就行了。不过，正如其他 API 一样，W3C 的规范总是滞后于浏览器的实现，注意以下几点。

万能事件对象的创建。由于是使用事件对象的基类 Event 创建，因此理应能触发所有事件的回调。但早期，浏览器不支持 document.createEvent(“Events”)创建万能事件对象，而要使用 HTMLEvents 参数，即 document.createEvent(“HTMLEvents”)。

万能事件对象也不能触发鼠标事件的回调，要用 MouseEvents 参数创建鼠标对象事件。

键盘事件在 IE9、FF 与 Opera 的传参不叫 KeyboardEvents，而是叫 KeyboardEvent，webkit 系则两者都支持。

FF 不支持 mousewheel，因此我们需要用 document.createEvent(“MouseScrollEvents”)创建 DOMMouseScroll 事件对象。

如果在控制台执行以下函数，我们会找到更多事件构造器。

```
Object.getOwnPropertyNames(window).filter(function(p){
    return typeof window[p] == "function" && (window[p].prototype instanceof Event);
})
```

得到这样一个数组：

```
["OverflowEvent", "CustomEvent", "CompositionEvent", "AudioProcessingEvent", "Before
LoadEvent", "TextEvent", "MessageEvent", "SVGZoomEvent", "UIEvent", "PopStateEvent",
"MouseEvent", "SpeechInputEvent", "ErrorEvent", "DeviceOrientationEvent", "ProgressEvent",
```

```
"CloseEvent", "MutationEvent", "PageTransitionEvent", "HashChangeEvent", "WheelEvent",
"StorageEvent",  "XMLHttpRequestProgressEvent",  "WebGLContextEvent",  "TouchEvent",
"WebKitTransitionEvent", "MediaStreamEvent", "OfflineAudioCompletionEvent", "Keyboard
Event", "WebKitAnimationEvent"]
```

但常用的交互都集中在 HTML4.0 已经定义好的事件上，我们无需理会什么 message、storage、popstate 事件，更别提什么变动事件。我认为支持以下事件就足够了，如表 11.2 所示。

表 11.2

HTMLEvents	Load、unload、abort、error、select、change、submit、reset、focus、blur、resize、scroll 及其他
KeyboardEvent	Keypress、keyup、keydown
MouseEvents	contextmenu、click、dblclick、mouseout、mouseover、mouseenter、mouseleave、mousemove、mousedown、mouseup、mousewheel

我们根据上面的表格弄一个叫 eventMap 的 hash 出来，那么 trigger 方法最大限制可以压缩成如下样子。

```
trigger: function( type, target){
    var doc = target.ownerDocument || target.document || target || document;
    event = doc.createEvent(eventMap[type] || "CustomEvent");
    if(/^(focus|blur|select|submit|reset)$/.test(type)){
        target[type] && target[type]();//触发默认行为
    }
    Event.initEvent( type, true,true);
    target.dispatchEvent(event);
}
```

由于这样 trigger 出来的对象是只读，不能覆盖原生属性或方法，因此你可以为它自定义一个 more 属性，里面装载着你要改写的东西，然后在 dispatch 将它包装成一个 jQuery 伪事件对象后，再把循环加在伪事件对象就行了。

11.10 jQuery 对事件对象的修复

jQuery 在这里分两步走，首先创建一个伪事件类 jQuery.Event，统一处理 preventDefault、stopPropagation、stopImmediatePropagation 方法，然后在 jQuery.event.fix 针对不同的事件类型修复特定的属性。

```
jQuery.Event = function(src, props) {
    //无 "new" 实例化
    if (!(this instanceof jQuery.Event)) {
        return new jQuery.Event(src, props);
    }
    if (src && src.type) {
        this.originalEvent = src;
        this.type = src.type;
        this.isDefaultPrevented = (src.defaultPrevented || src.returnValue === false ||
                src.getPreventDefault && src.getPreventDefault()) ? returnTrue : returnFalse;
```

```
        // Event type
    } else {
        this.type = src;
    }

    // 如果是一个对象，复制它的属性
    if (props) {
        jQuery.extend(this, props);
    }

    this.timeStamp = src && src.timeStamp || jQuery.now();//修复 FF 的问题

    // 标识已经修正过
    this[ jQuery.expando ] = true;
};

function returnFalse() {
    return false;
}
function returnTrue() {
    return true;
}

jQuery.Event.prototype = {
    preventDefault: function() {
        this.isDefaultPrevented = returnTrue;
        var e = this.originalEvent;
        if (!e) {
            return;
        }
        if (e.preventDefault) {
            e.preventDefault();
        } else {
            e.returnValue = false;
        }
    },
    stopPropagation: function() {
        this.isPropagationStopped = returnTrue;
        var e = this.originalEvent;
        if (!e) {
            return;
        }
        if (e.stopPropagation) {
            e.stopPropagation();
        }
        e.cancelBubble = true;
    },
    stopImmediatePropagation: function() {
        this.isImmediatePropagationStopped = returnTrue;
        this.stopPropagation();
    },
    isDefaultPrevented: returnFalse,
    isPropagationStopped: returnFalse,
    isImmediatePropagationStopped: returnFalse
};
```

jQuery.event.fix 也很简单，它把大部分逻辑分担给其他钩子对象，如 mouseHooks、keyHooks，因此插件作者可以自制 wheelHooks 用来屏蔽 mousewheel 的差异性，touchHooks 来模拟触摸事件。fix 方法只用于包装事件对象为伪事件对象，修正 target,metaKey 属性。

```
fix = function(event) {
    if (event[ jQuery.expando ]) {
        return event;
    }
    var i, prop,
            originalEvent = event,
            fixHook = jQuery.event.fixHooks[ event.type ] || {},
            copy = fixHook.props ? this.props.concat(fixHook.props) : this.props;

    event = jQuery.Event(originalEvent);
    //不同的事件类型,它的属性也不一样
    for (i = copy.length; i; ) {
        prop = copy[ --i ];
        event[ prop ] = originalEvent[ prop ];
    }

    if (!event.target) {//FireFox IE6~IE8 Safari2
        event.target = originalEvent.srcElement || document;
    }
    if (event.target.nodeType === 3) {// FireFox Safari
        event.target = event.target.parentNode;
    }
    // FireFox IE6~IE8
    event.metaKey = !!event.metaKey;
    return fixHook.filter ? fixHook.filter(event, originalEvent) : event;
}
```

jQuery 只修正两个重要类型，键盘事件与鼠标事件。

键盘事件的重点在于修正 keyCode，不过 W3C 已经规定 which 才是标准属性，因此做一下映射就轻松搞定。

```
if(event.which == null ) {
     event.which = event.charCode != null ? event.charCode : event.keyCode;
 }
```

鼠标事件的重点是修复 pageX/Y、relatedTarget 与用于左中右键的 which 属性。

这里得介绍一下事件对象的几对与位置有关的属性。

clientX clientY 为事件触发时鼠标在当前浏览器可视区的坐标。

screenX screenY 为事件触发时鼠标在屏幕的坐标。

pageX pageY 为事件触发时鼠标在整个 document 的坐标。

offsetX offsetY 为事件触发时鼠标在事件源元素的坐标。不过元素的区域也有许多计算方式，比如 border box、padding box、context box。按规范来说，参考点为 padding box（即由 padding 围成的区域，不包括滚动条与边框）的左上角。如果存在 border，可能出现负值。

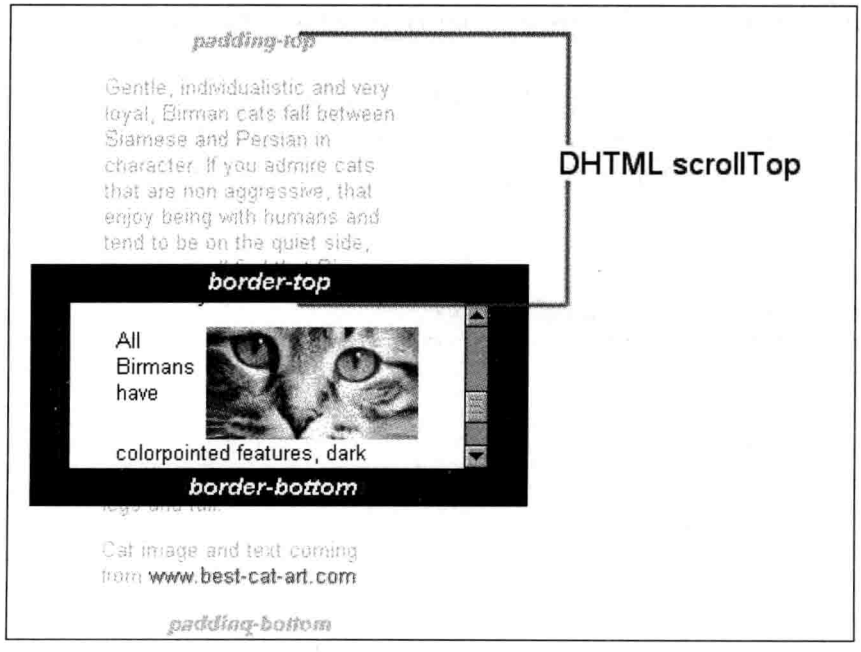
layerX layerY 为事件发生时鼠标相当于事件源元素的 offsetParent 的坐标。

x,y 可以说是 IE 的 layerX layerY，为事件发生时鼠标相当于事件源元素的 offsetParent 的坐标。但是 IE6～IE8 的 offsetParent 存在 BUG，比如 td 的元素的 offsetParent 总为 table，最好不要用这两个东西。支持一览表如表 11.3 所示。

表 11.3

	clientX/Y	offsetX/Y	layerX/Y	pageX/Y	screenX/Y	x/y
IE6～IE8	√	√	×	×	√	√
Firefox15	√	×	√	√	√	×
Chrome23	√	√	√	√	√	√
Opera12	√	√	×	√	√	√
Safari5	√	√	√	√	√	√
IE9	√	√	√	√	√	√
IE10	√	√	√	√	√	√

好了，我们尝试修复旧版本 IE 下的 pageX/Y 问题。pageX/Y 是相当于整个文档的坐标，而我们在 IE 只能拿相对于可视区的坐标 clientX/Y，因此我们需要补上可视区之外的高度或宽度，换言之，加上滚动过的位移就行了。这个我们可以通过 scrollLeft、scrollTop 拿到，如图 11.11 所示。



▲图 11.11

但 scrollLeft、scrollTop 也存在兼容性问题，它本来属性 IE 的 dthml 对象模型的东西，但被标准浏览器抄去了，直到 2012 才列入 W3C 标准草案。对于的一般元素，取这两个属性值，是没有差异的，问题是 HTML 与 BODY，因为它们的滚动条同时也代表着窗口的滚动条。在 IE 的兼容模式下，窗口滚动条是出现在 body 中，标准模式下，窗口滚动条出现在 html 中；webikt 下，窗口滚动条只出现在 body 中；Firefox 与 Opera 的滚动条只出现在 html 中。当窗口滚动条出现在 body 中，document.body.scrollTop（或 scrollLeft）为我们想要的高宽，

document.documentElement.scrollTop 为 0，反之亦然。幸运的是，我们不用理会标准浏览器，只管好 IE 就行了。

```
var scrollTop = document.documentElement.scrollTop || document.body.scrollTop;
var scrollLeft = document.documentElement.scrollLeft || document.body.scrollLeft;
```

然后

```
e.pageX = e.clientX + scrollLeft
e.pageY = e.clientY + scrollTop
```

在 IE5～IE7 标准模式及 IE 全系列的怪异模式下，页面左上角存在 2px 的偏移值，我们可以在 MSDM 可以找到描述：

http://msdn.microsoft.com/en-us/library/ms536433

This method retrieves an object that exposes the left, top, right, and bottom coordinates of the union of rectangles relative to the client's upper-left corner.

In Microsoft Internet Explorer 5, the window's upper-left is at ***2,2*** *(pixels) with respect to the true client.*

这会导致我们在 IE 下取得的 pageX/Y 可能比其他浏览器大 2px。

归根结底，这 2px 是怎么出来的，归属于谁管的呢？网上虽然有许多人在研究，给出的见解却鲜有一致，也很粗糙，我们还是自己做一个页面测试一下。

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>2px 偏移值 by 司徒正美</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <style>
            html{
                background:lime;
                /* border:0px;*/
            }
            body{
                background:orange;
            }
            #log {
                padding:20px;
                background: red;
            }
        </style>
        <script>
```

```
            var log = function(s) {
                document.getElementById("log").innerHTML = s;
            }
            document.onclick = function(e) {
                e = e || event;
                log(document.documentElement.clientTop + "  "
+ document.body.clientTop);
            };
        </script>
    </head>
    <body>

        <pre id="log">

        </pre>
    </body>
</html>
```

首先我们要明白，在 IE 全系列触发标准模式下很简单，使用<!DOCTYPE HTML>这个最简洁的文档类型，它的前面不能再有任何字符就行了。经测试，IE8～IE10 下为 0 0，IE6、IE7 下为 2 0。当在标准模式，页面背景为绿色。

然后我们切换成怪异模式，做法有如下几种。

- IE6 下，在最前面加上 XML 声明，如< ?xml version="1.0" encoding="utf-8"?>。
- 在最前面加上非法的标签，如<ddd></ddd>。
- 在最前面加上一段文本，如 ddddddddddd。

为了通用与布局好看，我们先用第二个方案吧。进入怪异模式下，页面背景变成橙色。IE 全系列的怪异模式，打印出 0 2。如果结合我们之前对 scrollTop、scrollLeft 做的测试，可以初步得出一结论:

IE 在怪异模式，页面显示用的顶级容器是 body！什么背景颜色、滚动条、装饰用的 2px 偏移量都在 body 中实现。

我们在怪异模式可以设置

```
body{
    background:orange;
    Border: 0;
}
```

让 clientTop 归零。

当然我们也惊奇地发现，如果这个 border:0 的样式规则移到 html 中，也能让 clientTop 归零。那估计是它也把样式进行修正，如果 body 没有设置相关样式，就用 html 的。总之，在怪异模式下，我们可以通过修改样式来消除这 2px 偏移量。

在标准模式下，IE6 也可以通过修改样式消除这 2px 偏移量，但 IE7 不能！IE8+在标准模式下不存在什么编移量就不说了，如表 11.4 所示。

表 11.4

	背景颜色	html.clientTop	body.clientTop	能否通过修改样式消除 2px
IE6	绿	2	0	能
IE6Q	橙	0	2	能
IE7	绿	2	0	不能
IE7Q	橙	0	2	能
IE8	绿	0	0	不需要
IE8Q	橙	0	2	能

因此完美方案如下。

```
if ( event.pageX == null && event.clientX != null ) {  // 处理鼠标事件
    var doc = event.target.ownerDocument || document;
    var box = document.compatMode == "BackCompat" ?
 doc.body : doc.documentElement
    event.pageX = event.clientX + (box && box.scrollLeft  || 0) - (box && box.clientLeft
|| 0);
    event.pageY = event.clientY + (box && box.scrollTop  || 0) - (box && box.clientTop  ||
0);
}
```

如果查看 jQuery 的源代码，发现它比上面的长一点，不过效果是一样的，当 HTML 不是渲染用顶层元素时，它的相关值都为 0，短路运算符就会取 body 的相关值。最后一个是，当文档不是 HTML 文档，是 XML 或 SVG 文档时，就没有 scrollTop、clientTop 了，这时它们的值就为 0。

```
//jQuery1.82
if ( event.pageX == null && original.clientX != null ) {
    eventDoc = event.target.ownerDocument || document;
    doc = eventDoc.documentElement;
    body = eventDoc.body;

    event.pageX = original.clientX + ( doc && doc.scrollLeft || body && body.scrollLeft
|| 0 )
        - ( doc && doc.clientLeft || body && body.clientLeft || 0 );
    event.pageY = original.clientY + ( doc && doc.scrollTop  || body && body.scrollTop  || 0 )
        - ( doc && doc.clientTop  || body && body.clientTop  || 0 );

}
```

11.11 滚轮事件的修复

jQuery 核心库没有对 mousewheel 事件的差异性进行处理，但作为一个常用的事件，本文稍带说上几句。其实解决它的兼容性问题还是简单的。

在浏览器中，现有五种滚轮事件：mousewheel、DOMMouseScroll、MozMousePixelScroll、wheel、mousemultiwheel。Mousewheel 现在除 Firefox 外都支持，接着两种是 Firefox 的私有实现，最后两种是 DOM3 事件，能同时支持了 x、y、z 3 个轴向的滚轮值。

mousewheel 用于取得滚动距离的属性名为 event.wheelDelta，往上滚一圈为 120，往下滚一圈为–120。但 Opera9x 系列却实现错误，与 IE 滚动方向一致，不过版本 10 后又修复。Safari 早期版本，wheelDelta 会出现浮点数的情况，我们需要自行取整。在 Opera 中，它的事件对象同时存在 wheelDelta 与 detail 属性。在 IE6～IE8 中，window 无法绑定 mousewheel 事件，Opera、Safari、Chrome 可以。

DOMMouseScroll 是最常用的 mousewheel 代替品，用于取得滚动距离的属性名为 event.detail，上滚一圈为–3，往下滚一圈为 3；当我们按住 ctrl、alt、shift 键之后，再滑动鼠标滚轮，detail 的值就会成为–1 和–1；并且在按住 ctrl 键时，滑动滚轮还有缩放页面的功效。我们可以通过 event.axis == event.HORIZONTAL_AXIS 或 event.axis == event.VERTICAL_AXIS 得知是水平滚动还是垂直滚动。

MozMousePixelScroll，Firefox 的私有实现，与 DOMMouseScroll 不同的是，event.detail 代表的是滚动过的像素值。不过这个值会视用户的分辨率与缩放比有何不同，通常在 50～60，向上滚是负数，向下滚是正数。当我们按住 Ctrl、Alt、Shift 键的其中一个后，再滑动鼠标滚轮，detail 的值为不按下这 3 个键时的 detail 值的三分一大小。同时，在按住 Ctrl 键时，滑动滚轮还能缩放页面，再用正常的方式滑动滚轮，会发现 detail 比原来大或小几像素。

wheel 现在只有 IE9、IE10 支持，Opera12,FF16,Chrome23 都不支持。它没有 wheelDelta 与 detail 属性，但有 deltaX、deltaY、deltaZ 与 deltaMode 属性。当我们向下滚动时，deltailY 为 53，反之为–53。有关它的使用基本不明朗，自己可以到这里查看：

http://msdn.microsoft.com/zh-cn/subscriptions/downloads/ff975254(v=vs.85).aspx

mousemultiwheel 与 wheel 事件一样，相关资料非常少，也没有什么使用示例，至今仍是 W3C 的玩物。

下面给出兼容方案。

//addEvent 略，见开章部分

```
var wheel = function(obj,callback){
    var wheelType = "mousewheel"
    try{
        document.createEvent("MouseScrollEvents")
        wheelType = "DOMMouseScroll"
    }catch(e){}
    addEvent(obj, wheelType,function(event){
        if ("wheelDelta" in event){//统一为±120，其中正数表示向上滚动，负数表示向下滚动
            var delta = event.wheelDelta
            //Opera 9x 系列的滚动方向与 IE 保持一致，10 后修正
            if( window.opera && opera.version() < 10 )
                delta = -delta;
            //由于事件对象的原有属性是只读，我们只能通过添加一个私有属性 delta 来解决兼容问题
            event.delta = Math.round(delta) /120; //修正 Safari 的浮点 bug
        }else if( "detail" in event ){
            event.wheelDelta = -event.detail * 40//为 FF 添加更大众化的 wheelDelta
            event.delta = event.wheelDelta /120  //添加私有的 delta
        }
        callback.call(obj,event);//修正 IE 的 this 指向
    });
}
```

一个完整的测试页面如下。

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>滚轮事件 mousewheel by 司徒正美</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <style>
            body{
                padding:10px 100px;
            }
            .slider{
                width:48px;
                height:200px;
                padding: 5px 0px;
                background:#eee;
                cursor:n-resize;
            }
            .slider-slot {
                width:16px;
                margin:10px 15px;
                height:180px;
                background:#eee;
                border:1px solid gray;
                border-color:#999 white white #999;
                position:relative;
            }
            .slider-trigger {
                width:14px;
                height:18px;
                font:1px/0 arial;
                border:1px solid gray;
                border-color:white #999 #999 white;
                background:#ccc;
                position:absolute;
            }

        </style>
        <script type="text/javascript">// <![CDATA[
            window.onload = function(){
                function log(s){
                    window.console && console.log(s);
                }
                var get = function(i) {
                    return document.getElementById( i );
                }
                function addEvent(el, type, callback, useCapture ){
                    if(el.dispatchEvent){//W3C 方式优先
                        el.addEventListener( type, callback, !!useCapture );
                    }else {
                        el.attachEvent( "on"+type, callback );
                    }
                    return callback;//返回 callback 方便卸载时用
                }
                var wheel = function(obj,callback){
```

```
                var wheelType = "mousewheel"
                try{
                    document.createEvent("MouseScrollEvents")
                    wheelType = "DOMMouseScroll"
                }catch(e){}
                addEvent(obj, wheelType,function(event){
                    if ("wheelDelta" in event){//统一为±120，其中正数表示向上滚动，
                                               //负数表示向下滚动
                        var delta = event.wheelDelta
                        //Opera 9x 系列的滚动方向与 IE 保持一致，10 后修正
                        if( window.opera && opera.version() < 10 )
                            delta = -delta;
//由于事件对象的原有属性是只读，我们只能通过添加一个私有属性 delta 来解决兼容问题
            event.delta = Math.round(delta) /120; //修正 Safari 的浮点 BUG
                    }else if( "detail" in event ){ event.wheelDelta = -event.detail * 40
//为 FF 添加更大众化的 wheelDelta
                     event.delta = event.wheelDelta /120  //添加私有的 delta
                    }
                    callback.call(obj,event);//修正 IE 的 this 指向
                });
            }
            function  preventDefault(e){
                if( e.preventDefault )
                    e.preventDefault();
                e.returnValue = false;
            }

            wheel(get("number"), function(e){
                this.value = (Number(this.value) || 0) + e.delta  //120  ;
                this.select();
                preventDefault(e)
            })

            wheel(get("img"), function(e){
                this.style.width =  this.offsetWidth + e.delta + 'px';
                this.style.height =  this.offsetHeight + e.delta + 'px';
                preventDefault(e)

            })

            function range( num, max,min ) {
                return Math.min( max, Math.max( num,min ) );
            }
            var  tar = get('sliderTrigger');
            wheel(get("slider"), function(e){
                preventDefault(e)
                tar.style.top = range( tar.offsetTop + ( -1 * e.delta * 10 ),160,0 ) + 'px';
            })
        }
    </script>
</head>
<body>
    <h2>文本框增加/减少值</h2>
    <div><input type="text" value="300" id="number">
```

```
            <span>文本框获得焦点后滚动鼠标滚轮</span>
        </div>
        <h2>鼠标滚动缩放图片</h2>
        <div>
            <img id="img" style="cursor:-moz-zoom-in" title="鼠标滚动 缩放大小" src="http:
//www.baidu.com/img/baidu.gif" />
        </div>
        <h2>鼠标滚动控制滑块移动</h2>
        <div id="slider" class="slider">
            <div class="slider-slot">
                <div id="sliderTrigger" class="slider-trigger">
                </div>
            </div>
        </div>

    </body>
</html>
```

11.12 mouseenter 与 mouseleave 事件的修复

在前端开发，我们经常遇到隐藏弹出层或下拉菜单的需要。这个层本来是隐藏的，当我们以某种条件叫它显示出来后，我们或者通过点击某个按钮关闭它，这是最简单的，或者点击这个层以外的其他位置让它自动隐藏，这个就有点麻烦。这正是本节要做的事。

从上面的描述看，最直接的做法是使用事件代理，把点击事件绑到 document 上，然后比较事件源元素与弹出层的包含关系。如果不存在包含，就隐藏它。我们可以通过第 6 章给出的 contains 方法搞定它，但即使动用的是原生 API，在这种情形下也非常不妥。因为页面上的交互最频繁的操作就是点击，不小心点一下就要检测一次……此方案驳回！

我们必须要把这判定的调用减到最少。注意“这个层以外的其他位置”这几个字眼没有？我们可以从元素与元素的包含关系，转移到鼠标与某个边界的比较。当我们的鼠标在这个层的上方时，我们设置个标记为 false，这时点击事件发生，不执行隐藏层的逻辑，当鼠标离开这个层的上方时，我们把这标记改为 true，接着你应该明白是什么回事了！

下面挑选用来编程的事件。大部分人会用 mouseover、mouseout，因为他们也想不出其他的了。它们的确是不错的选择，这只限于弹出层非常简单的情况，比如它只由一两个 DIV 组成。假设真这样做，就掉落入陷阱。作为弹出层的最外围容器，它上面会放置了无数子孙元素。而我们的 mouseout 只监听最外的大容器，加之我们又很容易就“移出”这个大容器——鼠标移到大容器的子元素上，对于 mouseout 来说也算“移出”，导致判定失败。

这时就用到 mouseenter 与 mouseleave。museenter 是鼠标进入这元素的内部就触发了，并且以后在这内部的子元素上移来移去都不会触发 mouseleave，而 mouseleave 只有鼠标跑到这元素的外面才触发。造成这新效果的原因是它不会冒泡。

下面是示例。

```
<!DOCTYPE html>
<html>
    <head>
        <title>点击弹出层以外的地方隐藏层 by 司徒正美</title>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<style>
    .menu_wrap{
        display: none;
        padding:5px;
        width:120px;
        border:1px solid #006dcc;
    }
    .menu{
        list-style: none;
        padding:0;
        margin:0;
    }
    .menu li{
        width:120px;
        text-indent: 1em;
        height: 25px;
        line-height: 25px;
    }
    .menu li:hover{
        background: #06a3d6;
        color:#fff;
    }
</style>
<script>
    window.onload = function(){
        function log(s){
            window.console && console.log(s);
        }
        var get = function(i) {
            return document.getElementById( i );
        }
        function addEvent(el, type, callback, useCapture ){
            if(el.dispatchEvent){//W3C 方式优先
                el.addEventListener( type, callback, !!useCapture );
            }else {
                el.attachEvent( "on"+type, callback );
            }
            return callback;//返回 callback 方便卸载时用
        }
        addEvent(get("button"),"click",function(){
            get("menu_wrap").style.display = "block";
        })
        var flag
        addEvent(get("menu_wrap"),"mouseenter",function(){
            flag = false;
        })
        addEvent(get("menu_wrap"),"mouseleave",function(){
            flag = true;
        })
        addEvent(document,"click",function(){
            if(flag){
                get("menu_wrap").style.display = "none";
            }
        })
    }
</script>
```

```
    </head>
    <body>
        <div>
            <strong id="button">显示下拉菜单 </strong>
            <div class="menu_wrap" id="menu_wrap">
                <ul class="menu">
                    <li>item1</li>
                    <li>item2</li>
                    <li>item3</li>
                </ul>
            </div>
        </div>
    </body>
</html>
```

上面，在 webkit 浏览器下无法隐藏下拉菜单，因为它们不支持这两事件，Firefox 也是到 10 才支持。如何在不支持 mouseenter、mouseleave 的浏览器中模拟它们，这是本节的第二个任务。

```
//如果浏览器不支持我们才进入修复分支
if (!+"\v1" || eventSupport("mouseenter")) {
//IE6～IE8 不能实现捕获，
//2013 年 10 月之前的 safari 和 chrome 版本也不支持 mouseenter，chrome30 修复
    jQuery.each({
        mouseenter: "mouseover",
        mouseleave: "mouseout"
    }, function(orig, fix) {
        jQuery.event.special[ orig ] = {
            delegateType: fix, //用 mouseover, mouseout 伪装 mouseenter 和 mouseleave
            bindType: fix,
            handle: function(event) {//这时 event 已经打补丁了
                var ret, target = this, handleObj = event.handleObj
                //mouseover 时相当于 IE 的 formElement，mouseout 时相当于 IE 的 toElement
                related = event.relatedTarget
                //判定要进入的节点与绑定的节点不存在包含关系并且不相等才调用函数
                if (!related || (related !== target && !jQuery.contains(target, related))) {
                    event.type = handleObj.origType;
                    ret = handleObj.handler.apply(this, arguments);
                    event.type = fix;
                }
                return ret;
            }
        };
    });
}
```

mouseenter 的支持情况如表 11.5 所示。

表 11.5

IE	Firefox8+	Chrome	Safari	Opera11+
√	√	×	×	√

注：Firefox8+是说它从 8 这个版本开始支持，Opera11 也是同样意思。

11.13 focusin 与 focusout 事件的修复

这两个事件也是 IE 的私有实现，能冒泡的获得或失去焦点的事件。现在只有 Firefox 不支持 focusin、focusout 事件，并且它也不支持 DOMFocusIn、DOMFocusOut。由于不支持冒泡，就需要我们手动模拟冒泡过程，前面的章节也提到过，在 jQuery 的事件系统中，只有 trigger 方法是干这活的。不过 jQuery 又提供了一个 simulate 方法，用于包装 trigger 与 dispatch，用第四个参数的真伪来区分它们。simulate 相对于 trigger 多干的一件事是，它会修正事件类型。

jQuery 的实现如下。

```
if ( !jQuery.support.focusinBubbles ) {
    jQuery.each({ focus: "focusin", blur: "focusout" }, function( orig, fix ) {
        //只绑定一次 focusin 与 focusout，通过 document 进行监听，
        //然后捕获事件源，进行人工冒泡
        var attaches = 0,
        handler = function( event ) {
            jQuery.event.simulate( fix, event.target, jQuery.event.fix( event ), true );
        };
        jQuery.event.special[ fix ] = {
            setup: function() {
                if ( attaches++ === 0 ) {
                    document.addEventListener( orig, handler, true );
                }
            },
            teardown: function() {
                if ( --attaches === 0 ) {
                    document.removeEventListener( orig, handler, true );
                }
            }
        };
    });
}
```

示例:

```
<!DOCTYPE html>
<html>
    <head>
        <title>focusin/focusout by 司徒正美</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <script type="text/javascript">
            window.onload = function(){
                var get = function(i) {
                    return document.getElementById( i );
                }
                function addEvent(el, type, callback, useCapture ){
                    if(el.dispatchEvent){//W3C 方式优先
                        el.addEventListener( type, callback, !!useCapture );
                    }else {
                        el.attachEvent( "on"+type, callback );
```

```
                }
                return callback;//返回 callback 方便卸载时用
            }
            var form = get("form")
            function focusin (event) {
                var target = event.target ? event.target : event.srcElement;
                if (target) {
                    target.style.color = "blue";
                }
            }
            function focusout (event) {
                var target = event.target ? event.target : event.srcElement;
                if (target) {
                    target.style.color = "";
                }
            }
            if ("onfocusin" in form) {
                addEvent(form,"focusin",focusin, false)
                addEvent(form,"focusout",focusout, false)
            }else{
                addEvent(form,"focus",focusin, true)
                addEvent(form,"blur",focusout, true)
            }
        }

    </script>
</head>
<body>
    <form id="form">
        姓名: <input placeholder="请填写用户名"/><br />
        邮箱: <input placeholder="请填写邮箱"/>
    </form>
</body>
</html>
```

我们可以把这两个事件当成实现 focus、blur 事件的手段，通过监听 form 即可监听各个输入域，当失去焦点以及得到焦点时进行处理，如表 11.6 所示。

表 11.6

	IE6～IE9 keydown	IE6～IE9 keyup	IE6～IE9 keypress	Firefox16 keydown	Firefox16 keyup	Firefox16 keypress
区分大小写	×	×	√	×	×	√
监听 A 类功能键	√	√	√	√	√	√
监听 B 类功能键	√	√	×	√	√	×
获取 charCode	×	×	√	×	×	×
监听 tab	√	×	×	√	√	√

A 类功能键是指 enter、del、insert，方向。

B 类功能键是指上下翻页、shift、win、alt、ctrl、caps、退格。

Chrome23、Safari5 的情况同 IE。Opera12 的情况与 FF 相近，但不能获取 charCode 值时返回 undefined，并且只能通过 keydown，keyup 监听 tab 键。

11.14 旧版本 IE 下 submit 的事件代理的实现

在旧版本 IE 下，submit 不会冒泡到顶层，它只执行 form 元素的 submit 回调，并立即执行提交跳转，因此只能用事件冒充的方式来实现。

那么，我们看看浏览器在什么情况才触发 submit 事件吧。submit 事件与鼠标事件、键盘事件是不一样的。它是一种复合事件，既可以通过鼠标事件实现，也可以通过键盘事件实现，重要的是结果，能实现表单提交即可。

当焦点聚集于 input[type=text]、input[type=password]、input[type=checkbox]、input[type=radio]、input[type=checkbox]、input[type=button]、input[type=image]、input[type=submit]，按回车键会触发提交。

当鼠标在 input[type=image]、input[type=submit]上方，通过点击事件会触发提交。

浏览器差异性：IE 还可以在 input[type=file]，回车会触发提交。

我们也可以使用 form.submit()这样的编程手段触发提交。

如果我们是通过鼠标或键盘事件提交，那么 submit 回调的执行情况如下。

- IE9+与其他标准浏览器 会触发 form 元素及其祖先元素（一直到 window）的 submit 事件才跳转。
- IE8 及 IE 以下版本，只触发 form 元素的的 submit 事件。
- 它们都不会触发 form 元素之内的元素绑定的 submit 事件。

由于问题只存在于旧版本 IE，我们可利用前面章节提供的 eventSupport 方法进行检测。不过 jQuery 在此处的实现非常绕，我就贴一下 mass Framework 的代码吧。

```
//https://github.com/RubyLouvre/mass-Framework/blob/0.6/event_fix.js
"submit,reset".replace($.rword, function(type) {
    adapter[ type ] = {
        setup: delegate(function(node) {
            $(node).bind("click._" + type + " keypress._" + type, function(event) {
                var el = event.target;
                if (el.form && (adapter[ type ].keyCode[ event.which ] || adapter[ type ].
input[  el.type ])) {
                    facade._dispatch([node], event, type);
                }
            });
        }),
        keyCode: $.oneObject(type == "submit" ? "13,108" : "27"),
        input: $.oneObject(type == "submit" ? "submit,image" : "reset"),
        teardown: delegate(function(node) {
            $(node).unbind("._" + type);
        })
    };
});
```

这里是同时处理 IE 的 submit、reset 的事件代理，delegate 用于检测它有没有用事件代理，不用管它。重点是它在代理元素上一下子绑定了两个事件，click、keypress。如果是键盘事件，根据 keyCode 是否为 13108（回车键）；如果是点击事件，根据 input 元素的 type 属性值是否为 submit、image。是，就手动冒泡，这里和 jQuery.event.trigger 逻辑差不多，逐层取得元素的缓存数据，再得到相应的回调队列，执行它们，没有阻止冒泡则继续上一个层，一直到最顶的 window。如果最后也没有阻止默认行为，就通过 el.submit() 触发提交行为。

el.submit()方法有个特异之外，它是不会执行 submit 回调的，像其他的 click、blur、focus、select 这样的 DOM 方法都会同时执行回调与默认行为。

reset 事件代理与 submit 的差不多，不同的就是 keyCode 与 type 属性值而已。

11.15 oninput 事件的兼容性处理

在做搜索框的智能提示，微博发布区@好友出现列表等功能时，我们不免要监听输入内部的变化。如果使用 change 事件，只能等在失去焦点时才触发回调，如果使用 onkeydown、onkeypress、onkeyup 这几个键盘事件来监测的话，监听不了右键的复制、剪贴和粘贴这些操作，这时我们就需要 onput 事件了。

onput 事件最早是由 W3C 阵营那边提出来的，IE9 才支持。但 IE9 的那个也是半成品，它对回退建、粘贴复制操作的监听也失灵，解决方法可能是见招拆招，用 onkeydown 应付回退键，oncut 与 onpaste 对应后两个。IE9 也算有这事件，IE6～IE8 怎么办？它有一个 onpropertychange，能监听元素一切属性与特性的变化，因此我们只要稍做限制就能完美模拟 oninput 事件（通过事件对象的 propertyName 属性获取当前变动的属性名）。

现在我们回过头来搞定 IE9。在 IE9 下，onpropertychange 只能用了，不支持回退键、粘贴复制操作的监听。这时我从强大的候补团队拉个“人”出来干这事，它就是 onselectionchange 事件。

http://msdn.microsoft.com/en-us/library/ie/ms536968(v=vs.85).aspx

onselectionchange 事件的触发条件五花八门，我们只看倒数第 2 个，就发现它完美搞定回退键、粘贴复制操作的监听需求。

（1）Cause the selection object's type property to change.

（2）Return a range from a different location when using the createRange method of the selection object.

（3）Move the insertion point in an editable region of the document using the mouse or keyboard.

（4）Refresh the page when an editable region has focus.

（5）Start or extend a text selection by dragging the mouse or using SHIFT+an arrow key.

（6）Make a control selection in an editable region of the document.

（7）Add or remove an element from a multiple selection by pressing SHIFT while clicking on the element.

（8）Delete text or an element in an editable region of the document by using the BACKSPACE key, DELETE key, CTRL+X, or the Delete command.

（9）Insert text or an element in an editable region of the document by using CTRL+V or the Paste

command.

