

JavaScript 核心及 实践

第 0.9.9 版 alpha 版

作者：邱俊涛

版权声明

1. 未经作者书面许可,任何其他个人或组织均不得以任何形式将本书的全部或部分內容用作商业用途。
2. 本书的电子版本可以在作为学习,研究的前提下,自由发布,但均需保留完整的作者信息及此声明。
3. 作者保留其他一切与本作品相关之权利。

邱俊涛

2011 年 1 月 25 日

本书历史简表:

第 0 版 alpha 版	2010 年 1 月
第 0 版 bate 版	2010 年 5 月
第 0 版正式版	2011 年 1 月 25 日
第 0.1 版 alpha 版	2011 年 2 月 16 日
第 0.2 版 alpha 版	2011 年 2 月 18 日
第 0.3 版 alpha 版	2011 年 5 月 2 日
第 0.4 版 alpha 版	2012 年 1 月 26 日
第 0.9 版 alpha 版	2012 年 1 月 28 日
第 0.9.9 版 alpha 版	2012 年 2 月 3 日

作者信息:

姓名	邱俊涛 (abruzzi)
邮件	juntao.qiu@gmail.com
主页	http://www.icodeit.org/

目录

前言	8
本书组织结构	9
如何使用本书	11
致谢	12
第一章 概述	13
1.1 JavaScript 简史	13
1.1.1 动态网页	13
1.1.2 浏览器之战	14
1.1.3 标准	14
1.2 JavaScript 语言特性	14
1.2.1 动态性	14
1.2.2 弱类型	15
1.2.3 解释与编译	16
1.3 JavaScript 应用范围	16
1.3.1 客户端 JavaScript	17
1.3.2 服务端 JavaScript	19
1.3.3 其他应用中的 JavaScript	22
基础部分	26
第二章 基本概念	26
2.1 数据类型	26
2.1.1 基本数据类型	26
2.1.2 对象类型	27
2.1.3 基本类型与对象间的转换	28
2.1.4 类型的判断	30
2.2 变量	31
2.2.1 基本类型和引用类型	31
2.2.2 变量的作用域	32
2.3 运算符	33
2.3.1 中括号运算符([])	33
2.3.2 点运算符(.)	34
2.3.3 相等与等同运算符	35
第三章 对象与 JSON	38
3.1 Javascript 对象	38
3.1.1 对象的属性	38
3.1.2 属性与变量	39
3.1.3 原型对象及原型链	40
3.1.4 this 指针	42
3.2 使用对象	43
3.3 JSON 及其使用	44

第四章 函数.....	47
4.1 函数对象.....	47
4.1.1 创建函数.....	47
4.1.2 函数的参数.....	48
4.2 函数作用域.....	50
4.2.1 词法作用域.....	50
4.2.2 调用对象.....	52
4.3 函数上下文.....	52
4.4 call 和 apply.....	53
4.5 使用函数.....	54
4.5.1 赋值给一个变量.....	54
4.5.2 赋值为对象的属性.....	54
4.5.3 作为参数传递.....	55
4.5.4 作为函数的返回值.....	55
第五章 数组.....	57
5.1 数组的特性.....	57
5.2 使用数组.....	58
5.2.1 数组的基本方法使用.....	58
5.2.2 删除数组元素.....	62
5.2.3 遍历数组.....	64
第六章 正则表达式.....	65
6.1 正则表达式基础概念.....	65
6.1.1 元字符与特殊字符.....	65
6.1.2 范围及重复.....	66
6.1.3 分组与引用.....	68
6.2 使用正则表达式.....	69
6.2.1 创建正则表达式.....	69
6.2.2 String 中的正则表达式.....	71
6.3 实例: JSFilter.....	72
第七章 闭包.....	74
7.1 闭包的特性.....	74
7.2 闭包的用途.....	76
7.2.1 匿名自执行函数.....	76
7.2.2 缓存.....	77
7.2.3 实现封装.....	77
7.3 应该注意的问题.....	79
7.3.1 内存泄漏.....	79
7.3.2 上下文的引用.....	79
第八章 面向对象的 Javascript.....	81
8.1 原型继承.....	81
8.1.1 引用.....	83
8.1.2 new 操作符.....	84
8.2 封装.....	85
8.3 工具包 Base.....	86