下面的代码取自 avalon.js，只要将当中的 updateModel 改成你的回调函数就可用。

```
https://github.com/RubyLouvre/avalon/blob/0.972/avalon.js#L2336
if (window.addEventListener) { //先执行 W3C
    element.addEventListener("input", updateModel)
} else {
    element.attachEvent("onpropertychange", function(e) {
        if (e.propertyName === "value") {
            updateModel()
        }
    })
}

if (document.documentMode === 9) {
    var selectionchange = function(e) {
        if (e.type === "focus") {
            document.addEventListener("selectionchange", updateModel)
        } else {
            document.removeEventListener("selectionchange", updateModel)
        }
    };
    element.addEventListener("focus", selectionchange)
    element.addEventListener("blur", selectionchange)
}
```

第12章 章异步处理

浏览器环境与后端的 nodejs 存在着各种消耗巨大或堵塞线程的行为，对于 JavaScript 这样单线程的东西唯一的解耦方法就提供异步 API。异步 API 是怎么样的呢？简单来说，它是不会立即执行的方法。比方说，一个长度为 1000 的数组，在 for 循环内，可能不到几毫秒就执行完毕，若在后端的其他语言，则耗时更少。但有时候，我们不需要这么快的操作，我们想在页面上能用肉眼看到它执行的每一步，那就需要异步 API。还有些操作，如加载资源，你想快也快不了，它不可能一下子提供给你，你必须等待，但你也不能一直干等下去什么也不干，得允许我们跳过这些加载资源的逻辑，执行下面的代码。于是浏览器首先搞出的两个异步 API，就是 setTimeout 与 setInterval。后面开始出现各种事件回调，它只有用户执行了某种操作后才触发。再之后，就更多，XMLHttpRequest、postMessage、WebWorkor、setImmediate、requestAnimationFrame 等。

这些东西都有一个共同的特点，就是拥有一个回调函数，描述一会儿要干什么。有的异步 API 还提供了对应的中断 API，比如 clearTimeout、clearInterval、clearImmediate、cancelAnimationFrame。

早些年，我们就是通过 setIimeout 或 setInterval 在网页上实现动画的。这种动画其实就是通过这些异步 API 不断反复调用同一个回调实现的，回调里面是对元素节点的某些样式进行很小范围的改动。

随着 iframe 的挖掘与 XMLHttpRequest 的出现，无缝刷新让用户驻留在同一个页面上的时间越来越长，许多功能都集成在同一个页面。为实现这些功能，我们就得从后端加载数据与模板，来拼装这些新区域。这些加载数据与模板的请求可能是并行的，可能是存在依赖的。只有在所有数据与模板都就绪时，我们才能顺利拼接出 HTML 子页面插入到正确的位置上。面对这些复杂的流程，人们不得不发明一些新模式来应对它们。最早被发明出来的是“回调地狱（callback hell）”，这应该是一个技能。事实上，几乎 javaScript 中的所有异步函数都用到了回调，连续执行几个异步函数的结果就是层层嵌套的回调函数，以及随之而来的复杂代码。因此有人说，回调就是程序员的 goto 语句[①]。

此外，并不是每一个工序都是一帆风顺的，如果有一个出错了呢，对于 JavaScript 这样单线程的语言，往往是致命的，必须 try...catch，但 try...catch 语句只能捕捉当前抛出的异常，对后来执行的代码无效。

```
function throwError() {
    throw new Error('ERROR');
```

① http://tirania.org/blog/archive/2013/Aug-15.html

```
}
try {
    setTimeout(throwError, 3000);
} catch (e) {
    alert(e);//这里的异常无法捕获
}
```

这些就是本章所要处理的课题。不难理解，domReady、动画、Ajax 在骨子里都是同一样东西，假若能将它们抽象成一个东西，显然是非常有用的。

12.1 setTimeout 与 setInterval

首先我们得深入学习一下这两个 API。一般的书籍只是简单介绍它们的用法，没有对它们内在的一些隐秘知识进行描述。它们对我们创建更有用的异步模型非常有用。

（1）如果回调的执行时间大于间隔间隔，那么浏览器会继续执行它们，导致真正的间隔时间比原来的大一点。

（2）它们存在一个最小的时钟间隔，在 IE6～IE8 中为 15.6ms[①]，后来精准到 10ms，IE10 为 4ms，其他浏览器相仿。我们可以通过以下函数大致求得此值。

```
function test(count, ms) {
    var c = 1;
    var time = [new Date() * 1];
    var id = setTimeout(function() {
        time.push(new Date() * 1);
        c += 1;
        if (c <= count) {
            setTimeout(arguments.callee, ms);
        } else {
            clearTimeout(id);
            var tl = time.length;
            var av = 0;
            for (var i = 1; i < tl; i++) {
                var n = time[i] - time[i - 1]; //收集每次与上一次相差的时间数
                av += n;
            }
            alert(av / count); // 求取平均值
        }
    }, ms);
}
winod.onload = function() {
    var id = setTimeout(function() {
        test(100, 1);
        clearTimeout(id);
    }, 3000);
}
```

具体如表 12.1 所示。

① http://ie.microsoft.com/testdrive/Performance/setImmediateSorting/Default.html

表 12.1

Firefox 3.6.3	Firefox 18.1	Chrome 10.53	Chrome 10.53	Opera 12.41	Safari 5.01	IE 8	IE 10
15.59	3.98	3.92	3.6	4.01	4.12	15.91	3.91

但上面的数据很难与官方给出的数值一致，因为它太容易受外部因素影响，比如电池快没电了，同时打开的应用程序太多了，导致 CPU 忙碌，这些都会让它的数值偏高。

如果嫌旧版本 IE 的最短时钟间隔太大，我们或许有办法改造一下 setTimeout，利用 image 死链时立即执行 onerror 回调的情况进行改造。

```
var orig_setTimeout = window.setTimeout;
window.setTimeout = function (fun, wait) {
    if (wait < 15) {
        orig_setTimeout(fun, wait);
    } else {
        var img = new Image();
        img.onload = img.onerror = function () {
            fun();
        };
        img.src = "data:,foo";
    }
};
```

（3）有关零秒延迟，此回调将会放到一个能立即执行的时段进行触发。JavaScript 代码大体上是自顶向下执行，但中间穿插着有关 DOM 渲染、事件回应等异步代码，它们将组成一个队列，零秒延迟将会实现插队操作。

（4）不写第二参数，浏览器自动配时间，在 IE、Firefox 中，第一次配可能给个很大数字，100ms 上下，往后会缩小到最小时钟间隔，Safari、Chrome、Opera 则多为 10ms 上下。Firefox 中，setInterval 不写第二参数，会当作 setTimeout 处理，只执行一次。

```
window.onload = function() {
    var a = new Date - 0;
    setTimeout(function() {
        alert(new Date - a);
    });
    var flag = 0;
    var b = new Date,
        text = ""
    var id = setInterval(function() {
        flag++;
        if (flag > 4) {
            clearInterval(id)
            console.log(text)
        }
        text += (new Date - b + " ");
        b = new Date
    })
}
```

（5）标准浏览器与 IE10，都支持额外参数，从第三个参数起，作为回调的传参传入！

```
setTimeout(function() {
 alert([].slice.call(arguments));
}, 10, 1, 2, 4);
```

IE6～IE9 可以用以下代码模拟。

```
if (window.VBArray && !(document.documentMode > 9)) {
    (function(overrideFun) {
        window.setTimeout = overrideFun(window.setTimeout);
        window.setInterval = overrideFun(window.setInterval);
    })(function(originalFun) {
        return function(code, delay) {
            var args = [].slice.call(arguments, 2);
            return originalFun(function() {
                if (typeof code == 'string') {
                    eval(code);
                } else {
                    code.apply(this, args);
                }
            }, delay);
        }
    }
    );
}
```

（6）setTimeout 方法的时间参数若为极端值（如负数、0、或者极大的正数），则各浏览器的处理会出现较大差异，某些浏览器会立即执行。幸好最近所有最新的浏览器都立即执行了。

12.2 Mochikit Deferred

Deferred[①]是当今最著名的异步模型。它原来是 Python 的 Twisted 框架的一个类，后来被 Mochikit 框架引进来，再后来又被 dojo 抄去。现在你们又看到，同名的东西又出现在 jQuery1.5 上。

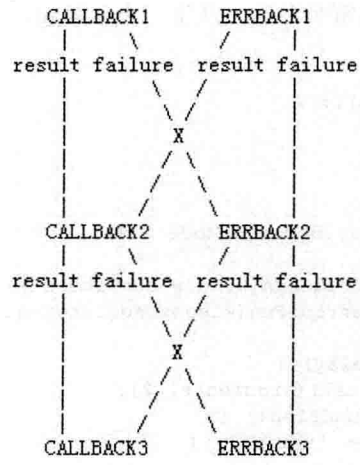
怎么描述 Deferred 好呢？它是一个双链参数加工的流水线模型。双链是指它内部把回调分成两种，一种叫成功回调，用于正常时执行，一种叫错误回调，用于出错时执行。各自组成两个队列，这里姑且叫成功队列与错误队列。添加回调时是一组组添加的。每组回调的参数都是上一组回调的处理结果，当然只有第一组的参数是用户传入的。说是流水线，是说每个回调可能不是紧挨着执行，有时要耗些时间，可能是异步 API 引起的，也可能是我们调用了诸如 wait 这样的方法。假若出错，由后一组的错误回调捕获处理，没有问题尝试再转回成功队列，如图 12.1 所示。

http://pythonhosted.org/defer/defer.html

现在我们看看源码吧，这个是我从 Mochikit 框架下描下来的，经改进能独立运行。

```
//https://github.com/mochi/mochikit/blob/master/MochiKit/Async.js
Deferred = function(/* optional */ canceller) {
    this.chain = [];
```

① http://wiki.ecmascript.org/doku.php?id=strawman:deferred_functions



▲图 12.1

```
    this.id = setTimeout("1")
    this.fired = -1;
    this.paused = 0;
    this.results = [null, null];
    this.canceller = canceller;
    this.silentlyCancelled = false;
    this.chained = false;
};

function curry(fn, scope, args) {
    return function() {
        var argv = [].concat.apply(args, arguments)
        return fn.apply(scope, argv);
    };
}
Deferred.prototype = {
    //3 种状态，未触发，触发成功，触发失败
    state: function() {
        if (this.fired == -1) {
            return 'unfired';
        } else if (this.fired === 0) {
            return 'success';
        } else {
            return 'error';
        }
    },
    //取消触发，类似于 ajax 的 abort
    cancel: function(e) {
        if (this.fired == -1) { //只有未触发时才能 cancel 掉
            if (this.canceller) {
                this.canceller(this);
            } else {
                this.silentlyCancelled = true;
            }
```

```
            if (this.fired == -1) {
                if (!(e instanceof Error)) {
                    e = new Error(e + "");
                }
                this.errback(e);
            }
        } else if ((this.fired === 0) && (this.results[0] instanceof Deferred)) {
            this.results[0].cancel(e);
        }
    },
    //这里决定是用哪个队列
    _resback: function(res) {
        this.fired = ((res instanceof Error) ? 1 : 0);
        this.results[this.fired] = res;
        if (this.paused === 0) {
            this._fire();
        }
    },
    //判定是否触发过
    _check: function() {
        if (this.fired != -1) {
            if (!this.silentlyCancelled) {
                throw new "此方法已经被调用过";
            }
            this.silentlyCancelled = false;
            return;
        }
    },
    //触发成功队列
    callback: function(res) {
        this._check();
        if (res instanceof Deferred) {
            throw new Error("Deferred instances can only be chained if they are the result
of a callback");
        }
        this._resback(res);
    },
    //触发错误队列
    errback: function(res) {
        this._check();
        if (res instanceof Deferred) {
            throw new Error("Deferred instances can only be chained if they are the result
of a callback");
        }
        if (!(res instanceof Error)) {
            res = new Error(res + "");
        }
        this._resback(res);
    },
    //同时添加成功与错误回调
    addBoth: function(a, b) {
        b = b || a
        return this.addCallbacks(a, b);
    },
```

```
//添加成功回调
addCallback: function(fn) {
    if (arguments.length > 1) {
        var args = [].slice.call(arguments, 1);
        fn = curry(fn, window, args);
    }
    return this.addCallbacks(fn, null);
},
//添加错误回调
addErrback: function(fn) {
    if (arguments.length > 1) {
        var args = [].slice.call(arguments, 1);
        fn = curry(fn, window, args);
    }
    return this.addCallbacks(null, fn);
},
//同时添加成功回调与错误回调，后来 Promise 的 then 方法就是参考它设计
addCallbacks: function(cb, eb) {
    if (this.chained) {
        throw new Error("Chained Deferreds can not be re-used");
    }
    if (this.finalized) {
        throw new Error("Finalized Deferreds can not be re-used");
    }
    this.chain.push([cb, eb]);
    if (this.fired >= 0) {
        this._fire();
    }
    return this;
},
//将队列的回调依次触发
_fire: function() {
    var chain = this.chain;
    var fired = this.fired;
    var res = this.results[fired];
    var self = this;
    var cb = null;
    while (chain.length > 0 && this.paused === 0) {
        var pair = chain.shift();
        var f = pair[fired];
        if (f === null) {
            continue;
        }
        try {
            res = f(res);
            fired = ((res instanceof Error) ? 1 : 0);
            if (res instanceof Deferred) {
                cb = function(res) {
                    self.paused--;
                    self._resback(res);
                };
                this.paused++;
            }
        } catch (err) {
            fired = 1;
```

```
                if (!(err instanceof Error)) {
                    try {
                        err = new Error(err + "");
                    } catch (e) {
                        alert(e)
                    }
                }
                res = err;
            }
        }
        this.fired = fired;
        this.results[fired] = res;
        if (cb && this.paused) {
            res.addBoth(cb);
            res.chained = true;
        }
    }
};
```

我们先通过它的 addCallback、addErrback、addBoth 方法来添加回调函数。

addCallback 用于正常返回时执行，第一个参数为函数，允许第二、第三等额外参数，反正内部一个 curry 搞定。

addErrback 用于出错时执行，参数同 addCallback。

addBoth 方便同时添加正常回调与错误回调。

它们都是内部调用于 addCallbacks 方法，参数是两个函数或一个函数一个 null，上面三个会设法弄成这种格式传到。Deferred 实例有一个 chain 数组属性，它的每个元素都是一个双元素的数组，换言之是这个样子：

```
deferred.chain = [ [fn1,fn2],[fn3,fn4],[fn5,fn6]/*略*/]
```

另一个更直接的例子：

```
var d = new Deferred();
d.addCallback(myCallback);
d.addErrback(myErrback);
d.addBoth(myBoth);
d.addCallbacks(myCallback, myErrback);
```

它的 chain 结构就是这样：

```
[
    [myCallback, null],
    [null, myErrback],
    [myBoth, myBoth],
    [myCallback, myErrback]
]
```

触发这些回调是通过 callback 与 ertback 方法，当然通常我们放到 XMLHttpRequest 对象的回调中执行它们。我们可以查看 XMLHttpRequest 的 status（状态码），即便是成功返回还是服务器错误，决定调用 Deferred 对象的 callback 还是 ertback，将返回值传入到它们里面。

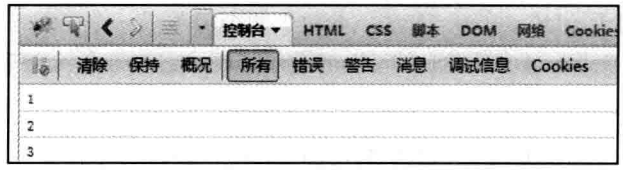
callback 与 errback 里面的流程都很一致，首先检测此 Deferred 对象有没有被调用过，它们就好像一次性用品，然后抛给_resback 方法。

_resback 很简单，就是把这参数放到一个数组中，正常放在第一个位置，出错放到第二个位置，然后判定有没有被暂停，调用_fire 方法执行回调。

_fire 就是不断弹出 chain 中的那组函数，根据状态取第一个或第二个回调，放入 results 中的值 res 执行，执行后得到的结果会立即重写 res。因此这是个瀑布模型，不像事件那样，每一个回调都是接受相同的参数。Deferred 中的每一个回调都是接受上一组回调的返回值。在这个过程中有个 try...catch，如果出错就转到错误队列去，下次能正常执行又跑回成功队列。

```
function increment(value) {
    console.log(value);
    return value + 1;
}

var d = new Deferred();
d.addCallback(increment);
d.addCallback(increment);
d.addCallback(increment);
d.callback(1);
```

如图 12.2 所示。



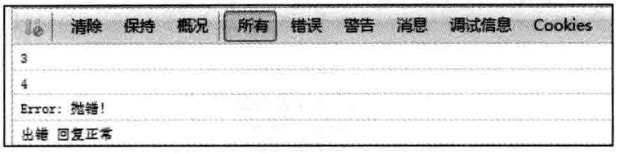
▲图 12.2

再看一个复杂点的。

```
var d = new Deferred();
d.addCallback(function(a) {
    console.log(a)
    return 4
}).addBoth(function(a) {
    console.log(a);
    throw "抛错"
}, function(b) {
    console.log(b)
    return "xxx"
}).addBoth(function(a) {
    console.log(a);
    return "正常"
}, function(b) {
    console.log(b + "!")
    return "出错"
}).addBoth(function(a) {
    console.log(a + " 回复正常");
```

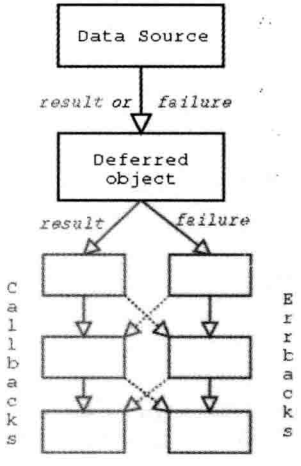
```
    return "正常 2"
}, function(b) {
    console.log(b + "  继续出错")
    return "出错 2"
})
d.callback(3)
```

具体如图 12.3 所示。



▲图 12.3

这里从上到下共有四组，先进入第一组的成功回调，没有问题；进入第二组的成功回调，使用第一组的返回值做参数，里面抛错，转入错误队列，下次交由第三组的错误回调处理；错误回调中在执行时没有发生异常，于是再转入成功队列；第四组的成功回调执行上一组返回的结果，具体如图 12.4 所示。



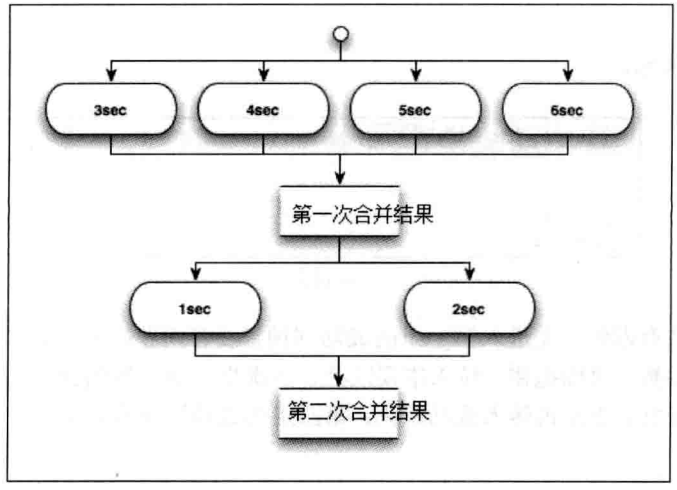
▲图 12.4

在 Mochikit 中，它只是作为 Ajax 的主要部件而存在。有了它，它就能实现两大功能：（1）能同时绑定多成功回调或错误回调，而不用借助观察者模式或事件系统。

能有效地捕捉回调中的异常，众所周知，Ajax 的调试是很复杂的。

其实，它还有一个最重要的功能，并归多个 Ajax 的结果，然后再做一次依赖它们的处理。我称这个最后的回调为“回调的回调”。不过，Mochikit 是用 DeferredList 实现的，你可以看作是 Deferred 的“五星战队合体版”。早期的 jQuery 与 Prototype 没有这个东西，自己用计数器实现比较痛苦。

比如有一个业务，需要发起 4 个请求，这 4 个请求的地址与返回时间都不一样，必须等到它们都到齐时才能整理，合并成一个数据，然后根据这个数据的某些值再发出两个请求，等到它们都返回时，最后处理一次才算成功。如果用 Mochikit 来实现是非常简单明了的，具体如图 12.5 所示。

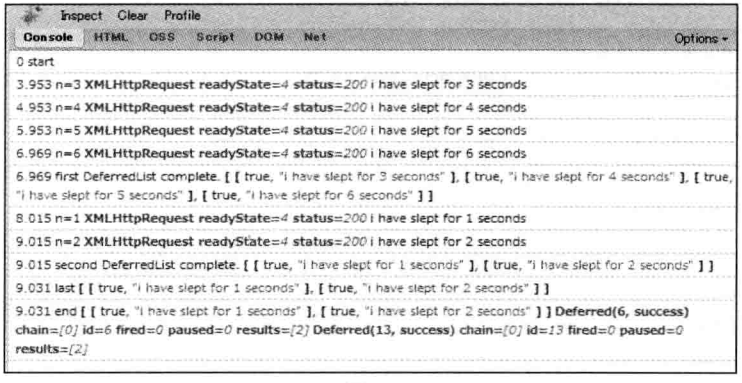


▲图 12.5

```
var elapsed = (function() {
    var start = null;
    return function() {
        if (!start)
            start = Date.now();
        return ((Date.now() - start) / 1000)
    }
})();

console.log(elapsed(), "start");

var dl1, dl2;

dl1 = new DeferredList([
    doXHR('/sleep.php?n=3').addCallback(function(res) {
        console.log(elapsed(), "n=3", res, res.responseText);
        return res.responseText;
    }),
    doXHR('/sleep.php?n=4').addCallback(function(res) {
        console.log(elapsed(), "n=4", res, res.responseText);
        return res.responseText;
    }),
    doXHR('/sleep.php?n=5').addCallback(function(res) {
        console.log(elapsed(), "n=5", res, res.responseText);
        return res.responseText;
    }),
    doXHR('/sleep.php?n=6').addCallback(function(res) {
        console.log(elapsed(), "n=6", res, res.responseText);
        return res.responseText;
```

```
    })]).addCallback(function(res) {
    console.log(elapsed(), "first DeferredList complete.", res);

    return dl2 = new DeferredList([
        doXHR('/sleep.php?n=1').addCallback(function(res) {
            console.log(elapsed(), "n=1", res, res.responseText);
            return res.responseText;
        }),
        doXHR('/sleep.php?n=2').addCallback(function(res) {
            console.log(elapsed(), "n=2", res, res.responseText);
            return res.responseText;
        }), ]).addCallback(function(res) {
        console.log(elapsed(), "second DeferredList complete.", res);
        console.log(elapsed(), "last", res);
        return res;
    })
}).addCallback(function(res) {
    console.log(elapsed(), "end", res, dl1, dl2);
});
```

具体如图 12.6 所示。



▲图 12.6

作为最早的先进者，它能做到预想到如此复杂的情景应用，是难能可贵的。但由于 Mochikit 的推广不力，它最终没有发展起来。

12.3 JSDeferred

JSDeferred 是日本高手 cho45 搞出来的，其易用性远胜于 Mochikit Deferred，其实现更堪称一绝，它的实现形态基本奠定了后来称为“Promise/A”的范式，是 javascript 在异步编程上的一个里程碑式作品，不禁让我联想到“比特币创世论文[①]”这类东西。因此我们这节将会花大篇幅探讨其

① 2009 年，一个名叫中本聪（Satoshi Nakamoto）的神秘黑客率先提出了“比特币”这个概念，并描述了一种利用计算机网络创造一种“秘密货币”的方法——比特币不由某家公司或某个央行发行，也不与任何现实货币挂钩，却可以用来购买现实世界中的物品和服务。读者们可以上网搜索《Bitcoin: A Peer-to-Peer Electronic Cash System》

源码是怎么运转的。不可否认，看得懂这代码需要有很强的时空感，需要从某一个时间跳跃到另一个时间。某一个阶段可能是一下子绑定许多回调，另一个阶段则是执行它们，如果穿插了 wait 方法，则划分更多阶段。然而，它本身不像 Mochikit Deferred 那样用数组保存回调，作为载体的竟然是 Deferred 对象自身，换言之，每一个回调都至少涉及一个 Deferred，我们的流程就是在这个 Deferred 连中流转。

源代码位于 github，自己下回来，引入到页面。看懂这库对自己的能力提升很大。

https://github.com/cho45/jsdeferred/blob/master/jsdeferred.js

12.3.1　得到一个 Deferred 实例

```
Deferred.define();
next(function(){
       /* 处理 */
})
```

上面代码能让我们得到一个匿名的 Deferred 实例，绑定一个成功时执行的回调。它使用 next，而不是 addCallback，API 属于 DSL 风格。不过，这样做非常不好，它把 next、wait 等方法都放到全局作用域下。或许会更喜欢下面这种无侵入的写法。

```
var o = {}; //定义一个对象
Deferred.define(o);//把 Deferred 的方法加持到它上面，让 o 成为一个 Deferred 子类

for(var x in o)alert(x);
// parallel
// wait
// next
// call
// loop

o.next(function(){
  /* 处理 */
})
```

或者直接使用 Deferred：

```
Deferred.next(function(){//在暗地里创建一个 Deferred 实例，然后我们直接“链”下去就行了
  /* 处理 */
})
```

无论哪一个，最初的 next 都是一个静态方法，目的是用于提供了一个 JSDeferred 实例与执行第一个异步操作。异步操作在 JSDeferred 中有许多实现方式，如 setTimeout、img.onerror 或 script.onreadystatechange，它会视浏览器选择最快的 API。现在我们看最简单的实现方式，setTimeout。它在 next_default 静态方法中实现。

```
Deferred.next_default = function(fun) {
    var d = new Deferred();
    var id = setTimeout(function() {
        clearTimeout(id);
```

```
        d.call()
    }, 0);
    d.canceller = function() {
        try {
            clearTimeout(id)
        } catch (e) {
        }
    };
    if (fun)
        d.callback.ok = fun;
    return d;
};
```

无论何种 next 静态方法，最终都会返回一个 Deferred 实例，因此第二个 next 方法与第一个 next 是完全不同的。它是一个实例方法，并设置其_post 的第一个参数为“ok”，并在_post 中方法重置 callback.ok 与设置其_next，从而构造一个 Deferred 链。

为了方便讲解，我先给出一个例子，并贴出相关源码。

```
Deferred.define();
next(function func1() {
    alert(1)
})
.next(function func2() {
    alert(2)
});
alert(3);
```

依次弹出 3，1，2。因为 Deferred 是个异步列队，它的回调都是延迟执行的。

```
function Deferred() {
    return (this instanceof Deferred) ? this.init() : new Deferred()
}
Deferred.ok = function(x) {
    return x
};
Deferred.ng = function(x) {
    throw  x
};
Deferred.prototype = {
    init: function() {
        this._next = null;
        this.callback = {
            ok: Deferred.ok,
            ng: Deferred.ng
        };
        return this;
    },
    next: function(fun) {
        return this._post("ok", fun)
    },
    error: function(fun) {
        return this._post("ng", fun)
    },
    call: function(val) {
```

```
        return this._fire("ok", val)
    },
    fail: function(err) {
        return this._fire("ng", err)
    },
    cancel: function() {
        (this.canceller || function() {
        })();
        return this.init();
    },
    _post: function(okng, fun) {
        this._next = new Deferred();
        this._next.callback[okng] = fun;
        return this._next;
    },
    _fire: function(okng, value) {
        var next = "ok";
        try {
            value = this.callback[okng].call(this, value);
        } catch (e) {
            next = "ng";
            value = e;
        }
        if (value instanceof Deferred) {
            value._next = this._next;
        } else {
            if (this._next)
                this._next._fire(next, value);
        }
        return this;
    }
};
```

我们整理一下思路。

（1）最初的 next 会创建一个新的 Deferred 实例（以下简称为 d1）,然后将 func1 放入 d1.callback.ok 中。

（2）第二个的 next 会创建一个新的 Deferred 实例（以下简称为 d2）,然后将 func2 放入 d2.callback.ok 中。

（3）在_post 中将 d1._next 设置为 d2。

（4）setTimeout 零秒延迟暂时挂起它里面的回调函数，让代码后面的 alert(3)先执行。当 setTimeout 回调执行时，它的 d.call 会先打到_fire 方法，从而执行 d1.callback.ok。如果 d1.callback.ok 的返回值不为 Deferred 实例，并且 d1._next 不为空，则执行 d1._next.callback.ok，相当于执行 d2.callback.ok。

（5）就这样，两个绑定好的回调安然执行，不需要 shift 什么，也不需要做什么检测来阻止 Deferred 被重复触发。

12.3.2　Deferred 链的实现

它每绑定一个回调函数就需要一个全新的 Deferred 对象，从而形成一个 Deferred 链。两个

Deferred 是否能连到一块，取决了两个东西，当前 Deferred 实例在_fire 方法执行 callback 时得到的返回值和_next 属性。

_next 属性会在两个地方用到，_post 实例方法与_fire 实例方法。想调用_post 方法，就要调用 next 实例方法。想调用_fire 方法，我们目前只是通过 call 方法实现，而 call 位于 setTimeout 中。由于 setTimeout 的零秒延迟，换言之，当我们执行 call 方法时，程序已经为 d1 准备好 callback 与_next 了，然后再进入_fire 方法中。

要注意一下，Deferred.prototype._fire 内的 this 实质为 d1。

```
value =this.callback[okng].call(this, value);
//相当于 d1.callback.ok.call(d1,null)
//相当于 func1.call(d1,null)
//相当于 d1.func1();
```

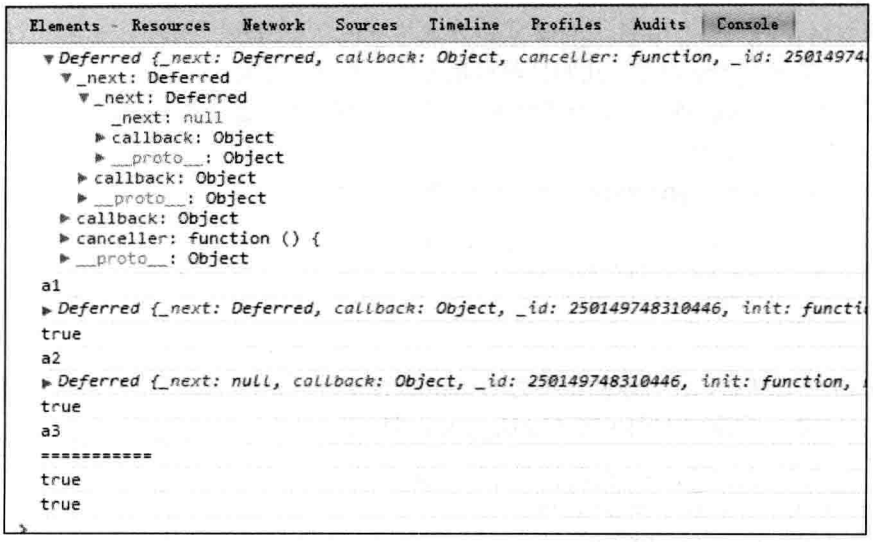
如果 func1 正常执行，并且返回值不为 Deferred 的实例，由于它已经拥有_next，因此接着便再一次执行_fire 方法，这次它的调用者为 this._next，即 d2。

```
if (this._next)this._next._fire(next, value);
  //相当于 if(d2)d2.fire(next,value)
  //相当于 if(d2)d2.fire("ok",null)
  //相当于 d2.call()
```

由于 d2 的_next 为 null，func2 也没有返回值，程序就此结束。最后提一下，func1 与 func2 是同步执行的，因为它们之间没有用 setTimeout 隔开。

```
window.onload = function() {
    //最好把 Deferred 放到 window.onload 中,因为当它的 next 用
    //image.onload 实现时会用 document.body.appendChild(image)
    var a, b, c;
    Deferred.
          next(function() {
        console.log(this);
        a = this;
        console.log("a1")
    })
          .next(function() {
        console.log(this);
        b = this;
        console.log(a !== b)
        console.log("a2");
    })
          .next(function() {
        console.log(this);
        c = this;
        console.log(b !== c)
        console.log("a3");
        console.log("===========");
        console.log(a._next === b);
        console.log(b._next === c);
    })
}
```

在 Chrome 的控制台观察到的 Deferred 链结构，如图 12.7 所示。



▲图 12.7

如果想让它们也异步，可以插入一个 wait 方法。由于 wait 是用 setTimeout 实现的，它受到最小时钟间隔的限制，你传 0 进去，它不会真的 0ms 后执行，最小要 4ms。

```
Deferred.next(function() {
    alert(1)
})
        .wait(0)
        .next(function() {
    alert(2)
})
```

不过我们在源码中没有看到原型链有 wait 这个方法，倒是找到一个同名的静态方法。再认真看，就会发现它是通过 register 方法进行函数转换，并绑到原型上了。

```
Deferred.wait = function(n) {
    var d = new Deferred(), t = new Date();
    var id = setTimeout(function() {
        d.call((new Date()).getTime() - t.getTime());
    }, n * 1000);
    d.canceller = function() {
        clearTimeout(id)
    };
    return d;
};

Deferred.register = function(name, fun) {
    this.prototype[name] = function() {
        var a = arguments;
        return this.next(function() {
```

```
            return fun.apply(this, a);
        });
    };
};
Deferred.register("wait", Deferred.wait);
```

这里的代码可以重点关注 3 处。它只是一个 curry 函数，里面调用了一次 next。上面介绍过，它是一个_post 方法的包装，里面会产生一个 Deferred 对象，为了方便称我们叫它为 d_hide。而 Deferred 的 wait 方法又会产生一个 Deferred 实例 d_wait，并且最后返回了它。因此，上例短短几句涉及到四个 Deferred 对象。

```
Deferred.next(function fun1() { //产生 Deferred 对象 d1
    alert(1)
})
        .wait(0) // 这里相当于.next(function curry(){ return Deferred.wait(0) })
        // 产生 Deferred 对象 d_hide 与 d_wait
        .next(function fun2() { //产生 Deferred 对象 d2
    alert(2)
})
```

为什么要这样做呢？我们看核心的_fire 方法（好像所有 Deferred 的实现，_fire 都是非常长而且非常重要的）。

```
_fire : function(okng, value) {
    var next = "ok"; //每次都尝试从成功队列执行
    try { //决定是执行成功回调还是错误回调
        value = this.callback[okng].call(this, value);
    } catch (e) {
        next = "ng";
        value = e; //反正返回值
        if (Deferred.onerror)
            Deferred.onerror(e);
    }
    if (value instanceof Deferred) {
        value._next = this._next;
    } else { //执行链表中的下一个 Deferred 的_fire
        if (this._next)
            this._next._fire(next, value);
    }
    return this;
}
```

假令我们在第一个静态方法 next 搞个断点，在它的 setTimeout 回调还没有执行时，它的 Deferred 链其实已经完成。这时只有 3 个 Deferred 对象，d_wait 则要在 wait 方法的 setTimeout 中产生。

```
d1
  .callback.ok = fn1
  ._next       = d_hide
                    .callback.ok = curry
                    ._next       = d2
                                       .callback.ok = fn2
                                       ._next       = null
```

如果中途插入 wait 方法，那么：

```
value = this.callback[okng].call(this, value);
就相当于
// value =  d_hide.callback.ok.call(d_hide,value)
// value =  curry.call(d_hide,value)
// value = (function curry(){ return Deferred.wait(0) }).call(d_hide,value)
// value = d_wait
```

接着在 value instanceof Deferred 这一行，把_next 对象进行转移，防止它在 setTimeout 中执行。

```
if (value instanceof Deferred) {
    value._next = this._next;
    //相当于 d_wait._next = d_hide._next
    //相当于 d_wait._next = d2
} else {
    if (this._next)
        this._next._fire(next, value);
}
```

wait 是个好方法，能让我们的程序就像时钟那样一下一下地执行回调。

```
var d = Deferred.next(function() {
    console.log("0")
});
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10].forEach(function(index) {
    d = d.wait(1).next(function() {
        console.log(index + "!!")
    })
});
```

12.3.3　JSDeferred 的并归结果

JSDeferred 最厉害之处是它的 parallel 的实现，它想实现 12.2 节提到的那种 Ajax 套嵌请求更加轻松，假若那个 Ajax 方法是返回 Deferred 对象，就更轻松。不难想象，它里面存在一个数组，用于收集每一个阶段 N 个并行执行的回调的结果，同时，还应该有一个计数器，如果回调都成功执行了，开始执行“回调的回调”，这个回调当然是我们的框架提供的，然后处理所有收集到的结果，放进用户绑定的后续回调中。

```
Deferred.parallel(function() {
    return 1
}, function() {
    return 2
}, function() {
    return 3
}).next(function(a) {
    console.log(a)//[1,2,3]
})
```

但 parallel 里面都是同步的，我们弄个异步的，搞个假的 Ajax 函数就知道很好用了！

```
window.onload = function() {
```

```
    function ajax(url) {
        var d = Deferred();
        var delay = url.match(/time=(\d+)/)[1];
        // 真正的实现应该如下:
        // var xhr = new (self.XMLHttpRequest || ActiveXObject)("Microsoft.XMLHTTP");
        // xhr.onreadystatechange = function() {
        //     if (ajax.readyState == 4 && ajax.status == 200) {
        //         d.call(ajax.responseText)
        //     }
        // }
        // xhr.onload = function() {
        //     d.call(ajax.responseText)
        // }
        // xhr.onerror = function() {
        //      d.fail(ajax.responseText)
        //  }
        //xhr.open(ooooo )
        setTimeout(function() {
            d.call(delay)
        }, +delay);
        return d
    }
    Deferred.parallel(
            ajax("rubylovre?time=2000"),
            ajax("rubylovre?time=3000"),
            ajax("rubylovre?time=4000")).next(function(a) {
        //大概等 4 秒
        console.log(a) //[2000,3000,4000]
    })
}
```

我们来看看其实现：

```
Deferred.parallel = function(dl) {
    var isArray = false; //它可以放一个数组或对象，或 N 个函数、或 Deferred 对象做参数
    if (arguments.length > 1) {
        dl = Array.prototype.slice.call(arguments);
        isArray = true;
    } else if (Array.isArray && Array.isArray(dl) || typeof dl.length == "number") {
        isArray = true;
    }
    //并归用的 Deferred
    var ret = new Deferred(),
            //收集结果
            values = {},
            //计数器
            num = 0;
    for (var i in dl) {
        if (dl.hasOwnProperty(i)) {
            (function(d, i) {
                //统统转成 Deferred 对象
                if (typeof d == "function")
                    dl[i] = d = Deferred.next(d);//转换为 Deferred 对象
```

```
                d.next(function(v) {//然后让它们添加两个回调：next 与 error
                    values[i] = v;
                    if (--num <= 0) {
                        //到这一步，说明用户最初传入的函数都成功返回数据了，组成 values
                        if (isArray) {
                            values.length = dl.length;
                            values = Array.prototype.slice.call(values, 0);
                        }
                        ret.call(values);
                    }
                }).error(function(e) {
                    ret.fail(e);
                });
                num++;
            })(dl[i], i);
        }
    }
    //如果里面没有内容立即执行
    if (!num)
        Deferred.next(function() {
            ret.call()
        });
    ret.canceller = function() {
        for (var i in dl)
            if (dl.hasOwnProperty(i)) {
                dl[i].cancel();
            }
    };
    return ret;
};
```

除此之外，JSDeferred 还提供了一个 loop 方法，用于模拟 for 循环的串行模式。它相当于内置了 range 方法，能生成我们想要的数组，然后逐个放进回调中执行。10.3.2 节的例子可改成这样：

```
Deferred.loop(10, function(n) {
    console.log(n);
    return Deferred.wait(1);
});
```

12.3.4　JSDeferred 的性能提速

JSDeferred 的每一个对象都允许自动执行它绑定的回调，不需要手动调用 call、fail 方法。但它们不是立即执行，而是异步的，它需要等待用户用 next、errot 绑定好所需要的回调。12.2.1 小节介绍过它的默认 next 方法，它还有其他几种方法用于提速的。

首先是针对 IE 的提速。

```
Deferred.next_faster_way_readystatechange = (typeof window === 'object')
        && (location.protocol == "http:")
        && window.VBArray && function(fun) {// MSIE
    var d = new Deferred();
    var t = new Date().getTime();
    //浏览器的并发请求数是有限的，在 IE6、IE7 中为 2～4，IE8、IE9 为 6
```

```
    //如果超出这数目，可能造成堵塞，必须待这几个处理完才继续处理
    //因此这里添加一个阈值，如果上次与这次相隔超过 150 还没有处理完
    //那么就退化到原始的 setTimeout 方法
    if (t - arguments.callee._prev_timeout_called < 150) {
        var cancel = false;
        //创建一个 script 节点，加载一个不存在的资源来引发 onerror
        //由于旧版本 IE 不区分 onerror 与 onload，都只会触发 onreadystatechange
        //那么就用 onreadystatechange
        //只要造成异步效果，让用户有足够时间绑定回调就行了
        var script = document.createElement("script");
        script.type = "text/javascript";
        script.src = "data:text/javascript,";
        script.onreadystatechange = function() {
            if (!cancel) {
                d.canceller();
                d.call();
            }
        };
        //清掉事件与移出 DOM 树
        d.canceller = function() {
            if (!cancel) {
                cancel = true;
                script.onreadystatechange = null;
                document.body.removeChild(script);
            }
        };
        //由于这里的逻辑，我们的 Deferred 最好延迟到 domReady 或 onload 后执行
        document.body.appendChild(script);
    } else {
        arguments.callee._prev_timeout_called = t;
        var id = setTimeout(function() {
            d.call()
        }, 0);
        d.canceller = function() {
            clearTimeout(id)
        };
    }
    if (fun)
        d.callback.ok = fun;
    return d;
};
```

针对标准浏览器的加速如下。

```
Deferred.next_faster_way_Image = (typeof window === 'object')
        && (typeof(Image) != "undefined")
        && !window.opera && document.addEventListener && function(fun) {
    // 用于 Opera 外的标准浏览器
    var d = new Deferred();
    var img = new Image();
    //创建一个 image 加载一个不存在的图片（为了防止万分之一的图片存在的情况，onload 也绑上了）
    var handler = function() {
        d.canceller();
```

```
        d.call();
    };
    img.addEventListener("load", handler, false);
    img.addEventListener("error", handler, false);
    d.canceller = function() {
        img.removeEventListener("load", handler, false);
        img.removeEventListener("error", handler, false);
    };
    img.src = "data:image/png," + Math.random();
    if (fun)
        d.callback.ok = fun;
    return d;
};
```

根据 JSDeferred 官方的数据，用上这两个后至少比原有的 setTimeout 异步方式快上 700%以上[①]，如图 12.8 所示。

环境	setTimeout	faster_way	%
Opera 9.5 (Mac)	80sec	1.7sec	4706%
Internet Explorer (Win), calc time	120sec	6sec	2000%
Safari 3.1 (Mac)	80sec	6sec	1300%
Google Chrome (Win)	31sec	2.4sec	1200%
Firefox 3.1b2 (Mac)	83sec	10sec	830%
Internet Explorer (Win)	120sec	17sec	700%

▲图 12.8

同时，JSDeferred 不单单是运行于普通网页，还能在浏览器的插件环境或 node.js 上运行[②]。它存在多个版本，自己可以针对需要使用不同的版本。node.js 适配方法则已经包含在原始版本中。

```
Deferred.next_tick = (typeof process === 'object')
        && (typeof process.nextTick === 'function') && function(fun) {
    var d = new Deferred();
    process.nextTick(function() {
        d.call()
    });
    if (fun)
        d.callback.ok = fun;
    return d;
};
```

这里说一下，node.js 已经支持快得惊人的 setImmediate API 了，读者可以尝试做一个 next_setImmediate。

当然，最快的方式就不使用这些具有延迟效果的异步 API，我们通过精妙的数组结构也能达到目的[③]，但是这样一来就不能自动触发了。jQuery Deferred 就是采取这种方式。因为我们的 Deferred 对象通常是在 XMLHttpRequest 等的事件回调中被触发的，没有自己再异步一次。

① http://subtech.g.hatena.ne.jp/cho45/20090125/1232831437
② http://cho45.stfuawsc.com/jsdeferred/
③ http://www.cnblogs.com/rubylouvre/archive/2011/03/18/1984336.html

12.4 jQuery Deferred

最先浮出水面的是_Deferred，一个单链结构，稍微改一改就能实现 domReady。它们合起来，就是 Deferred，外加一个 when 函数，就是实现 JSDeferred 那种并归多个异步操作的效果。当时马上就投入 ajax 模块的改造，立即让此模块难读十倍以上。

jQuery Deferred 的存在感与受众面一直很低，1.52 正式独立成一个模块，对外宣传，但没什么人埋单。

jQuery Deferred 的实现也堪称一绝，实现得十分玄乎，其 API 也不易懂，如 then、promise、resolve、reject……

因为 promise 是一个非常学术化的东西，假若 jQuery 官方不讲解，不要指望外界能理解它。

回顾 Deferred 的实现，它就是将一系列异步操作以优雅的形态链起来，然后在某个时刻一下子执行。其中有个要点，必须保证异步链有足够的时间构建。

JSDeferred 发掘 setTimeot 0、image.onerror、script.onreadystatechange 等的缘由也在此。此外，保存回调时也很有特点，是一组组保存的。比如 Mochikit Deferred、dojo Deferred,是一个 2 维数组，里面的小数组们每次只装两个回调。JSDeferred 则每个实例必有 ng、ok 这两个回调。但 jQuery 不一样，完全的离经叛道。首先，构成 Deferred 的子结构_Deferred，利用闭包手段，实现返回的那个对象能通过它的某些方法操作其引入一个数组与一些开头。这个数组可以不断添加回调。

合体之后，一个_Deferred 负责添加成功回调，一个负责添加错误回调。但 Deferred 这对象其实并不负责添加回调，它是负责执行的。添加回调的任务交给一个叫 Promise 的东西，这个由 promise 方法产生。

jQuery 使用一个巧妙的方法保证每个 Deferred 每次调用 promise 总是返回相同的一个对象，换言之，一个 Deferred 只能有一个 Promise。Promise 拥有 Deferred 除 resolve、reject、resolveWith、rejectWith 外等一切成员。

换言之，它是一个只读的 Deferred，目的是防止在构建“Deferred 链”的过程中就执行回调，改变其状态。之所以用引号，是因为 jQuery 一个 Deferred 对象，其实是相当其他 Deferred 库 N 个类似的对象组成的结构体。

John Resig 认为这样更节能，封装性更好。

但 jQuery Deferred 的实现很糟糕。首先，它没有对异常进行处理，在异步中捕捉异常这个重要功能缺席了。其次，jQuery Deferred 没有对原始参数进行流水化加工。比如想从文本上读取某东西到控制器，经过加工，从原始的 Buffer，到符合某种编码的字符串，再整成对象，才好处理。又比如审批流程，每流转到下一个步骤，必须会在原对象上修改一些东西。这是一个非常普遍的需求。

由于 jQuery 的创新，让其 Deferred 变成一个普通的回调集合。事实上，jQuery Callbacks 的出现，证明 jQuery 团队彻底理解错这东西了。即便是后来幡然悔悟，推出 pipe 方法，也大势已去。但我们也不必过于苛求 jQuery Deferred，毕竟人家已经完成整合所有异步操作这一使命，对于普通用户来说，也已够用。

为了方便学习 jQuery，明白这些 API 是什么意思，我们把它与早期的 Deferred 库放在一起，

就一目了然。如表 12.2 所示。

表 12.2

	Mochikit Deferred	JSDeferred	jQuery Deferred
添加一个成功回调	addCallback	next	done
添加一个错误回调	addErrback	error	fail
添加一组回调	addBoth		Then|pipe
触发成功队列	callback	call	resolve|resolveWith
触发错误队列	errback	fail	reject|rejectWith
取消	cancel	cancel	cancel
询问当前状态	state		state
并归结果	DeferredList	parallel	when

接下来，我们看一下 jQuery1.51 的源码。

```
(function(jQuery) {
   var
         promiseMethods = "then done fail isResolved isRejected promise".split(" "),
         //用于转换 arguments 对象为真正的数组
         sliceDeferred = [].slice;
   jQuery.extend({
      //这是最早期的实现，Deferred 还随大众那样是一个双链结构，只不过它的每一条链都由_Deferred 实现
      //_Deferred 方法会返回一个_Deferred 对象，它可以简单地理解为一个操作函数数组的特殊对象
      _Deferred: function() {
         var //函数列表
               callbacks = [],
               //这一个数组，用于保存传入的上下文对象与参数 [ context , args ]
               fired,
               // 判定是否在触发过程中
               firing,
               // 判定是否被消
               cancelled,
               //这是要返回的对象
               deferred = {
            done: function() {
               if (!cancelled) { //cancel 就不能再绑定了
                  //与 jQuery 其他 API 一样，允许太多种传参形式，因此这里需要各种处理
                  var args = arguments,
                        i,
                        length,
                        elem,
                        type,
                        _fired;
                  if (fired) {
                     _fired = fired;
                     fired = 0;
                  }
                  for (i = 0, length = args.length; i < length; i++) {
```

```
                elem = args[i];
                type = jQuery.type(elem);
                if (type === "array") { //递归调用自身
                    deferred.done.apply(deferred, elem);
                } else if (type === "function") {
                    callbacks.push(elem);
                }
            }
            if (_fired) { //如果已经触发了，那么就立即执行它们
                deferred.resolveWith(_fired[0], _fired[1]);
            }
        }
        return this;
    },
    //触发时允许指定上下文与传参
    resolveWith: function(context, args) {
        if (!cancelled && !fired && !firing) {
            args = args || [];
            firing = 1;
            try {
                while (callbacks[0]) {
                    //注意这里，每次都是弹出一个回调，因此此对象不能反复利用
                    callbacks.shift().apply(context, args);
                }
            } finally {
                fired = [context, args];
                firing = 0;
            }
        }
        return this;
    },
    //触发时允许指定传参
    resolve: function() {
        deferred.resolveWith(this, arguments);
        return this;
    },
    // 判定是否调用了 resolve 或 resolveWith
    isResolved: function() {
        return !!(firing || fired);
    },
    // 取消
    cancel: function() {
        cancelled = 1;
        callbacks = [];
        return this;
    }
};

return deferred;
},
// 真正给人用的 Deferred，一个双链对象，由两个_Deferred 对象合体而成
Deferred: function(func) {
    var deferred = jQuery._Deferred(),
        failDeferred = jQuery._Deferred(),
```

```
            promise;
        // Add errorDeferred methods, then and promise
        jQuery.extend(deferred, {
            then: function(doneCallbacks, failCallbacks) {
                deferred.done(doneCallbacks).fail(failCallbacks);
                return this;
            },
            fail: failDeferred.done,
            rejectWith: failDeferred.resolveWith,
            reject: failDeferred.resolve,
            isRejected: failDeferred.isResolved,
            // Get a promise for this deferred
            // If obj is provided, the promise aspect is added to the object
            promise: function(obj) {
                //这是一个单例方法，无论调用多少次，只会返回同一个对象
                //它与 promiseMethods 有着相同的方法
                if (obj == null) {
                    if (promise) {
                        return promise;
                    }
                    promise = obj = {};
                }
                var i = promiseMethods.length;
                while (i--) {
                    obj[promiseMethods[i]] = deferred[promiseMethods[i]];
                }
                return obj;
            }
        });
        //将这两个 cancel 放进队列目的是，一旦执行 fail 与 done 就不能用了
        deferred.done(failDeferred.cancel).fail(deferred.cancel);
        // 移除 cancel 方法
        delete deferred.cancel;
        if (func) {
            func.call(deferred, deferred);
        }
        return deferred;
    },
    when: function(firstParam) {
        var args = arguments,
                i = 0,
                length = args.length,
                count = length, //计数器
            //如果只是传入一个 Deferred/Promise 对象，那么就直接利用此对象，否则生成一个新的
            Deferred
                //此 Promise 对象（下称 "并归 Promise"）用于绑定下一个阶段的回调
                deferred = length <= 1 && firstParam && jQuery.isFunction(firstParam.
                promise)
                ? firstParam : jQuery.Deferred();

        function resolveFunc(i) {
            return function(value) {
                args[i] = arguments.length > 1 ? sliceDeferred.call(arguments, 0) : value;
                if (!(--count)) {
```

```
                    //如果 count 减为零，那么就将这些结果放到下一个阶段绑定的回调执行
                    deferred.resolveWith(deferred, sliceDeferred.call(args, 0));
                }
            };
        }
        if (length > 1) {
            for (; i < length; i++) {
                //只对 Deferred/Promise 对象进行操作，因为只有它们有 promise 方法
                if (args[i] && jQuery.isFunction(args[i].promise)) {
                    //绑定过渡用的 resolveFunc 方法，让它们决定“并归 Promise”的触发
                    args[i].promise().then(resolveFunc(i), deferred.reject);
                } else {
                    --count;
                }
            }
            if (!count) { //如果为零立即执行
                deferred.resolveWith(deferred, args);
            }
        } else if (deferred !== firstParam) {
            //如果什么也不传，也立即执行
            deferred.resolveWith(deferred, length ? [firstParam] : []);
        } //换言之，只传一个 Deferred/Promise 对象是不会执行的，只会转成 Promise 对象
        return deferred.promise();
    }
});

})(jQuery);
```

这是第一代 Deferred 模块，Deferred 完全为 Promise 准备，when 的参数都要求是一个 Promise 对象，而它本身也有返回 Promise 对象，从而实现链式调用。

在 jQuery1.6 中，jQuery Deferred 添加两个方法，一个是 always，一个是 pipe。

always 是用于添加回调，无论正常与出错都会执行。这很好设计，我们可以类比一下 XMLHttpRequest2.0，它的回调也整合成这三个东西：onload、onerror、onloadend。

pipe 就是管道，对回调使用瀑布模型，上一个回调的返回值供下一个回调使用，然后再吐出来给下下一个回调用。我们可以通过下面的例子区别 pipe 与 then 的差别。

```
var deferred = $.Deferred()
//注意，这里不能链起来
//因为 Deferred()返回的是 Deferred 对象
//pipe()返回的是 Promise 对象
var promise = deferred.pipe(function(a) {
    console.log(a); //5
    return a * 2;
}).pipe(function(a) {
    console.log(a); //10
    return a * 4;
}).pipe(function(a) {
    console.log(a); //40
})
console.log(deferred === promise); //false
deferred.resolve(5);
```

```
//这里则可以链起来，始终都是那个 Deferred 对象
$.Deferred()
       .then(function(a) {
    console.log(a); //10
    return a * 2;
}).then(function(a) {
    console.log(a); //10
    return a * 4;
}).then(function(a) {
    console.log(a); //10
}).resolve(10);
```

此时期， Deferred 开始在内部大规模应用了，包括 queue 模块、effects 模块、Ajax 模块。

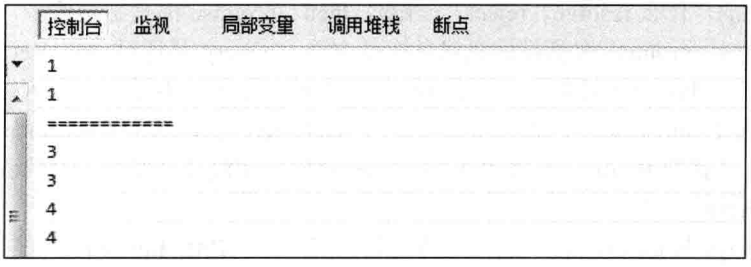
```
var stuff1 = function(deferred) {
    setTimeout(function() {
        console.log("Stuff #1 is done!");
        deferred.resolve();
    }, 1000);
};
var stuff2 = function(deferred) {
    setTimeout(function() {
        console.log("Stuff #2 is done!");
        deferred.resolve();
    }, 500);
};
var stuff3 = function(deferred) {
    setTimeout(function() {
        console.log("Stuff #3 is done!");
        deferred.resolve();
    }, 500);
};
$.when(
       $.Deferred(stuff1),
       $.Deferred(stuff2),
       $.Deferred(stuff3)
       ).then(function() {
    console.log("done!");
});
```

jQuery1.7 添加 callbacks 模块，里面的 Callbacks 函数其实就是用_Deferred 改改，进化而来，用它实现观察者模式非常简单。不用多言，Deferred 也得重新实现。这时，它是三链结构（doneList、failList 与 progressList ），为 progressList 的添加回调与触发回调新增了 3 个 API，progress、notify、notifyWith。那 progressList 是干什么用的呢？阅读文档，我们知道成功队列 doneList 与错误队列 failList 都是一次的，触发了一次就不能再触发，有时需要反复触发的情况，Callbacks 这个东西现在就能提供这个可能性。

```
var a = $.Deferred();
a.done(function(x) {
    console.log(x);
}).done(function(x) {
    console.log(x);
```

```
});
a.resolve(1);
a.resolve(2);
console.log("============")
var b = $.Deferred();
b.progress(function(y) {
    console.log(y);
}).progress(function(y) {
    console.log(y);
});
b.notify(3);
b.notify(4);
```

具体如图 12.9 所示。



▲图 12.9

jQuery1.8 又进行了大重构，思路是一致的，就是设计压缩一下代码，许多类似的方法改成循环生成。这时期它做了一种重要的决定，将 then 方法改为 pipe 的别名方法。而 pipe 是实现 Promise 规范中标准的 then 方法。显然这就是现在的兼容陷阱，只不过由于 jQuery Deferred 的不畅销，因此也没有把问题暴露出来罢了。

12.5 Promise/A 与 mmDeferred

Promise/A 隶属于 Promise 规范，而 Promise 规范则又隶属于 CommonJS。显然异步模型不是那么好设计的，因此才会出现这么多分类，而当中的每个子规范都相当松散。

目前最成功的 Promise/A 大致是这样的：一个带有 then 方法的对象，它拥有 3 个状态，pending、fulfilled、rejected。一开始是 pending，执行 then 方法后，当其回调被执行，会进入 fulfiled 或 rejected 状态。

then 方法可传入两个函数，一个是成功时执行，第一个是失败时执行，分别叫做 onFulfill、onReject。then 还有第 3 个参数叫做 onNotify，它不会改变对象的状态。这 3 个函数都是可选的，非函数时会忽略掉。

then 方法在添加了 onFufill 或 onReject 会返回一个新的 Promise 对象。这样一来，就能形成一个 Promise 链。显然，jQuery Deferred 并不满足后面的那个条件。

后来人们在 Promose/A 那粗糙的准则上添加更多的细节，形成了 Promise/A+，http://promises-aplus.github.io/promises-spec/，重点见 Requirements 与 Notes。

Notes 还着重提到，为了防止 Promise 链还没有形成就被用户触发回调，强烈要求使用 setTimeout、setImmediate、process.nextTick 等异步 API 来提供足够的构建时间。这思路其实与著名的 JSDeferred 如出一辙。

此外，大家在实现 Promise/A+时，渐渐达成一些共识，添加了 all、any 等方法，来并归结果或处理竞态状态。现在市面上有 3 大 Promise/A+库。

Q，RSVP，when。其中，Q 的微缩版被整进 angular.js，RSVP 被整进 ember.js，angular 和 ember 我后面会重点介绍，都是著名的 MVVM 库。微软出品的 WinJs MVVM 框架，也内置一个 Promise 模块。

jQuery Deferred 虽然不标准，但它的出现已表明 Deferred/Promise 这种异步范式的重要了。

现在让我们看看如何实现一个完全 Promise/A 规范的库。首先，要确认有什么 API。这个好办，直接看 Q 或 when.js，什么 resolve、reject、notify、then、promise 都是必不可少。

有争议的地方是 promise 究竟是做成对象还是方法，jQuery 是通过 promise(obj)这方式将一个普通的对象变成一个 Promise 对象，让 Promise 的种类更多元化。不过这个也可以使用 mix 实现。加之，Deferred 与 Promise 是一一对应关系，一个 Deferred 对象必对应 Promise 对象，延迟生成 Promise 对象与一早就把 Promise 对象附在 Deferred 对象其实没什么影响。最终我是决定使用后者——直接做成 promise 对象。

此外还有 always 与 ensure 方法之争。它们都是实现类似 XMLHttpRequest 的 onloadend 回调功能，用来做收尾工作。可详看下面这里的讨论。

https://github.com/cujojs/when/issues/103

一个主要论点是 ensure 更像 finally 语句：

```
try{
   doSomething(x, y)
}.catch(e){
   handleError(e)
}.finally{
   cleanup()
};
```

与

```
var fn = require('when/function');
fn.call(
       doSomethiny(x, y)
       ).otherwise(
       handleError(e)
       ).somethingThatIsNotTheExistingAlways(
       cleanup()
       );
```

顺着这思路，如何从语法层面做类似，就会发现一个很有意思的现象。其中，resolve 相当于正常的 call、apply 语句，reject 相当于 throw 语句，otherwise 相当于 catch 语句，ensure 相当于 finally 语句。

我们用 JSDeferred 修改一下，让它内部多出一个 promise 对象，专门用于收集回调，原来的

Deferred 仅用于触发回调，这样就减少其状态被改变的可能性。虽然这样也很难与 jQuery Deferred 那种以闭包手段实现 Deferred 工厂相媲美。

```
function Deferred() {
    return (this instanceof Deferred) ? this.init() : new Deferred()
}
Deferred.ok = function(x) {
    return x
};
Deferred.ng = function(x) {
    throw  x
};
Deferred.prototype = {
    init: function() {
        this.callback = {
            resolve: Deferred.ok,
            reject: Deferred.ng,
            notify: Deferred.ok,
            ensure: Deferred.ok
        };

        var that = this
        this.state = "pending"
        this.dirty = false;
        this.promise = {
            then: function(onResolve, onReject, onNotify) {
                return that._post(onResolve, onReject, onNotify)
            },
            otherwise: function(onReject) {
                return that._post(null, onReject, null)
            },
            ensure: function(onEnsure) {
                return that._post(0, 0, 0, onEnsure)
            },
            _next: null
        }
        return this;
    },
    _post: function(fn0, fn1, fn2, fn3) {
        var deferred
        if (!this.dirty) {
            deferred = this;
        } else {
            deferred = this.promise._next = new Deferred();
        }
        var index = -1, fns = arguments;
        "resolve,reject,notify, ensure".replace(/\w+/g, function(method) {
            var fn = fns[++index];
            if (typeof fn === "function") {
                deferred.callback[method] = fn;
                deferred.dirty = true;
            }
        })
        return deferred.promise;
```

```
    },
    _fire: function(method, value) {
        var next = "resolve";
        try {
            if (this.state == "pending" || method == "notify") {
                var fn = this.callback[method]
                value = fn.call(this, value);
                if (method !== "notify") {
                    this.state = method
                } else {
                    next = "notify"
                }
            }
        } catch (e) {
            next = "reject";
            value = e;
        }
        var ensure = this.callback.ensure
        if (Deferred.ok !== ensure) {
            try {
                ensure.call(this)
            } catch (e) {
                next = "reject";
                value = e;
            }
        }


        if (Deferred.isPromise(value)) {
            value._next = this.promise._next
        } else {
            if (this.promise._next) {
                this.promise._next._fire(next, value);
            }
        }
        return this;
    }
};
"resolve,reject,notify".replace(/\w+/g, function(method) {
    Deferred.prototype[method] = function(val) {
        //http://promisesaplus.com/ 4.1
        if (!this.dirty) {
            var that = this;
            setTimeout(function() {
                that._fire(method, val)
            }, 0)
        } else {
            return this._fire(method, val)
        }
    }
})
Deferred.isPromise = function(obj) {
    return !!(obj && typeof obj.then === "function");
};
```

```
function some(any, promises) {
    var deferred = Deferred(), n = 0, result = [], end
    function loop(promise, index) {
        promise.then(function(ret) {
            if (!end) {
                result[index] = ret//保证回调的顺序
                n++;
                if (any || n >= promises.length) {
                    deferred.resolve(any ? ret : result);
                    end = true
                }
            }
        }, function(e) {
            end = true
            deferred.reject(e);
        })
    }
    for (var i = 0, l = promises.length; i < l; i++) {
        loop(promises[i], i)
    }
    return deferred.promise;
}
Deferred.all = function() {
    return some(false, arguments)
}
Deferred.any = function() {
    return some(true, arguments)
};
Deferred.nextTick = function(fn) {
    setTimeout(fn, 0)
}
```

我们可以看到传统的类机制实现，会把一些内部方法以“_”前缀暴露出来。

我们用闭包方式进行封装吧。首先，Deferred 对象不是 new 出来的，是函数的一个对象，直接 return 出来。能影响 Deferred 状态的_post、_fire 做成内部函数，标识其状态的 state、 dirty 也做成内部变量。想改变状态，只能通过执行 reject、resolve 方法实现。onFulfill、onReject 函数会在用户传入时，外包一层，将 state 变量包进去，以此实现状态的改变！

```
if (method === "resolve" || method === "reject") {
    dfd.callback[method] = function() {//重写 onFulfill, onReject
        try {
            var value = fn.apply(this, arguments)
            state = "fulfilled"//偷偷改变状态
            return value
        } catch (err) {
            state = "rejected"
            return err
        }
    }
}
```

如果我们想检测对象的状态如何（指 pedding，fulfilled，rejected）或其默认回调是否被重写

(dirty，这与 Deferred 能否立即执行 resolve、reject 方法有关，因为 Deferred 要求先存在回调才能触发回调，虽然回调我们默认已经准备了一些，但还是希望有个真实的用户回调才让用户执行 resolve、reject 方法)，我们就调用相应的同名方法，.state()、.dirty()。这样对象的状态就不会被随意窜改了。

```
(function() {

    var noop = function() {
    }
    function Deferred(mixin) {
        var state = "pending", dirty = false
        function ok(x) {
            state = "fulfilled"
            return x
        }
        function ng(e) {
            state = "rejected"
            throw e
        }

        var dfd = {
            callback: {
                resolve: ok,
                reject: ng,
                notify: noop,
                ensure: noop
            },
            dirty: function() {
                return dirty
            },
            state: function() {
                return state
            },
            promise: {
                then: function(onResolve, onReject, onNotify) {
                    return _post(onResolve, onReject, onNotify)
                },
                otherwise: function(onReject) {
                    return _post(0, onReject)
                },
                ensure: function(onEnsure) {
                    return _post(0, 0, 0, onEnsure)
                },
                _next: null
            }
        }
        //允许传入一个可选的对象或函数，修改整条 Deferred 链的所有 Promise 对象
        //这相当于 jQuery 的 promise(obj)，将一个普通对象转换为 Promise 对象的功能
        if (typeof mixin === "function") {
            mixin(dfd.promise)
        } else if (mixin && typeof mixin === "object") {
            for (var i in mixin) {
                if (!dfd.promise[i]) {
```

```
            dfd.promise[i] = mixin[i]
        }
    }
}

"resolve,reject,notify".replace(/\w+/g, function(method) {
    dfd[method] = function() {
        var that = this, args = arguments
        //http://promisesaplus.com/ 4.1
        if (that.dirty()) {
            _fire.call(that, method, args)
        } else {
            Deferred.nextTick(function() {
                _fire.call(that, method, args)
            })
        }
    }
})
return dfd

function _post() {
    var index = -1, fns = arguments;
    "resolve,reject,notify,ensure".replace(/\w+/g, function(method) {
        var fn = fns[++index];
        if (typeof fn === "function") {
            dirty = true
            if (method === "resolve" || method === "reject") {
                dfd.callback[method] = function() {
                    try {
                        var value = fn.apply(this, arguments)
                        state = "fulfilled"
                        return value
                    } catch (err) {
                        state = "rejected"
                        return err
                    }
                }
            } else {
                dfd.callback[method] = fn;
            }
        }
    })
    var deferred = dfd.promise._next = Deferred(mixin)
    return deferred.promise;
}

function _fire(method, array) {
    var next = "resolve", value
    if (this.state() === "pending" || method === "notify") {
        var fn = this.callback[method]
        try {
            value = fn.apply(this, array);
        } catch (e) {//处理 notify 的异常
            value = e
        }
```

```
                if (this.state() === "rejected") {
                    next = "reject"
                } else if (method === "notify") {
                    next = "notify"
                }
                array = [value]
            }
            var ensure = this.callback.ensure
            if (noop !== ensure) {
                try {
                    ensure.call(this)//模拟 finally
                } catch (e) {
                    next = "reject";
                    array = [e];
                }
            }
            var nextDeferred = this.promise._next
            if (Deferred.isPromise(value)) {
                value._next = nextDeferred
            } else {
                if (nextDeferred) {
                    _fire.call(nextDeferred, next, array);
                }
            }
        }

    }
    window.Deferred = Deferred;
    //……下面就与原来的一模一样了
})()
```

有关此 Deferred 的完整实现可以访问以下地址，我将它做成一个项目，名为 mmDeferred：https://github.com/RubyLouvre/mmDeferred。

12.6 JavaScript 异步处理的前景

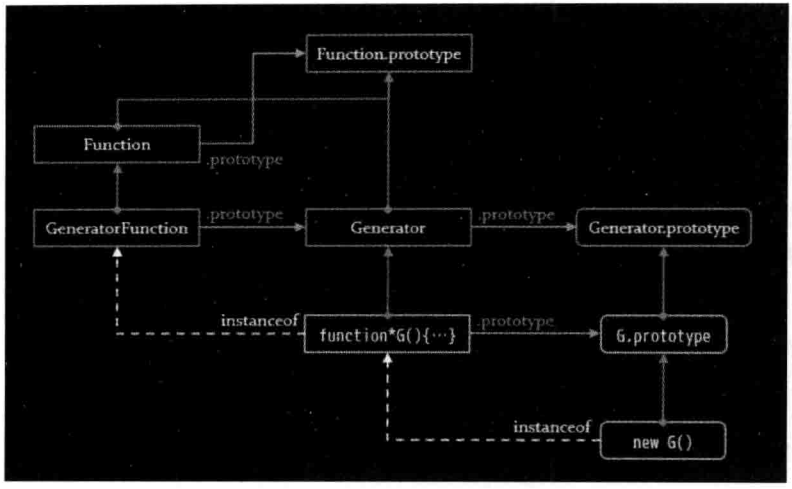
以后 JavaScript 将支持 yield 生成器（generator）。目前只有 Firefox 装备，但是要自己打开实验性开关。具体如下：

```
<script type="text/javascript; version=1.7"> // 这里改为 1.8 也行
// 然后就可以使用 yield 了
</script>
```

如果先前没有学过 Ruby、 Python 等语言，可以到下面地址了解一下生成器是什么东西，如图 12.10 所示。

https://developer.mozilla.org/en-US/docs/JavaScript/New_in_JavaScript/1.7?redirectlocale=en-US&redirectslug=New_in_JavaScript_1.7#Generators

https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Iterators_and_Generators



▲图 12.10

我们用其中一个例子做分析吧。

```
function fib() {
    var i = 0,
            j = 1;
    while (true) {
        yield i; // 每次在这里把 i 返回
        var t = i;
        i = j;
        j += t;
    }
}
var g = fib();
for (var i = 0; i < 10; i++) {
    console.log(g.next());
}
```

这个包含 yield 关键字的函数就是一个生成器。当我们调用时，它内部的代码是没有被执行的，它连同我们的传参将被 curry，返回生成器迭代器（generator-iterator ）。这是一个特殊的对象，拥有 send、next、end 3 个方法。然后在第一次调用它时，这些参数才起效，驱动程序运转，但它只运行到 yield 关键字就停下来，将 yield 后面的变量保存一下，作为结果返回。下次再调用 next 方法，程序不是从头执行，而在 yield 那个地方开始，参数为上次保存的那个变量，若再遇到 yield，重复刚才的工作。

如果有疑惑，可以再看个简单的例子。

```
function f() {
    console.log(1);
    var value = yield 2;
    console.log(value);
    console.log("这是跟在 3 后面")
}
```

```
var g = f(); // 这里什么也不行
console.log(g.next()); //这里打迎 f 函数以 yield 切开的前半部分,及其返回值。于是有 1，2
g.send(3);  //这里打印 3，  与我们最后那一行提示
```

从上例我们也可看到 send 的用途，用于覆写生成器暂时保存起来的变量的值。

我们也看到上例最后竟然会抛错了，一个 StopIteration 异常，因此我们得 try catch 一下，并用 close 方法关闭它。

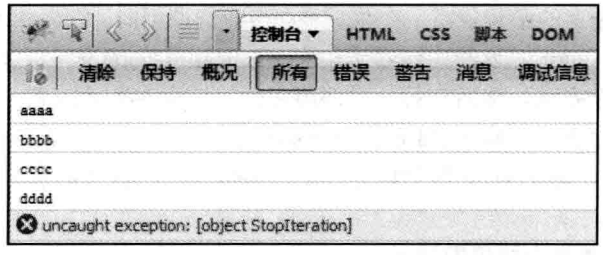
一个函数里面允许存在多个 yield 关键字，将程序拆成 n+1 块，相对应的，需要执行 next 方法 n+1 次。

```
function Generator() {
    yield "aaaa";
    yield "bbbb";
    yield "cccc";
    yield "dddd";
}

var g = Generator();
console.log(g.next());//aaaa
console.log(g.next());//bbbb
console.log(g.next());//cccc
console.log(g.next());//dddd
```

但这样也太多 next 了，我们让它变得智能化些，让 console.log 这个逻辑抽取成一个函数，然后递归自身。

```
var g = Generator();
function process(g) {
    console.log(g.next())
    process(g)
}
process(g)
```

具体如图 12.11 所示。



▲图 12.11

最后一行报错，我们需要 try...catch 一下，并抽取处理逻辑。

```
var g = Generator();
function process(g, f) {
    try {
        f(g.next())
```

```
        process(g, f)
    } catch (e) {
    }
}
process(g, function(a) {
    console.log(a)
})
```

我们再将它镶入一些异步代码，如 setTimeout，就会看到一些美妙的效果。

```
function Generator(next) {
    setTimeout(next, 1000)
    yield;
    console.log("aaaa")

    setTimeout(next, 1000)
    yield;
    console.log("bbbb")

    setTimeout(next, 1000)
    yield;
    console.log("cccc")
}

function process(f) {
    var g
    //这是传参 next，next 里面再次驱动它执行生成器的 next，形成递归调用
    g = f(function() {
        try {
            g.next();
        } catch (e) {
        }
    });
    g.next(); //执行它
}

process(Generator);
```

可以看到 aaaa、bbbb、cccc 都是每隔 1 秒才打印出来。原本异步的代码，被整得同步一样，阻塞在那里，只有前一块执行完才轮到下一块。如果没有 yield，我们就不得不套嵌它们。

```
setTimeout(function() {
    console.log("aaaa")
    setTimeout(function() {
        console.log("bbbb")
        setTimeout(function() {
            console.log("cccc")
        }, 1000)
    }, 1000)
}, 1000)
```

再来个明显的例子。

```
Function.prototype.wait = function() {
    var me = this;
    var g = me(function(t) {
```

```
        try {
            g.send(t)
        } catch (e) {
        }
    });
    g.next();
}
Requester = function(a) {
    this.resume = a;
};

Requester.prototype.send = function(time) {
    var resume = this.resume
    setTimeout(function() {
        resume(time)
    }, time)
};


(function(resume) {

    var r = new Requester(resume);
    //这里隔 1000 秒才执行
    var json = yield r.send(1000);
     console.log(json);
    //这里隔 1000 秒才执行
    json = yield  r.send(1000);
     console.log(json);

}).wait()
```

通过这程序应该明白，有了 yield，我们就可以轻松以同步的形式写异步的代码，只要将它们放到某个函数体内。基于这思路，有几个库被开发出来，如 deferred-generator[①]、taskjs[②]、gens[③]、co[④]、tamejs[⑤]、windjs[⑥]。前四个是基于原生的 yield，后两个基于代码预编译。

大家可以看一下 deferred-generator 给出的例子，实现异步出乎意外的简洁。而从性能角度来看，它也一点也不逊色。在《Analysis of generators and other async patterns in node》[⑦]一文，对各种不同实现的异步库进行性能评测。基于原生生成器的库总体上占上风，当然这也一个废话，天生丽质与整容是不一样。由于本来就是语言自身提供的东西，yield 语句比回调更能及时地工作，耗时就更少。但在目前来看，想实现生成器的代码量非常大，像 windjs 那样需要引入一个编译器，相当于把用户代码回炉再造。而实现 Promise 则相对轻松些，是当下最廉价的异步方案[⑧]。

负责 ecma262 制订的 TC39，对于本章提到两种方案 Promise 与生成器其实不置可否，各有各的优势，不能取舍，干脆都实现了。生成器在 ecma262v6 中实现，而 Promise 也打算从 ecma262v7 搬到 ecma262v6。

① https://github.com/fujidig/deferred-generator
② http://taskjs.org/
③ https://github.com/Raynos/gens
④ https://github.com/visionmedia/co
⑤ https://github.com/maxtaco/tamejs
⑥ http://windjs.org/cn/
⑦ http://spion.github.io/posts/analysis-generators-and-other-async-patterns-node.html
⑧ http://www.cnblogs.com/rubylouvre/p/3292745.html

第13章 数据交互模块

前端与后端进行交互的方式有许多种，比如直接通过地址栏请求页面，script 节点加载脚本，以及 XMLHttpRequest 对象进行更可控的数据加载。尤其是 XMLHttpRequest 对象，它带来的 Ajax 无缝刷新，直接焕发 JavaScript 的“第 2 春”，让前端向富应用发展。在许多框架与类库中，作者都是把基于 script 节点的 JSONP 作为 Ajax 的补充，放在一起的。本章也不例外，还会介绍一下基于 iframe 的文件上传。

13.1 Ajax 概览

```
var xhr = new (self.XMLHttpRequest || ActiveXObject)("Microsoft.XMLHTTP")
xhr.onreadystatechange = function() {//先绑定事件后 open
    if (this.readyState === 4 && this.status === 200) {
        var div = document.createElement("div");
        div.innerHTML = this.responseText;
        document.body.appendChild(div)
    }
}
xhr.open("POST", "/ajax", true);
//必须,用于让服务器端判断 request 是 Ajax 请求(异步)还是传统请求(同步)
xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest")
xhr.send("key=val&key1=val2");
```

这是一个完整的 Ajax 程序，包括跨平台取得 XMLHttpRequest 对象，绑定事件回调，判定处理状态，发出请求，设置首部，以及在 POST 请求时，通过 send 方法发送数据。上面七个步骤每一步都有兼容性问题或易用性处理。如果是跨域请求，IE8 可能为 XDomainRequest 更为方便。

13.2 优雅地取得 XMLHttpRequest 对象

Ajax 的核心就是那个 XMLhttpRequest 对象，以前，人们还花大精力用 iframe 来模拟它，但现在可以不管了。在 IE5 时，微软用一个 ActiveXObject 对象来加载数据，并且在数据返回不会导致地址栏跳转和页面刷新。除此以外，微软的 ActiveXObject 还可以做许多事情，比如创建一个 HTML 页面或 XML 文档，解析 XSLT，外挂 Windows 的日历，拖动条什么，因此要知道这个 ActiveXObject 是干什么用，必须传参。像 XMLhttpRequest 这样重要的对象，在其发展过程中，一直面临外界的

剧烈竞争，因此它也不断升级，这些版本号与名字连在一起，组成参数传入 ActiveXObject 才能正确生成 XMLhttpRequest。但它们之间没有什么规则，Msxml2.XMLHTTP.6.0，Msxml2.XMLHTTP.5.0，Msxml2.XMLHTTP.4.0，Msxml2.XMLHTTP.3.0，Msxml2.XMLHTTP，Microsoft.XMLHTTP，逐个试验（这取决于用户有没有打补丁与操作系统自带的版本）。

毋庸置疑，越新的版本功能越多，因此我们在试验时，把最新的版本放在前面。可能微软最后也发觉这用户体验太差了，也学标准浏览器那样，直接提供一个 XMLHttpRequest 对象给你 new，然后内部总是对应当前可用的最新版本。

IE7 这个决定是明智的，但 IE7 在 IE 的发展史，是个半成品，这个 XMLHttpRequest 对象也一样不好用。它不支持本地 file 协议，会出现拒绝访问，需要倒退到 ActiveXObject 对象。如果服务器不发送任何 header 来禁止浏览器进行缓存的话，IE7 的 XMLHttpRequest 对象可能会缓存和重用对 GET 请求的响应。使用以下的解决方式仍然会出现该问题：使用 POST 方法、使用随机的查询串（加时间戳等）、配置并使服务器发送某些缓存的指令。注意：单独使用一个非随机的查询串并不能阻止浏览器进行缓存。此外，IE7 的 XMLHTTPRequest 对象与标准浏览器的出入还是很大，人家有 prototype，有 onbort、onload、onerror 方法。它还不是一个 JavaScript 对象，是 MSXML2.XMLHTTP.3.0 的外壳。

如果我们查看一下各大类库的源码，发现它们并没有全部逐一试验。

```
//**********************jQuery1.4a2*****************
xhr: function(){
  return window.ActiveXObject ?
    new ActiveXObject("Microsoft.XMLHTTP") :
    new XMLHttpRequest();
},
//**********************mootools1.2.4*****************
Browser.Request = function(){
  return $try(function(){
    return new XMLHttpRequest();
  }, function(){
    return new ActiveXObject('MSXML2.XMLHTTP');
  }, function(){
    return new ActiveXObject('Microsoft.XMLHTTP');
  });
};
//**********************prototype1.61rc2*****************
getTransport: function() {
  return Try.these(
  function() {return new XMLHttpRequest()},
  function() {return new ActiveXObject('Msxml2.XMLHTTP')},
  function() {return new ActiveXObject('Microsoft.XMLHTTP')}
) || false;
},
```

Microsoft.XMLHTTP 是最早的版本，Msxml2.XMLHTTP 是 IE6 的，但打了补丁后自然有 Msxml2.XMLHTTP.3.0 和 Msxml2.XMLHTTP.4.0，Msxml2.XMLHTTP.5.0 不是给浏览器使用的，属于旁支，存在一定的兼容问题。其中 5.0 是为 office 所开发的，甚至带有一些特性是后来的 6.0 所没有的（如 xml 数字加密）。

MSXML 4.0 is a separate download that was released by Microsoft in October 2001. The latest or current service pack release of MSXML 4.0 is available through the Microsoft Web site. MSXML 4.0 must be installed separately and is not currently included with other Microsoft products. MSXML 4.0 installs side-by-side with earlier versions of MSXML without affecting any existing functionality.

MSXML 5.0 for Microsoft Office Applications is only available with current versions of Microsoft Office. MSXML 5.0 for Microsoft Office Applications installs side-by-side with earlier versions of MSXML without affecting any existing functionality.

Msxml2.XMLHTTP.6.0 是已知 ActiveXObject 系的 XMLHttpRequest 对象的最高版本。可能大家在网上还看到 Msxml2.XMLHTTP.7.0，很遗憾，那只是以讹传讹，我用 IE8 测试并不存在这个版本。

根据 IE blog 的建议，应该仅使用 6.0 版本和 3.0 版本，不要使用旧的 microsoft.xmlhttp，那它也应该出现在上面的数组中。此外还有一个 MSXML2.XMLHTTP.2.6，但不知为什么，总之，如果你的浏览器打了某些升级补丁，new ActiveXObject("Msxml2.XMLHTTP") 调用的是 2.6 版本或 3.0 版本，非常混乱。此后的版本，才正确对应它的版本号。因此上面没有列举 Msxml2.XMLHTTP.2.6 与 Msxml2. XMLHTTP.3.0 的必要，都给 Msxml2.XMLHTTP 代表了。

因此我们要检测的 ActiveXObject 的 ProgID 收窄为 Msxml2.XMLHTTP.6.0、Msxml2. XMLHTTP.3.0、Msxml2.XMLHTTP 与 Microsoft.XMLHTTP 四个。

```
function xhr() {
  if(!xhr.cache){
    var fns = [
      function () { return new XMLHttpRequest(); },
      function () { return new ActiveXObject('Msxml2.XMLHTTP'); },
      function () { return new ActiveXObject('Microsoft.XMLHTTP'); },
    ];
    for (var i = 0,n=fns.length; i < n; i++) {
      try {
        fns[i]();
        xhr.cache = fns[i];
        break;
      }catch(e){}
    }
    return xhr.cache();
  }else{
    return xhr.cache();
  }
}
var xhrObject = xhr();//调用
alert(xhrObject) //[object XMLHttpRequest]
```

凭心而论，上面的方法已经很高效了，只判定了一次，然后缓存生成方式。但有没有更优雅的设计呢？！

因为即便我们缓存了生成方式，每次还要判断一下 xhr.cache 的值。这时用惰性函数处理。这是一种覆写自身的模式。

```
var xhr =  function() {
  var fns = [
    function () { return new XMLHttpRequest(); },
    function () { return new ActiveXObject('Msxml2.XMLHTTP'); },
    function () { return new ActiveXObject('Microsoft.XMLHTTP'); },
  ];
  for (var i = 0,n=fns.length; i < n; i++) {
    try {
      fns[i]();
      xhr = fns[i];//注意这里，覆写自身
      break;
    }catch(e){}
  }
  return xhr()
}
```

我们再认真思考一下，既然我们是写框架，那么这些检测其实是放在 IIFE 里面，因此基本不用覆写，检测好哪个可用，就把它加到命名空间上就好了。最后的版本就出来了，使用 new Function、eval，反正只用一次，耗不了多少性能。

```
window.$ = window.$ = {}
var s = ["XMLHttpRequest", "ActiveXObject('Msxml2.XMLHTTP.6.0')",
    "ActiveXObject('Msxml2.XMLHTTP.3.0')", "ActiveXObject('Msxml2.XMLHTTP')"];
if (!"1"[0]) {//判定 IE67
    s[0] = location.protocol === "file:" ? "!" : s[0];
}
for (var i = 0, axo; axo = s[i++]; ) {
    try {
        if (eval("new " + axo)) {
            $.xhr = new Function("return new " + axo);
            break;
        }
    } catch (e) {
    }
}
```

13.3 XMLHttpRequest 对象的事件绑定与状态维护

最早的 XMLHTTPRequest 对象只有 onreadystatechange 方法，然后标准浏览器抄来，在易用性上进行增强。首先将 XMLHTTPRequest 改造成一个事件派发者（EventDispatcher）。很早以前，浏览器只有三种原生的事件派发者，window 对象、文档对象与元素节点。既然是事件派发者，就有 addEventListener、removeEventListener、dispatchEvent 等多投事件 API，IE 到 9.0 才支持 W3C 那套，因此它的 XMLHTTPRequest 也在 IE9 才有 addEventListener 这些 API。

早些年，W3C 的大旗一直是 Firefox 在扛，许多 API 都是 Firefox 搞出来的，在漫长的 Firefox3.X 时代，onload、onerrot、onabort、onprogress 这 4 个方法被首先搞出来。其他标准浏览器只是跟进。微软在 IE8 才为 XDomainRequest 添加 onerror、onload、onprogress、ontimeout 这几个事件，或许觉得过期事件非常有用，IE8 为 XMLHTTPRequest 多加个 ontimeout 事件。IE9 又加个 onabort 事件。

Firefox 在 3.6 开始支持用 XMLHttpRequest 上传二进制文件，为此搞出 FileReader 对象，此对

象所支持的事件类型与后来的 XMLHTTPRequest2 的一模一样（onabort、onerror、onload、onloadend、onloadstart、onprogress）。其中 onloadend 与 jQuery ajax 的 complete 回调一模一样，无论成功与错误都会触发，用于收尾工作非常适合。IE10 终于把这些事件一口气实现了，如表 13.1 所示。

表 13.1

事　件	描　述
loadstart	在请求开始时触发
progress	在请求发送或接收数据期间，在服务器指定的时间间隔触发
abort	在请求被取消时触发，例如，在调用 abort() 方法时
error	在请求失败时触发
load	在请求成功完成时触发
timeout	在作者指定的时间段已经结束时触发
loadend	在请求完成时触发，无论请求是成功还是失败
readystatechange	在 XMLHTTPRequest 对象的 readyState 值发生改变时触发

从实现角度来看，由于 IE6～IE8 的 XMLHTTPRequest 无法进展原型，我们需要用包裹的方式创建一个伪 XMLHTTPRequest 对象，在它里面操作原生对象。对于事件绑定，为了对同一种事件绑定多个回调，我们需要继承一个自定义事件对象，换言之，一个观察者模式的东西。loadstart 就是在一开头执行，没什么难度，成功与失败我们可以判定 status 状态码，ontimeout 可以用 setTimeout 实现，onabort 就是一个开关，loadend 就是在回调里肯定会执行的方法。最麻烦是 onprogress，在标准浏览器中我们可以通过事件对象的 loaded 与 total 属性轻易计算得进度，IE 在 readyState==3 时 Content-Length 的值并不可靠。

至于请求是成功还是失败，IE 就要在 readystatechange 回调中查看 status 值与转换目标类型是否成功。

2xx 状态与表示从缓存中直接取出的 304 可以看是成功，但浏览器还是有一些例外情况需要我们注意。IE（非原生的 XHR 对象）中会将 204 设置为 1223，Opera 会在取得 204 时将 status 设置为 0，而 Safari 3 之前的版本会将 status 设置为 undefined。最终验证请求是否成功的代码将会是：

```
( xhr.status >= 200 && xhr.status < 300 ) ||
xhr.status === 304 || xhr.status === 1223 || xhr.status === 0
```

IE 的 1223 请求算是一个著名的 BUG，在各大类库的 bugstack 中都有介绍。

XMLHTTPRequest implementation in MSXML HTTP (at least in IE 8.0 on Windows XP SP3+) does not handle HTTP responses with status code 204 (No Content) properly; the `status' property has the value 1223.

dojo - http://trac.dojotoolkit.org/ticket/2418

prototype - https://prototype.lighthouseapp.com/projects/8886/tickets/129-ie-mangles-http-response-status-code-204-to-1223

YUI - http://developer.yahoo.com/yui/docs/connection.js.html (handleTransactionResponse)

JQuery - http://bugs.jquery.com/ticket/1450

ExtJS - http://www.sencha.com/forum/showthread.php?85908-FIXED-732-Ext-doesn-t-normalize-IE-s-crazy-HTTP-status-code-1223

IE 下甚至会返回五位数的状态码，这是 WinInet 错误代码：

http://support.microsoft.com/kb/193625

另外，Firefox 在本地使用 XMLHttpRequest 时，成功时 status 为 0。由于很少在本地发请求，因此主流框架没有对它加以处理。这个当作常识，自己留意一下吧。

13.4 发送请求与数据

XMLHTTPRequest 对象发送请求是使用 open 方法，在这之前请先绑定好各种事件回调。

语法如下：

```
open(method, url, async, username, password)
```

method 参数是用于请求的 HTTP 方法。值包括 GET、POST、PUT、DELETE 和 HEAD。有的浏览器还允许你自定义 method[①]，不过要求全是大写。比如 IE6、Firefox3～Firefox19、Chrome、Opera。

url 参数是请求的主体。大多数浏览器实施了一个同源安全策略，并且要求这个 URL 与包含脚本的文本具有相同的主机名和端口。在 GET 请求，我们需要将参数转换成 querystring 的形式放在问号后面。

async 参数指示请求使用应该异步地执行。如果这个参数是 false，请求是同步的，后续对 send() 的调用将阻塞，直到响应完全接收。如果这个参数是 true 或省略，请求是异步的，且通常需要一个 onreadystatechange 事件句柄。

username 和 password 参数是可选的，为 url 所需的授权提供认证资格。如果指定了，它们会覆盖 url 自己指定的任何资格。

发送数据要用 send 方法，网上通常教我们在 POST 请求时发送 querystring。后来增加了 FormData、ArrayBuffer、Blob、Document 这几种数据类型。

FormData 是一个不透明的对象，无法序列化，但能简化人工提交数据的过程。以前，我们点击按钮提交表单，浏览器会自动将这个表单的所有 disabled 为 false 的 input、texteara、select、button 元素的 name 与 value 抽取出来，变成一个 querystring。当我们用 Ajax 提交时，这个过程就成为人工的。jQuery 把它抽象成一个 serialize 方法，代码量不是少数。而 FormData 直接可以 new 一个实例出来，我们只需遍历表单元素，用 append 方法传入其 name 与 value 就行了。

```
var formdata = new FormData();
formdata.append("name", "司徒正美");
formdata.append("blog", "http://www.cnblogs.com/rubylouvre/");
```