8.4 实例：事件分发器	89
第九章 函数式的 Javascript	99
9.1 匿名函数	100
9.2 高阶函数	100
9.2.1 JavaScript 中的高阶函数	100
9.2.2 C 语言中的高阶函数	102
9.2.3 Java 中的高阶函数	103
9.3 闭包与柯里化	104
9.3.1 柯里化的概念	105
9.3.2 柯里化的应用	105
9.4 一些例子	107
9.4.1 函数式编程风格	107
9.4.2 Y-结合子	109
9.4.3 其他实例	110
高级主题	112
第十章 核心概念深入	112
10.1 原型链	112
10.1.1 原型对象与原型链	112
10.1.2 构造器	115
10.2 执行期上下文	117
10.3 活动对象	119
10.4 作用域链	120
10.5 this 值	122
10.5.1 词法作用域	122
10.5.2 this 的上下文	123
第十一章 客户端的 JavaScript	125
11.1 客户端 JavaScript 执行环境	125
11.2 文档对象模型(DOM)	126
11.3 事件驱动模型	127
11.4 与服务器端交互(Ajax)	128
11.5 调试	130
11.5.1 FireFox	131
11.5.2 Chrome	134
11.6 客户端的 MVC	135
11.7 Javascript/Ajax 框架	138
第十二章 前端 JavaScript 框架:jQuery	139
12.1 jQuery 简介	139
12.2 jQuery 基础	142
12.2.1 jQuery 选择器	142
12.2.2 对 DOM 的操作	143
12.2.3 对 CSS 的操作	146
12.2.4 事件处理	148
12.2.5 实用函数	149
12.3 jQuery 实例	151

第十三章 JavaScript 引擎	155
13.1 使用 SpiderMonkey	155
13.1.1 SpiderMonkey 简介	155
13.1.2 JavaScript 对象与 C 对象间的转换关系	156
13.1.3 基本代码模板	157
13.1.4 执行 JavaScript 代码	160
13.1.5 C 程序调用 JavaScript 函数	162
13.1.6 JavaScript 程序调用 C 函数	163
13.1.7 在 C 程序中定义 JavaScript 对象	165
13.2 SpiderMonkey 的实现	167
13.2.1 虚拟机概述	167
13.2.2 SpiderMonkey 体系结构	168
13.2.3 jsval 类型	168
13.2.4 对象	169
13.3 V8 引擎概览	169
13.3.1 V8 引擎基本概念	169
13.3.2 V8 引擎使用示例	170
13.3.3 使用 C++ 变量	171
13.3.4 调用 C++ 函数	173
13.3.5 使用 C++ 类	174
第十四章 Java 应用中的 JavaScript	178
14.1 脚本化基础	178
14.1.1 脚本化框架	178
14.2 使用 Rhino 引擎	179
14.2.1 直接对脚本求值	179
14.2.2 传递 Java 对象	179
14.2.3 调用脚本内的函数	181
14.2.4 在脚本中使用 Java 资源	183
14.2.5 实现 Java 接口	184
14.3 实例: sTodo	185
14.3.1 sTodo 简介	185
14.3.2 sTodo 的插件机制	187
14.3.3 sTodo 中的脚本	189
14.4 实例: 可编程计算器 phoc	191
14.4.1 phoc 简介	191
14.4.2 phoc 中的脚本	193
第十五章 服务器端的 JavaScript	197
15.1 node.js	197
15.1.1 node.js 简介	197
15.1.2 node.js 使用示例	198
15.1.3 node.js 实例	201
15.2 CouchDB	205
15.2.1 CouchDB 简介	205
15.2.1 CouchDB 使用	207

15.2.3 CouchDB 实例	209
附录一 一些 JavaScript 技巧.....	217
创建对象	217
访问对象的属性	217
遍历对象	217
名称空间	218
附录二 使用 graphviz 绘图	221
graphviz 简介	221
基础知识	222
第一个 graphviz 图.....	222
定义顶点和边的样式	223
进一步修改顶点和边样式.....	224
子图的绘制	225
数据结构的可视化	226
一个 hash 表的数据结构	226
绘制 hash 表的数据结构	227
hash 表的实例.....	228
软件模块组成图	230
Apache httpd 模块关系.....	230
模块组成关系	231
状态图	233
有限自动机示意图.....	233
OSGi 中模块的生命周期图	234
其他实例	235
一棵简单的抽象语法树(AST)	235
简单的 UML 类图.....	236
状态图	238
附录.....	239
附录三 ExtJs 简介及示例.....	241
ExtJs 简介	241
ExtJS 示例	242
后记	245