更简洁的方式，它本来就可以用 getFormData 生成，并得到此表单的所有数据：

① http://www.mnot.net/javascript/xmlhttprequest/

```
var formobj =  document.getElementById("form");
var formdata = formobj.getFormData()
```

它也可以用传参方法填充内容：

```
var formobj =  document.getElementById("form");
var formdata = new FormData(formobj);
```

最后就是提交：

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "http://ajaxpath");
xhr.send(formData);
```

如果是 document，就自己生成一个 XML 对象发上去吧，但我们没有好的手段辨识浏览器是否支持 send(document)。对于其他新数据，一般来说，只要它们的构造器是出现在全局作用域下，浏览器已经同步好 send 方法了。

在标准浏览器支持二进制的过程中，无节操地实现了各种各样的对象。有的只是昙花一现，很快被废弃掉，如 BlobBuilder。剩下的还有 Blob、File、FileReader、FileWriter、BlobURL 及庞大的 TypedArray 家族。

下面是发送 ArrayBuffer 与 Blob 的例子。

```
var myArray = new ArrayBuffer(512);
var longInt8View = new Uint8Array(myArray);
for (var i = 0; i < longInt8View.length; i++) {
   longInt8View[i] = i % 255;
}
var xhr = new XMLHttpRequest;
xhr.open("POST", url, false);
xhr.send(myArray);
var xhr = new XMLHttpRequest();
xhr.open("POST", url, true);
var blob = new Blob(['abc123'], {type: 'text/plain'});
xhr.send(blob);
```

Firefox 很早以前还实现了一个私有的 sendAsBinary ，可直接发送二进制数据。Chrome 可以用以下方法模拟。

```
//http://objectprog.blogspot.com/2010/11/sendasbinary-for-google-chrome.html
XMLHttpRequest.prototype.sendAsBinary = function(datastr) {
   var bb = new WebKitBlobBuilder();
   var data = new ArrayBuffer(1);
   var ui8a = new Uint8Array(data, 0);
   for (var i in datastr) {
      if (datastr.hasOwnProperty(i)) {
         var chr = datastr[i];
         var charcode = chr.charCodeAt(0)
         var lowbyte = (charcode & 0xff)
         ui8a[0] = lowbyte;
         bb.append(data);
      }
   }
```

```
    var blob = bb.getBlob();
    this.send(blob);
}
```

13.5 接收数据

早期 XMLHTTPRequest 对象拥有两种接收数据的属性，responseText 对应解码后的字符串（默认解码为 utf-8），responseXML 对应一个 XML 文档，IE 还支持第三种，responseBody 对应未解码的二进制数据。JSON 传输格式兴起后，我们会对 responseText 进行加工，用 JSON.parse 得到 JSON 数据。至于后端返回什么类型的数据，在项目开发过程，这个有对应文档看，如果比较悲催，我们还可以通过 getResponseHeader(“Content-Type”)得知。

随着浏览器着手对二进制的支持，它新增的 responseType 和 response 属性，告知浏览器我们希望返回什么格式的数据。

responseType，在发送请求前，根据您的数据需要，将 xhr.responseType 设置为“text”、“arraybuffer”、“blob”或“document”。请注意，设置（或忽略）xhr.responseType =，会默认将响应设为“text”。

response 成功发送请求后，xhr 的响应属性会包含 DOMString、ArrayBuffer、Blob 或 Document 形式（具体取决于 responseType 的设置）的请求数据。

```
//适用范围：Chrome8～Chrome24、Firefox6～Firefox18、IE10
var BlobBuilder = window.MozBlobBuilder || window.WebKitBlobBuilder || window.MSBlob
Builder || window.BlobBuilder
if (!BlobBuilder) {
    console.log("BlobBuilder 已被废弃")
}
var xhr = new XMLHttpRequest();
var img = document.getElementById("img")
xhr.open('POST', 'image.jpg', true);
xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest");
xhr.responseType = 'arraybuffer';
xhr.onload = function(e) {
    if (this.status === 200) {
        var bb = new BlobBuilder();
        bb.append(this.response);
        var blob = bb.getBlob('image/jpeg');
        img.src = blob;
    }
};
xhr.send();

//适用范围：Chrome19+、Firefox6、IE10、Opera12、Safari5
var xhr = new XMLHttpRequest();
var img = document.getElementById("img");
xhr.open('POST', 'image.jpg', true);
xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest");
xhr.responseType = 'blob';
//https://developer.mozilla.org/zh-CN/docs/DOM/window.URL.createObjectURL
window.URL = window.URL || window.webkitURL;
```

```
img.addEventListener("DOMNodeRemoved", function() {
    window.URL.revokeObjectURL(img.src);
});
xhr.onload = function(e) {
    if (this.status === 200) {
        var blob = this.response;
        img.src = window.URL.createObjectURL(blob);
    }
}
xhr.send();

//适用范围：Chrome10+、Firefox6、IE10、Opera11.60
var xhr = new XMLHttpRequest();
xhr.open('POST', '/path/to/image.jpg', true);
xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest");
xhr.responseType = 'arraybuffer';
xhr.onload = function(e) {
    if (this.status == 200) {
        var uInt8Array = new Uint8Array(this.response);
        var i = uInt8Array.length;
        var binaryString = new Array(i);
        while (i--) {
            binaryString[i] = String.fromCharCode(uInt8Array[i]);
            }
        var data = binaryString.join('');
        var base64 = window.btoa(data);
        document.getElementById("img").src = "" + base64;
        }
    };
xhr.send();

//适用范围：支持 overrideMimeType 与 btoa 的浏览器
var xhr = new XMLHttpRequest();
xhr.open('POST', 'image.jpg', true);
xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest");
//重写了默认的 MIME 类型，强制浏览器将该响应当成纯文本文件来对待，使用一个用户自定义的字符集
//这样就是告诉了浏览器，不要去解析数据,直接返回未处理过的字节码
xhr.overrideMimeType('text/plain; charset=x-user-defined')
xhr.onload = function(e) {
    if (this.status == 200) {
        var responseText = xhr.responseText;
        var responseTextLen = responseText.length;
        var binary = ''
        for (var i = 0; i < responseTextLen; i += 1) {
            //扔掉的高位字节(f7)
            binary += String.fromCharCode(responseText.charCodeAt(i) & 0xff);
        }
        var base64 = window.btoa(binary);
        document.getElementById("img").src = "" + base64;
    }
};
xhr.send();
```

btoa 其实就是一个 encodeBase64 方法。如果浏览器不支持，可以自己实现一个，让适用范围更广。

```
(function() {
var chars =
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'.split('');
   window.bota = window.btoa || function(str) {
      if (/[^\u0000-\u00ff]/.test(str)) {
         throw new Error('encodeBase64 : INVALID_CHARACTER_ERR');
      }
      var hash = chars;
      var i = 2;
      var len = str.length;
      var output = [];
      var link;
      var mod = len % 3;
      len = len - mod;
      for (; i < len; i += 3) {
         output.push(
               hash[(link = str.charCodeAt(i - 2)) >> 2],
               hash[((link & 3) << 4 | ((link = str.charCodeAt(i - 1)) >> 4))],
               hash[((link & 15) << 2 | ((link = str.charCodeAt(i)) >> 6))],
               hash[(link & 63)]
               );
      }
      if (mod) {
         output.push(
               hash[(link = str.charCodeAt(i - 2)) >> 2],
               hash[((link & 3) << 4 | ((link = str.charCodeAt(i - 1)) >> 4))],
               link ? hash[(link & 15) << 2] + '=' : '=='
               );
      }
      return output.join('');
   }
})()
```

下面讲解如何对付旧版本 IE。IE 支持两种脚本语言，JavaScript 与 VBScript。JavaScript 不能做的事，VBScript 可以殿后。整个思路是参照方式四，搞一个类似的 Uint8Array 的东西出来。

你可以使用以下两种方式创建一个 VBS 的转换函数。

```
(function() {
   function vbs() {
      /*
Function BinaryToArray(binary)
   Dim length,array()
   length = LenB(binary) - 1
   ReDim array(length)
   For i = 0 To length
      array(i) = AscB(MidB(binary, i + 1, 1))
   Next
   BinaryToArray = array
End Function
      */
   }
```

```
    var str = vbs.toString();
    execScript(str.slice(str.indexOf("*") + 1, str.lastIndexOf("*")), "VBScript");
})();

(function() {
    var str =
           'Function BinaryToArray(binary)\r\n\
               Dim oDic\r\n\
               Set oDic = CreateObject("scripting.dictionary")\r\n\
               length = LenB(binary) - 1\r\n\
               For i = 1 To length\r\n\
                  oDic.add i, AscB(MidB(binary, i, 1))\r\n\
               Next\r\n\
               BinaryToArray = oDic.Items\r\n\
            End Function'
    execScript(str, "VBScript");
})();
```

第一种是使用多行注释实现 heredocument，但不防压缩。第二种为了保持格式漂亮加入了许多空白，注意不能去掉后面的换行符，否则解释错误。

通过 BinaryToArray 方法得到一个数字数组，但它是一个 VBScript 对象，我们需要把它放到 VBArray 构造器内，转换为真正的 JavaScript 数组。剩下的处理就是一样的了。

```
var xhr = new XMLHttpRequest();
xhr.open('POST', 'image.jpg', true);
xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest");
xhr.onload = function(e) {
    if (this.status === 200) {
        var byteArray = new VBArray(BinaryToArray(this.responseBody)).toArray();
        var n = byteArray.length;
        var binary = '';
        for (var i = 0; i < n; i++) {
            //扔掉的高位字节(f7)
            binary += String.fromCharCode(byteArray[i] & 0xff);
        }
        var base64 = window.btoa(binary);
        document.getElementById("img").src = "" + base64;
    }
};
xhr.send();
```

13.6 上传文件

这个与前面提到的传送数据有点相似，不过它与 input[type=file]结合得更紧密。

假设页面上有一个 ID 为 upload 的上传域与一个 ID 为 progress 用于显示进度的 SPAN 元素，那么使用 XMLHTTPRequest2 来上传文件是这样实现的：

```
window.addEventListener("load", function() {
    var el = document.querySelector('#file');
    var progress = document.querySelector('#progress');
    el.addEventListener('change', function() {
```

```
        var file = this.files[0];
        if (file) {
            var xhr = new XMLHttpRequest();
            xhr.upload.addEventListener('progress', function(e) {
                //处理兼并性问题，不同版本其名字不一样
                var done = e.position || e.loaded, total = e.totalSize || e.total;
                progress.innerHTML = (Math.floor(done / total * 1000) / 10) + "%";
            });
            xhr.addEventListener('load', function() {
                progress.innerHTML = "上传成功";
            });
            xhr.open('PUT', '/upload', true);
            xhr.setRequestHeader('X-Requested-With', 'XMLHttpRequest');
            xhr.setRequestHeader('X-File-Name', encodeURIComponent(file.fileName || file.name));
            xhr.setRequestHeader('Content-Type', 'application/octet-stream');
            xhr.send(file);
        }
    })
});
```

如果文件很大，我们利用文件对象的 slice 方法切割一下，分块上传。

```
window.addEventListener("load", function() {
    var el = document.querySelector('#file');
    var progressBar = document.querySelector('#progress');
    function upload(name, index, file, total) {
        var xhr = new XMLHttpRequest();
        xhr.addEventListener('load', function() {
            progressBar.innerHTML = "100% 上传成功";
        });
        xhr.open('PUT', '/upload', true);
        xhr.setRequestHeader('X-Requested-With', 'XMLHttpRequest');
        xhr.setRequestHeader('X-File-Index', index);
        xhr.setRequestHeader('X-File-total', total);
        xhr.setRequestHeader('X-File-Name', encodeURIComponent(name));
        xhr.setRequestHeader('Content-Type', 'application/octet-stream');
        xhr.send(file);
    }
    el.addEventListener('change', function() {
        var blob = this.files[0];
        var meanSize = 512 * 1024; // 1MB chunk sizes.
        var totleSize = blob.size;
        var start = 0, end = 0;
        var i = 0;
        var name = blob.fileName || blob.name;
        var total = Math.ceil(totleSize / meanSize)
        while (start < totleSize) {
            if ('mozSlice' in blob) {
                var chunk = blob.mozSlice(start, end);
            } else if ("webkitSlice" in blob) {
                chunk = blob.webkitSlice(start, end);
            } else {
                chunk = blob.slice(start, end);
            }
            upload(name, i, chunk, total);
```

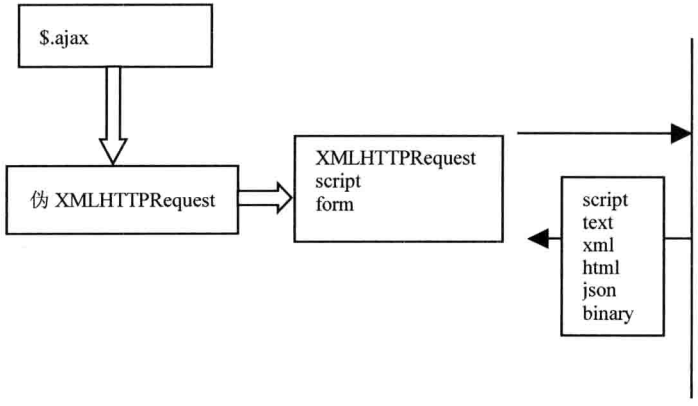
```
            i++
            start = end;
            end = start + meanSize;
        }
    });
})
```

我们再来看如何在旧版本 IE 中上传文件。由于我们无法序列化二进制文件，只好通过原始的 form 提交方法实现，这时需要将 enctype 的值设为 multipart/form-data。为了防止提交后刷新本页面，我们创建一个临时的 iframe，将 form 的 target 值指向它。这个我在下一节完整介绍它的实现。

13.7 一个完整的 Ajax 实现

我们先来考虑它有什么参数，首先原生 XMLHTTPRequest 的参数一定要有，然后是 method、url、async、成功回调、失败回调、过时回调、传输数据。再之是与缓存相关，如返回什么数据类型，不区分成功或失败都执行的 complete 回调，过期时间等。为了跨域，我们还要把 JSONP 整合进去。这就涉及回调名。

实现正如 13.1 节提到的那样，需要一个伪 XMLHTTPRequest 对象。它将实现大部分 XMLHTTPRequest2.0 的接口。这由一个主函数返回它，为方便起见，就叫 Ajax。它会将传递的参数处理成 querystring 的形式。发送请求时，我们视跨域或上传，可能采用 script 与 form 来发送数据，因此需要一个传送器（transport）的概念，搞个适配器，视情况用不同的传输器发出请求与绑定回调。数据回来时，我们需要转换原始数据为用户目标数据，因此需要有转换器（converter）的概念，如图 13.1 所示。



▲图 13.1

本节讲解的代码为 mass Framework 的 Ajax 模块，可以到下面地址查看它们。

https://github.com/RubyLouvre/mass-Framework/blob/master/ajax.js

https://github.com/RubyLouvre/mass-Framework/blob/master/ajax_fix.js

接口尽量保持与 jQuery 一致，但不掺杂 Promise 这么复杂的机制。主函数的实现如下。

```
$.ajax = function(opts) {
    if (!opts || !opts.url) {
        $.error("参数必须为 Object 并且拥有 url 属性");
    }
    opts = setOptions(opts); //处理用户参数，比如生成 querystring, type 大写化
    //创建一个伪 XMLHttpRequest,能处理 complete、success、error 等多投事件
    var dummyXHR = new $.XMLHttpRequest(opts);
    "complete success error".replace($.rword, function(name) {//绑定回调
        if (typeof opts[name] === "function") {
            dummyXHR.bind(name, opts[name]);
            delete opts[name];
        }
    });
     var dataType = opts.dataType; //目标返回数据类型
     var transports = $.ajaxTransports;
     var name = opts.form ? "upload" : dataType;
     var transport = transports[name] || transports.xhr;
    $.mix(dummyXHR, transport )//取得传送器的 request, respond, preproccess
    if (dummyXHR.preproccess) { //这用于 jsonp upload 传送器
         dataType = dummyXHR.preproccess() || dataType;
     }
    //设置首部 1. Content-Type 首部
    if (opts.contentType) {
        dummyXHR.setRequestHeader("Content-Type", opts.contentType);
    }
    //2. Accept 首部
    dummyXHR.setRequestHeader("Accept", accepts[dataType] ? accepts[dataType] + ", */*;
q=0.01" : accepts["*"]);
    for (var i in opts.headers) {//3 haders 里面的首部
        dummyXHR.setRequestHeader(i, opts.headers[i]);
    }
    // 处理超时
    if (opts.async && opts.timeout > 0) {
        dummyXHR.timeoutID = setTimeout(function() {
            dummyXHR.abort("timeout");
        }, opts.timeout);
    }
    dummyXHR.request();
    return dummyXHR;
};
```

根据 jQuery 的设计，Ajax 这个主函数的传参为一个对象，并且可以非常复杂，当然也可以简单到只有 url、type、success 这 3 个键值对。因此为了整理这个对象，我们需要一个 setOptions 方法。然后就到达伪 XMLHTTPRequest 的实例 dummyXHR，简单来说它就是一个代理，没有收送数据与接收数据的方法，这些都在传送器上。为此搞了四个传送器，xhr、script、jsonp、upload，视用户传参选定一个，然后 mix 进 dummyXHR，接下来的路线就是，绑定回调，设置首部，设置定时器，发出请求！

setOptions 方法的目的是将 data 对象变成一个 querystring，如果是 GET 请求，那么就把

querystring 加到 url 后，如果不缓存，url 再加时间戳。此外，它还要判定一下 URL 是否跨域，就是取得当前页面的 URL 与用户的 URL 进行比较。

伪 XMLHTTPRequest 作为一个中介者，许多方法与真正的一样。如果不看 dispatch 方法，它的代码还算简单。

```
$.XMLHttpRequest = $.factory($.Observer, {
    init: function(opts) {
        $.mix(this, {
            responseHeadersString: "",
            responseHeaders: {},
            requestHeaders: {},
            querystring: opts.querystring,
            readyState: 0,
            uniqueID: setTimeout("1"),
            status: 0
        });
        this.addEventListener = this.bind;
        this.removeEventListener = this.unbind;
        this.setOptions("options", opts); //创建一个 options 对象保存原始参数
    },

    setRequestHeader: function(name, value) {
        this.requestHeaders[name] = value;
        return this;
    },
    getAllResponseHeaders: function() {
        return this.readyState === 4 ? this.responseHeadersString : null;
    },
    getResponseHeader: function(name, match) {
        if (this.readyState === 4) {
            while ((match = rheaders.exec(this.responseHeadersString))) {
                this.responseHeaders[match[1]] = match[2];
            }
            match = this.responseHeaders[name];
        }
        return match === undefined ? null : match;
    },
    overrideMimeType: function(type) {
        this.mimeType = type;
        return this;
    },
    toString: function() {
        return "[object XMLHttpRequest]";
    },
    // 终止请求
    abort: function(statusText) {
        statusText = statusText || "abort";
        if (this.transport) {
            this.respond(0, statusText);
        }
        return this;
    },
    //用于触发用户绑定的 success、error、complete 回调
```

```
    dispatch: function(status, statusText) {/*略*/
    }
});
```

可以看到之前的 setRequestHeader 调用，都把首部放到 requestHeaders 对象上，这个留给真正的 XMLHTTPRequest 对象调用。

$.Observer 很简单，其实就是用于收集回调。源代码如下。

```
$.Observer = $.factory({
    init: function(target) {
        this._events = {};
        this._target = target || this;
    },
    bind: function(type, callback) {
        var listeners = this._events[type];
        if (listeners) {
            listeners.push(callback);
        } else {
            this._events[type] = [callback];
        }
        return this;
    },
    unbind: function(type, callback) {
        var n = arguments.length;
        if (n === 0) {
            this._events = {};
        } else if (n === 1) {
            this._events[type] = [];
        } else {
            var listeners = this._events[type] || [];
            var i = listeners.length;
            while (--i > -1) {
                if (listeners[i] === callback) {
                    return listeners.splice(i, 1);
                }
            }
        }
        return this;
    },
    fire: function(type) {
        var listeners = (this._events[type] || []).concat(); //防止影响原数组
        if (listeners.length) {
            var target = this._target, args = [].slice.call(arguments);
            if (this.rawFire) {
                args.shift();
            } else {
                args[0] = {
                    type: type,
                    target: target
                };
            }
            for (var i = 0, callback; callback = listeners[i++]; ) {
                callback.apply(target, args);
            }
```

```
        }
    }
});
```

为了与原生 XMLHTTPRequest 一致，我们还将其 bind、unbind 方法改名为 addEventListener 与 removeEventListener。

接着是传送器部分，每一个传送器都必有 request 方法与 respond 方法，视情况还有 preproccess 方法。

xhr 传送器是万能传送器。在非常新的浏览器内，它基本上打包一切。它的 request 方法思路如下：得到真正的 XMLHTTPRequest 对象，调用 open 方法发出请求，调用 overrideMimeType 方法（这个见第 13.5 节，用于在旧版本的标准浏览器接收二进制数据，参数为'text/plain; charset=x-user-defined'），调用 setRequestHeader 方法将之前放在 requestHeaders 中的首部释数放出（在这之前，我们强制添加 X-Requested-With 首部），视情况设置 XMLHTTPRequest2 新增的 responseType 属性（这里我们只允许 blob、arraybuffer、text 通过此分支），发送数据（我们可能在这里放上 formData 或 querystring），最后是绑定回调。回调有两种绑法，一种是使用 onload，一种是使用传参的 onreadystatechange。

```
xhr.request = function() {
    var opts = this.options;
    $.log("XhrTransport.request.....");
    var transport = this.transport = new $.xhr;
    if (opts.crossDomain && !("withCredentials" in transport)) {
        $.error("本浏览器不支持 crossdomain xhr");
    }
    if (opts.username) {
        transport.open(opts.type, opts.url, opts.async, opts.username, opts.password);
    } else {
        transport.open(opts.type, opts.url, opts.async);
    }
    if (this.mimeType && transport.overrideMimeType) {
        transport.overrideMimeType(this.mimeType);
    }
    this.requestHeaders["X-Requested-With"] = "XMLHTTPRequest";
    for (var i in this.requestHeaders) {
        transport.setRequestHeader(i, this.requestHeaders[i]);
    }
    var dataType = this.options.dataType;
    if ("responseType" in transport && /^(blob|arraybuffer|text)$/.test(dataType)) {
        transport.responseType = dataType;
        this.useResponseType = true;
    }
    transport.send(opts.hasContent && (this.formdata|| this.querystring) || null);
    //在同步模式中,IE6、IE7 可能会直接从缓存中读取数据而不会发出请求,因此我们需要手动发出请求
    if (!opts.async || transport.readyState === 4) {
        this.respond();
    } else {
        var self = this;
        if (transport.onerror === null) { //如果支持 onerror、onload 新 API
            transport.onload = transport.onerror = function(e) {
```

```
                this.readyState = 4; //IE9+
                this.status = e.type === "load" ? 200 : 500;
                self.respond();
            };
        } else {
            transport.onreadystatechange = function() {
                self.respond();
            };
        }
    }
}
```

xhr 传送器的 respond 方法思路如下：首先判定这个方法被用过没有，这是一个一次性方法，只能调用一次。但在 onreadychange 回调中，它可能会被调用两次以上，因此我们得做些限制，只有是被强制中止或 readyState 为 4 时，才能进入主逻辑，把 transport 属性删掉。因此有没有被用过就根据这个 transport 属性。XMLHTTPRequest 的 abort 方法是能真正中断请求的，旧版本 IE 我们只好做个标记，方便以下不再调用任何回调，相当什么也没发生一样。然后我们从 XMLHTTPRequest 取得 response、responseXML、responseText 进行处理。不过取 responseXML 属性时，如果后台不按规则编写 XML，浏览器解析发生错误，这时这个属性对应是一个 DOMException 对象，我们访问它时会抛“An attempt was made to use an object that is not, or is no longer, usable”异常。在跨域的情况下，Firefox 在访问 statusText 也会抛错，因此我们对它们都进行 try catch。之后，根据 responseText 修正是 200 还是 204 状态码，初步修正 IE 的 1223 状态码。最后调用 dispatch 方法。

```
xhr.respond = function(event, forceAbort) {
    var transport = this.transport;

    if (!transport) {
        return;
    }
    try {
        var completed = transport.readyState === 4;
        if (forceAbort || completed) {
         transport.onerror = transport.onload = transport.onreadystatechange = $.noop;
            if (forceAbort) {
                if (!completed && typeof transport.abort === "function") {
                    transport.abort();
                }
            } else {
                var status = transport.status;
                this.responseText = transport.responseText;
                try {
//当 responseXML 为[Exception: DOMException]时
//访问它会抛 "An attempt was made to use an object that is not, or is no longer, usable" 异常
                    var xml = transport.responseXML
                } catch (e) {
                }
                if (this.useResponseType) {
                    this.response = transport.response;
                }
                if (xml && xml.documentElement) {
                    this.responseXML = xml;
```

```
                }
                this.responseHeadersString = transport.getAllResponseHeaders();
                //Firefox 跨域请求时访问 statusText 值会抛出异常
                try {
                    var statusText = transport.statusText;
                } catch (e) {
                    statusText = "firefoxAccessError";
                }
//用于处理特殊情况，如果是一个本地请求，只要我们能获取数据就假当它是成功的
                if (!status && isLocal && !this.options.crossDomain) {
                    status = this.responseText ? 200 : 404;
                    //IE 有时会把 204 当作为 1223
                   //returning a 204 from a PUT request - IE seems to be handling the 204
                   from a DELETE request okay.
                } else if (status === 1223) {
                    status = 204;
                }
                this.dispatch(status, statusText);
            }
        }
    } catch (e) {
// 如果网络问题时访问 XHR 的属性，在 FF 会抛异常
// http://helpful.knobs-dials.com/index.php/Component_returned_failure_code:_0x80040111
_ (NS_ERROR_NOT_AVAILABLE)
        if (!forceAbort) {
            this.dispatch(500, e + "");
        }
    }
}
```

jsonp 传送器用于发出 JSONP 跨域请求。它的原理是后端根据一个函数名，生成一个 JavaScript 文件，把一个对象放到它的小括号内当传参。当浏览器加载完后，就会立即执行这个函数，于是就得到目标数据。为此，我们在发出请求前，需要动态创建一个全局函数。但问题是全局数据无法 delete 掉，因此我们退一步，将这个函数放到一个全局对象上，这个对象就是框架的命名空间。要传给后端，我们需要约定一个参数，通常叫做 callback，不过我们写成可配置，让它在$.ajax 的配置对象中叫 jsonp，而那个全局函数名也写成可配置的，叫做 jsonpCallback。如果用户没有传，那么我们动用"jsonp"加随机数创建一个。这些逻辑被我们抽象为一个 preproccess 方法。

```
script.preproccess = function() {
    var namespace = DOC.URL.replace(/(#.+|\W)/g, '');//得到 mass 框架的命名空间
    var opts = this.options;
    var name = this.jsonpCallback = opts.jsonpCallback || "jsonp" + setTimeout("1");
    opts.url = opts.url + (rquery.test(opts.url) ? "&" : "?") +
           opts.jsonp + "=" + namespace + "." + name;//rquery = /\?/
    //将后台返回的 json 保存在惰性函数中
    global[namespace][name] = function(json) {
        $[name] = json;
};
    return "script"
};
```

jsonp 传送器的 request 方法，主要流程是创建一个 script 方法，插入到 DOM 树，然后绑定 onload 或 onreadychange 等回调，最后设置 src 属性以发出请求。在较新的浏览器中，script 标签都支持 onerror 方法，与 XMLHTTPRequest 一样好用，因此统一使用 script 来实现 JSONP，不搞什么 Ajax 跨域了。

```
script.request = function() {
    var opts = this.options;
    var node = this.transport = DOC.createElement("script");
    if (opts.charset) {
        node.charset = opts.charset;
    }
    var load = node.onerror === null;//判定是否支持 onerror
    var self = this;
    node.onerror = node[load ? "onload" : "onreadystatechange"] = function() {
        self.respond();
    };
    node.src = opts.url;
    $.head.insertBefore(node, $.head.firstChild);
};
```

jsonp 传送器的 respond 方法纠结于如何判定成功与否，在旧版本 IE 下只有一个 onreadystatechange 事件，因此我在上面的 preproccess 方法留了一手。如果成功执行了我们的全局函数，那是一个惰性函数，会重写自身，因此对应位置就变成一个对象，如果还是一个函数，说明失败。

```
script.respond = function(event, forceAbort) {
    var node = this.transport;
    if (!node) {
        return;
    }
    var execute = /loaded|complete|undefined/i.test(node.readyState);
    if (forceAbort || execute) {
        node.onerror = node.onload = node.onreadystatechange = null;
        var parent = node.parentNode;
        if (parent) {
            parent.removeChild(node);
        }
        if (!forceAbort) {
            var args = typeof $[this.jsonpCallback] === "function" ?
                    [500, "error"] : [200, "success"];
            this.dispatch.apply(this, args);
        }
    }
};
```

script 传送器与 jsonp 传送器几乎一样，就是没有 preproccess 方法。

upload 传送器有两个实现，一个是利用 FormData，另一个原生 form.submit 提交，然后欺骗浏览器刷新临时生成的 iframe。第一种用到 XMLHTTPRequest 对象，因此它的 request/respond 方法重用 xhr 传送器就行了，我们只需多加个 preproccess 方法。

```
$.ajaxTransports.upload = {
    preproccess: function() {
        var opts = this.options;
        var formdata = new FormData(opts.form);//将二进制打包到 formdata
        $.each(opts.data, function(key, val) {
            formdata.append(key, val);//添加客外数据
        });
        this.formdata = formdata;
    }
};
$.each($.ajaxTransports.xhr, function(key, val) {
    $.ajaxTransports.upload[key] = val;//重用 xhr 传送器的方法
});
```

不支持 FormData 的浏览器，我们使用第二种，它被放到 ajax_fix 模块内。ajax_fix 里面有一个补丁函数，当它最后被执行后，会重写 upload 传送器。

新的 request 方法主要是在原生 form 元素上做文章，重写它 target、action、enctype 等四个属性，如果用户有额外的传参（opts.data），那么一个键值对就对应一个隐藏域，收集到 fields 数组内。然后创建一个隐藏的 iframe，id 与 name 值都等于 target 的值，并绑定 onload 事件。当后端有什么要返回时，浏览器会默认写到 body 的一个 pre 元素内。然后是手动触发 submit 提交，提交后再还原 form 最初的状态。

```
function createIframe(ID) {
    var iframe = $.parseHTML("<iframe " + " id='" + ID + "'" +
      " name='" + ID + "'" + " style='position:absolute;left:-9999px;top:-9999px;/>").
firstChild;
    return (DOC.body || DOC.documentElement).insertBefore(iframe, null);
}
function addDataToForm(form, data) {
    var ret = [],
            d, isArray, vs, i, e;
    for (d in data) {
        isArray = Array.isArray(data[d]);
        vs = isArray ? data[d] : [data[d]];
        // 数组和原生一样对待，创建多个同名输入域
        for (i = 0; i < vs.length; i++) {
            e = DOC.createElement("input");
            e.type = 'hidden';
            e.name = d;
            e.value = vs[i];
            form.appendChild(e);
            ret.push(e);
        }
    }
    return ret;
}

upload.request = function() {
    var self = this;
    var opts = this.options;
    var ID = "iframe-upload-" + this.uniqueID;
    var form = opts.form;
```

```
    var iframe = this.transport = createIframe(ID);
    //form.enctype 的值
    //1:application/x-www-form-urlencoded   在发送前编码所有字符（默认）
    //2:multipart/form-data 不对字符编码。在使用包含文件上传控件的表单时，必须使用该值。
    //3:text/plain  空格转换为 "+" 加号，但不对特殊字符编码。
    var backups = {
        target: form.target || "",
        action: form.action || "",
        enctype: form.enctype,
        method: form.method
    };
    var fields = opts.data ? addDataToForm(form, opts.data) : [];
    //必须指定 method 与 enctype，要不在 FF 报错
    //表单包含文件域时，如果缺少 method=POST 以及 enctype=multipart/form-data，
    //文件将不会被发送。
    form.target = ID;//防止整页刷新
    form.action = opts.url;
    form.method = "POST";
    form.enctype = "multipart/form-data";
    this.uploadcallback = $.bind(iframe, "load", function(event) {
        self.respond(event);
    });
    form.submit();
    //还原 form 的属性
    for (var i in backups) {
        form[i] = backups[i];
    }
    //移除之前动态添加的节点
    fields.forEach(function(input) {
        form.removeChild(input);
    });
};
```

新的 respond 方法主要是在 onload 事件中，进行 responseXML 与 responseText 的抽取工作，最后把整个 iframe 移除。

```
upload.respond = function(event) {
    var node = this.transport, child
    // 防止重复调用,成功后 abort
    if (!node) {
        return;
    }
    if (event && event.type === "load") {
        var doc = node.contentWindow.document;
        this.responseXML = doc;
        if (doc.body) {//如果存在 body 属性,说明不是返回 XML
            this.responseText = doc.body.innerHTML;
            //当 MIME 为'application/javascript' 'text/javascript',
            //浏览器会把内容放到一个 PRE 标签中
            if ((child = doc.body.firstChild) && child.nodeName.toUpperCase()
                    === 'PRE' && child.firstChild) {
                this.responseText = child.firstChild.nodeValue;
```

```
            }
        }
        this.dispatch(200, "success");
    }
    this.uploadcallback = $.unbind(node, "load", this.uploadcallback);
    delete this.uploadcallback;
    setTimeout(function() {  // Fix busy state in FF3
        node.parentNode.removeChild(node);
        $.log("iframe.parentNode.removeChild(iframe)");
    });
}
```

下面回过头来看伪 XMLHTTPRequest 的 dispatch 方法了。之前所有传送器的 respond 方法都会执行 dispatch 方法，把状态码与描述传给它，并且把原始数据保存到 responseText, responseXML 属性中。现在我们要做的是，根据状态码判定成功与否，进一步修正状态描述，如 204 更正为 nocontent，304 更正为 notmodified。如果浏览器支持 response 这个新属性，那么传送器会把 response 数据从真正的 XMLHTTPRequest 对象转移到伪对象上，否则使用转换器进行转换。如果用户没有指定目标类型，我们可以根据 mimeType 或 Content-Type 进行揣测。最后是把状态码、转换好的数据、伪 XMLHTTPRequest 对象传入到 success，error，complete 回调中。

```
function dispatch(status, statusText) {
    if (!this.transport) {
        return;
    }
    this.readyState = 4;
    var eventType = "error";
    if (status >= 200 && status < 300 || status === 304) {
        eventType = "success";
        if (status === 204) {
            statusText = "nocontent";
        } else if (status === 304) {
            statusText = "notmodified";
        } else {
//如果浏览器能直接返回转换好的数据就最好不过了，否则需要手动转换
            if (typeof this.response === "undefined") {
                var dataType = this.options.dataType || this.options.mimeType;
                if (!dataType) { //如果没有指定 dataType，则根据 mimeType 或 Content-Type 进行揣测
                    dataType = this.getResponseHeader("Content-Type") || "";
                    dataType = dataType.match(/json|xml|script|html/) || ["text"];
                    dataType = dataType[0];
                }
                try {
                    this.response = $.ajaxConverters[dataType].call(this, this.response
                    Text, this.responseXML);
                } catch (e) {
                    eventType = "error";
                    statusText = "parsererror : " + e;
                }
            }
        }
    }
    this.status = status;
```

```
        this.statusText = statusText;
        if (this.timeoutID) {
            clearTimeout(this.timeoutID);
            delete this.timeoutID;
        }
        this.rawFire = true;
        this._transport = this.transport;
        // 到这里，要么成功，调用 success，要么失败，调用 error，最终都会调用 complete
        if (eventType === "success") {
            this.fire(eventType, this.response, statusText, this);
        } else {
            this.fire(eventType, this, statusText);
        }
        this.fire("complete", this, statusText);
        delete this.transport;
    }
```

像二进制数据，标准浏览器已经帮我们转换好，我们只需要转一些简单的数据类型。旧版本 IE 转二进制比较苦逼，目前我只实现 arraybuffer（见 10.4 节）。

```
$.ajaxConverters = {//转换器，返回用户想要做的数据
    text: function(text) {
        return text || "";
    },
    xml: function(text, xml) {
        return xml !== void 0 ? xml : $.parseXML(text);
    },
    html: function(text) {
        return $.parseHTML(text); //一个文档碎片,方便直接插入 DOM 树
    },
    json: function(text) {
        return $.parseJSON(text);
    },
    script: function(text) {
        $.parseJS(text);
    },
    jsonp: function() {
        var json = $[this.jsonpCallback];
        delete $[this.jsonpCallback];
        return json;
    }
}
```

到这里主函数正式运行完，然后就是 get、post、getJSON、getScript、upload 等方法，它只是在内部调用一下$.ajax 而已。通常本模块还会包含一些将表单转换一个对象或一个 querystring 的方法。

第14章 动画引擎

浏览网页，尤其是国外的网页，我们经常被老外的创意所感动。无论是极具视觉冲击的动画，还是很平缓但舒适的细微变动，都是这东西所创造的。动画引擎听起来是很高级的东西，但原理却很简单。

以前听说日本的动画是这样做出来：一个人在一本空白的书中画了许多画，这些画每一张与上一张都有细微的差异，然后快速翻动书页，就看起来像动一样。动画只不过是我们眼睛的残影，叫做视觉暂留现象。这里有两个关键字：差异与快速。在网页中，控制样式的任务已经交由 CSS 掌控，让 JavaScript 第一次拥有时间处理的 API，setTimeout 与 setInterval 早在 CSS 诞生前已出现。CSS 可划分两种，一种的值近似一个无限集合，第一种只有寥寥几个值。能够度量与变化的也是那个无限集。就是像颜色值，red、yellow、black，我们也可以转换为 RGB 进行计算。我们可以在定时器里面，每隔 20ms~30ms 改变这些样式值，于是就有了动画。改变宽高，就叫缩放；改变坐标，就叫位移；改变坐标轴，就叫旋转；改变透明度，就叫淡入淡出……

14.1 动画的原理

在标准浏览器中，可计算的样式值基本上它已经为你转换好，如 width、height、margin-x、border-x-width、padding-x，这些样式的单位为 px，color、background-color 则被分解为 RGB，这个很容易就格式化为一个数组，透明度就自不用说。不过，最恶心的是新引进的变形样式 transfrom，它有两类传值方式，一种面向人类，是 rotate() / skew() / scale() / translate() ，分别还有 x、y 之分，比如 rotateX() 和 rotateY() ，依此类推；一类面向计算机，传入矩阵进去，matrix()。无论是传什么，它最后都转成矩阵。

如果换是旧版本 IE，那么就得自己转了，比如你原来的单位是 em, currentStyle 会返回 em，原来填的颜色是 red，它不会返回 rgb(255,0,0)给你。

因此搞动画引擎的第一步是设法获得元素的精确样式值，这个我在样式模块这一章有介绍。那里给出的$.css 方法基本上除颜色值都是已转换好的。

现在我们尝试让一个方块动起来吧。动起来，换言之就是改变位置，在 CSS 就是对应 top 与 left，当然你还可以用 margin-left，现在我们只讲最通用的方式。要想用 top 与 left 还需要让它相对定位或绝对定位。通常的做法是父元素相对定位，成为包含块，子元素绝对定位。只有定位了，top 与 left 才不会返回 auto，而是可计算的像素值。我们首先要得到它原来的位置。简单起见，我们先

实现平移，那么只需要取 left，读者可以在 Firefox 下做实现，直接使用 getComputedStyle，然后让用户传结束位置，起始位置与结束位置之间的距离，就是我们要一点点操作的变量。动画还涉及到时间，一个时长，也就是动画执行的总时间，另一个是每次变动的相隔时间，这个通常由引擎决定，当然也可以暴露出来，它有个学名叫做 fps。

fps 通俗地说，叫刷新率，在 1 秒内更新多少次画面。根据人眼睛的视觉停留效应，若前一幅画像留在大脑中的印象还没消失，后一幅画像就接踵而至，而且两幅画面间的差别很小，就会有“动”的感觉。那么停留多少毫秒最合适呢？我们不但要照顾人的眼睛，还要顾及一下显示器的显示速度与浏览器的渲染速度。根据外国的统计[①]，25 毫秒为最佳数值。其实，这个数值我们应该当作常识来记住。联想一下，日本动画好像有个规定是 1 秒 30 张画，中国的是 1 秒 24 张。用 1 秒去除以张数，就得到每幅画面停留的时间。日本的那个 27.77 毫秒已经很接近我们的 25 毫秒了，因为浏览器的渲染速度明显不如电视机的渲染速度，尤其是 IE6 这个拉后腿的。距离与时间出来了，那么我们再求一下速度就行了。

为此我们建一个新页面，里面有一个方块，它位于包含块这个轨道上。它要从这一端跑到另一个端。动画时间为 2 秒，fps 为 30 帧。

```
<style type="text/css">
    #taxiway{
        width:800px;
        height:100px;
        background:#E8E8FF;
        position: relative;
    }
    #move{
        position: absolute;
        left:0px;
        width:100px;
        height:100px;
        background:#a9ea00;
    }
</style>
<div id="taxiway">
      <div id="move" ></div>
</div>
<script>
    window.onload = function() {
        var el = document.getElementById("move");
        var parent = document.getElementById("taxiway")
        var distance = parent.offsetWidth - el.offsetWidth;                    //总移动距离
        var begin = parseFloat(window.getComputedStyle(el, null).left); //开始位置
        var end = begin + distance;                                    //结束位置
        var fps = 30;                                                  //刷新率
        var interval = 1000 / fps;                                     //每相隔多少 ms 刷新一次
        var duration = 2000;//时长
        var times = duration / 1000 * fps;                             //一共刷新这么多次
        var step = distance / times;//每次移动多少距离
```

① http://www.nczonline.net/blog/2009/08/11/timed-array-processing-in-javascript/。

```
        el.onclick = function() {
            var now = new Date
            var id = setInterval(function() {
                if (begin >= end) {
                    el.style.left = end + "px";
                    clearInterval(id);
                    alert(new Date - now)
                } else {
                    begin += step;
                    el.style.left = begin + "px"
                }
            }, interval)
        }
    }
</script>
```

上例中，我们使用了最简单的累加来实现。现在我们改写一下，加入进度这个变量，让其更具广泛性。

```
el.onclick = function() {
    var beginTime = new Date
    var id = setInterval(function() {
        var t = new Date - beginTime;//当时已用掉的时间
        if (t >= duration) {
            el.style.left = end + "px";
            clearInterval(id);
            alert(t)
        } else {
            var per = t / duration;//当前进度
            el.style.left = begin + per * distance + "px"
        }
    }, interval)
}
```

如果我们能随意控制 per 这个数值，那么就能轻易实现加速减速。于是乎，人们发明了缓动公式。所谓缓动公式其实来自数学上的三角函数，二次项方程式，高阶方程式，它们最初由 flash 界的 Robert Penner[①]整理收来。

有了缓动公式，我们就能轻松模拟现实中加速、减速、急刹车、重力、摇摆、弹簧、来回弹动等效果。

14.2 缓动公式

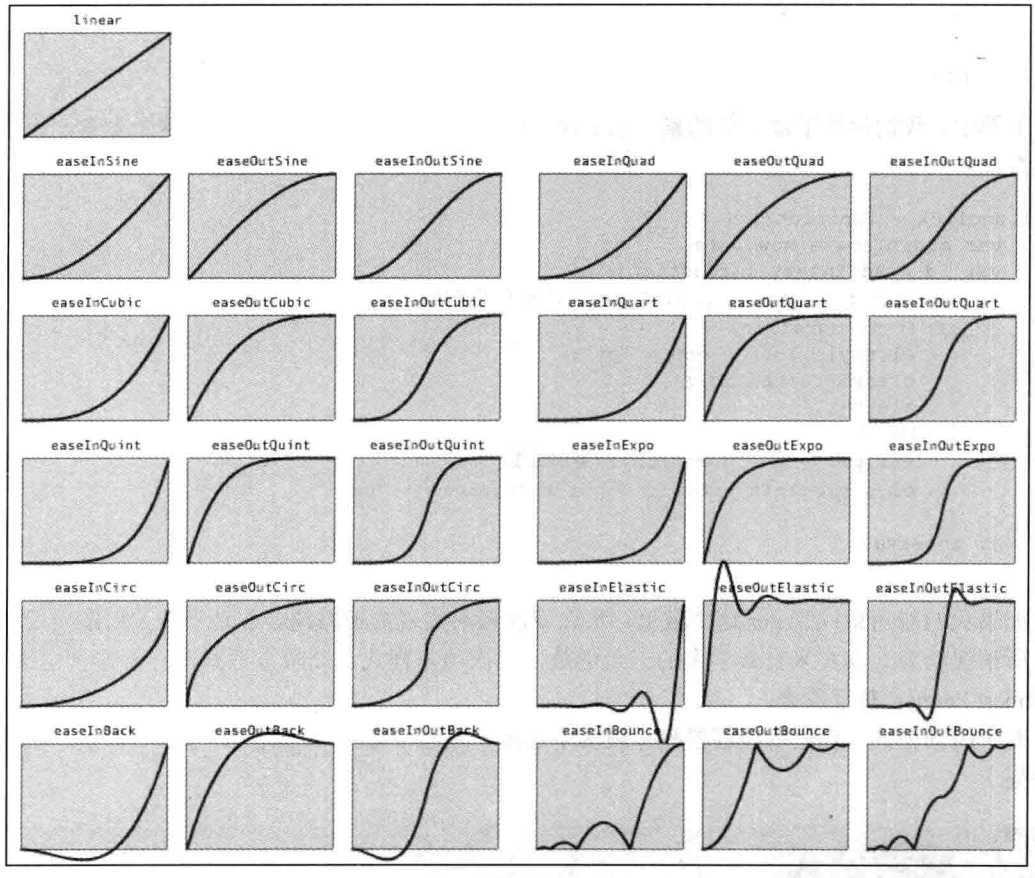
经过这么多的发展，缓动公式的各项规范都稳定下来。虽然现在还有人源源不断地发掘出新公式，但一般业务中，绝对多数人都是用默认的 easeIn 或 linear。因此如果对库的大小有顾虑，那么就将它们独立成一个模块吧。

现在所有缓动公式，基本上除了 linear 外（但它也常被称为 easeNone），它们都以 ease 开头命名。

① http://www.robertpenner.com/easing/

添加 3 种后缀，In 表示加速，Out 表示减速，InOut 表示加速到中途又开始减速，于是就有 easeIn、easeOut、easeInOut 之分。如果单是这样命名，说明，它们没有介入高阶函数与三角函数。像 linear 就是匀速。

然后再以实现方式与指数或开根进行区分。Sine 表示由三角函数实现，Quad 是二次方，Cubic 是三次方，Quart 是四次方，Quint 是五次方，Circ 使用开平方根的 Math.sqit，Expo 使用开立方根的 Math.pow, Elastic 则是结合三角函数与开立三方根的初级弹簧效果，Back 是使用了一个 1.70158 常数来计算的回退效果，Bounce 则是高级弹簧效果，如图 14.1 所示。



▲图 14.1

图片出自 http://hosted.zeh.com.br/tweener/docs/en-us/misc/transitions.html。

这些公式我们可以在 AS 库[①]，或 jquery.easing.js[②]、mootools[③]等库里面扒下来，基本上大同小异。其中 jquery.easing.js 最规范，mootools 使用循环生成方式实现，代码最简。读者想实现自己的

① https://github.com/danro/tweenman-as3/blob/master/Easing.as
② https://github.com/danro/jquery-easing/blob/master/jquery.easing.js
③ https://github.com/mootools/mootools-core/blob/master/Source/Fx/Fx.Transitions.js

动画库，可以参考这两者。

jQuery 标准库里面只有两条：

```
linear: function( p ) {
    return p;
},
swing: function( p ) {
    return 0.5 - Math.cos( p*Math.PI ) / 2;
}
```

但显然与其他缓动库的参数形式不一样：

```
var easing = {
    easeInQuad: function( t, b, c, d) {
        return c * (t /= d) * t + b;
    },
    easeOutQuad: function( t, b, c, d) {
        return -c * (t /= d) * (t - 2) + b;
    },
    easeInOutQuad: function( t, b, c, d) {
        if ((t /= d / 2) < 1)
            return c / 2 * t * t + b;
        return -c / 2 * ((--t) * (t - 2) - 1) + b;
    }
    //……
}
```

这是因为 jQuery 在上面的计算过程已经做了这一步。我们看这四个参数分别代表什么。

- T：timestamp，指缓动效果开始执行到当前帧所经过的时间段，单位 ms。
- B：beginning val，起始值
- C：change，要变化的总量。
- D：duration ，动画持续的时间。

返回的是直接可用的数值，或许我们只要加个单位进行赋值就行了。

而 jQuery 那一个参数的风格，其实是当前时间减去动画开始时间除以总时间的比值，一个 0 到 1 的小数，它用于乘以总变化量，然后加上起始值，就是现在此样式的情况。

下面是缓动的应用，还是使用上例：

```
window.onload = function() {
    var el = document.getElementById("move");
    var parent = document.getElementById("taxiway")
    var change = parent.offsetWidth - el.offsetWidth;                    //★总变化量
    var begin = parseFloat(window.getComputedStyle(el, null).left);//★起始值
    var end = begin + change;                                    //★结束值
    var fps = 30;                                                //刷新率
    var interval = 1000 / fps;                                   //每相隔多少 ms 刷新一次
    var duration = 2000;                                         //★时长
    var bounce = function(per) {                                 //★缓动公式，弹簧
        if (per < (1 / 2.75)) {
            return (7.5625 * per * per);
        } else if (per < (2 / 2.75)) {
            return (7.5625 * (per -= (1.5 / 2.75)) * per + .75);
        } else if (per < (2.5 / 2.75)) {
```

```
            return (7.5625 * (per -= (2.25 / 2.75)) * per + .9375);
        } else {
            return (7.5625 * (per -= (2.625 / 2.75)) * per + .984375);
        }
    }
    el.onclick = function() {
        var beginTime = new Date
        var id = setInterval(function() {
            var per = (new Date - beginTime) / duration;//★进度
            if (per >= 1) {
                el.style.left = end + "px";
                clearInterval(id);
            } else {
                el.style.left = begin + bounce(per) * change + "px";
            }
        }, interval)
    }
}
```

点击它就发现方块到终点后还弹回来，再过去，又弹回来，渐渐平息……

14.3 API 的设计

现在我们看如何写动画引擎。由于选择器的流行，注定这个 API 到用户里也是集化操作，一下子处理 N 个元素。但这也无所谓，关键是函数名与如何传参。

jQuery 的 API 易用性很好，我们看一下它的 animate。它有两种用法。

```
.animate( properties [, duration ] [, easing ] [, complete ] )
.animate( properties, options )
```

第一个参数恒为要进行动画的属性的映射，在第一种情况下，其他参数都是可选的，因为 duration 除了 show、fast、default 这三个字符串就是数字，easing 为特殊的缓动公式的名字，complete 是函数，options 是对象。不过尽管说得轻松，jQuery 在这里也花了几十行进行转换。最后转换两个对象的形式其实与后来出现的 CSS3 keyframe animation 的定义非常吻合。

我们还是利用上例的方块，加个类名 animate，就能看到效果了。

```
.animate {
    animation-duration: 3s;
    animation-name: slidein;
    animation-timing-function:ease-in-out;
    animation-fill-mode: forwards
}

@keyframes slidein {
    from {
    left: 0%;
    background: white;
}

    to {
```

```
    left: 700px;
    background: red;
}
}
```

这个动画分为两部分：一个是普通的样式规则，用于描述动画所需时长，缓动公式，结束后保留状态，重复多少次，及关键帧动画的引用名字（slidein）；第二个是关键帧规则。这里只剩入两个关键帧，实际上可以插入更多，最开始与最尾的可以用 from、to 命名，其实当你用 JavaScript 去取时，它们都会转换为百分比，0%或 100%。其中以最尾的最重要，没有浏览器会补上，它就是对应 jQuery. animate 方法的第一个参数。至于初始值，浏览器会自动计算，如果我们是使用纯 JavaScript 实现方式，它们就要我们来计算了。动画引擎的强大与否，这时就取决于 CSS 模块的强大了。animate 方法的第二个参数等对应 animation-duration、animation-timing-function、animation-fill-mode 等设置。

除此之外，jQuery 还提供了一个 queue 参数，目的让作用于同一个元素的动画进行排队，先处理完这个再处理后一个。要不，同时运行于浏览器中的定时器就太多了。加之集化操作，会把它放大化，队列的重要性就更突出。为此 jQuery 把这个参数默认为 true。

从实现上看，jQuery 的队列是放在元素对应的缓存体上，里面是一个 Promise 对象，complete 之后，会自动弹出下一个动画对象。所有动画对象都有自己的 setInterval 驱动。在 YUI、kissy、mass Framework 则有一个中央队列，所有不排队的动画全部放在这数组中，然后有一个 setInterval 来驱动它们，排队的动画作为它的兄弟的属性而存在（有的中央队列可能是一条二维数组），当前面的动画执行完，排队的动画就会翻身。

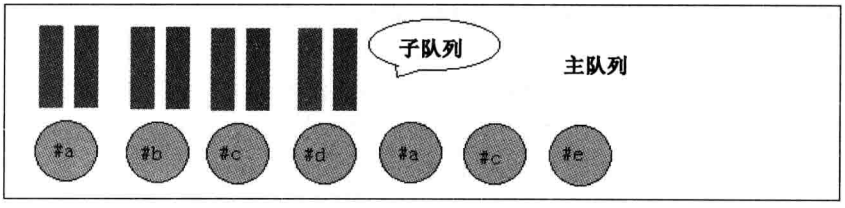
14.4 mass Framework 基于 JavaScript 的动画引擎

现在我们看一下，动画引擎是如何做的。首先，我们搞一个中央队列（也叫时间轴，我们在里面插入关键帧，两个关键帧之间就是我们的补间动画），其实就是一个数组。只要它里面有一个元素，它就驱动 setInterval 执行动画。如果动画执行完毕，它就会删掉其 node 属性，并且从数组中删除此元素。检测一下数组还有没有元素，没有就 clearInterval，否则就继续。

```
https://github.com/RubyLouvre/mass-Framework/blob/master/fx.js
var timeline = $.timeline = [];                    //时间轴

function insertFrame(frame) {                      //插入包含关键帧原始信息的帧对象
    if (frame.queue) {                             //如果指定要排队
        var gotoQueue = 1;
        for (var i = timeline.length, el; el = timeline[--i];) {
            if (el.node === frame.node) {          //★★★第一步
                el.positive.push(frame);           //子队列
                gotoQueue = 0;
                break;
            }
        }
        if (gotoQueue) {                           //★★★第二步
```

```
            timeline.unshift(frame);
        }
    } else {
        timeline.push(frame);
    }
    if (insertFrame.id === null) {//只要数组中有一个元素就开始运行
        insertFrame.id = setInterval(deleteFrame, 1000 / $.fps);
    }
}
insertFrame.id = null;
```

具体如图 14.2 所示。



▲图 14.2

主队列的动画是立即执行的，一个元素可以对应多个动画，比如它的宽与高与背景色都要同时改变。在上图中，#a 元素就有两个同时进行的动画。但在某些场合，我们要求一个方块像星星一样一闪一闪地眨眼睛，就需要子队列了。子队列是放置等待执行的动画，只有前面的动画执行完毕，才执行它们。在 CSS3 中，有 animation-fill-mode，能够轻松实现倒带的效果。

```
<div id="test" class="animate" ></div>
<style type="text/css">
    #test{
        width:100px;
        height:100px;
        background:blue;
    }
  /* 此动画先放大再缩回原状*/
    .animate{
        animation-duration:3s;
        animation-name: cycle;
        animation-iteration-count:2;
        animation-direction:alternate;
    }
    @keyframes cycle{
        to{
        width:200px;
        height:200px;
    }
    }
</style>
```

用 JavaScript 实现就是，把第一帧与最后一帧调换。我们把这些动画放到 negative 子队列中。

如果不是倒带就放在 positive 队列。

```
var effect = $.fn.animate = $.fn.fx = function(props) {
    //将多个参数整成两个，第一参数暂时别动
    var opts = addOptions.apply(null, arguments), p
    //第一个参数为元素的样式，我们需要将它们从 CSS 的连字符风格统统转为驼峰风格
    //如果需要私有前缀，也在这里加上
    for (var name in props) {
        p = $.cssName(name) || name;
        if (name !== p) {
            props[p] = props[name];    //添加 borderTopWidth、styleFloat
            delete props[name];        //删掉 border-top-width、float
        }
    }
    for (var i = 0, node; node = this[i++];) {
        //包含关键帧的原始信息的对象到主队列或子队列
        insertFrame($.mix({
            positive: [],              //正向队列
            negative: [],              //外队队列
            node: node,                //元素节点
            props: props               //@keyframes 中要处理的样式集合
        }, opts));
    }
    return this;
}
```

接着我们看 deleteFrame 方法，它的任务就是把已经完成或强制完成的动画从主队列中删掉。

```
function deleteFrame() {
        var i = timeline.length;
        while (--i >= 0) {
            if (!timeline[i].paused) { //如果没有被暂停
                if (!(timeline[i].node && enterFrame(timeline[i], i))) {
                    timeline.splice(i, 1);
                }
            }
        }
        timeline.length || (clearInterval(insertFrame.id), insertFrame.id = null);
}
```

那么我们如果添加关键帧呢？当然是使用 animate 方法（它也有个更短的别名叫 fx），参数与 jQuery 一样多态化，那也意味着我们需要花些精力调整它们成可用状态。

调整用户传参需要用到这么多代码：

```
function addOptions(properties) {
    if (isFinite(properties)) {//如果第一个为数字
        return {
            duration: properties
        };
    }
    var opts = {};
    //如果第二参数是对象
```

```
    for (var i = 1; i < arguments.length; i++) {
        addOption(opts, arguments[i]);
    }
    opts.duration = typeof opts.duration === "number" ? opts.duration : 400;
    opts.queue = !!(opts.queue == null || opts.queue); //默认进行排队
    opts.easing = $.easing[opts.easing] ? opts.easing : "swing";
    return opts;
}
function addOption(opts, p) {
    switch ($.type(p)) {
        case "Object":
            addCallback(opts, p, "after");
            addCallback(opts, p, "before");
            $.mix(opts, p);
            break;
        case "Number":
            opts.duration = p;
            break;
        case "String":
            opts.easing = p;
            break;
        case "Function":
            opts.complete = p;
            break;
    }
}
function addCallback(target, source, name) {
    if (typeof source[name] === "function") {
        var fn = target[name];
        if (fn) {
            target[name] = function(node, fx) {
                fn(node, fx);
                source[name](node, fx);
            };
        } else {
            target[name] = source[name];
        }
    }
    delete source[name];
}
```

然后才进入我们刚才的 insertFrame 方法，insertFrame 会间接调用 enterFrame 方法。由于它是在 setInterval 内运行的，因此它才是动画真正的实现者。

```
function enterFrame(fx, index) {
    //驱动主队列的动画实例进行补间动画(update)
    //并在动画结束后，从子队列选取下一个动画实例取替自身
    var node = fx.node,
        now = +new Date;
    if (!fx.startTime) {                                    //第一帧
        callback(fx, node, "before");                       //动画开始前做些预操作
        fx.props && parseFrames(fx.node, fx, index);        //分解原始材料为关键帧
        fx.props = fx.props || [];
        AnimationPreproccess[fx.method || "noop")](node, fx); //parse 后也要做些预处理
```

```
            fx.startTime = now;
        } else {  //中间自动生成的补间
            var per = (now - fx.startTime) / fx.duration;
            var end = fx.gotoEnd || per >= 1;//gotoEnd 可以被外面的 stop 方法操控,强制中止
            var hooks = effect.updateHooks;
            if(fx.update){
                for (var i = 0, obj; obj = fx.props[i++]; ) {// 处理渐变
                   (hooks[obj.type] || hooks._default)(node, per, end, obj);
                }
            }
            if (end) { //最后一帧
                callback(fx, node, "after");            //动画结束后执行的一些收尾工作
                callback(fx, node, "complete");         //执行用户回调
                if (fx.revert && fx.negative.length) { //如果设置了倒带
                    Array.prototype.unshift.apply(fx.positive, fx.negative.reverse());
                    fx.negative = []; // 清空负向队列
                }
                var neo = fx.positive.shift();
                if (!neo) {
                    return false;
                }//如果存在排队的动画,让它继续
                timeline[index] = neo;
                neo.positive = fx.positive;
                neo.negative = fx.negative;
            } else {
                callback(fx, node, "step"); //每执行一帧调用的回调
            }
        }
        return true;
    }
```

这里面涉及几个辅助函数，callback 用于执行回调，包括内部用的 before 与 after，用户设置的 step 与 complete。

```
function callback(fx, node, name) {
    if (fx[name]) {
        fx[name](node, fx);
    }
}
```

parseFrames 是相当复杂庞大的，这里我就不贴出来了。总而言之，就是从已有的材料中分成两个关键帧，每个关键帧包括样式名、缓动公式（之前只是名字）、开始值、结束值、单位与类型。类型分为三种，颜色值、滚动与默认处理，根据它们，程序会选用不同的钩子函数进行分解、刷新。

```
effect.updateHooks = {
    _default: function(node, per, end, obj) {
        $.css(node, obj.name, (end ? obj.to : obj.from + obj.easing(per) *
          (obj.to - obj.from)) + obj.unit)
    },
    color: function(node, per, end, obj) {
        var pos = obj.easing(per),
                rgb = end ? obj.to : obj.from.map(function(from, i) {
```

```
            return Math.max(Math.min(parseInt(from + (obj.to[i] - from) * pos, 10), 255), 0);
        });
        node.style[obj.name] = "rgb(" + rgb + ")";
    },
    scroll: function(node, per, end, obj) {
        node[obj.name] = (end ? obj.to : obj.from + obj.easing(per) * (obj.to - obj.from));
    }
};
```

这里可留意一下颜色值的刷新函数，它是转换为 RGB 的形式，再变成数组。刷新函数都有个 end 参数，用于立即跳到最后一帧。

下面是分解颜色值的代码，只处理 16 进制、RGB 与颜色名这 3 种。难点在于 IE 的颜色转化。

```
var colorMap = {
    "black": [0, 0, 0],
    "gray": [128, 128, 128],
    "white": [255, 255, 255],
    "orange": [255, 165, 0],
    "red": [255, 0, 0],
    "green": [0, 128, 0],
    "yellow": [255, 255, 0],
    "blue": [0, 0, 255]
};

function parseColor(color) {
    var value;//在 iframe 下进行操作
    $.applyShadowDOM(function(wid, doc, body) {
        var range = body.createTextRange();
        body.style.color = color;
        value = range.queryCommandValue("ForeColor");
    });
    return [value & 0xff, (value & 0xff00) >> 8, (value & 0xff0000) >> 16];
}

function color2array(val) { //将字符串变成数组
    var color = val.toLowerCase(),
            ret = [];
    if (colorMap[color]) {
        return colorMap[color];
    }
    if (color.indexOf("rgb") === 0) {
        var match = color.match(/(\d+%?)/g),
                factor = match[0].indexOf("%") !== -1 ? 2.55 : 1;
        return (colorMap[color] = [parseInt(match[0]) * factor, parseInt(match[1]) * factor,
parseInt(match[2]) * factor]);
    } else if (color.charAt(0) === '#') {
        if (color.length === 4)
            color = color.replace(/([^#])/g, '$1$1');
        color.replace(/\w{2}/g, function(a) {
            ret.push(parseInt(a, 16));
        });
        return (colorMap[color] = ret);
    }
    if (window.VBArray) {
```

```
            return (colorMap[color] = parseColor(color));
        }
        return colorMap.white;
    }
    $.parseColor = color2array;
```

在 enterFrame 方法中有一个预处理的过程，主要是用于 show、hide 等方法。AnimationPreproccess 里面有四方法：noop、show、hide、toggle。

noop 不处理。

show 是将隐藏的元素的 display 改为 block 等值（有时未必改为 block，像 li、td、tr、tbody、table 都有默认的 display 值，如果强行改 block，布局就会走形；对于内联元素，span、em 等，要使用它们进行缩放操作，则要设置为 inline-block，但众所周知，旧版本 IE 要开启 hasLayout 才能生效，这又是一番周折）。

hide 是将显示的元素隐藏起来，由于它对应的动画效果是从大到小，这时进行动画的那个元素的子元素可能会超出父元素的大小，被挤出来。因此我们需要强制设置它的 overflow 为 hidden，在动画结束后才还原（因为这时 display 为 none，你怎么玩也没人知道）。此外，要还原的样式值还有宽高、边框、补白、外界、透明度等值，这方便我们在 show→hide→show 这样的连续动画中能运行起来。

toggle 就是对隐藏元素进行 show 操作，显示元素进行 hide 操作。

```
var AnimationPreproccess = {
    noop: $.noop,
    show: function(node, frame) {
        if (node.nodeType === 1 && $.isHidden(node)) {
            var display = $._data(node, "olddisplay");
            if (!display || display === "none") {
                display = $.parseDisplay(node.nodeName);
                $._data(node, "olddisplay", display);
            }
            node.style.display = display;
            if ("width" in frame.props || "height" in frame.props) {
                //如果是缩放操作
                //修正内联元素的 display 为 inline-block，让其可以进行 width/height 的动画渐变
                if (display === "inline" && $.css(node, "float") === "none") {
                    if (!$.support.inlineBlockNeedsLayout) { //W3C
                        node.style.display = "inline-block";
                    } else { //IE
                        if (display === "inline") {
                            node.style.display = "inline-block";
                        } else {
                            node.style.display = "inline";
                            node.style.zoom = 1;
                        }
                    }
                }
            }
        }
    },
    hide: function(node, frame) {
```

```
        if (node.nodeType === 1 && !$.isHidden(node)) {
            var display = $.css(node, "display"),
                s = node.style;
            if (display !== "none" && !$._data(node, "olddisplay")) {
                $._data(node, "olddisplay", display);
            }
            var overflows;
            if ("width" in frame.props || "height" in frame.props) {
                //如果是缩放操作
                //确保内容不会溢出,记录原来的 overflow 属性
                //因为 IE 在改变 overflowX 与 overflowY 时, overflow 不会发生改变
                overflows = [s.overflow, s.overflowX, s.overflowY];
                s.overflow = "hidden";
            }
            var fn = frame.after || $.noop;
            frame.after = function(node, fx) {
                if (fx.method === "hide") {
                    node.style.display = "none";
                    for (var i in fx.orig) { //还原为初始状态
                        $.css(node, i, fx.orig[i]);
                    }
                }
                if (overflows) {
                    ["", "X", "Y"].forEach(function(postfix, index) {
                        s["overflow" + postfix] = overflows[index];
                    });
                }
                fn(node, fx);
            };
        }
    },
    toggle: function(node, fx) {
        $[$.isHidden(node) ? "show" : "hide"](node, fx);
    }
};
```

至此，整个动画引擎就完成了。而那些 show、hide、toggle、slideUp、slideDown、slideToggle、show、hide、toggle、slideUp、slideDown、slideToggle、fadeIn、fadeToggle、fadeOut 等特效，其实就是利用 animate 这个主函数，预先传些样式进去。比如 fadeIn，就是将透明度逐渐由 0 变成 1，fadeOut 就是从 1 到 0。 slideUp 就是将高度逐渐减为 0，slideDown 就是从 0 逐渐还原为之前的高度。show 与 hide 则是同时对宽高边界边框补白等盒子属性与透明度进行动画。

Prototype.js 重要特性 script.aculo.us 可能是 JavaScript 最强大的动画框架了，它的许多特效都是其他动画库的重要参考，如表 14.1 所示。

表 14.1

script.aculo.us	jQuery 或 jQuery UI	说　明
Appear	fadeIn	淡入
Fade	fadeOut	淡出
Puff	puff	整体扩大一倍并同时进行淡出

续表

script.aculo.us	jQuery 或 jQuery UI	说　明
DropOut	drop	整体往下堕落并同时进行淡出，最后不占空间
Shake	shake	像蛇信子一样左右震荡几下
SwitchOff	clip	像电视关闭时的收场动画一样两边往中间叠起一线消失
BlindDown	slideDown	像展开卷轴般从上到下呈现元素
BlindUp	slideUp	像收起卷轴般从下到上折起元素
SlideDown	blind	像升降机从我们眼前落下，jQuery UI 的 blind、slide 恰好与 script.aculo.us 的相反
SlideUp	blind	像升降机从我们眼前升起
Pulsate	pulsate	闪动几下
Squish	show	四边往左上角收缩，有时类似 jQuery 的 show 效果，但 jQuery 同时进行淡出
Fold	fold	先是高度从下往上收起，然后宽度从右到左收起
Grow		从中间一点向四边扩张
Shrink	scale	四边往中间一点收缩

基本上 script.aculo.us 能做的，jQuery UI 也能轻松实现。不过 jQuery UI 每个子特效设计得非常强大，因此体积也不遑多让。

14.5 requestAnimationFrame

如果一个页面运行许多定时器，那么你无论怎么优化，最后肯定是超过指定时间才能完成动画，定时器越多，延时越严重。为此，YUI、kissy、mass 等采用中央队列的方式，将定时器减少至一个。浏览器商也不是吃素的，最早 Firefox 也想到这点，早在 4.0 时就推出 mozRequestAnimationFrame。当然，它与现在的标准相差很远。它不是个定时器，甚至也不能传递回调，它只是用于触发 MozBeforePaint 这个私有事件。由于是浏览器在维护队列，它内部掌握 DOM 渲染，事件队列排队等情况，因此它大抵能保证 fps 在 60 帧上下。

```
<script>
  /*firefox4-10*/
  var startTime,
        duration = 3000;
  function animate(event) {
     var now = event.timeStamp;
     var per = (now - startTime) / duration;
     if (per >= 1) {
        window.removeEventListener('MozBeforePaint', animate, false);
     } else {
        document.getElementById("test").style.left = Math.round(600 * per) + "px";
        window.mozRequestAnimationFrame();
     }
  }
  function start() {
```

```
        startTime = Date.now();
        window.addEventListener('MozBeforePaint', animate, false);
        window.mozRequestAnimationFrame();
    }
</script>
<button onclick="start()">点我</button>
<div id="test" style="position: absolute; left: 10px; background: blue;"> go! </div>
```

谷歌发现这个思路不错，立即整到自己的 Chrome 中去，但相对而言，精简很多，与最后定案的标准差距很少。webkitRequestAnimationFrame 有点像定时器，第一个是回调，第二个可选，传执行动画的元素节点进去，返回一个 ID，然后允许像 clearTimeout 一样有个 webkitCancelRequestAnimationFrame 函数进行中止动画。不过名字有点长了，后来模仿 Firefox 一样使用 cancelAnimationFrame，不同的只是前面的私有前缀，即改成 webkitCancelAnimationFrame。

```
<script>
  /*chrome10-23*/
  var startTime,
        duration = 3000;
  function animate(now) {
      var per = (now - startTime) / duration;
    if(per >= 1) {
        window.webkitCancelRequestAnimationFrame(requestID);
    } else {
        document.getElementById("test").style.left = Math.round(600 * per) + "px";
        window.webkitRequestAnimationFrame(animate);//不断地递归调用 animate
    }
}
function start() {
    startTime = Date.now();
    requestID = window.webkitRequestAnimationFrame(animate);
}
</script>
<button onclick="start()">点我</button>
<div id="test" style="position: absolute; left: 10px; background: red;"> go! </div>
```

IE 与 Opera 起步最晚，IE 到 10 才支持，不过那时标准已经成形，没有私有前缀，也没有兼容性问题。Opera 则到 12 还不支持。

网上流行一个非常不负责的写法，以为光是换个名字就行：

```
window.requestAnimFrame = (function(){
  return  window.requestAnimationFrame       ||
          window.webkitRequestAnimationFrame ||
          window.mozRequestAnimationFrame    ||
          function( callback ){
            window.setTimeout(callback, 1000 / 60);
          };
})();
```

还有另一个看起来非常周到的写法：

```
(function() {
    var lastTime = 0;
```

```
    var vendors = ['webkit', 'moz'];
    for(var x = 0; x < vendors.length && !window.requestAnimationFrame; ++x) {
        window.requestAnimationFrame = window[vendors[x]+'RequestAnimationFrame'];
        window.cancelAnimationFrame =
          window[vendors[x]+'CancelAnimationFrame'] || window[vendors[x]+'CancelRequest
AnimationFrame'];
    }

    if (!window.requestAnimationFrame)
        window.requestAnimationFrame = function(callback, element) {
            var currTime = new Date().getTime();
            var timeToCall = Math.max(0, 16 - (currTime - lastTime));
            var id = window.setTimeout(function() { callback(currTime + timeToCall); },
              timeToCall);
            lastTime = currTime + timeToCall;
            return id;
        };

    if (!window.cancelAnimationFrame)
        window.cancelAnimationFrame = function(id) {
            clearTimeout(id);
        };
}());
```

别说早期 Firefox 不支持传参，它到 Firefox11 才与 webkit 那个有点相近，并且才有了 mozCancelAnimationFrame。webkit 也有 BUG，它在某个版本，竟然忘了返回 id 给我们清除。在另一个版本发现它竟然没有给回调传参。

下面我给出真正可用的兼容版本：

```
// by 司徒正美 基于网友屈屈与月影的版本改进而来
// https://github.com/wedteam/qwrap-components/blob/master/animation/anim.frame.js
function getAnimationFrame() {
    //不存在 msRequestAnimationFrame，IE10 与 Chrome24 直接用:requestAnimationFrame
    if (window.requestAnimationFrame) {
        return {
            request: requestAnimationFrame,
            cancel: cancelAnimationFrame
        }
        //Firefox11-没有实现 cancelRequestAnimationFrame
        //并且 mozRequestAnimationFrame 与标准出入过大
    } else if (window.mozCancelRequestAnimationFrame && window.mozCancelAnimationFrame) {
        return {
            request: mozRequestAnimationFrame,
            cancel: mozCancelAnimationFrame
        }
    } else if (window.webkitRequestAnimationFrame && webkitRequestAnimationFrame(String)) {
        return {//修正某个特异的 webKit 版本下没有 time 参数
            request: function(callback) {
                return window.webkitRequestAnimationFrame(
                        function() {
                            return callback(new Date - 0);
                        }
                );
```

```
            },
            cancel: window.webkitCancelAnimationFrame ||
              window.webkitCancelRequestAnimationFrame
        }
    } else {
        var millisec = 25;    //40fps;
        var callbacks = [];
        var id = 0, cursor = 0;
        function playAll() {
            var cloned = callbacks.slice(0);
            cursor += callbacks.length;
            callbacks.length = 0; //清空队列
            for (var i = 0, callback; callback = cloned[i++]; ) {
                if (callback !== "cancelled") {
                    callback(new Date - 0);
                }
            }
        }
        window.setInterval(playAll, millisec);
        return {
            request: function(handler) {
                callbacks.push(handler);
                return id++;
            },
            cancel: function(id) {
                callbacks[id - cursor] = "cancelled";
            }
        };
    }
}
```

当然，requestAnimationFrame 不是没有缺点，它不能控制 fps，比如我们做一些慢放动作，许多回调都是做无用功。另一个极端，在动作、枪战、飞车等动态场景时，如果帧数不够高，画面会发虚或模糊。利用原始的 setTimeout（在 IE9、IE10、Firefox10、Chrome 等浏览器中，它的最短时钟间隔已经压缩至 4ms，能轻松跑 100 帧以上的动画），能让画面更清楚，细节逼真，特写镜头丝丝入扣。

另外，我们还可以尝试一下 postMessage 这个异步方法，能实现超高度的动画（IE10 有 setImmediate，速度也相当不错）。下面有个实验，可以看到它们的性能差异。

```
<script>
    var fps_arr, fps_min, fps_max, last_time, loop_iteration;
    var testing = false;
    var fps_label;
    function stop_test() {
        testing = false;
    }
    var init_test = function init_test() {
        fps_arr = [];
        fps_min = 1000;
        fps_max = last_time = loop_iteration = 0;
        if (typeof fps_label === 'undefined') {
            fps_label = document.getElementById('fps_label');
        }
        testing = true;
```

```
    }
    function main() {
        var i, fps_avg = 0;
        var now = new Date().getTime();
        if (last_time !== 0 && last_time !== now) {
            var fps = Math.round(1000 / (now - last_time));

            fps_arr.push(fps);
            if (fps_arr.length > 100) {
                fps_arr.shift();
            }
            for (i = 0; i < fps_arr.length; i++) {
                fps_avg += fps_arr[i];
            }
            fps_avg /= fps_arr.length;
            fps_avg = Math.round(fps_avg);//平均帧数

            if (++loop_iteration > 1) {
                if (fps < fps_min) {
                    fps_min = fps;//最小帧数
                }
                if (fps > fps_max) {
                    fps_max = fps;//最大帧数
                }
            }
            fps_label.innerHTML = fps + ' FPS (' + fps_min + ' - ' + fps_max + ') [平均为
' + fps_avg + ']';
        }
        last_time = now;
    }
    /* Pure Timers */
    function run_timers() {
        main();
        if (testing === true) {
            setTimeout(run_timers, 1);
        }
    }
    window.requestAnimFrame =
            window.requestAnimationFrame ||
            window.webkitRequestAnimationFrame ||
            window.mozRequestAnimationFrame;
    /* requestAnimationFrame */
    function run_raf() {
        main();
        if (testing === true) {
            requestAnimFrame(run_raf, 1);
        }
    }
    /* for loop */
    function run_loop() {
        var iterations = 15;
        while (iterations--) {
            main();
        }
        if (testing === true) {
```

```
            setTimeout(run_loop, 1);
        }
    }
    /* postMessage */
    function run_message() {
        main();
        if (testing === true) {
            window.postMessage('', '*');
        }
    }
    window.addEventListener('message', run_message, false);
</script>
<p id="fps_label"># fps (# - #) [#]</p>
<button onClick="stop_test();">中止</button>

<br /><br />
<strong>Tests</strong><br />
<button onClick="init_test();
      run_timers();">Pure Timers</button>
<button onClick="init_test();
      run_raf();">requestAnimationFrame</button>
<button onClick="init_test();
      run_loop();"> Loop</button>
<button onClick="init_test();
      run_message();">postMessage</button>
```

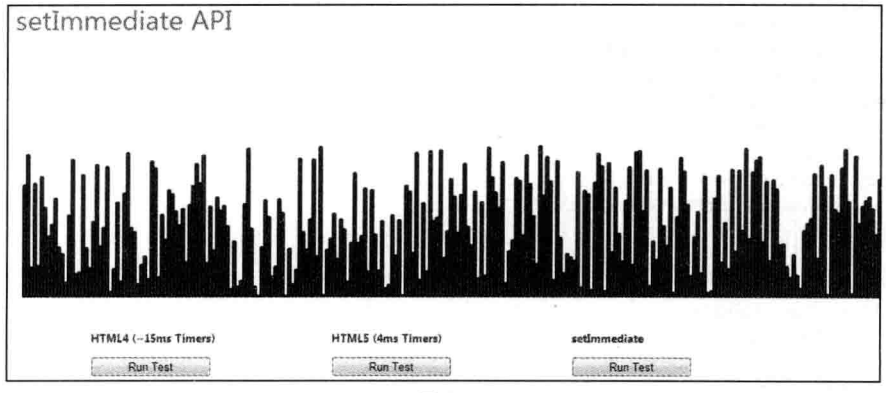
结果大致如下，视各位的电脑性能而言，如表 14.2 所示。

表 14.2

类型	setTimeout	requestAnimationFrame	loop	postMessage
平均帧数	200	60	200～300	900～1000

微软官方也放出使用高性能异步方法 setImmediate 与原始 setTimeout 的对比实验，流畅度很好，如图 14.3 所示。

http://ie.microsoft.com/testdrive/Performance/setImmediateSorting/Default.html



▲图 14.3

在现实中，尤其是游戏中，结合多种异步 API 是很有必要的。如作为背景的树木、流水，NPC 用 requestAnimationFrame 实现，而玩家角色，由于需要点击，再配合速度、体力、耐力等元素，其走路的速度是可变的，那么用 setTimeout 比较合适。一些非常炫的动画，可能就需要 postMessage、Image.onerror、 setImmediate、MessageChannel 等 API 了。

14.6 CSS3 transition

transition 是 CSS3 的一个重要模块，是 CSS 对入侵行为层的主要行为。W3C 标准中对它是这样描述的："CSS 的 transition 允许 CSS 的属性值在一定的时间区间内平滑地过渡。这种效果可以在鼠标单击、获得焦点、被点击或对元素任何改变中触发，并圆滑地以动画效果改变 CSS 的属性值。"

transition 主要包含四个属性值：transition-property，样式名；transition-duration，持续时间；transition-timing-function，缓动公式；transition-delay，延迟多长时间才触发。下面分别来看这四个属性值。

1. transition-property

transition-property 是用来指定当元素其中一个属性改变时执行 transition 效果，主要有以下几个值：none（没有属性改变）；all（所有属性改变），这个也是其默认值；indent（元素属性名）。当其值为 none 时，transition 马上停止执行，当指定为 all 时，元素产生任何属性值变化时都将执行 transition 效果，ident 是可以指定元素的某一个属性值。其对应的类型如下。

（1）与颜色相关的样式，如 background-color、border-color、color、outline-color 等。

（2）与盒子模块、字体大小、间距、行高有关的样式，如 word-spacing、width、vertical-align、top、right、bottom、left、padding、outline-width、margin、min-width、min-height、max-width、max-height、line-height、height、border-width、border-spacing、background-position 等。

（3）透明度，opacity。

（4）变形相关，即 transform 样式。

（5）阴影，如 text-shadow、box-shadow。

（6）线性渐变与径向渐变，用于背景色径向渐变，如-webkit-gradient、-ms-linear-gradient，各浏览器间的差异不是一般的大。

2. transition-duration

动画持续时间，单位可以是 s，也可以是 ms。我们可以连续写两个持续时间，对应两个不同的样式的变换。如：

```
transition-duration: 6s
transition-duration: 120ms
transition-duration: 1s, 15s
transition-duration: 10s, 30s, 230ms
transition-duration: inherit
```

3. transition-timing-function

缓动公式，根据时间的推进去改变属性值的变换速率。它有 6 个可能的值。

（1）ease：（逐渐变慢）默认值，ease 函数等同于贝塞尔曲线（0.25, 0.1, 0.25, 1.0）。

（2）linear：（匀速），linear 函数等同于贝塞尔曲线（0.0, 0.0, 1.0, 1.0）。

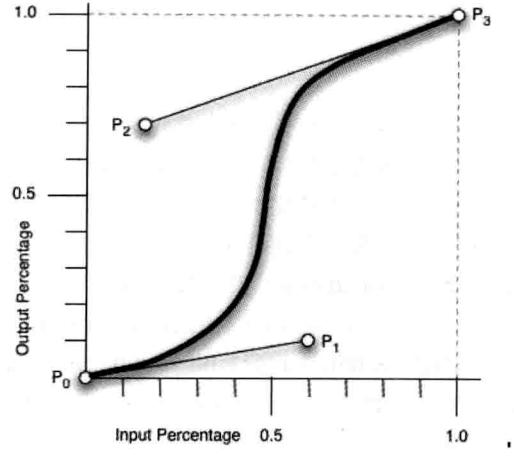
（3）ease-in：(加速)，ease-in 函数等同于贝塞尔曲线（0.42, 0, 1.0, 1.0）。

（4）ease-out：（减速），ease-out 函数等同于贝塞尔曲线（0, 0, 0.58, 1.0）。

（5）ease-in-out：（加速然后减速），ease-in-out 函数等同于贝塞尔曲线（0.42, 0, 0.58, 1.0）。

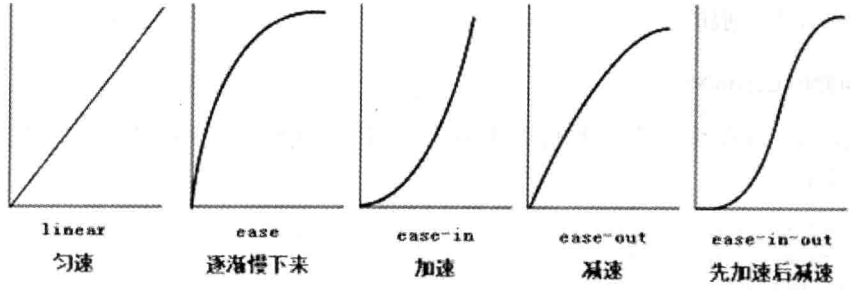
（6）cubic-bezier：（该值允许你去自定义一个时间曲线），特定的 cubic-bezier 曲线。（x1, y1, x2, y2）四个值特定于曲线上点 P_1 和点 P_2。所有值需在［0, 1］区域内，否则无效。

其中 cubic-bezier 为通过贝赛尔曲线来计算“转换”过程中的属性值，如下曲线所示，通过改变 P_1（x1, y1）和 P_2（x2, y2）的坐标可以改变整个过程的 Output Percentage。初始默认值为 default，如图 14.4 所示。



▲图 14.4

其他几个属性的示意如图 14.5 所示。



▲图 14.5

4. transition-delay

延迟执行时间。可选单位有 s 与 ms。

接着我们看如何应用。它必须是放在基于某些延迟触发的伪类或后来才添加到元素上的类名才有效。原因很简单，就是区分出初始状态与结束状态。比如一个元素的背景色原来是绿色，然后动态加了个类名或在:hover 中将它变成红色，这样 transition 才有用武之地。

```
<div id="move">移上去试试</div>
<style>
    #move {
        position: absolute;
        left:0px;
        width:100px;
        height:100px;
        background:red;
        font-size: 14px;
    }