前言

大概很少有程序设计语言可以担当得起“优美”这两个字的，我们可以评论一个语言的语法简洁，代码可读性高(尽管这一点主要依赖于开发人员的水平，而并非语言本身)，但是几乎不会说哪个语言是优美的，而 JavaScript 则是一个例外。

程序设计语言，主要可以分为两种，一种是我们平时接触较多的，工业级的程序设计语言如 C/C++， JAVA， Object Pascal(DELPHI)等，从本质上来讲，这些语言是基于程序存储原理，即冯·诺依曼体系的，一般被称为命令式编程语言，而另一种，是根据阿隆左·丘奇的 lambda 演算而产生的，如 Lisp， Scheme， 被称为函数式编程语言。这两个体系一般情况下是互不干涉，泾渭分明的，这一现象直到 JavaScript 的逐渐成熟之后才被打破。函数式语言被认为是晦涩难懂的，学院派的，使用 Lisp 的似乎都是些披头散发，满口之乎者也而且性情古怪的大学教授。Emacs， 这个被它的爱好者誉为世界上最强大，最好用的编辑器的插件机制，就是基于一个 Lisp 的方言完成的，Emacs 应该可以算是函数式语言比较成功的运用案例之一，后来又出现了 Gimp， 一个 Linux 平台下的图形图像处理软件，它使用另一个 Lisp 的方言来进行自己的扩展。如此看来，函数式编程似乎已经被人们所接受了，然而事实并非如此简单，那种“前缀的操作符”，“一切皆函数”的理念在短时间内是无法被诸如“数据结构+算法=程序”之类箴言束缚住思想的冯·诺依曼程序员所接受，直到 JavaScript 的出现。

JavaScript 被称为具有 C 的语法的 Lisp，它完美的结合了这两个体系。C 的语法使得它迅速的被习惯命令式编程的程序员所接受，从而得到了推广，而 Lisp 的内核则使得其代码以优美的形式和内涵超过了其他的命令式语言，从而成为非常流行的一门语言，根据 TIOBE 的编程语言排行统计，JavaScript 一直排在前十位(在第 8-第 9 之间徘徊)。然而，要转变长时间形成的编程习惯殊非易事，两个体系之间的一些理念具有根本性的差异，解决这个问题正是本书的一个目的，通过深入的学习 JavaScript 的内核思想，我们可以将另一个体系的思想应用在日常的工作中，提高代码的质量。

JavaScript 并不像它表现出来的那样简单，大多数 JavaScript 程序员在无需深入理解 JavaScript 运行机制的情况下也可以写出可运行的代码，但是这样的代码几乎没有可维护性，当出现了一个隐藏的较深的 bug 的情况下，程序员往往无法很快的定位错误可能的源头，从而花费大量的时间来进行 alert。因此，理解 JavaScript 运行机制，以及澄清其容易被误解的特性将有助于杜绝这种现象。