    #move:hover {
        background:green;
        font-size: 26px;
        left:700px;
        -moz-transition: all 2s ease 0.3s;
        -webkit-transition: all 2s ease 0.3s;
        -o-transition: all 2s ease 0.3s;
        transition: all 2s ease 0.3s;
    }
</style>
```

它的支持情况如下，基本上现在我们都可以去掉私有前缀使用，如图 14.6 所示。

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	1.0 -webkit 26.0	4.0 (2.0) -moz 16.0 (16.0)	10.0	11.6 -o 12.10	3.0 -webkit

▲图 14.6

另外，浏览器还提供了一个动画结束事件给我们监听。著名的 Bootstrap 的动画就是基于 transition 的。它的动画都是很简单的淡入淡出。另一个移动库 zepto 提供了更为强大的动画封装。

动画结束事件的麻烦在于，它的名字在各浏览器内严重不一致，在 Opera 竟然存在三种写法，如图 14.7 所示。

Feature	Chrome	Firefox (Gecko)	IE	Opera	Safari (WebKit)
1.0 -webkit 26.0	4.0 (2.0) -moz 16.0 (16.0)	10	10.5 -o 12.10	3.2 -webkit	
event name	1.0 as webkitTransitionEnd 26.0	4.0 (2.0)	10	10.5 as oTransitionEnd 12 as otransitionend 12.10 as transitionend	3.2 as webkitTransitionEnd

▲图 14.7

那么如何精确取得可用的事件名呢？方法一，动态创建一个元素与一个样式表插入DOM树中，然后改变目标样式值，从而触发回调，得到事件名字。

```
(function() {
    var span = document.createElement("span");
    span.id = "mass_transition";
    span.innerHTML = "test"
    var body = document.body || document.documentElement;
    var style = document.createElement("style");
    window.transitionend;//外面调用，如果是框架，请附到框架的命名空间下
    body.appendChild(span)
    body.appendChild(style);
    style.innerHTML = "#mass_transition{ color:red;opacity:0;height:1px;overflow:hidden ;" +
          "-moz-transition: color 0.1s; -o-transition:color 0.1s;" +
          "-webkit-transition:color 0.1s; transition:color 0.1s;  }"
    "transitionend otransitionend oTransitionEnd  webkitTransitionEnd".replace(/\w+/g,
function(a) {
       span.addEventListener(a, function(e) {
          if (!window.transitionend) {
             window.transitionend = e.type;
             alert(e.type) //想测试时自己去掉
          }
       }, false)
    });
    setTimeout(function() {
       span.style.color = "black";
    });
    setTimeout(function() {
       body.removeChild(span)
       body.removeChild(style);
    }, 1000)
})()
```

方法二，从事件的构造器着手。浏览器在全局作用域下暴露了许多事件的构造器，比如MouseEvent、MessageEvent、KeyboardEvent、UIEvent、MutationEvent、PopStateEvent、CloseEvent、StorageEvent、WheelEvent、WebKitTransitionEvent、WebKitAnimationEvent 等。这些构造器的名字，如果直接传入到 createEvent 这个最底层的 API 中，不抛错，说明就支持这种事件。我们将此事件与对应类型组成个表，循环一下就能得到可用事件名了。

```
var getTransitionEndEventName = function() {
    var obj = {
       'TransitionEvent': 'transitionend',
       'WebKitTransitionEvent': 'webkitTransitionEnd',
       'OTransitionEvent': 'oTransitionEnd',
       'otransitionEvent': 'otransitionEnd'
    }
    var ret
    //有的浏览器同时支持私有实现与标准写法，比如 webkit 支持前两种，Opera 支持 1、3、4
for (var name in obj) {
       if(window[name]){
          ret = obj[name]
          break;
```

```
        }
        try {
            var a = document.createEvent(name);
            ret = obj[name]
            break;
        } catch (e) {
        }
    }//这是一个惰性函数，只检测一次，下次直接返回缓存结果
    getTransitionEndEventName = function() {
        return ret
    }
    return ret
}
alert(getTransitionEndEventName());
```

我们用 JavaScript 让一个元素动起来，然后再让它停下来。

```
<div id="test"></div>
<br/><br/><br/><br/><br/>
<button id="run">开始动画</button>
<button id="stop">中断动画并移除 transition 效果</button>
<button id="invalid">这次赋值将不会出现动画</button>
<style>
    #test{
        width:100px;
        height:100px;
        position: absolute;
        background: red;
        left:0;
        -moz-transition: left 5s;
        -o-transition:left 5s;
        -webkit-transition:left 5s;
        transition:left 5s;
    }
</style>
<script>
    var $ = function(a) {
        return document.getElementById(a)
    }
    var el = $("test")
    $("run").onclick = function() {
        el.style.left = "700px"
    }
    $("stop").onclick = function() {
        var left = window.getComputedStyle(el, null).left;
        el.style.left = left; //暂停
        ["", "-moz-", "-o-", "-webkit-"].forEach(function(prefix) {
            el.style.removeProperty(prefix + "transition")
        })
    }
    $("invalid").onclick = function() {
        el.style.left = "340px";
    }
</script>
```

上面的实验暴露出 transition 的弱点了，虽然暂停时我们是通过取当时值再重赋的手段实现了，但只要 transition 这个样式没有被清除掉，那么每一次变动 left 这个样式值都会发生动画。而想移除 transition 这个样式，唯有它是写在标签时，我们才能通过 removeProperty 的方法实现，如果它是定义于内部样式或外部样式，那就不好处理了。如果外部样式还是跨域了，那么删除样式规则的这一手段也失灵了。因此这动画的 transition 可控度太差，不适宜作为一个框架的动画引擎的实现手段。

14.7 CSS3 animation

animation 是 CSS3 另一个重要的模块，它成形得比 transition 晚，吸引了 Flash 的关键帧的理念，并克服了 transition 的一些缺陷，实用性非常高。

animation 是一个复合样式，它可以细分为 8 个更细的样式，情况与 background 和 background-color、background-position、background-repeat、background-attachment、background-image 的关系相仿。

1. animation-name

它所制约的关键帧样式规则的名字，关键帧样式规则也就是以@keyframes 开头的样式规则。animation-name 可以同时对应多个关键帧样式规则名，以“,”号分开，说明此样式规则对多个关键帧样式规则都有效。

2. animation-duration

动画持续时间。单位是为 s 或 ms，与 transition 相仿。

3. animation-timing-function

缓动公式，请参看 transition 同名项目的讲解。

4. animation-delay

动画延迟多久才开始，此时间不计入 animation-duration。

5. animation-iteration-count

动画播放次数，值可以为正整数或 infinite，默认只执行一次。这是一个很好的设计，不像 transition，只要对指定样式重新赋值，就会触发意外的动画执行。

6. animation-direction

动画执行的方向，有四个值：normal、alternate、reverse、alternate-reverse。normal 是指每次都是从第 1 帧（@keyframes 中 0%或 from，不写，浏览器会自动补上）开始。alternate，这个值显然在 animation-iteration-count 的值大于 1 时才有效，它是指动画像钟摆一样从 0%到 100%，下次再

从 100%到 0%，再一次又是 0%到 100%……reverse，这个有兼容问题，animation 最早是 Safari 发明的 CSS 样式，它最早制定的规范中并没有 reverse 这个值，它的行为与 normal 相反，每次都是从 100%开始。alternate-reverse，也有兼容性问题，行为也是一个钟摆，第一次从 100%摇到 0%，下次是 0%到 100%……

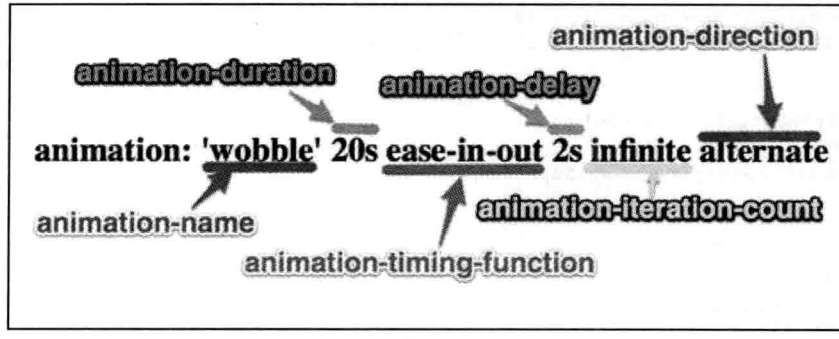
7. animation-fill-mode

指动画跑完一圈（从 0%到 100%或从 100%到 0%）后，是保持动画前的状态 forwards，还是此时的状态 backwards。

8. animation-play-state

用于暂停（paused）或继续（running）此动画。

除了最后两个，前六个可以连写在一块，如图 14.8 所示。



▲图 14.8

此外，animation 还配套了 3 种事件，分别用于开始时（animationstart）、结束时（animationend）、重复播放时（animationiteration）。当然，这些事件名可能不能直接用，需要加上私有前缀。基于上一节对 transitionend 取名字的经验，这也不是什么难事。

```
var getAnimationEndEventName = function() {
    //大致上有两种选择
    //IE10+, Firefox 16+ & Opera 12.1+: animationend
    //Chrome/Safari: webkitAnimationEnd
    //http://blogs.msdn.com/b/davrous/archive/2011/12/06/introduction-to-css3-animat
ions.aspx
    //IE10 也可以使用 MSAnimationEnd 监听，但是回调里的事件 type 依然为 animationend
    //  el.addEventListener("MSAnimationEnd", function(e) {
    //    alert(e.type)// animationend!!!
    // })
    var obj = {
        'AnimationEvent': 'animationend',
        'WebKitAnimationEvent': 'webkitAnimationEnd'
    }
    var ret ;
    for (var name in obj) {
```

```
        if(window[name]){
            ret = obj[name];
            break;
        }
    }//这是一个惰性函数，只检测一次，下次直接返回缓存结果
    getAnimationEndEventName = function() {
        return ret;
    }
    return ret;
}
```

以下代码可以不断将一个正方体变成圆形，然后圆形再变成正方形，周而复始……

```
<div id="test"></div>
<style>
    #test{
        width:100px;
        height:100px;
        background: red;
        -webkit-animation: circle 1s infinite alternate;
        animation: circle 1s infinite  alternate;

    }
    @-webkit-keyframes circle{
        100%{
        -webkit-border-radius: 50px;
    }
    }
    @keyframes circle{
        100%{
        border-radius: 50px;
    }
    }
</style>
```

14.8 mass Framework 基于 CSS 的动画引擎

本章最后一节，我们尝试利用 CSS3 animation 做个动画引擎。基于浏览器的动画 API，性能比较高，尤其在移动端，它的优势就更明显了。由于 animation 在 IE10 才支持，因此如果想应用于 PC 端，自己要做一下适配。如果条件不满足，则退回基于 JavaScript 的动画引擎。如果我们的框架是基于 AMD，这个实现起来就很简单了。

首先，我们看一下判定条件，方便切换。上面说过，浏览器把所有事件类型的构造器都放在 window，只不过不可遍历，我们用 Object.getOwnPropertyNames 加 filter 一下子就能得到所有事件构造器。不难看出，只要存在 window.AnimationEvent 或 winodw.WebKitAnimationEvent 就可以使用我们基于 CSS 的动画引擎。另一个方法是判定有没有 keyframe 样式规则的构造器。它也是放在 window 上，我们利用短路或把它所有可能的名字都放在一起就能判定出来:

var ok = window.MozCSSKeyframeRule || window.WebKitCSSKeyframeRule || window.CSSKeyframeRule;

用 CSS 实现动画引擎，有几个好处：它自带了缓动参数给你用；不用你计算原始值，它自行内部计算；颜色值不用你转换为 RBG 数组；如果想做倒带动画，那么直接设置 animation-iteration-count 为 2，animation-direction 为 alternate 就行了；像 hide 这个特效需要我们在动画结束时，将原来的进行动画的样式还原为初始值，在 CSS3，我们只需 animation-fill-mode 设置为 backwards；至于暂停与继续，其实就是控制 animation-play-state 的事。

与 JavaScript 动画引擎，CSS 动画引擎在操作元素进行动画是通过添加类名与插入样式规则实现的。由于已进入 IE10 时代，我们可以直接使用 el.classList.add 来添加类名，动态插入样式可以有点偏门，但在支持 animation 的浏览器中，相关 API 已经没有兼容性问题了。

在浏览器中，有两个元素能生产样式表，link 与 style。我们可以访问其 sheet 访问其样式表对象，然后在它的下面有个 CSSRules 类数组对象，里面就包含所有样式规则。为了方便操作，我们把动画引擎自己产生的样式规则全部放到一个动态插入的 style 元素中，以后删除就在这个元素里面找，这样可以减少一重遍历。而样式规则至少有五种类型，看以下代码。

```
<style type="text/css" id="aaa">
    .move {
        animation: move 4s linear;
    }
    @keyframes move {
        from { margin-left:-20%; }
        to { margin-left:100%; }
    }
    @font-face {
        font-family: 'YourWebFontName';
        src: url('YourWebFontName.eot?') format('eot');/*IE*/
        src:url('YourWebFontName.woff') format('woff'), url('YourWebFontName.ttf') format
('truetype');/*non-IE*/
    }
    @media screen {
        #element { background:lightgreen;
    }
</style>
```

从上到下依次是 CSSStyleRule、CSSKeyframesRule、CSSFontFaceRule、CSSMediaRule。别忘了，CSSKeyframesRuke 还镶嵌着以百分比命名的 CSSKeyframeRule。CSSStyleRule 是最早的类型，我们可以通过其 selectorText 取得指定的样式规则，比如这里 selectorText 为.move。CSSKeyframes-Rule 就是以@keyframes(视浏览器也可能是@-webkit-keyframe、@-moz-keyframe)开头的样式规则，我们可以通过专有的 name 属性判定。它里面指定进度该呈现的样式规则，用户在定义可能用到 to、from，但到 DOM 时全部转换为百分比了，它们可以通过 keyText 属性进行区分。CSSFontFaceRule 用于加载自定义字体，CSSMediaRule 则应用于著名的响应式布局，这两个都没有什么名字可区分。不过没关系，我们只用到最前面两个。

下面 mass Framework 用于操作样式规则的方法。

```
//https://github.com/RubyLouvre/mass-Framework/blob/master/fx_neo.js
var styleElement;

function insertCSSRule(rule) {
```

```
    //动态插入一条样式规则
    if (styleElement) {
        var number = 0;
        try {
            var sheet = styleElement.sheet;// styleElement.styleSheet;
            var cssRules = sheet.cssRules; // sheet.rules;
            number = cssRules.length;
            sheet.insertRule(rule, number);
        } catch (e) {
            $.log(e.message + rule);
        }
    } else {
        styleElement = document.createElement("style");
        styleElement.innerHTML = rule;
        document.head.appendChild(styleElement);
    }
}

function deleteCSSRule(ruleName, keyframes) {
    //删除一条样式规则
    var prop = keyframes ? "name" : "selectorText";
    var name = keyframes ? "@keyframes " : "cssRule ";//调试用
    if (styleElement) {
        var sheet = styleElement.sheet;// styleElement.styleSheet;
        var cssRules = sheet.cssRules;// sheet.rules;
        for (var i = 0, n = cssRules.length; i < n; i++) {
            var rule = cssRules[i];
            if (rule[prop] === ruleName) {
                sheet.deleteRule(i);
                $.log("已经成功删除" + name + " " + ruleName);
                break;
            }
        }
    }
}

function deleteKeyFrames(name) {
    //删除一条@keyframes 样式规则
    deleteCSSRule(name, true);
}
```

接着是引擎的主函数，$.fn.animtate，我们要设计与 jQuery 保持一致，降低学习成本，也意味着需要花许多代码处理参数多态化。还要考虑如何实现排队。以前我们是放在一个中央队列中，每一个元素都是一个很复杂的对象，用于分解成关键帧。现在我们没有必要搞队列了，我们可以在元素的 animationend 回调中自动执行下一个动画，所有排队的动画全部放到元素对应的缓存体中就行了。

```
function addOption(opts, p) {
    switch (typeof p) {
        case "object":
            $.mix(opts, p);
            delete p.props;
            break;
```

```
            case "number":
                opts.duration = p;
                break;
            case "string":
                opts.easing = p;
                break;
            case "function":
                opts.complete = p;
                break;
        }
    }

    function addOptions(duration) {
        //这里与 JavaScript 动画引擎大同小异
        var opts = {};
        for (var i = 1; i < arguments.length; i++) {
            addOption(opts, arguments[i]);
        }
        duration = opts.duration;
        duration = /^\d+(ms|s)?$/.test(duration) ? duration + "" : "1000ms";
        if (duration.indexOf("s") === -1) {
            duration += "ms";
        }
        opts.duration = duration;
        opts.effect = opts.effect || "fx";
        opts.queue = !!(opts.queue == null || opts.queue); //默认使用队列
        opts.easing = easingMap[opts.easing] ? opts.easing : "easeIn";
        return opts;
    }
```

上面用到一个 easingMap 对象，里面包含 Robert Penner 整理的所有缓动公式名及其对应的贝塞尔曲线实现。由于 CSS 动画引擎体积很少，我们有足够空间将它们全部打包。

```
var easingMap = {
    "linear": [0.250, 0.250, 0.750, 0.750],
    "ease": [0.250, 0.100, 0.250, 1.000],
    "easeIn": [0.420, 0.000, 1.000, 1.000],
    "easeOut": [0.000, 0.000, 0.580, 1.000],
    "easeInOut": [0.420, 0.000, 0.580, 1.000],
    //……略……
    "custom": [0.000, 0.350, 0.500, 1.300],
    "random": [Math.random().toFixed(3),
        Math.random().toFixed(3),
        Math.random().toFixed(3),
        Math.random().toFixed(3)]
}
```

此外，外国还有些网站提供可视化的界面让你自己设计喜欢的曲线，比如：

http://www.roblaplaca.com/examples/bezierBuilder/

http://cubic-bezier.com/

接着就是 3 个重要的内容函数：startAnimation、nextAnimation 与 stopAnimation。

startAnimation，用于立即执行此元素的动画，具体做法是分解原始材料构建两个样式规则，一个是用于集中定义动画的运作情况，另一个是定义第一帧与最后一帧的样式情况。第一个样式规则

是普通的 CSSStyleRule，selectorText 为一个类名，方便添加到目标元素上，另一个不用说是 CSSKeyframesRule。由于是多个元素共用此类名，如果样式表有此类名，我们就不用重复分解与插入了。这里我们可以做个 flag！最后我们要绑定一下 animationend 事件，在它的回调中保存指定样式到元素的 style 中，然后移除类名（因为类名对应的样式规则迟早会被移除，因此必须将它们转移到内联样式里），并调用 nextAnimation 与 stopAnimation。

nextAnimation 决定是否调用 startAnimation，里面有个 setTimeout，用于模拟 delay 效果。

stopAnimation 用于移除 startAnimation 插入的两个样式规则。

```
var AnimationRegister = {};
function startAnimation(node, id, props, opts) {
    var effectName = opts.effect;
    var className = "fx_" + effectName + "_" + id;
    var frameName = "keyframe_" + effectName + "_" + id;
    var hidden = $.css(node, "display") === "none";
    var preproccess = AnimationPreproccess[effectName];
    if (typeof preproccess === "function") {
        var ret = preproccess(node, hidden, props, opts);
        if (ret === false) {
            return;
        }
    }
    //各种回调
    var after = opts.after || $.noop;
    var before = opts.before || $.noop;
    var complete = opts.complete || $.noop;
    var from = [],
            to = [];
    var count = AnimationRegister[className];
    node[className] = props;//保存到元素上，方便 stop 方法调用
    //让一组元素共用同一个类名
    if (!count) {
        //如果样式表中不存在这两条样式规则
        count = AnimationRegister[className] = 0;
        $.each(props, function(key, val) {
            var selector = key.replace(/[A-Z]/g, function(a) {
                return "-" + a.toLowerCase();
            });
            var parts;
            //处理 show、toggle、hide 3 个特殊值
            if (val === "toggle") {
                val = hidden ? "show" : "hide";
            }
            if (val === "show") {
                from.push(selector + ":0" + ($.cssNumber[key] ? "" : "px"));
            } else if (val === "hide") { //hide
                to.push(selector + ":0" + ($.cssNumber[key] ? "" : "px"));
            } else if (parts = rfxnum.exec(val)) {
                var delta = parseFloat(parts[2]);
                var unit = $.cssNumber[key] ? "" : (parts[3] || "px");
                if (parts[1]) { //操作符
                    var operator = parts[1].charAt(0);
```

```
                var init = parseFloat($.css(node, key));
                try {
                    delta = eval(init + operator + delta);
                } catch (e) {
                    $.error("使用-=/+=进行递增递减操作时,单位只能为 px, deg", TypeError);
                }
            }
            to.push(selector + ":" + delta + unit);
        } else {
            to.push(selector + ":" + val);
        }
    });
    var easing = "cubic-bezier( " + easingMap[opts.easing] + " )";
    //CSSStyleRule 的模板
    var classRule = ".#{className}{#{prefix}animation:#{frameName} #{duration}#{easing} " +
            "#{count} #{direction}; #{prefix}animation-fill-mode:#{mode}  }";
    //CSSKeyframesRule 的模板
    var frameRule = "@#{prefix}keyframes #{frameName}{ 0%{ #{from}; } 100%{  #{to}; }  }";
    var mode = effectName === "hide" ? "backwards" : "forwards";
    //填空数据
    var rule1 = $.format(classRule, {
        className: className,
        duration: opts.duration,
        easing: easing,
        frameName: frameName,
        mode: mode,
        prefix: prefixCSS,
        count: opts.revert ? 2 : 1,
        direction: opts.revert ? "alternate" : ""
    });
    var rule2 = $.format(frameRule, {
        frameName: frameName,
        prefix: prefixCSS,
        from: from.join("; "),
        to: to.join(";")
    });
    insertCSSRule(rule1);
    insertCSSRule(rule2);
}
AnimationRegister[className] = count + 1;
$.bind(node, animationend, function fn(event) {
    $.unbind(this, event.type, fn);
    var styles = window.getComputedStyle(node, null);
    // 保存最后的样式
    for (var i in props) {
        if (props.hasOwnProperty(i)) {
            node.style[i] = styles[i];
        }
    }
    node.classList.remove(className); //移除类名
    stopAnimation(className);         //尝试移除 keyframe
    after(node);
    complete(node);
    var queue = $._data(node, "fxQueue");
```

```
            if (opts.queue && queue) { //如果在列状,那么开始下一个动画
                queue.busy = 0;
                nextAnimation(node, queue);
            }
        });
        before(node);
        node.classList.add(className);
    }
```

这里面有两个 flag，AnimationRegister 里面装着许多类名，类名的值为数字，表示有多少个元素在共用它。我们只有在这个值为零时进行分解与插入样式规则。然后每当动画结束时，这个值就减一，归零时我们就移除它们。第二个 flag 是缓存体中的动画队列中 busy，进行动画或被延迟时为真值，其他时间为假，我们只有在假时，才能进入执行 startAnimation 的分支。

在 startAnimation 里，有时还会调用 AnimationPreproccess 里面的预处理函数，因此 CSS3 规定 display 为 none 的元素无法进行动画，因此我们想实现 show 特效，在提前修改 display 值。hide 特效也要求我们对 overflow 做些处理。总体上说比 JavaScript 动画引擎简单多了。

nextAnimation 与 stopAnimation 的源码如下。

```
function nextAnimation(node, queue) {
    if (!queue.busy) {
        queue.busy = 1;
        var args = queue.shift();
        if (isFinite(args)) {//如果是数字
            setTimeout(function() {
                queue.busy = 0;
                nextAnimation(node, queue);
            }, args);
        } else if (Array.isArray(args)) {
            startAnimation(node, args[0], args[1], args[2]);
        } else {
            queue.busy = 0;
        }
    }
}

function stopAnimation(className) {
    var count = AnimationRegister[className];
    if (count) {
        AnimationRegister[className] = count - 1;
        if (AnimationRegister[className] <= 0) {
            var frameName = className.replace("fx", "keyframe");
            deleteKeyFrames(frameName);
            deleteCSSRule("." + className);
        }
    }
}
```

最后，我们看看 delay, pause, resume 这几方法是如何实现的。

```
var playState = $.cssName("animation-play-state");
$.fn.delay = function(number) {
    return this.fx(number);
};
```

```
$.fn.pause = function() {
    return this.each(function() {
        this.style[playState] = "paused";
    });
};
$.fn.resume = function() {
    return this.each(function() {
        this.style[playState] = "running";
    });
};
```

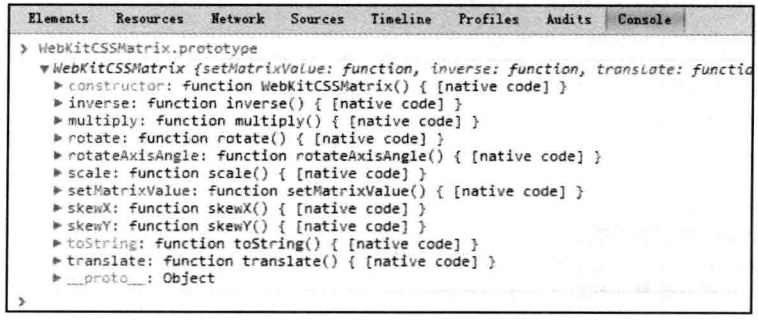
一切就是这么简洁。

当然，基于 CSS 的动画引擎不是没有缺点，比如它对 scrollTop、scrollLeft 的动画就无能为力，它们是元素的属性。此外，我们也无法对 canvas 元素里面的矢量图形进行动画。要想打包这一切，我们需要一个更强大的动画引擎。但 canvas 还涉及 stage 等概念，个人觉得还是针对它打包专用的引擎比较好。

有了基于 CSS 的动画引擎，我们就可以安心使用 CSS3 的 transform2D 或 3D，要不用 JavaScript 来实现，在旧版本 IE 下连 2D 也很勉强。因为浏览器对元素进行变形，是基于矩阵，而不是 rotate、scale、translate 等原始函数。标准浏览器好办，getComputedStyle 直接转换，IE 需要自己提取参数值，转换角度为弧度，使用万能矩阵滤镜也惨兮兮的，另外，变形中心（transform-origin）也很难调校。如果两个连续的动画都涉及变形，那就是矩阵相乘。这里哗啦啦就要几百行，来实现矩阵的加减乘除……

矩阵相乘后，我们还得把这些值还原为 rotate、scale、translate、skew 等方法的传参。在 http://www.w3.org/TR/css3-transforms/这里，我们可以看到如何分解与还原矩阵的伪代码。显然，这一切加起来，让我们的引擎增加千行。这还不算 transform3D。不过，它本来就无法在旧版本 IE 下实现。

在 webkit 与 IE10 中，各提供了一个 CSSMatrix 类（WebKitCSSMatrix 与 MSCSSMatrix），里面包含了我们所有想要的方法，否则自己实现有点残酷，如图 14.9 所示。



▲图 14.9

目前只是视 IE11 是否支持 WebGL 了，如果支持，到时肯定会提供更多的矩阵方法，我们的引擎玩 3D 才有意义。

第15章 插件化

插件化与模块化看起来很像，都是在一个系统上添加新功能。有人说，插件化是对已有功能进行扩展，比如 jQuery 的 css 方法，它里面有个 cssHooks，里面可以加一些新函数，用于支持取得一些偏门的样式，而模块化则是添加一些之前系统没有的新功能。但这样说来，jQuery 那上万个插件岂不是要归类为模块。显然，大多数人是从另一个角度来看——模块化是从开发流程上说，插件是从功能上说。模块化是把一堆相同的接口打包在一起，它们可能会依赖于其他模块，但总的来说，只会返回一个对象或函数供其他人调用。我们只需关注它如何被加载就行了。插件化，需要系统与插件间有一套规范，让插件集中放到某个位置，一个对象或一个数组，方便让我们一下子扫描出来，因此有的模块可能包含某个方法的 N 个插件。

15.1 jQuery 的插件的一般写法

jQuery 有成千上万个插件，经过这么多年的实践，人们早已摸索出一个套路出来。不过，jQuery 本来就为此做准备，其中 extend 方法就是一个很好的设计。extend 同时存在于命名空间与原型中，而 jQuery 的原型有个便捷的名字，叫 fn，因此为它扩展一个新的原型方法，可以直接$.fn.xxx 也可以$.fn.extend({a:fnc})。得益于 IIFE（立即调用函数表达式）的发掘，我们可以最大限度地减少全局污染。

```
(function($) {//这个东西叫 IIFE
    //扩展这个方法到 jQuery
    $.fn.extend({
        //插件名字
        pluginname: function() {
            //遍历匹配元素的集合
            return this.each(function() {
                //在这里编写相应代码进行处理
            });
        }
    });

 //传递 jQuery 到内层作用域去，如果 window、document 在里面用得多，也可在这里传入
})(jQuery);
```

由于 jQuery 是集化操作，应该一个元素对应一个配置对象，我们可以使用$.extend({}, defaults,

options)来为每个元素分配独立的配置对象。defaults 为默认配置对象，方便我们什么对象也不传时，插件也能正常运作。如果配置对象比较复杂，对象的属性也是一个对象，那么必要时我们可以使用深拷贝，让它的第一个参数为 true 就行。如果你习惯使用面向过程的方式写码，那么接下来在 each 迭代器里面加东西就可以，但插件规模很大，我还是愿意使用面向对象的写法。我们在 IIFE 里面编写一个类，然后在 each 迭代器内为类传入当前元素与私有配置对象来实例化。然后，用户在 domReady 通过$(“.selector”).plugname()的方式使用，有时传参，有时没传参。因此我们需要区分一下这个元素对应的 UI 实例化了没有，这时就要用到数据缓存系统了。

```
(function($) {//这个东西叫 IIFE
    //扩展这个方法到 jQuery
    var Plugin = function() {
    }
    Plugin.prototype = {};
    $.fn.extend({
        //插件名字
        pluginname: function(options) {//用户的统一配置对象或方法名
            //遍历匹配元素的集合
            var args = [].slice.call(arguments, 1)
            return this.each(function() {
                //在这里编写相应代码进行处理
                var ui = $._data(this, pluginname);
                if (!ui) {
                    var opts = $.extend(true, {}, $.fn.pluginname.defaults,
                            typeof options === "object" ? options : {});
                    ui = new Plugin(opts, this);
                    $._data(this, pluginname, ui);
                }
                if (typeof options === "string" && typeof ui[options] == "function") {
                    ui[options].apply(ui, args);//执行插件的方法
                }
            });
        }
    });
    $.fn.pluginname.defaults = {/*略*/};//默认配置对象
    //传递 jQuery 到内层作用域去
})(jQuery);
```

这种结构最著名的执行者是 jQuery tools。boostrap 在这基础上进行了 3 大改进。首先是插件的无冲突处理，因为 jQuery 的插件太多了，好用的名字早就有人占去。它使用 noConflict 进行规避。以前这东西是 jQuery 防止别的框架与它争$用的，这个用在插件上一样很有用。第二项就是将内部构造器放到$.fn.pluginname.Constructor 上，方便人们扩展这个类。第三个就是利用事件代理自动初始化实例，目的是让用户只需引用 JavaScript 与按照文档编写 HTML，不用写一行 JavaScript 代码就能让插件运行起来。

Bootstrap 流行起来后，jQuery 插件又开始模拟它那种只需引用 JavaScript 就能用的编写方式。原理是，这些插件最后几行都是一些事件代理，当用户触发某些事件，就会自动实例化它们。因此用户不用写一行 JavaScript 代码，只要引入 JavaScript 文件，HTML 按照某些模式来编写，标签上

有着指定的类名就能运行了。

Bootstrap 的 Dropdown 插件的主体骨架如下。

```
//https://github.com/twitter/bootstrap/blob/master/js/bootstrap-dropdown.js
!function($) {
    "use strict"; // ecma262v5 的新东西,强制使用更严谨的代码编写
    /* 内部工作的类
     * ========================== */
    var toggle = '[data-toggle=dropdown]';
    var Dropdown = function(element) {
        var $el = $(element).on('click.dropdown.data-api', this.toggle);
        $('html').on('click.dropdown.data-api', function() {
            $el.parent().removeClass('open');
        });
    };
    Dropdown.prototype = {
        constructor: Dropdown,
        toggle: function(e) {
            /*略*/
        },
        keydown: function(e) {
            /*略*/
        }
    };
    /* 主接口
     * ========================== */
    var old = $.fn.dropdown;
    $.fn.dropdown = function(option) {
        return this.each(function() {
            var $this = $(this),
                    data = $this.data('dropdown');
            if (!data)
                $this.data('dropdown', (data = new Dropdown(this)));
            if (typeof option === 'string')//调用它的实例方法
                data[option].call($this);
        });
    };

    $.fn.dropdown.Constructor = Dropdown;//暴露类名

    /* 无冲突处理
     * ==================== */
    $.fn.dropdown.noConflict = function() {
        $.fn.dropdown = old;
        return this;
    };
    /*事件代理，智能初始化
     * ================================== */
    $(document)
            .on('click.dropdown.data-api', clearMenus)
            .on('click.dropdown.data-api', '.dropdown form', function(e) {
        e.stopPropagation();
    })
```

```
            .on('click.dropdown-menu', function(e) {
        e.stopPropagation();
    })
            .on('click.dropdown.data-api', toggle, Dropdown.prototype.toggle)
            .on('keydown.dropdown.data-api', toggle + ', [role=menu]',
            Dropdown.prototype.keydown);

}(window.jQuery);
```

我们可以把这两个看作是编写 jQuery 插件的最佳实践。插件名即新增的原型函数名，内部类使用，默认参数，智能初始化。

15.2 jQuery UI 对内部类的操作

作为官方出的东西，jQuery 一直不为人们所看重，一是它没有 datagrid、tree 等 UI 库必备的东西，二是它修改太过频繁，体积庞大。它所有以 ui 开头的插件其实都来自社区，为统一它们的接口，jQuery 在这耗时过久。在 jqury1.9 中它共有 accordion、autocomplete、button、dateplcker、dialog、menu、spinner、tabs、slider、tooltip 十个 UI，其中最受欢迎的 dateplcker 日历组件还没有统一化，其他都是基于$.Widget 构建。

```
//一个 jquery UI 的基本骨架如下，通过 widget 入口函数反刍出来，将 each 什么隐藏掉了。
$.widget("ui.button", {//jquery.ui.button.js
    version: "@VERSION",
    defaultElement: "<button>", //使用什么元素作为它的最外围元素
    options: {
        //默认参数
    },
    _create: function() {
        //根据当前元素的情况重置一些参数与绑定事件
    },
    widget: function() {
        return this.buttonElement;
    },
    _destroy: function() {
//移除各种类名，属性与事件
    },
    _setOption: function(key, value) {
    },
    refresh: function() {
        //略
    },
    //略
});
```

jQuery 团队最擅长的就是提供优雅的接口，实现我们暂且不看。15.1 节我们提前剧透了如何操作实例方法，jQuery UI 已经把这些操作规范下来。如果第一个传参为“options”就进入配置模式，来修改配置对象。如果传参与它的原型方法名一致，就调用其方法。

以当中的 accordion 为例。

初始化时传一个对象，方便设置 N 个配置项。

```
$( ".selector" ).accordion({ heightStyle: "fill" ,{ active: 2 }});
```

初始化后，第一个参数为字符串“option”时即进入配置模式。如果后面只有一个属性或方法名，那么就是读方法（getter）。如果它们之后还有参数，那就是写方法（setter），作为一个方法的参数或这个属性的新值。

```
// getter
var active = $( ".selector" ).accordion( "option", "active" );
// setter
$( ".selector" ).accordion( "option", "active", 2 );
```

不同的控件会有不同的方法或属性，但由于都是同一个基类，因此会有如下相同的操作。

让控件不可用。它有以下两种操作方式。

第一种，使用配置模式。

```
$( ".selector" ).accordion( "option", "disabled", true );
```

第二种，直接传入“disable”。

```
$( ".selector" ).accordion( "disable" );
```

让控件可用，也对应两种。

第一种，使用配置模式。

```
$( ".selector" ).accordion( "option", "disabled", false);
```

第二种，直接传入“enable”。

```
$( ".selector" ).accordion( "enable" );
```

销毁控件，可传入“destroy”。

```
$( ".selector" ).accordion( "destroy" );
```

如果想得此 UI 最外围的元素节点的 jQuery，可传入“widget”。

```
var widget = $( ".selector" ).accordion( "widget" );
```

至于实现，修改配置与调用方法是很容易的，它们都是走_setOptions 方法。问题在于让插件是否可用。一般地，它只对控件的类名下手，添加一个叫做 ui-state-disabled 的类名，并将 options.disable 改为 false。那么在修改配置时，就无法进入实际操作的那个分支。由于控件必然绑定了许多方法，因此它不是使用 jQuery 的 on、bind、delegate 进行绑定，而是使用一个_on 的方法。

```
$.Widget.prototype._on = function(suppressDisabledCheck, element, handlers) {
   var delegateElement,
         instance = this;
   // 第一个参数决定是否检测 disabled 状态
   if (typeof suppressDisabledCheck !== "boolean") {
      handlers = element;
```

```
            element = suppressDisabledCheck;
            suppressDisabledCheck = false;
        }

    // 处理参数多态化,可能用户不会传这么多参数,不足部分自己设计补上
        if (!handlers) {
            handlers = element;
            element = this.element;
            delegateElement = this.widget();
        } else {
            element = delegateElement = $(element);
            this.bindings = this.bindings.add(element);
        }
    //开始绑定
        $.each(handlers, function(event, handler) {
            function handlerProxy() {
    //这里的分支最关键,用于决定用户的操作是否无效化
                if (!suppressDisabledCheck &&
                        (instance.options.disabled === true ||
                                $(this).hasClass("ui-state-disabled"))) {
                    return;
                }
                return (typeof handler === "string" ? instance[ handler ] : handler)
                        .apply(instance, arguments);
            }

    // 这里就是模拟 jQuery 核心库 proxy 方法的实现,加个 UUID,方便移除
            if (typeof handler !== "string") {
                handlerProxy.guid = handler.guid =
                        handler.guid || handlerProxy.guid || $.guid++;
            }

            var match = event.match(/^(\w+)\s*(.*)$/),
                    eventName = match[1] + instance.eventNamespace,
                    selector = match[2];
            if (selector) {
                delegateElement.delegate(selector, eventName, handlerProxy);
            } else {
                element.bind(eventName, handlerProxy);
            }
        });
    }
```

有的插件的 disable 与 enable 可能非常复杂，需要专门设计一个原型方法。

jQuery 通过字符串传参来调用原型方法的设计非常绝妙，基本成为一个套路。

15.3 jQuery easy UI 的智能加载与个别化制定

由于 jQuery 官方 UI 库不好用，导致人们不得不致力于设计自己的 UI 库。在这个群雄纷争的世界，国外最顶尖的几位是 Wijmo、KendoUI、jQuery tools、bootstrap，国内则有 DWZ、easyUI、tigerUI、angelaUI、OperaMasksUI。其中对于国人而言，easyUI 应该是最熟悉的，其次是 bootstrap

这个后起之秀。但后台应用，还是 easyUI 比较有优势。

早期 easyUI 的脱颖而出，原因只是对手太过笨重或无能，但后期它渐渐有了自己的东西。标题中的两个东西就是 jQuery 1.3 新加的特征，而且是非常优势的东西。

第一个是智能加载。UI 库通常是非常庞大的，jquery UI 只有这么少插件也要你加许多东西，而且依赖关系非常复杂。easyUI 有两种应用插件的方式，一种类似于 EXT，一种类似于 boostrap。EXT 方式就免谈了。boostrap 就是要求你按照文档的范本那样写 HTML，里面有指定的类名。当中有些类名很重要，easyUI 只要求你引入核心库与 parse.js 这个文件。其中 parsejs 有一段脚本在 domReady 之后，会扫描 DOM 树，把这些带特定类名的元素全部找出来，并且根据这些类名来加载对应的 UI 插件的 JavaScript 文件，最后初始化它们。有关加载的实现与依赖关系全部写在 easyloader，虽然与 AMD 没得比，但还算凑合。

```
//jquery.parser.js
(function($) {
    $.parser = {
        auto: true, //由于加载与初始化是在 domReady 之后才开始
        //因此我们可以早早把这个改为 false,或干脆不加载这个 JavaScript 文件就行
        onComplete: function(context) {
        },
        //插件名的集合，它们与元素的那些类名同名
        plugins: ['draggable', 'droppable', 'resizable', 'pagination',
            'linkbutton', 'menu', 'menubutton', 'splitbutton', 'progressbar',
            'tree','combobox','combotree','combogrid','numberbox','validatebox','searchbox',
            'numberspinner', 'timespinner', 'calendar', 'datebox', 'datetimebox', 'slider',
            'layout','panel','datagrid','propertygrid','treegrid','tabs','accordion','window',
'dialog'
        ], parse: function(context) {
            var aa = [];
            for (var i = 0; i < $.parser.plugins.length; i++) {
                var name = $.parser.plugins[i];
                //搜索 DOM 树
                var r = $('.easyui-' + name, context);
                if (r.length) {
                    if (r[name]) {//如果 jQuery 的原型已经有这个插件方法,就实例化它
                        r[name]();
                    } else {//没有就加载它们
                        aa.push({name: name, jq: r});
                    }
                }
            }
            if (aa.length && window.easyloader) {
                var names = [];
                for (var i = 0; i < aa.length; i++) {
                    names.push(aa[i].name);
                }
                easyloader.load(names, function() {//加载好就初始化它们
                    for (var i = 0; i < aa.length; i++) {
                        var name = aa[i].name;
                        var jq = aa[i].jq;
                        jq[name]();
```

```
                }
                $.parser.onComplete.call($.parser, context);
            });
        } else {
            $.parser.onComplete.call($.parser, context);
        }
    },

    parseOptions: function(target, properties) {
    }
};
$(function() {
    if (!window.easyloader && $.parser.auto) {
        $.parser.parse();
    }
});

})(jQuery);
```

第二个好东西就是个性化制定。由于 jQuery 是集合操作，$(“.selector”)可能得到五个匹配的元素，而$(“.selector”).tabs(opts) 这样操作，其实只是对它们应用了相同的配置。有时我们需要根据元素本来的情况修改一下，jQuery 在 1.43 支持抽取 data-*属性来做配置对象，easyUI 这个新特征可以看作是其强化版，它会对 data-options 的值的两边加个花括号，然后 new Function 转换为一个配置对象。它还支持传入一个数组，通过 style 或 attr 取得指定的目标值，如果这个数组元素恰好是一个对象，那么就直接混入。

```
$.parser.parseOptions = function(target, properties) {
    var t = $(target);
    var options = {};
    var s = $.trim(t.attr('data-options'));
    if (s) {
        var first = s.substring(0, 1);
        var last = s.substring(s.length - 1, 1);
        if (first != '{')
            s = '{' + s;
        if (last != '}')
            s = s + '}';
        options = (new Function('return ' + s))();
    }
    if (properties) {
        var opts = {};
        for (var i = 0; i < properties.length; i++) {
            var pp = properties[i];
            if (typeof pp == 'string') {
                if (pp == 'width' || pp == 'height' || pp == 'left' || pp == 'top') {
                    opts[pp] = parseInt(target.style[pp]) || undefined;
                } else {
                    opts[pp] = t.attr(pp);
                }
            } else {
                for (var name in pp) {
                    var type = pp[name];
                    if (type == 'boolean') {
```

```
                    opts[name] = t.attr(name) ? (t.attr(name) == 'true') : undefined;
                } else if (type == 'number') {
                    opts[name]=t.attr(name)=='0' ? 0 : parseFloat(t.attr(name))|| undefined;
                }
            }
        }
        $.extend(options, opts);
    }
    return options;
}
```

那么针对于每一个元素，它所得到配置对象是：

newOptions = $.extend({}, $.fn.pluginName.defaults, $.parser.parseOptions(el), options)

网上有文章介绍如何利用它更换更强大的 my97 日历的。

```
http://easyui.btboys.com/esayui-extension-extension----my97.html
$.fn.my97.parseOptions = function(target) {
    return $.extend({}, $.parser.parseOptions(target, ["el", "vel", "weekMethod",
        "lang", "skin", "dateFmt", "realDateFmt", "realTimeFmt", "realFullFmt",
        "minDate", "maxDate", "startDate", {
            doubleCalendar: "boolean",
            enableKeyboard: "boolean",
            enableInputMask: "boolean",
            autoUpdateOnChanged: "boolean",
            firstDayOfWeek: "number",
            isShowWeek: "boolean",
            highLineWeekDay: "boolean",
            isShowClear: "boolean",
            isShowToday: "boolean",
            isShowOthers: "boolean",
            readOnly: "boolean",
            errDealMode: "boolean",
            autoPickDate: "boolean",
            qsEnabled: "boolean",
            autoShowQS: "boolean",
            opposite: "boolean"
        }
    ]));
};
```

easyUI 这个设计对 webpage 来说是非常贴心的。这也是它成功的原因之一。

15.4 更直接地操作 UI 实例

jquery tools 如果配置 API 为 true，那么它会返回最后一个匹配元素对应的 UI 实例。对应的源码如下。

```
//https://github.com/jquerytools/jquerytools/tree/master/src/tooltip
$.fn.tooltip = function(conf) {
    //已保存就返回，由于 data 原型方法是 get first set all 操作，这时是返回第一个 UI
    var api = this.data("tooltip");
```

```
    if (api) {
        return api;
    }
    //合并配置
    conf = $.extend(true, {}, $.tools.tooltip.conf, conf);
    if (typeof conf.position === 'string') {
        conf.position = conf.position.split(/,?\s/);
    }
    //实例化每个元素的 UI，返回最后一个，因此与上方矛盾，小心被坑
    this.each(function() {
        api = new Tooltip($(this), conf);
        $(this).data("tooltip", api);
    });
    //决定是返回最后一个实例还是进行链式操作
    return conf.api ? api : this;
};
```

显然直接操作实例，比如 jQuery UI 那种传字符串的形式更直观些。但只处理一个，并且是最后一个，有违我们的常识。虽然这个只要保证我们每次且选择一个元素进行 UI 操作就安全，但这个通常保证不了。简单修正如下（略过大多数逻辑）。

```
$.fn.tooltip = function(opt) {
    var api = [];
    this.each(function() {
        //全部保存到一个数组里
        api.push(new tooltip());
    });
    //这时 jQuery 对象里面就是一个个 UI，而不是节点
    return opt.api ? $(api) : this;
}
    那么下次就直接在 each 迭代器内用 UI 的原型方法了！
var tooltip = $('#sample1,#sample2').tooltip({ api : true });
tooltip.each(function(index){
    this.show();//this 为一个个 UI 实例
    this.hide();
});
```

但这也产生一个违和点，返回的 jQuery 对象不能直接操作 UI 方法。

```
var tooltip = $('#sample1,#sample2').tooltip({ api : true });
alert(tooltip.show) // undefind

tooltip[0].show();
tooltip.get(1).show();
```

因此又一个 jQuery 插件产生了！

```
(function($) {
    $.ui = $.ui || {};
    $.ui.api = function(opts) {
        var api = $(opts), first = opts[0];
        for (var name in first)
            (function(name) {
```

```
            if (typeof first[name] === "function")
                api[ name ] = (/^get[^a-z]/.test(name)) ?
                        function() {//作为一个约定，get 开头的方法只处理第一个元素
                            //与 jQuery 的 get first set all 原则一致
                            return first[name].apply(first, arguments);
                        } :
                        function() {//否则处理所有匹配元素
                            var arg = arguments;
                            api.each(function(idx) {
                                var apix = api[idx];
                                apix[name].apply(apix, arg);
                            })
                            return api;
                        }
        })(name);//遍历 UI 实例对象，取得其原型方法
        return api;
    }
})(jQuery);
```

那么上面的 tooltip 改成这样：

```
$.fn.tooltip = function(opt){
    var api = [];
    this.each(function(){
        api.push( new tooltip() );
    });
    //返回一个强化版 jQuery 对象
    return opt.api ? $.ui.api(api) : this;
}
```

使用如下。

```
//返回一个包含 UI 实例的 jQuery 对象
var tooltip = $('#sample1,#sample2').tooltip({api: true});

//同时处理#sample1,#sample2
tooltip.show();

//链式调用
tooltip.show().hide();
```

个人感觉以这种形式操作 UI 对象比 jQuery UI 的更优雅。读者可以把本书提到的四种东西都用到自己的 UI 库内。

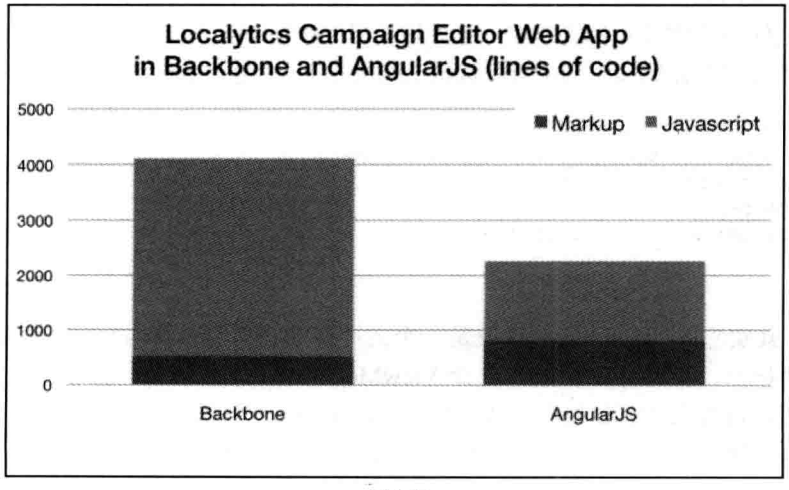
第16章 MVVM

人们总是爱探求完美的东西，编程界也有自己的追求，如完美的架构从 MVC 到 MVP，再到 MVVM 当然 MVC、MVP、MVVM 有它们不同的场景，但 MVVM 在微软试水后已被证实为界面开发最好的方案。

当 MV*应用到前端时，这个 V 与传统意义上的 V 已经不一样了。在后端，这只是字段串拼接，在前端，还涉及 DOM 操作。即便你用的是字符串模板，也要将 script、noscript、text 中取得内容，结合后端的返回数据，生成一个符合 HTML 结构的字符串，最后通过 innerHTML 或 createContextual Fragment 方法转换为节点，MVVM 特有的动态模板，是活生生对已有的文本节点、特性节点、注释节点的 nodeValue 进行抽取，再加工，再进行 DOM 操作。

此外，MVVM 另一个重要特性，双向绑定，更是方便你同时维护页面上都依赖某个字段的 N 个区域，而不是自己手动更新它们。即便是使用构建在选择器引擎的 jQuery、Prototypejs 等框架，也不得不写许多类名做钩子以及查找它们的代码及后续处理代码。MVVM 把这些都隐藏了，你只需专注于业务。外国有文章介绍，使用了 AngularJS（MVVM）代替（Backbone），代码减少了一半，如图 16.1 所示。

http://www.localytics.com/blog/2013/angularjs-at-localytics/



▲图 16.1

本章将深入介绍 MVVM 的运作机理，并教导大家开发一个简单的 MVVM。

16.1 当前主流 MVVM 框架介绍

MVVM 是一个很新的东西，在后端诞生于 2005 年，前端最早的 MVVM 框架 knockout 在 2010 年发布。由于里面的核心理念比较深奥，实现上也非常晦涩，涉足这个领域的人（指框架开发者，非使用者）很少，目前数量很少，主要有 knockoutjs、emberjs、 angularjs、 winjs、kendoui 还有本章讲解的迷你 MVVM 框架 avalon。在这些框架中，knockoutjs、winjs、kendoui 都有微软的影子，emberjs 是一位超级程序员开发，angular 是 Google 主持开发，不过 angular 到 1.0 才有 MVVM 的影子。

knockoutjs 基本上把 WPF 大多数概念搬过来，如属性绑定、集合绑定，对于命令绑定，则使用事件绑定取替，有原生的就用原生的，没有原生的就外包一层，做兼容处理。它的几个核心概念如下。

监控属性，Observables，可以来自 model 的字段与业务过程需要用到的中间量，比如表单中的 username、password，在后端的数据库的表里都有这些字段，而中间量是指重新输入密码这个 input 控件对应的值。

依赖监控属性，Dependent Observables，是基于两个或两个以上监控属性构建的属性。它自己本身是没有值的，它的值是“依赖”在其他对象的属性值上、通过 Binding 的传递和转换而得来的。比如 fullName 就是基于 firstName 与 lastName，在线用户数则是基于用户表中的某个属性来统计得到的长度。

监控数组：observableArray，对一个数组进行监控，通常是监听其元素的位置变化及数组的长度。至于元素的属性变化，由元素对应的 ViewModel 来处理。

声明式绑定：它拥有两种绑定形式，一种是以 data-bind 的特性节点做宿主，一种以 ko 开头的注释节点，knockout 称之为虚拟结点。比如说某表格根据某个集合做两重以上 foreach 循环赋值，由于 tr、tbody 之间不给你插入其他节点，这时只能用上注释节点了。注释节点在 IE6、IE7 中会引发 BUG，并且不会呆在你编写它的位置。因此只有 knockout 用它做宿主。

```
<p>Your value: <input data-bind="value: someValue, valueUpdate: 'afterkeydown'"/></p>
<p>You have typed: <span data-bind="text: someValue"></span></p>
<script type="text/javascript">
    var viewModel = {
        someValue: ko.observable("edit me")
    };
</script>
```

仔细观察，其实它的 data-bind 属性就是一个去掉两端花括号的 JavaScript 对象字面量。解析时，只要在两端补上，上面用 with 劫持几个 ViewModel 做上下文对象，就可以变成一个对象了。不过实际操作还是很复杂的，如果键名为关键字，new、float 什么的，旧版本 IE 会报错，因此你还要补上双引号。这个设计也不太好，它只是在扫描绑定时轻松一点，对于内部实现或是用户使用都不友好。

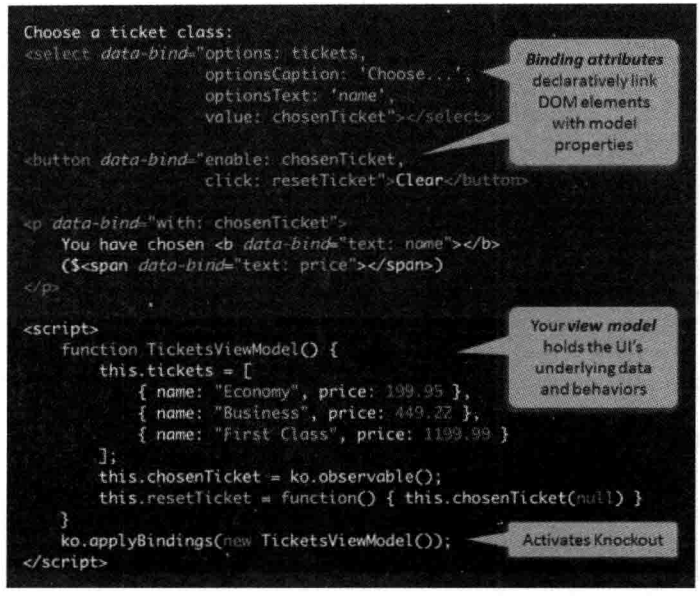
这些绑定都会变转换为来值函数，然后被绑定处理器调用，求值函数则是用户在 ViewModel 定义的函数的消费者，从而把整条链串起来。knockout 的世界就是函数世界，而这些函数又围着它们内部的_latestValue 变量团团转，只要它一改变，就派发消息给它的订阅者，让它们自己执行自己，然后把值传递上去。

```
//监控属性的实现
ko.observable = function (initialValue) {
    var _latestValue = initialValue;//重点

    function observable() {
        if (arguments.length > 0) {
            // setter 只有值不同时才发出通知
            if ((!observable['equalityComparer']) || !observable['equalityComparer'](_
latestValue, arguments[0])) {
                observable.valueWillMutate();//通知订阅者
                _latestValue = arguments[0];
                if (DEBUG) observable._latestValue = _latestValue;
                observable.valueHasMutated();//通知订阅者
            }
            return this; // Permits chained assignments
        }
        else {
            // getter
            ko.dependencyDetection.registerDependency(observable);
           // 收集订阅者，这个工作每次都执行
            return _latestValue;
        }
    }
    //这个通常情况不外泄
    if (DEBUG) observable._latestValue = _latestValue;
    //怎么也要执行一次，目的是收集订阅，方便派发报纸
    ko.subscribable.call(observable);
    //添加观察者模式必要的装备
    observable.peek = function() { return _latestValue };
    observable.valueHasMutated = function () { observable["notifySubscribers"](_latest
Value); }
    observable.valueWillMutate = function () { observable["notifySubscribers"](_latest
Value, "beforeChange"); }
    ko.utils.extend(observable, ko.observable['fn']);

    ko.exportProperty(observable, 'peek', observable.peek);
    ko.exportProperty(observable, "valueHasMutated", observable.valueHasMutated);
    ko.exportProperty(observable, "valueWillMutate", observable.valueWillMutate);

    return observable;
}
```

具体如图 16.2 所示。

假若我们再进一步，从宏观角度来看，其实上述东西分别是位于两个层面，如监控属性、依赖监控属性，监控数组是位于 ViewModel 上，通俗来说，是模型层，只不过它是类似于 ActiveRocored

这样的领域模型上。由于它们专门为视图服务，微软给它们一个新名词，视图模型。声明式绑定位于页面上，在 MVVM 里，页面不再是一个单纯的 HTML，而是作为一个模板而存在，归类为视图层。此外还存在一个模型层，它来自后端的数据库，用于抽象出视图模型。



▲图 16.2

这个层、那个层的出现，意味着前端终于进入分层架构的时代。像 Prototype.js、 mootools、mass Framework 等都以模块为单位组织代码。在分层架构更高的角度来划分它们。框架技术与架构技术的出现，都是为了解决软件系统日益复杂带来的困难而采取“分而治之”思维的结果——先大局后局部，就出现了架构；先通用而专用，就出现了框架。在前面一些章节，我们都是讨论通用功能的实现，如，如何选择元素，设置样式，绑定事件，操纵属性。这些在 MVVM 中，都归为视图层的东西，基于 MVVM 操作数据即操作 DOM 的理念，都是应该对用户隐藏。框架在水下偷偷干这些脏活，用户无需动手。

emberjs，是 jQuery、rails、Sproutecore、Merb、Handlebars 这几个著名框架的核心成员 Yehuda Katz 创建，前身是 Sproutecore2.0，是国外公认开发效率最高的前端 MVVM 框架。许多设计都是直接来自 rails3，如 restful 风格的路由，能指定数据格式的 Model，跳转相关的 helper 函数几乎一模一样。你看我说了多少东西，没错，它的特点就是打包一切。

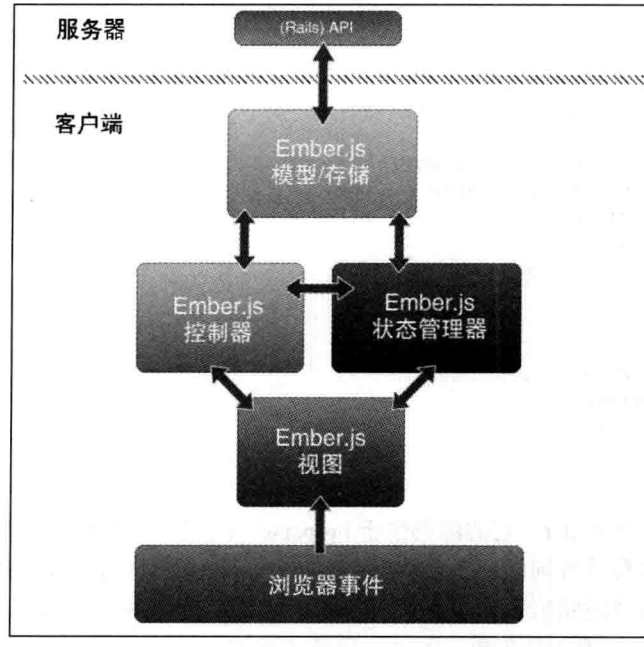
它在视图中依赖 handlebars 模板引擎，将所有绑定拆出来。不过它的是静态模板，用 script 标签做容器，但它在插入节点后会转换成动态模板。由于是基于词法分析，相当于重新发明一门新语言，可以像 rails 那样玩许多魔术。但成本非常大，需要一万行左右的模板引擎做支撑。

它的 ViewModel 需要你继承于某个类实现，不像 knockoutjs 那样一个个字段显式转换。这种做法也来源于 rails，语法的包装与 backbone 相似。

```
App.Person = Ember.Object.extend({
  firstName: null,
  lastName: null,

  fullName: function() {
    return this.get('firstName') +
          " " + this.get('lastName');
  }.property('firstName', 'lastName')
});
```

fullName 这里称为 Computed，计算属性，与 knockout 的依赖监控属性是同一个东西。用户在设置或读取模型的监控属性时，必须用 set、get 方法进行访问。如果深究下去，就会发现，其下面有一个对象，它会将用户的原 Model 的属性都改造成访问器[①]复制到其上，并且对对象属性的属性，以"aaa.bbb"的形式作为一个新属性也拷贝上去。Set、get 与一个自定义事件系统挂钩，方便互相通知。

emberjs 着重于提供于一体化的解决方案，双向依赖链的地位被掩盖了，图 16.3 是它的架构图。



▲图 16.3

angular 是 Google 组织开发的，也是走大而全的道路。它没有显式教你怎么定义监控属性，计算属性什么的，你只要定义一个控制器函数（controller），然后框架就会对它进行重新编译，实现双向绑定。对于复杂一点的计算属性，它提供了一个$watch 方法，也是内部进行编译，方便你在

① JavaScript 中有三种不同类型的属性:命名数据属性(named data properties)、命名访问器属性(named accessor properties)以及内部属性(internal properties)。文中的访问器就是命名访问器属性的简称，通过 Object.defineProperty 定义的属性，可以让我们做一些魔术效果，如图 16.3 所示。

改动它时，它能及时通知其依赖者进行更新。

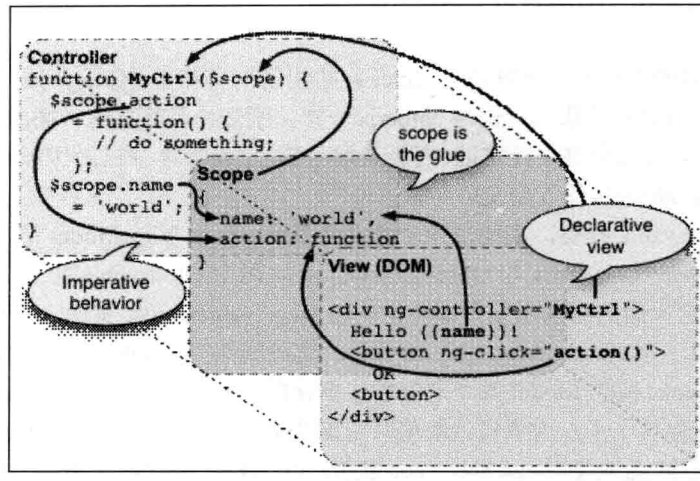
```
//angular 将字符串转换为函数的部分代码
$watch = function(watchExp, listener, objectEquality) {
    var scope = this,
            get = compileToFn(watchExp, 'watch'),
            array = scope.$$watchers,
            watcher = {
        fn: listener,
        last: initWatchVal,
        get: get,
        exp: watchExp,
        eq: !!objectEquality
    };
    // in the case user pass string, we need to compile it, do we really need this ?
    if (!isFunction(listener)) {
        var listenFn = compileToFn(listener || noop, 'listener');
        watcher.fn = function(newVal, oldVal, scope) {
            listenFn(scope);
        };
    }

    if (!array) {
        array = scope.$$watchers = [];
    }
// we use unshift since we use a while loop in $digest for speed.
// the while loop reads in reverse order.
    array.unshift(watcher);
    return function() {
        arrayRemove(array, watcher);
    };
}

function compileToFn(exp, name) {
    var fn = $parse(exp);
    assertArgFn(fn, name);
    return fn;
}
```

$parse 就是 $ParseProvider，它的源码位于 ng/parser 中，框架会逐个字符拆解出变量，如字符串、数字、关键词、运算符等词素。之后用 hasOnwerProperty 与现有的 ViewModel 比较一下，得到从属关系，可以把普通的赋值语句、取值语句转换为 getter、setter。getter、setter 内部与观察者模式挂钩，就是 knockout 那一套东西。天下原理是一致的。

在绑定上，它的实现也是可圈可点。它早期也是使用类似 knockout 的绑定结构，不同的是放到类名，这样更惨，因此后来改为以 ng-开头的特性节点。此外还有插值表达式，它可以出现 innerText 或 href 等字符串属性的值中。最后是自定义标签，根据 tagName 判定这整个东西是个绑定。angular 统统将它们称之为指令（directive）。在绑定属性的值或插值表达式的内部，我们可以写很复杂的表达式，因此 angular 又要动用上面的 parser，如图 16.4 所示。

相对于 knockoutjs 一边扫描一边处理绑定，knockout 是全部扫描时才处理，因为它允许用户直接视图中定义模型，这不是一个好的设计。所有被抽出的绑定会根据之前的位置确认它的作用域（通

过 ng-controller），然后再根据内部决定好的优先级逐条解析，挂到双向绑定链的顶部，它们怎么也会先执行一次。只要执行一次，就能得到它的依赖。当然也是出于渲染视图的需要，将占位符替换为真实数据。



▲图 16.4

```
<div ng-controller="Ctrl">
    Enter name: <input type="text" ng-model="name"><br>
    Hello <span ng-bind="name"></span>!
</div>

function Ctrl($scope) {
  $scope.name = 'Whirled';
}
```

在 angular 中，许多概念都被颠覆了。比如 ng-controller 是指定了一个 ViewModel 的作用范围，不过它的 ViewModel 本来就可以混杂许多函数，相当于 action。ng-model 是对应模型的一个字段，并且只能用于表单元素，在替换时，还绑定了诸如 input、change、click 事件做同步。

angular 支持管道符风格的过滤器，允许你对输出进行格式化。

另一个瞩目的设计是基于 HTML5 history 设计的路由系统，它会阻止页面刷新，并且根据 URL 地址加载不同的模块。至于旧版本浏览器，渐退到 hash hack 方式。这个的简单实现可以查看 backbone 的源码。

当然它还有许多特征，如国际化、端对端测试等，但偏离 MVVM 的双向绑定太远，请跳过它们。

winJS 是微软让人更多为其 Windows 应用商店开发应用而提供的 JavaScript 库，特点是拥有庞大的 UI 库，绑定设计与 knockout 好多了，绑定节点为 data-win 前缀的特性节点，值的形式也多样化。只不过在多数情况下，绑定所在的元素节点被当成一个容器、一个占位符以及传参的存在。大量的代码是写 JavaScript 的。或者它生成的 HTML 也像 emberjs 那样进行处理，可以最小化局部刷新。由于没看它的源码，个人不便发表什么评论。

```
<div id="ratingControlHost" data-win-control="WinJS.UI.Rating"
    data-win-options="{maxRating: 10, averageRating: 6}">
</div>

<div id="timepicker" data-win-control="WinJS.UI.TimePicker"
    data-win-options="{current: '10:29 am'}">
</div>
```

kendoui，是著名的.Net 组件开发公司 Telerik 推出了一套基于 jQuery 结合时下最潮 HTML5 技术的 UI 框架，拥有自己的模板（也与 emberjs 一样，刚开始是静态的，插入后变动态，可能考虑遍历 DOM 树代价大）、双向绑定链实现、高级查询函数、加载器等。核心代码分布于 kendo.core.js、kendo.data.js,、kendo.binder.js 这 3 个文件中。

core.js 提供 Observable 基类，其实是一个自定义系统，什么 ViewModel 都基于它。

data.js 提供了 ObservableArray 与 ObservableObject 这两个 Observable 的子类，ObservableArray 会调用 wrap 方法将它里面的对象元素全部再包装成 ObservableObject 实例。它有一个 Model 类，其实就是 ViewModel，为 ObservableObject 的子类。这些类都有个 toJSON 的方法，用于还原为原始数据。基于上 kendoui 检测 Model 某个属性发生变化的方式与 angular 是一致的，俗称脏检测，所有对象的属性都变包装一下，内部设值取值时用这些类的 set、get 方法，方便带动起事件系统。不过 angular 不会让你随意地使用 aaa.bbb = ccc 赋值，想这样干也可以，需要到$watch 里面处理，它有编译器伺候，取 toString 内部转$set(aaa, "bbb", ccc)。kendoui 的用法体验可能差一些，但作为一个 UI 框架，用户基本上在配置对象上忙碌，很少需要自己去调校这些属性。kendoui 使用 MVVM 也是贪图其维护状态，最小化刷新的好处。

binderjs 就是各种绑定的集合，每种绑定也一个 Observable 的子类，这样方便互相订阅互相通知。kendoui 就是一个 Observable 的世界。

avalon 是本人开发的 MVVM 框架，将通过它讲解 MVVM 的运行机理，这些大体都一致。

16.2 属性变化的监听

上面列举各大 MVVM 框架时，已经提到它的一些实现了，其中最重要的部分，莫过于 ViewModel 对它自身的那些属性变化的监听。只有做到这一点，才能实现双向绑定。

而“监听”这东西，在设计模式里，最靠近的模式莫过于观察者模式，或叫发布者订阅者模式。其工作原理是自己检测自己，发出变化时就把之前保持的回调执行一遍。在 JavaScript，一个变量发生变化，只能是它被赋值了。我们把外界对它的影响进行封装，包到两个方法中，再把它原来的值保存到另一个变量，就得到下面的代码。这是 ViewModel 能监听值变化的根源。

```
var name, oldValue, val
get = function() {
    oldValue = this[name];
    //这里用于收集订阅者，订阅者为调用这个 get 方法的某个视图函数
    //由于这个 get 事实上可能被包几层，因此可能是 caller.caller.caller
    //这需要更高的技术来收集，这里只是假设
    Observer.bind(name, arguments.callee.caller);
```

```
    return oldValue
}

set = function(val) {
    if (oldValue !== val) {//不一样就重写 oldValue
        oldVal = val
        Observer.fire(this, name, val);//通知订阅者更新
    }
}
```

至于如何让用户调用 set、 get 方法，各大框架的做法都不太一样。

像 knockout，对于一普通的属性，放到 ko. observable 函数里，返回一个监控函数，然后根据传参的不同判定是读是写。

emberjs 则拥有两个特性 setter、getter，对监控属性的访问都必须通过它们，从而确保观察者不会遗漏任何订阅函数，也不会在修改时忘了通知它们。

angular 是对$watch 回调、控制器函数等进行重新编译，得到它们的赋值取值关系，从而生成一大堆回调，交给观察者。

winjs 是使用 Object.defineProperty，Object.defineProperty 的第三个参数本来就可以接受两个函数当 getter、setter，从而控制着所有对此属性的访问。这应该是最好最自然的实现。不过缺憾是严重受制于浏览器的语法支持。

kendoui 则是将 VM 包得很紧的，只能通过它暴露那些方法进行修改属性，而这些方法里面有着像 emberjs 那样的上帝 getter、 setter。

avalon 的方案是使用 Object.defineProperties 与 VBScript 类，得到一个充满访问器属性的对象，这个对象就是 VM，访问这个 VM 的属性就会访问它的 setter、getter。setter、getter 是框架动态生成，里面包含着用户原始的写函数与读函数，还有框架内部的收集订阅者，通知订阅者更新的逻辑。这应该是目前最好的方案，起码保证原来 Model 是布尔的东西还是个布尔，是字符串还是字符串，不像 knockout 那样全部变成函数，也不像其他那样需要通过第三者调用。当然，angular 的做法也相当精明，就是工程太浩大了。

在 avalon 决定起用 VBScript 这老将时，我们还是把现有的所有与监听相关的语法或 API 都过目一遍吧。每个浏览器都私藏了不少好东西。

IE8 下有 set、get 关键字，它后来标准化了，不过可惜的是，如果浏览器不支持此语法，抛出的语法错误 try catch 不了。

IE 下元素节点有万能的 onpropertychange[①]，无论你是用 setAttribute 还是=号，只要值不一样，都能触发回调。事件对象的 propertyName 就是发生改变的属性名。大家可以通过下面的链接查看外国程序员是如何用它模拟 Object.defineProperty 的。但问题是，它是属性改变了，才触发回调，而通常我们在回调里才做这工作，并且属性是否变化这工作是我们来做的（当然不可能用!==，万一用户传 NaN 进来怎么办？）另外，它监听不了 get 操作，通常我们在这里收集订阅者——即依赖这个属性的其他属性（内部会转成操作属性的方法）。

① http://msdn.microsoft.com/zh-CN/library/ie/ms536956(v=vs.85).aspx

标准浏览器很早就支持__defineGeter__、__defineSetter__[①]，以前人们都用它模拟 IE 的 outerHTML 等。它们算是最纯粹的访问器设置机制，没有设置可配置性、可遍历性等功能。对于我们当前的场合已足够。这个作为旧版本标准浏览器的后备方案保留。

Firefox 支持用 Proxy.create[②]创建一个代理对象，相当于把一个对象包含起来，任何对这对象的读写删除都以回调方式通知原对象，我们可以在这里面放置我们的方法。是至今 JavaScript 出现的最神奇的东西。不过有了__defineGeter__、__defineSetter__与 Object.defineProperty，没有它出场的时间。

Firefox 支持 Object.prototype.watch[③]、Objet.prototype.unwatch，你可以看作是 Firefox 版的 protpertychange，不过要求作用对象为一个普通的 JavaScript 对象。另，它也不能实现 setter。

ecma6 打算支持 Object.observe[④]，可以看作是 Object.prototype.watch 的强化版，连数组的变动都能监听到。可惜缺点是一样的。

最后就是 VBScript 了，这是最后的手段，只有万不得已时，我才决定用 VBScript 与 JavaScript 进行混合编程。VBScript 的性能一直饱受诟病，并且与 JavaScript 有许多不同，比如说对象不具有扩展性，没有 toString 与 valueOf、hasOwnerProperty，this 不会指向自身（人家是用 me），不区分大小写。

好了，基本上就是这么多候补。标准浏览器基本上不用我们操心，即便是 Opera9 也支持__defineGetter__、__defineSetter__。IE 里只能用 VBScript，虽然有些困难，要用户注意一下大小写，但能让用户触发观察者做到像 Object.defineProperty 那样透明自然。当然如果我们能像 angular 那样写足够的代码，写个编译器也是可能的。

16.3 ViewModel

上一节我们讲解了如何监听一个属性的变化，现在让我们实现 ViewModel 吧。ViewModel 就是一堆监听属性或计算属性的集合。它是基于 Model 转换过来的，对一些需要渲染到视图的属性进行强化，方便下次我们改动它时自动同步到视图。使用了 MVVM，我们要转换思路，将注意的重心由 DOM 转换到数据。

不过正如 javaer 发明“贫血模型”，“充血模型”这些概念[⑤]时指出，光是有 get、set（贫血模型）不够面向对象，过于单薄，颗粒度过小，容易造成代码量膨胀，最重要的是业务逻辑的描述能力比较差，一个稍微复杂的业务逻辑，就需要太多类和太多代码去表达。而充血模型则包含操作数据的方法，自治程度高，表达能力强。缺点也不是没有，会降低复用性。鉴于前端改需求重构推倒重来的事太多了，写代码写得快才是王道（jQuery 已经完美证明这一点），因此这模式很适合 ViewModel。angular 放任我们在$scope 中添加任何数据与方法就是明证。

① http://www.iteye.com/topic/40946
② http://soft.vub.ac.be/~tvcutsem/proxies/
http://wiki.ecmascript.org/doku.php?id=harmony:proxies
https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Proxy
③ https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Object/watch
④ http://weblog.bocoup.com/javascript-object-observe/
⑤ http://wezly.iteye.com/blog/1601931

在MVVM中，其实也不是所有东西都是双向绑定（two way）的，也可能是单向（one way）的，或一次性（one time）的。在kendoui，它提出了用“_”前缀来表示那些不被监听的属性，这也是个好主意。因此如果只渲染一次，以后不改，将它转换为监控属性实在太浪费了。

不过，我认为用“_”作标识不是一个好主意。它通常表示内部的私有状态，既然是状态，就至少有两种或以上的值，会变动。变动的东西我们都应该交由双向绑定链去维护。此外，现在jQuery这么流行，MVVM与jQuery混在一起用的可能性太高了，当用户想在ViewModel引用一个jQuery对象，如果不阻止MVVM将它转换为一个监控数组会吃掉很多内存（jQeury是一个庞大的类数组对象）。

人们通常以$开头的变量来表示这是一个jQuery实例，在其他库或后端，$开头的变量通常表示系统级的东西，既然是系统级就是底层，底层就是非常稳定，因此我们无需让它改来改去，用不着监听，用“$”开头做标识更为妥当。假若用户由于其他原因，不得不以其他字符开头呢？我们就设一个字符串数组，表示这里面的变量都不被监控。avalon将这个数组命名$skipArray，因为我们将Model转换一个ViewModel是在循环中进的，遇到它里面的变量名就跳过不处理。

让我们看看如何转换一个ViewModel。在avalon中负责此工作的函数叫define，传参也与AMD中的define方法相似。第一个为ID，第二个是依赖列表，第三个是工厂函数。有ID方便寻找与排序，有依赖列表是因为视图中某一个区域需要复数个ViewModel联袂演出，使用函数形式，是方便定义内部函数与一些不直接放到ViewModel中的私有变量。

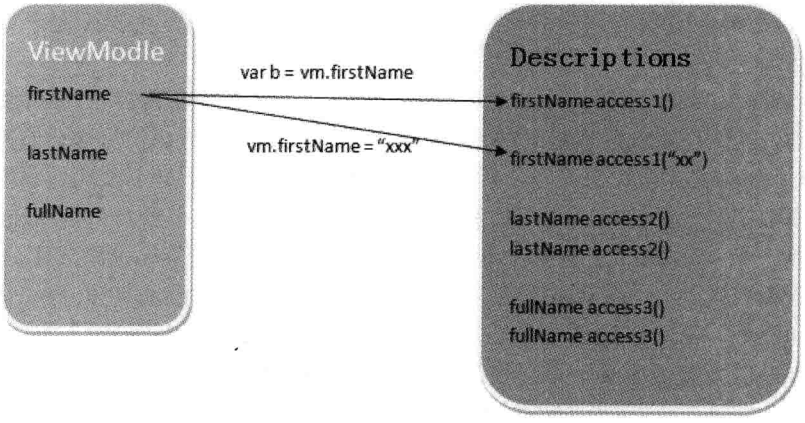
```
avalon.define("user", [], function(vm) {
    vm.firstName = "司徒"//监控属性
    vm.lastName = "正美" //监控属性
    //计算属性
    //一个包含 set 或 get 的对象方便加工为 Object.defineProperties 的第 2 个参数
    vm.fullName = {
        set: function(val) {
            var array = (val || "").split(" ");
            this.firstName = array[0] || "";
            this.lastName = array[1] || "";
        },
        get: function() {
            return this.firstName + " " + this.lastName;
        }
    }
})
```

当然如果页面只有一个ViewModel，ID就没有意义，依赖也同理，因此我们最后将它们设为可选。只有最后一个工厂为必须，其他通过unshift与splice凑够为3个。

由于VBScript对象不具扩展性，因此最好我们一次性在工厂里定义好。另外，由于使用了VBScript后，意味着最后产出的是一个VBScript类实例，与传参对象不是一个东西，而标准浏览器则直接在原对象上修改，为保持统一都返回一个新对象。

思路是，先补全参数，然后遍历参数对象的键值对，将所有允许监听的属性取出来，每个配一个方法，方法内部包含我们操作这属性的set、get方法，以及发出通知与收集订阅者的代码。最后组成Object.defineProperties的第2个参数Descriptions对象。然后新建一个对象（ViewModel），将

这两个对象作为 Object.defineProperties 参数传进去。如果不支持，则放到 VBScript 类中，它的所有访问器属性在内部都调用同一个方法 VBMediator，不同的是传参与固化在内部的方法名，然后用它们去操作 Descriptions。之后就到一个 VBScript 类实例。但无论何种方法，这时得到的对象的所有属性都是 undefine，我们需要再遍历一次，帮它赋值，如果遇到是计算属性需要求值，函数被再覆盖一次也无所谓，可以不管。最后 ViewModel 补上 ID 名就大功告成了，如图 16.5 所示。



▲图 16.5

用代码表示过程如下。

```
avalon.define(function(vm) {
    vm.salutation = 'Hello';
    vm.name = 'World';
    vm.greet = function() {
        vm.greeting = vm.salutation + ' ' + vm.name + '!';
    }
})

//第 1 阶段，在循环中拼凑出一个这样的对象
var description = {
    salutation: {
        set: accessor1,
        get: accessor1,
        enumerable: true
    },
    name: {
        set: accessor2,
        get: accessor2,
        enumerable: true
    }
}
//第 2 阶段，得到一个充满访问器的对象
var viewmodel = {}
if (Object.defineProperties) {
    Object.defineProperties(viewmodel, description)
} else {
```

```
    //names 包含原 model 所有属性名,方法名及你以后打算动态添加的属性名\方法名
    viewmodel = Object.defineProperties(description, names)
}
//第 3 个阶段，赋值与填充函数
viewmodel.salutation = vm.salutation;
viewmodel.name = vm.name;
viewmodel.greet = vm.greet
viewmodel.$id = "xsddfsdrfr" //一个 UUID
```

如果细心，你就会发现 greet 方法中的 vm 还是指向最初传入的空对象。为了将刚得到 ViewModel 放入工厂中执行一次，重置它。但这样做，有的监控属性就会再一次被赋值，对于它们来说这是毫无意义的。监控属性也罢了，如果是计算属性，这时就会赋一个对象，监控数组会再一次转换。为了阻止它们，只针对函数，我们需要对这些监控对象内部的 set 方法做个开关，让它们在这时不做处理。

下面是 avalon 的 define 方法。

```
avalon.define = function(name, deps, factory) {
    var args = [].slice.call(arguments);
    if (typeof name !== "string") {
        name = !avalon.models["root"] ? "root" : modleID();
        args.unshift(name);
    }
    if (!Array.isArray(args[1])) {
        args.splice(1, 0, []);
    }
    deps = args[1];
    if (typeof args[2] !== "function") {
        avalon.error("factory 必须是函数");
    }
    factory = args[2];
    var scope = {};
    deps.unshift(scope);
    factory(scope);                          //得到所有属性与方法
    var model = modelFactory(scope);         //得到 ViewModel
    stopRepeatAssign = true;                 //阻止已经初始化的监控属性在这时被赋值
    deps[0] = model;
    factory.apply(0, deps);                  //重置所有函数中的 scope
    deps.shift();
    stopRepeatAssign = false;
    model.$id = name;                        //重置 ID
    return avalon.models[name] = model;      //全部保存起来，方便以后查找
};
```

modelFactory 的实现如下。

```
function modelFactory(scope) {
    var skipArray = scope.$skipArray,
          description = {},
          model = {},
          callSetters = [],
          callGetters = [],
```

```
        VBPublics = [];
skipArray = Array.isArray(skipArray) ? skipArray : [];
avalon.Array.ensure(skipArray, "$skipArray");
forEach(scope, function(name, value) {
    if (typeof value === "function") {
        VBPublics.push(name);
    } else {
        if (skipArray.indexOf(name) !== -1) {
            return VBPublics.push(name);
        }
        if (name.charAt(0) === "$") {
            if (skipArray.indexOf(name) !== -1) {
                return VBPublics.push(name);
            }
        }
        var accessor, oldValue, oldArgs;
        if (value && typeof value === "object" && typeof value.get === "function"
                && Object.keys(value).length <= 2) {
            accessor = function(neo) { //创建计算属性
                if (arguments.length) {
                    if (stopRepeatAssign) {
                        return; //阻止重复赋值
                    }
                    if (typeof value.set === "function") {
                        value.set.call(model, neo);
                    }
                    if (oldArgs !== neo) {
                        oldArgs = neo;
                        notifySubscribers(accessor); //通知订阅者改变
                    }
                } else {
                    if (openComputedCollect) {
                        collectSubscribers(accessor);
                    }
                    oldValue = value.get.call(model);
                    return oldValue;
                }
            };
            //这里用到 ecma262v5 的 Function.prototype.bind
            callGetters.push(function() {
                var fn = this;
                Publish[expose] = fn;
                fn();
                delete Publish[expose];
            }.bind(accessor));
        } else {
            callSetters.push(name);
            accessor = function(neo) { //创建监控属性或数组
                if (arguments.length) {
                    if (stopRepeatAssign) {
                        return; //阻止重复赋值
                    }
                    if (oldValue != neo)) {
                        oldValue = neo;
```

```
                    notifySubscribers(accessor); //通知订阅者改变
                }
            } else {
                collectSubscribers(accessor); //收集视图函数
                return oldValue;
            }
        };
    }
    accessor[subscribers] = [];
    description[name] = {
        set: accessor,
        get: accessor,
        enumerable: true
    };
  }
});
if (defineProperties) {
    defineProperties(model, description);
} else {
    model = VBDefineProperties(description, VBPublics);
}
VBPublics.forEach(function(name) {
    model[name] = scope[name];
});
callSetters.forEach(function(prop) {
    model[prop] = scope[prop]; //为空对象赋值
});
callGetters.forEach(function(fn) {
    fn(); //为空对象赋值
});
model.$id = generateID();
return model;
}
```

里面依赖几个方法，讲解如下。

Array.isArray 是 ecma262v5 的方法，如果浏览器不支持就自己实现一个。

avalon.Array.ensure 的逻辑很简单，就是当数组不存在此元素时把它 push 进去。

```
avalon.Array = {
    ensure: function(target) {
        var args = [].slice.call(arguments, 1);
        args.forEach(function(el) {
            if (target.indexOf(el) === -1) {
                target.push(el);
            }
        });
        return target;
    }
}
```

generateID 就是返回一个 UUID。这种有 N 种实现，自己可以在下面挑一种。

方法一，网上流行的基于 rfc4122 UUID 的最精简实现。

```
var generateID = function() {
  return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function(c) {
    var r = Math.random()*16|0, v = c == 'x' ? r : (r&0x3|0x8);
    return v.toString(16);
  });
}
```

方法二，取自 Backbone.localStorage。

```
function S4() {
   return (((1+Math.random())*0x10000)|0).toString(16).substring(1);
};
function generateId() {
   return (S4()+S4()+"-"+S4()+"-"+S4()+"-"+S4()+"-"+S4()+S4()+S4());
};
```

方法三，取自 mootools。

```
var start = new Date - 0;
function generateID() {
    return (start++).toString(36)
}
```

方法四，取自 angularjs。

```
var uid  = ['0', '0', '0'];
function generateID() {
  var index = uid.length;
  var digit;

  while(index) {
    index--;
    digit = uid[index].charCodeAt(0);
    if (digit == 57 /*'9'*/) {
      uid[index] = 'A';
      return uid.join('');
    }
    if (digit == 90  /*'Z'*/) {
      uid[index] = '0';
    } else {
      uid[index] = String.fromCharCode(digit + 1);
      return uid.join('');
    }
  }
  uid.unshift('0');
  return uid.join('');
}
```

方法五，取自 mass。

```
function generateID() {
    return setTimeout("1")+""
}
```

当然还有许多，但体积都非常庞大，我们又不是专门搞 UUID 库，加之一个页面同时存在 200

个以上的 ViewModel 的情况太少了，够用就行了。

我们再来看看如何抹平 Object.defineProperties 在各浏览器的差异性。

//如果支持 Object.defineProperty，判定它是否为 IE8 那样只对 DOM 有效

```
var defineProperty = Object.defineProperty;
try {
    defineProperty({}, "_", {
        value: "x"
    });
    var defineProperties = Object.defineProperties;
} catch (e) {//否则使用__defineGetter__
    if ("__defineGetter__" in avalon) {
        defineProperty = function(obj, prop, desc) {
            if ('value' in desc) {
                obj[prop] = desc.value;
            }
            if ('get' in desc) {
                obj.__defineGetter__(prop, desc.get);
            }
            if ('set' in desc) {
                obj.__defineSetter__(prop, desc.set);
            }
            return obj;
        };
        defineProperties = function(obj, descs) {
            for (var prop in descs) {
                if (descs.hasOwnProperty(prop)) {
                    defineProperty(obj, prop, descs[prop]);
                }
            }
            return obj;
        };
    }
}
//最次的方案，使用 VBScript，想看明这代码要学些 VBScript，相关链接见下。
//http://webreflection.blogspot.com/2011/02/btw-getters-setters-for-ie-6-7-and-8.html
if (!defineProperties && window.VBArray) {
    window.execScript([
        "Function parseVB(code)",
        "\tExecuteGlobal(code)",
        "End Function"].join("\n"), "VBScript");
    function VBMediator(description, name, value) {
        var fn = description[name] && description[name].set;
        if (arguments.length === 3) {
            fn(value);//setter
        } else {
            return fn();//getter
        }
    }

    function VBDefineProperties(description, publics) {
        publics = publics.concat();
        avalon.Array.ensure(publics, "hasOwnProperty");
        avalon.Array.ensure(publics, "$id");
```

```
        var className = "VBClass" + setTimeout("1"),
                owner = {}, buffer = [];
        buffer.push(
                "Class " + className,
                "\tPrivate [__data__], [__proxy__]",
                "\tPublic Default Function [__const__](d, p)",
                "\t\tSet [__data__] = d: set [__proxy__] = p",
                "\t\tSet [__const__] = Me", //链式调用
                "\tEnd Function");
        publics.forEach(function(name) { //添加公共属性,如果此时不加以后就没机会了
            owner[name] = true; //因为 VBScript 对象不能像 JavaScript 那样随意增删属性
            buffer.push("\tPublic [" + name + "]"); //你可以预先放到 skipArray 中
        });
        Object.keys(description).forEach(function(name) {
            owner[name] = true;
            buffer.push(
                    //由于不知对方会传入什么,因此 set、let 都用上
                    "\tPublic Property Let [" + name + "](val)", //setter
                    "\t\tCall [__proxy__]([__data__], \"" + name + "\", val)",
                    "\tEnd Property",
                    "\tPublic Property Set [" + name + "](val)", //setter
                    "\t\tCall [__proxy__]([__data__], \"" + name + "\", val)",
                    "\tEnd Property",
                    "\tPublic Property Get [" + name + "]", //getter
                    "\tOn Error Resume Next",
                    //必须优先使用 set 语句,否则它会误将数组当字符串返回
                    "\t\tSet[" + name + "] = [__proxy__]([__data__],\"" + name + "\")",
                    "\tIf Err.Number <> 0 Then",
                    "\t\t[" + name + "] = [__proxy__]([__data__],\"" + name + "\")",
                    "\tEnd If",
                    "\tOn Error Goto 0",
                    "\tEnd Property");
        });
        buffer.push("End Class"); //类定义完毕
        buffer.push(
            "Function " + className + "Factory(a, b)", //创建实例并传入两个关键的参数
                "\tDim o",
                "\tSet o = (New " + className + ")(a, b)",
                "\tSet " + className + "Factory = o",
                "End Function");
        // console.log(buffer.join("\r\n"));
        window.parseVB(buffer.join("\r\n"));
        var model = window[className + "Factory"](description, VBMediator);
        model.hasOwnProperty = function(name) {
            return owner[name] === true;
        };
        return model;
    }
}
```

collectSubscribers 与 notifySubscribers，它们共同构成一个观察者模式，或叫发布者订阅者模式，是一切自定义事件系统的原型。核心方法有 3 个。

首先添加订阅者，通常是按照在某个名目（topic，事件系统叫做事件类型）进行储存，储存方

式可以是链表或数组。在 avalon 中，ViewModel 的监控属性或计算属性，它们的真面目为内部 Descriptions 的 accessor 函数。这些函数都有名字，就是 ViewModel 的键名，accessor 上有一个 subscribers 数组，用于放置监控属性的 accessor 函数或视图刷新函数。

其次是移除订阅者。在 avalon 中，移除操作是混杂在通知订阅者更新的逻辑中，判定依据是视图刷新函数上的元素节点有没被移出 DOM 树。由于 DOM 事件系统在这方面兼容性太差并且太低效，还是延迟到下次遍历这个 subscribers 数组检测为宜。

最后是通知订阅者。当 Descriptions 的 accessor 函数执行 setter 操作就会调用它。

//收集依赖于这个访问器的订阅者

```
function collectSubscribers(accessor) {
    if (Publish[expose]) {
        var list = accessor[subscribers];
         //只有数组不存在此元素时才 push 进去
        list && avalon.Array.ensure(list, Publish[expose]);
    }
}
//通知依赖于这个访问器的订阅者更新自身
function notifySubscribers(accessor) {
    var list = accessor[subscribers]
    if (list && list.length) {
        var args = aslice.call(arguments, 1)
        var safelist = list.concat()
        for (var i = 0, fn; fn = safelist[i++]; ) {
            var el = fn.element,
                state = fn.state,
//state 是来自它的父节点，用于此元素移出 DOM 树但又不想它的订阅函数被删除的情况
                remove
            if (el && (!state || state.sourceIndex !== 0)) {
//IE6~IE11 通过 sourceIndex 是否为零判定是否在 DOM 树
                if (typeof el.sourceIndex == "number") {
                    remove = el.sourceIndex === 0
                } else {
                    try {//尝试使用 contains 方法
                        remove = !root.contains(el)
                    } catch (e) { //如果不存在 contains 方法
                        remove = true
                        while (el == el.parentNode) {
                            if (el === root) {
                                remove = false
                                break
                            }
                        }
                    }
                }
            }
            if (remove) {
                avalon.Array.remove(list, fn)
            } else {
                fn.apply(0, args) //强制重新计算自身
            }
```

```
        }
    }
}
```

不过问题是如何收集订阅者。avalon 的思路是这样，描架会扫描到 DOM 树，抽取里面的绑定，比如，得到 firstName，假如页面上只有一个 ViewModel，我们就能确定它们的对应关系，结合 ms-controller 来对所有 ViewModel 进行排序，逐个检测它们是否有 firstName。然后转换成求值函数。

```
function val(){
    return vm.firstName
}
```

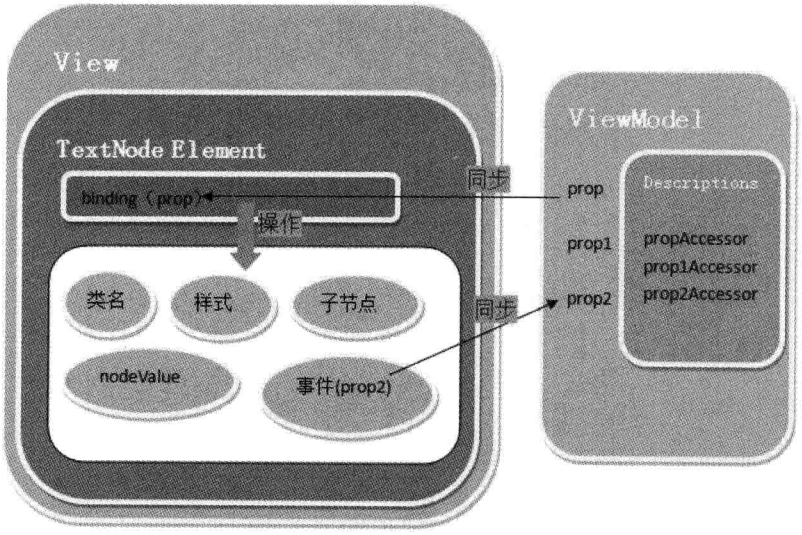
再根据 ms-html 分解出绑定处理器的名字 html，于是有：

```
function updateView(){
    node.innerHTML = val();
}
```

updateView 在执行前会把自己放到内部的一个 Publish 对象上，然后执行 updateView，updateView 继而执行 val 求值函数，触发 firstName 的 getter 操作，getter 操作会调用 Descriptions.firstName.get()方法，在 arguments.length == 0 的情况执行 collectSubscribers 方法，于是将 updateView 从 Publish 中取下，放到 subscribers 数组中。最后当此方法执行完，框架从 Publish 对象删除。

到此为止，ViewModel 就完成了，它的作用就相当于 Proxy.create 创建的那个代理对象。通过它调用水下的 Descriptions 中的方法，实现观察者模式。

下面是 avalon 的整个架构，如图 16.6 所示。



▲图 16.6

16.4 绑定

在 MVVM 中，绑定是 View 与 ViewModel 的纽带。正因为绑定的存在，我们才能确定要操作元素节点与文本节点，继而进行类名切换、样式修改、事件绑定、属性更新，如图 16.7 所示。

▲图 16.7

绑定在各大框架出入比较大，像 knockout 与 epoxyjs[①]是使用统一的 data-bind 特性节点，ember 则把它们放入界定符之内，但主流的方式是指定拥有某个特定前缀的特性节点作为我们的绑定属性，同时从它的 name 与 value 值中抽取有用的信息。除此之外，angular 还用到类名、标签名、以及 innerText 中的{{}}插值表达式，最后相当于 knockoutjs 的 text 绑定，但更易用，avalon 就厚颜无耻地收入囊中了。

我们看一下各 MVVM 框架都有什么绑定。

首先是 knockout，因为很重要，要重点看看，如表 16.1 所示。

表 16.1

html	替换 innerHTML
css	添加 style.cssText
style	操作 className
attr	操作属性与特性
foreach	与监控数组一起，将目标元素的 innerHTML 复制成 N 份，循环生成
if	如果求值函数为假就将目标元素移出 DOM 树
ifnot	如果求值函数为真就将目标元素移出 DOM 树
with	将目标元素的 innerHTML 根据对象的键值对数复制成 N 份，循环生成
event	绑定事件，此外还有 click、submit 等具名事件绑定
enable	将 disabled 属性移除
disable	添加 disabled 属性
value	修改 value 属性
hasfocus	就是 focus 绑定
checked	可以看作是 click 绑定的包装，只在 checked 改变时触发

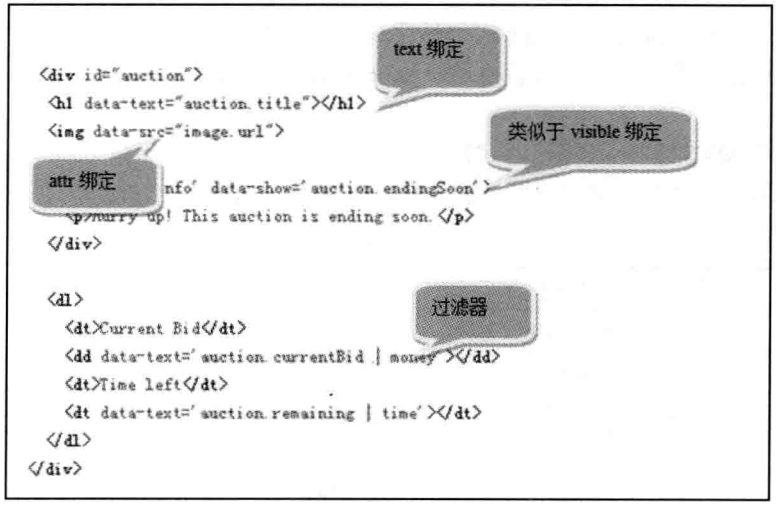
① http://epoxyjs.org/

续表

options	可以看作是 foreach 绑定的包装，用于循环生成 option 元素
selectedOptions	为 option 添加 selected 属性
uniqueName	为表单元素添加一个 name 值
template	是 forach、if、ifnot、with 等流程绑定者，它们全部由它实现的

epoxyjs 就是 knockout 的翻版，拥有如下绑定：attr、checked、collection、css、disabled、enabled、events、html、options、optionsDefaut、optionsEmpty、emplate、text、toggle、value。其中 collecion 相当 knockout 的 foreach，toggle 相当于 visible。epoxyjs 拥有过滤器机制，用于格式化输出，但用法不怎么管用。

rivetsjs[①]是一个相当小的 MVVM 框架，它的绑定也非常好。它让绑定变得更加灵活，特性节点的名字也包含绑定信息（只能是绑定类型或某个关键参数）的先河就是它开启的。它也率先引入了管道符风格的过滤器机制，如图 16.8 所示。



▲图 16.8

当然它也允许过滤器传参，但这个就不那么可爱了。

下面是它所拥有的绑定列表，用过 knockout 的人大抵能对得上号。

- data-text。
- data-html。
- data-value。
- data-show。
- data-hide。
- data-enabled。

① http://rivetsjs.com/

- data-disabled。
- data-checked。
- data-unchecked。
- data-[attribute]。
- data-class-[class]。
- data-on-[event]。
- data-each-[item]。

angular 拥有数量非常庞大的绑定，它称为指令。如果光是它的绑定属性就有 20 种以上，并且只要特性值包含{{}}插值表达式，就能转换为一个绑定属性。有特色的有如下这些。

- ng-cloak：要求在 CSS 中将它的 display 改为 none，当这元素被扫描后就会去掉此属性，重新显示。这是为了防止 reflow 引起页面抖动。不过由于 IE6、IE7 不支持属性选择器，需要多加一个 ng-cloak 类名。
- ng-include：用于加载其他页面的内部到此元素下。
- ng-init：用于动态成一些监听属性。
- ng-controller：指定此元素所包含的区域应该交由哪个$scope 对象来处理，$scope 对象就是其他框架的 ViewModel。
- ng-model：它只能用于表单元素，将 ViewModel 的某个字段与元素的 value 相绑定，并偷偷绑了一些事件进行同步。
- ng-non-bindable：对某个区域停止扫描，不处理它们的绑定，方便让它们原样输出。
- ng-list：只对 input[text]元素有效，用于将用户输出按","分割成一个字段串数组，同步到 ViewModel 中。
- ng-repeat：相当其他框架的 each 绑定，不过在循环生成的子 ViewModel 中，带有四个东西，$index、$first、$middle、$last 算是非常贴心的设计。
- ng-switch：相当于引入 switch 流程控制。
- 当然{{}}插值表达式应该是最值得一提的。它把框架操作的颗粒度细化为某一个文本节点、特性节点。

avalon 的绑定基本上就是 rivetsjs 与 angular 的合集。

- ms-class-*： 根据值绑定或移除类名。
- ms-on-*：绑定事件，*为事件名，必须。基于它，框架还构建了 ms-click、ms-mouseover 等绑定。
- ms-options-*：用于循环生成 option 元素。传参如果是一个数字时，则选中对应位置上的 option 元素，如果是一个数组，则把对 value 一致的 option 元素都选中。
- ms-visible： 通过 style.display 控制显隐。
- ms-each-*： 同其他框架，*为单个元素的引用名。
- ms-if：同 knockout，false 则元素移出 DOM 树。
- ms-href：用于设置 href，值里面可包含{{}}插值表达式，类似的绑定还有 ms-alt、ms-title。
- ms-disabled：用于操作 disabled，其他所有布尔属性都能用类似方式实现。

- ms-enabled：上面的反操作。
- ms-controller：功能同 angular。
- ms-model：功能同 angular。
- ms-skip：相当于 angular 的 ng-non-bindable。
- ms-html：操作 innerHTML。
- ms-important：类似于 CSS 的 important，只允许指定的 ViewModel 对它及其后代扫描，并且如果里面用到字段此 ViewModel 没有，也不会向往找其他 ViewModel 处理。
- ms-ui-*：绑定一个 UI 控件到目标元素。

在 angular 与 avalon 中都是由 controller 来控制某个 ViewModel 的作用范围，如果当前 ViewModel 找不到某字段，它就往上找定义在它上方的 ViewModel，然后看看它有没有，有就停止，没有就继续往上找，直接 documentElement。这有点像 avascript 变量沿着作用域链进行查询，采取就近原则，局部优先于全局。但有时，我们就是想让某个 ViewModel 负责这个区域，不想往上找，avalon 提供了 important 绑定。看下面的例子。

```
avalon.ready(function() {
    avalon.define("AAA", function(vm) {
        vm.name = "liger"
        vm.color = "green"
    });
    avalon.define("BBB", function(vm) {
        vm.name = "sphinx"
        vm.color = "red"
    });
    avalon.define("CCC", function(vm) {
        vm.name = "dragon"
    });
    avalon.define("DDD", function(vm) {
        vm.name = "sirenia"
    });
    avalon.scan()
})
```

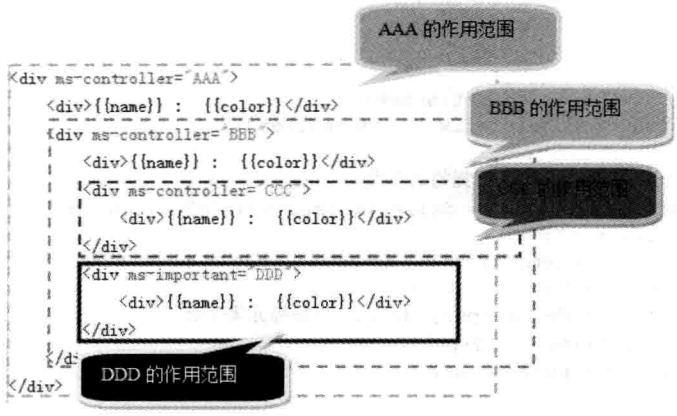
HTML 结构如下。

```
<div ms-controller="AAA">
    <div>{{name}} : {{color}}</div>
    <div ms-controller="BBB">
        <div>{{name}} : {{color}}</div>
        <div ms-controller="CCC">
            <div>{{name}} : {{color}}</div>
        </div>
        <div ms-important="DDD">
            <div>{{name}} : {{color}}</div>
        </div>
    </div>
</div>
```

输出的结果如下。

```
liger : green
sphinx : red
dragon : red
sirenia :
```

我们有线框表示各个 ViewModel 的作用域范围，如图 16.9 所示。



▲图 16.9

显然，controller 有利于我们团队协作开发，important 则用于某些特殊的沙箱环境的场合。

讲完设计，再看如何实现。第一步就是扫描，自顶向下，先找到元素节点，然后优先处理 ms-skip、ms-important、ms-controller 等三个绑定属性，根据情况停止扫描及切换 ViewModel。之后就开始扫描特性节点，边扫边处理。最后是扫描文本节点，抽取插值表达式，交由 text 绑定处理。最后处理子元素节点，递归这一过程。

为此，avalon 对外提供了 scan 方法，它的作用相当于 knockout 的 applyBindings。内部调用 scanTag、scanAttr、scanText，其中 scanTag 控制着整个流程，而 scanText、scanAttr 则是负责抽取与转换绑定。

```
avalon.scan = function(elem, scope) {
    elem = elem || document.documentElement;
    var scopes = scope ? [scope] : [];
    scanTag(elem, scopes, elem.ownerDocument || document);
};

function scanTag(elem, scopes, doc) {
    scopes = scopes || [];
    var a = elem.getAttribute(prefix + "skip");
    var b = elem.getAttribute(prefix + "important");
    var c = elem.getAttribute(prefix + "controller");
    //这三个绑定优先处理，其中 a > b > c
```

```
    if (typeof a === "string") {
        return;
    } else if (b) {
        if (!avalon.models[b]) {
            return;
        } else {
            scopes = [avalon.models[b]];
            elem.removeAttribute(prefix + "important");
        }
    } else if (c) {
        var newScope = avalon.models[c];
        if (!newScope) {
            return;
        }
        scopes = [newScope].concat(scopes);
        elem.removeAttribute(prefix + "controller");
    }
    scanAttr(elem, scopes); //扫描特点节点
    if (elem.canHaveChildren === false || !stopScan[elem.tagName]) {
        var textNodes = [];
        for (var node = elem.firstChild; node; node = node.nextSibling) {
            if (node.nodeType === 1) {
                scanTag(node, scopes, doc); //扫描元素节点
            } else if (node.nodeType === 3) {
                textNodes.push(node);
            }
        }
        for (var i = 0; node = textNodes[i++]; ) { //延后执行
            scanText(node, scopes, doc); //扫描文本节点
        }
    }
}
var stopScan =
avalon.oneObject("area,base,basefont,br,col,hr,img,input,link,meta,param,embed,wbr,sc
ript,style,textarea");
//扫描元素节点中直属的文本节点，并进行抽取

function scanText(textNode, scopes, doc) {
    var bindings = extractTextBindings(textNode, doc);
    if (bindings.length) {
        executeBindings(bindings, scopes);
    }
}
function scanAttr(el, scopes) {
    var bindings = [];
    for (var i = 0, attr; attr = el.attributes[i++]; ) {
        if (attr.specified) {
            var isBinding = false,
                    remove = false;
            if (attr.name.indexOf(prefix) !== -1) { //如果是以指定前缀命名的
                var type = attr.name.replace(prefix, "");
                if (type.indexOf("-") > 0) {
                    var args = type.split("-");
                    type = args.shift();
```

```
                }
                remove = true;
                isBinding = typeof bindingHandlers[type] === "function";
            } else if (bindingHandlers[attr.name] && hasExpr(attr.value)) {
                type = attr.name; //如果只是普通属性，但其值是个插值表达式
                isBinding = true;
            }
            if (isBinding) {
                bindings.push({
                    type: type,
                    args: args || [],
                    element: el,
                    remove: remove,
                    node: attr,
                    value: attr.nodeValue
                });
            }
        }
    }
    executeBindings(bindings, scopes);
}
```

scanTag 会跳过一些半闭包的标签与一些特殊的标签，加快扫描流程。

scanAttr 与 scanText 的抽取工作比较繁锁，允许我跳过，大多数是字符串处理。现在可说的是它们都返回一个 bindings 数组，里面是一个个对象，格式大致如下。

```
//对于 ms-each-post="posts"，抽取得到
var binding = {
    type: type,//"each"
    args: args || [],//["post"]
    element: el, //此特性节点所在元素节点
    remove: remove,//是否可以删除，
    node: attr,//特性节点
    value: attr.nodeValue//"posts"
};
```

第二步就是转换为求值函数了。我们在扫描时都携带着 scopes 这个参数，终于在这里发挥所长。data.value 为绑定属性的值，它会转换一个求值函数，然后通过传入 scope 作为参数，从而绑定属性在对应 VM 中的值。

```
var trimText = data.value.trim();//data 为 bindings 数组的一个元素
for (var i = 0, obj; obj = scopes[i++]; ) {
    if (obj.hasOwnProperty(trimText)) {
        target = obj;
        break;
    }
}
if (target) {
    updateView = function() {
        callback(target[trimText]);
    };
}
```

否则我们尝试使用 with 与 new Function 来编译一个函数。比如 ms-visible="posts.size()"，会转换为这样一个函数：

```
function fn(root1367310833258) {
    with (root1367310833258) {
        var ret1367310833258 = posts.size();
    }
    return ret1367310833258;
}
data.compileFn = fn;
```

然后在 updateView 中调用它：

```
updateView = function() {
    var fn = data.compileFn;
    val = fn.apply(fn, scopeLists);
    callback(val);
};
```

里面有个 callback，这是求值函数的最后归宿。整个真相是，上面生成 updateView 的逻辑是包裹在 watchView 方法中，而 watchView 又包含在每一个绑定处理函数中，它们基本上就是最后的回调不一样。我们看比较简单的 visible 绑定。

```
avalon.bindingHandlers.visible = function(data, scopes) {
    var element = data.element;
    watchView(data.value, scopes, data, function(val) {
        element.style.display = val ? "block" : "none";
    });
};
```

watchView 的大致实现如下。

```
function watchView(text, scopes, data, callback, tokens) {
    var updateView, target;
    var trimText = text.trim();
    //创建一个 updateView
    updateView = function() {
        //略
    };
    //这里非常重要,我们通过判定视图刷新函数的 element 是否在 DOM 树决定
    //将它移出订阅者列表
    updateView.element = data.element;
    Publish[expose] = updateView;//暴光此函数,方便 collectSubscribers 收集
    openComputedCollect = true;
    updateView();
    openComputedCollect = false;
    delete Publish[expose];
}
```

到这一步，View 与 ViewModel 就连成一体了。ViewModel 通过观察者模式操作这些视图刷新函数来更新视图，而视图会绑定一些事件，直接修改 ViewModel 里面的访问器属性。

为了让用户安心围绕数据（ViewModel）进行作业，这些绑定种类越多越好，让它们打包好这

一切。Angular、knockout、avalon 基本上可以单独处理 90%的情况，但面对复杂的 UI 控件操作还是力不从心，这主要受限于自身 DOM 的处理能力。Angular 里面有一个 jqLite，它的本意就是当你玩不转时，鸟枪换炮，让 jQuery 出马。DOM 操作最复杂的几个莫过于样式处理、属性处理与动画。如果你把这几个写得足够强大，也完全可以一脚踢开 jQuery 的。

16.5 监控数组与子模板

监控数组可以监听复数个属性的变化，如果这些属性分布在不同的节点上，那么它就是对一大片 HTML 进行绑定。在所有 MVVM 框架中，执行这个操作的绑定都叫 each 绑定（angular 的有点特殊，它叫做 ng-repeat，它是循环绑定元素本身，而不是循环绑定元素的子节点）。我们只需要定义第一个数组元素监控的 HTML 是什么样子，不需要所有 HTML 都写出来，让框架循环生成剩余元素对应的HTML 就行了，然后再让与之相对的元素来填充数据。在我们已有常识中，第一个 HTML 就是模板。由于在 MVVM 中，整个 DOM 树都被整成一个动态模板，这个只针对局部区域的 HTML，就该称为子模板。

子模板分两种形态，一种是页面有的，就像 each 绑定所在的元素所圈定的那片区域，一种是动态生成，这一般用于 UI 绑定。avalon 两种都支持。

我们先看监控数组的实现。在定义时，只要传入一个数组，avalon 就把它改造成一个监控数组。为此我们需要对 modelFactory 进行改造，具体可见这里：

```
https://github.com/RubyLouvre/mass-Framework/blob/1.4/avalon.js
if (oldArgs !== neo) {
    if (Array.isArray(neo)) {
        if (oldValue && oldValue.isCollection) {
            updateCollection(oldValue, neo);
        } else {
            oldValue = Collection(neo);
        }
    } else if (avalon.type(neo) === "Object") {
        if (oldValue && oldValue.$id) {
            updateModel(oldValue, neo);
        } else {
            oldValue = modelFactory(neo);
        }
    } else {
        oldValue = neo;
    }
    notifySubscribers(accessor); //通知顶层改变
    model.$events && model.$fire(name, neo, oldValue);
}
```

我们需要一个 Collection 方法，返回一个改造过的数组对象。被染指的 push、unshift、pop、shit 等方法只是为了方便通知订阅者更新。由于监控数组的每一个元素的属性变化太难，avalon 只对它的长度变化与排序变动进行监听。元素属性变化只会通知它们各自的视图刷新函数，不会通知装着它们的监控数组。

```
function convert(val) {//转换数组元素为一个个 ViewModel
    if (Array.isArray(val)) {
        return val.$id ? val : Collection(val);
    } else if (avalon.type(val) === "Object") {
        return val.$id ? val : modelFactory(val);
    } else {
        return val;
    }
}
function Collection(list) {
    var collection = list.map(convert);
    collection.$id = generateID();//这个用于标识它是个监控对象
    collection[subscribers] = []; //订阅者列表
    collection.isCollection = true;
    var dynamic = modelFactory({
        length: list.length //用于监控数组长度的变化
    });
    "push,pop,shift,unshift,splice".replace(rword, function(method) {
        var nativeMethod = Array.prototype[method];
        collection[method] = function() {
            var len = this.length;
            var args = [].slice.call(arguments);
            if (/push|unshift|splice/.test(method)) {
                args = args.map(convert);                    //处理新增元素
            }
            var ret = nativeMethod.apply(this, args);    //调用原生方法
            notifySubscribers(this, method, args, len);//向视图刷新函数发出消息
            dynamic.length = this.length;
            return ret;
        };
    });
    "sort,reverse".replace(rword, function(method) {
//.......
    });
    collection.clear = function() {
//.......
    };
    collection.update = function() { //强制刷新页面
        notifySubscribers(this, "update", []);
        return this;
    };
    collection.size = function() {
        return dynamic.length;
    };
    collection.remove = function(item) {
//.......
    };
    collection.removeAt = function(index) {
//.......
    };
    collection.removeAll = function(all) {
//.......
    };
```

```
    return collection;
}
```

与监控数组息息相关的 each 绑定，它里面会生成一个 updateListView 刷新函数，然后刷新函数作为订阅者放到监控数组的 subscribers 队列中。当数组的 push、unshift 等方法被调用时，它们除了调用原生数组方法干活外，还将“unshift”、“shift”、“pop”等方法名，用户参数及调用前的数组长度，传到 updateListView 中，以刷新页面的对应区域。

```
bindingHandlers["each"] = function(data, scopes) {
    var list = parseExpr(data.value, scopes, data);
    //.....
    function updateListView(method, args, len) {
        switch (method) {
            case "push":
                //略。这里进行 DOM 操作
                break;
            case "unshift":
                //略
                break;
            case "pop":
                //略
                break;
            case "shift":
                //略
                break;
            case "splice":
                //略
                break;
            case "clear":
                //略
                break;
            case "update":
                //略
                break;
        }
    }
    if ((list || {}).isCollection) {
        list[subscribers].push(updateListView);
    }
};
```

数组的大多数方法可拆解为 3 种操作，添加、删除、移动。在这之前，我们需要保存一份模板，用于复制。大多数框架的做法都很一般，把 each 绑定的那个元素的所有子节点都复制到一个文档碎片中。然后当监控数组添加新元素时，复制一份，在对应位置上插进去。例如，push、unshift、splice 都有添加操作，添加多少个元素就复制多少次。删除操作就是 removeChild。移动就是 insertBefore。听起来很简单，但如果我们的模板的结构并不是只有一个元素，例如下面的结构：

```
<div ms-each-el="array" ms-controller="array">
    <h3>{{el}}</h3>
    <p>这是第{{$index}}个了。</p>
```

```
    <p>这是第{{$index}}个了。</p>
    <hr/>
</div>
```

监控数组 array 有 10 个元素，我们需要将第 2 个移动第 7 个位置去。除非你总是保持每个片断中每个子节点的引用，这个否则很难实现。为此 avalon 发明了路标系统——在每个模板的前面插入一个注释节点，内容为这个监控数组的 ID 值。ID 为监控数组的 ID 值加上它自身的索引值，这样以后移动、删除等操作，先找到这个注释节点，然后再找到下一个注释节点，就能确认它所要操作的范围。

下面是它的 JavaScript 调用与生成的 HTML 结构。

```
avalon.ready(function() {
    avalon.define("array", function(vm) {
        vm.array = avalon.range(0, 10)
    })
    avalon.scan();
})
```

具体如图 16.10 所示。

▲图 16.10

这些注释元素的 ID 依次是：

```
avalonzactrhj9ty6z8k1910h7bcgxvf0,
avalonzactrhj9ty6z8k1910h7bcgxvf1,
avalonzactrhj9ty6z8k1910h7bcgxvf2,
avalonzactrhj9ty6z8k1910h7bcgxvf3,
```

```
avalonzactrhj9ty6z8k1910h7bcgxvf4,
.....
```

要实现这个，就要在最初的模板添加一个注释节点（见 each 绑定源码）。

```
var doc = parent.ownerDocument;
var view = doc.createDocumentFragment();
var comment = doc.createComment(list.$id);
view.appendChild(comment);
while (parent.firstChild) {
    view.appendChild(parent.firstChild);
}
```

然后每添加一个新元素就复制这个模板 view 一次，将它的第一个注释元素改一改。

```
nextTick(function() {
    forEach(list, function(index, item) {
        addItemView(index, item, data);
    });
});

function addItemView(index, item, data) {
    var scopes = data.scopes;
    var list = data.list;
    var parent = data.element;
    var doc = parent.ownerDocument;
    var scope = createItemModel(index, item, list, data.args);
    scopes = [scope].concat(scopes);
    var view = data.view.cloneNode(true);//复制模板
    var textNodes = [];
    var elements = [];
    for (var node = view.firstChild; node; node = node.nextSibling) {
        if (node.nodeType === 1) {
            elements.push(node);
        } else if (node.nodeType === 3) {
            textNodes.push(node);
        } else if (node.nodeType === 8) {
            node.id = node.nodeValue + index; //设置路标★★★★
            node.$scope = scope;
            node.$view = view.cloneNode(false);
        }
    }
    parent.insertBefore(view, list.place || null);//插入到正确的位置，没有就插到最后
    for (var i = 0; node = elements[i++]; ) {
        scanTag(node, scopes.concat(), doc); //扫描元素节点
    }
   //扫描文本节点
}
```

由于用户可能从数组中间删掉一些元素，会导致路标少了一块，比如 0、1、2、3、4 变成了 0、2、3、4，又如 unshift 方法，则会让路标变成 0、1、2、0、1、2、3、4，因此我们需要一个重置路标的方法。在中间插入某一个元素时，我们需要先找到这个注释节点，又需要一个方法。它们的实

现如下。

```
function findIndex(elem, index) { //寻找路标
    for (var node = elem.firstChild; node; node = node.nextSibling) {
        if (node.id === node.nodeValue + index) {
            return node;
        }
    }
}

function resetIndex(elem, name) { //重置路标
    var index = 0;
    for (var node = elem.firstChild; node; node = node.nextSibling) {
        if (node.nodeType === 8 && node.nodeValue === name) {
            if (node.id !== name + index) {
                node.id = name + index;
                node.$scope.$index = index;
            }
            index++;
        }
    }
}
```

除了添加与删除，我们还有移动操作，这主要发生在 reverse 与 sort 方法中。由于移动节点，难免会发生回流，因此各 MVVM 框架在这里做了大量文章。像 knockout，使用最小编辑距离算法，得到一个数组转换成一个数组最少的操作步骤，然后根据这些步骤进行移动。有兴趣可以参看下面的链接：

https://github.com/SteveSanderson/knockout/blob/master/src/binding/editDetection/compareArrays.js

avalon 通过一个 updateListView，确保子模板的数量与监控数组的元素个数一致，当用户调用 reverse、sort 方法，只需回填数据，就能达到目的。

不过我们还是有无法避免大量 reflow 的情况，比如新添加元素时，我们得复制一份文档碎片，结合当下的 ViewModel 进行数据填充。在现有的绑定中，html 绑定与 text 绑定都会导致节点增加或减少。因此，我们最好先处理完像 class、disabled 等消耗少的绑定，再处理它们。要做到这一点，就需要引入零秒延迟（setTimeout(fn, 0)），像复杂的 DOM 操作放到渲染队列的后面，先把简单的执行了。

在之前的异步操作章节中已经列举过一些比 setTimeout 0 更快的异步 API，我们也可以在这里应用一下。setTimeout 0 在 IE 下的最短时钟间隔为 15.6ms，显然这个时间段里，前面的 DOM 操作应该做完了，不会让浏览器等太久。avalon 从一个非常著名的 Promise 库中把它的定时器拆了出来，可以分享一下。

```
var nextTick;
if (typeof setImmediate === "function") {
    // In IE10, Node.js 0.9+, or https://github.com/NobleJS/setImmediate
    nextTick = setImmediate.bind(window);
} else {
    (function() { //否则用一个链表来维护所有待执行的回调
        var head = {
```

```
        task: void 0,
        next: null
    };
    var tail = head;
    var maxPendingTicks = 2;
    var pendingTicks = 0;
    var queuedTasks = 0;
    var usedTicks = 0;
    var requestTick = void 0;

    function onTick() {
        --pendingTicks;
        if (++usedTicks >= maxPendingTicks) {
            usedTicks = 0;
            maxPendingTicks *= 4;
            var expectedTicks = queuedTasks && Math.min(
                    queuedTasks - 1,
                    maxPendingTicks);
            while (pendingTicks < expectedTicks) {
                ++pendingTicks;
                requestTick();
            }
        }

        while (queuedTasks) {
            --queuedTasks;
            head = head.next;
            var task = head.task;
            head.task = void 0;
            task();
        }

        usedTicks = 0;
    }
    nextTick = function(task) {
        tail = tail.next = {
            task: task,
            next: null
        };
        if (
                pendingTicks < ++queuedTasks && pendingTicks < maxPendingTicks) {
            ++pendingTicks;
            requestTick();
        }
    };
    //然后找一个最快响应的异步 API 来执行这个链表
    //你可以用 postMessage、image.onerror、xhr.onreadychange、MutationObserver
    //最差还有个 setTimeout 0 殿后，见 http://jsperf.com/postmessage
    if (typeof MessageChannel !== "undefined") { //管道通信 API
        var channel = new MessageChannel();
        channel.port1.onmessage = onTick;
        requestTick = function() {
            channel.port2.postMessage(0);
        };
    } else {
```

```
            requestTick = function() { //IE6-8
                setTimeout(onTick, 0);
            };
        }
    })();
}
avalon.nextTick = nextTick;
```

可能有人觉得这样有点大题小做，但这其实只是 JavaScript 从刀耕火种到工业化的必经之路，就像后端数据库面对海量数据量与访问量，不也发展出分库、分表、主从、集群、负载均衡等技术吗？！emberjs 与 angular 不单单是引入异步 API 进行缓冲，还把整个 Promise 库搞进来，目的是在异步捕捉错误。在维护状态上，emberjs 不单单是依赖于 ViewModel，还引入了状态机。在移动复制删除一大片节点，emberjs 则视情况使用 W3C range API[①]，这些新 API 几乎没什么人知道，只有写框架的人“孤芳自赏”了。

总而言之，像监控数组那样成片操作 HTML 的东西，要注意的东西太多了，就像 jQuery 的集化操作那样，最后还是衍生了 set all get first、链式操作、无 new 实例化等概念。更何况 JavaScript 界才刚刚进入 MVVM 这个领域不到一两年，不要指望有什么成套理论产出，更多的是炒作与噱头。这时候，我们最可依赖的伙伴还是设计模式，经过本土化的设计模式。正如我在一篇博文提到的那样：

双向绑定实现的同步机制，像工业大革命那样极大地提高了生产效率和代码的维护性。 由于是大生产，就有大生产的范儿。Java 早进入这时代了，留下了 23 个设计模式与分层架构与 IoC 等经验。了解 WPF 框架的人都知道，一个简单的 dependent property，就把设计模式这本书里面的模式用掉一大半了。分析 WPF 框架代码的话，简直就是看一本设计模式的百科全书。这也一一应用到前端 MVVM！knockoutjs、avalonjs 这些框架里面闪烁着后端长期熔铸的精华！

结语

至此，本书就说完了，其实将本章放到最后是个投巧的做法。MVVM 基本上颠覆了 jQuery 那一套以 DOM 中心的体系，MVVM 的出发点是数据，核心是数据。数据是底层，是心脏，数据变了，作为表层的 UI 就会跟着变。如果用户修改了 UI 元素上的值，相当于透过 UI 元素直接修改了底层的数据。为了让用户专致于数据，许多绑定在名字上就带着各种操作节点的功能，如 ms-html、ms-class、ms-click 等，把这些原本是由用户处理的代码交给框架处理，用户只需要在目标节点上声明一下，最多传一两个参数，将它与 ViewModel 关联起来。DOM 原本如此常用的工作全部被一笔带过了。或者有时 DOM 很复杂，$watch 回调可以让你做这些边角处理。

难道 jQuery 就这样退出舞台吗？不，没有一个库比它处理 DOM 的能力更强，在浏览器的世界你总要与 DOM 打交道。把 jQuery 作为 MVVM 一个水下运作单元是个不错的选择。多亏了 jQuery 团队，许多生僻的浏览器特性与 Bug 都被发掘出来，给出侦测的手段与收复的办法。如果自己写，也只是实现了一个半成品的 jQuery。世界上有太多 jQuery like 库，在 angular 内有个 jqLite，在 avalon 里面也有一个迷你 jQuery 对象。网站越大，用户越多，兼容的浏览器就越多，这时就是 jQuery 发

① https://github.com/emberjs/ember.js/blob/master/packages/metamorph/lib/main.js

挥作用的时候。

在 MVVM 中，jQuery 的这几个功能最有用——样式操作、属性操作、事件系统。上面提过了，如果大规模地移动删除节点，knockout、emberjs 都提出一套更为卓绝的办法。数据缓存上，HTML5 的 data-*特性节点更为实用，起码在移除节点，我们不需要调用专门的 removeData 方法。不可否认，jQuery 的 Ajax 非常强大，但当它被路由系统覆盖起来时，我们是不需要这么多配置项的。动画引擎，bootstrap 那一套基于 CSS3 动画其实够用了。jQuery 引以为豪的选择器压底儿没用，因为我们会扫描 DOM，这比 jQuery 遍历 DOM 的次数更少，并且选择器其实会加重 HTML 与 JavaScript 的耦合度，特别是一些结构伪类。

MVVM 让我们换一个视角来看待浏览器的世界，会发现别样的精彩。在过去，jQuery 等把原生 DOM 接口涂上一层厚厚的“水泥”，因此前端的世界得以高速发展。但 jQuery 没有太多关于状态管理的东西，最接近的 data，其次是在元素上添加一个类名，有就进入分支 1，没有就进入分支 2。因此业务越复杂，前端就很容易变成意大利面了。MVVM 把这两个都干掉了，因此用它组织代码，会比 jQuery 少许多，功能越丰富就越能体现出其优势。为了与 DOM 相分离，更易于测试，这也有利于 JavaScript 朝更工业化的方向发展。

当然，最后可以说的，我们写框架、写库、写组件，只有别出心裁才能出奇制胜。过去 jQuery 已经证明了这一点。通过本书的学习，希望大家能构建一个完整的知识树，在这之上出奇制胜吧！