邱俊涛

2010 年 5 月于昆明

本书组织结构

- 第一章，介绍 JavaScript 的历史，语言特性及应用范围，从大的视角来概述 JavaScript。
- 第二章，介绍基本的 JavaScript 概念，这部分的概念十分重要，直接影响到后面章节的内容的理解。
- 第三章，对象，是 JavaScript 中最核心，也最容易被误解的部分，所以抽出一个章节来描述 JavaScript 的对象，涉及到 JSON(JavaScript Object Notation), 以及一些如何使用 JavaScript 对象的实例。
- 第四章，函数，是 JavaScript 中的另一个重要的概念，与大多数为人熟知的命令式语言中的函数(方法)概念不一样的是，JavaScript 中的函数涉及到更复杂的形式，比如匿名函数，闭包等。
- 第五章，数组 Array 在 JavaScript 中是一个保留字，与其他语言不同的是，Array 更像是一个哈希表，而对 Array 的操作则可以类比为栈结构，或者 Lisp 中的 List，总之，数组是一个复杂的对象，值得我们花时间深入探究。
- 第六章，正则表达式，正则表达式是一个伟大的发明，在很多的应用程序和程序设计语言中都会出现它的身影，我们当然需要讨论其在 JavaScript 中的使用。其中包括正则表达式的规则及一些简单的实例。
- 第七章，闭包，是函数式编程语言所特有的一种结构，使用它可以是代码更简洁，有是更是非它不可，但是，不小心的设计往往容易造成内存泄漏(特别是在 IE 浏览器的早期版本中)。
- 第八章，JavaScript 作为一个语言，它本身又是“可编程”(programmable)的，你可以使用你自己设想的任意方式来组建你的代码，当然包括流行的 OO。本章的最后包含一个事件分发器的实现，通过这个例子我们可以较好的掌握面向对象的 JavaScript。
- 第九章，这一章，我们来探讨 JavaScript 中的函数式编程的主题，如果有 Lisp 或者 Scheme 之类的语言经验，可以从某种程度上获得共鸣。如果不了解其他的函数式语言，则应该仔细阅读这一章，对你的编程思想大有裨益。
- 第十章，在前面的章节中，陆续而分散的讨论过 JavaScript 语言中的一些核心概念如对象，函数，作用域，闭包等等，但是不够深入，这一章则进行更详细的讨论。
- 第十一章，这一章主要讨论客户端 JavaScript 的一些基础知识，以及一些好的编程实践，为后两章做一个引子。
- 第十二章，讨论前端 JavaScript 框架 jQuery 的基本概念及使用，最后是一个简单的实例：一个简易的 todo 系统。
- 第十三章，讨论 JavaScript 引擎，主要包括目前较为流行的三种引擎：Mozilla 的 Spidermonkey，Google 的 V8，以及前 Sun 的 rhino，其中剖析 SpiderMonkey 的工作机制，V8 及 rhino 仅演示一些基本的使用。
- 第十四章，讨论 JavaScript 在 Java 应用程序中的使用，实例部分介绍笔者开发的待办事项管理工具 sTodo 及可编程计算器 phoc 的设计及实现，以及其中如何使用 JavaScript 引擎来完成脚本化。
- 第十五章，讨论 JavaScript 在服务器端的应用，分别讨论了 node.js 及一个面向文档的数据库系统 CouchDB 的基本使用。

- 附录一中讨论了一些常用 **JavaScript** 技巧。
- 附录二中讨论了 **graphviz** 绘图工具的使用，本书中大量的结构图等图例均采用 **graphviz** 绘制。
- 附录三中为前端 **JavaScript** 框架 **ExtJS** 的简单实例。

如何使用本书

本书中前半部分中讲解的大部分内容与客户端的 JavaScript 没有关系，如函数，对象，数组，闭包等概念都属于 JavaScript 内核本身，是与环境无关的，为了过早的陷入具体的应用之中，笔者开发了一个简单但可用的 JavaScript 执行环境(JSEvaluator)，核心采用 Mozilla 的一个开源的 JavaScript 引擎 Rhino，这个引擎为纯 Java 实现，不包含任何 DOM 元素，故可以较为轻便的运行书中的例子而不必纠缠与浏览器差异之类的问题中。

JSEvaluator 是一个简单的 JavaScript 的 IDE，提供基本的代码编辑功能，点击运行按钮可以运行当前活动标签中的脚本，结果将在 JSEvaluator 的控制台中打印出来。本书的后半部分，如第七章的事件分发器以及第九章的客户端 JavaScript，则需要在浏览器中运行。具体的章节会有详细说明。

程序设计是一门实践的艺术，读者在阅读本书的同时，应该做一些练习，那样才可能对书本中的知识点有好的理解。建议读者一边阅读，一边将书中的例子在 JSEvaluator 中运行，查看结果，并可以自己修改这些例子，以期得到更好的效果。

致谢

正如所有技术类书籍的前言部分所描述的那样,几乎没有任何一位作者宣称自己**独力**的完成了某一部著作。在进行广泛而深入的研究技术本身时,我们必须在别人研究的基础上展开工作,才能更好,更高效的进入该领域。

是的,本书的撰写过程中,参考了众多的资料,文献,以及相关的标准规范等,当然也和很多的朋友进行过讨论,这些朋友有现实世界中的同事,也有在虚拟网络中素未谋面的同好。在这里,一并感谢。

本书绸缪于 2009 年 12 月份,2010 年 1 月开始动笔,期间经历了很多生活上的杂事,感谢我的妻子孙女士在此期间对我的支持,没有她,此书无法与诸位读者见面。在本书的动笔之前的研究期间,笔者得到前公司的胡东先生的谆谆的教诲和不厌其烦的启发,胡东先生是一位沉湎于自己精心构筑的技术世界而不能自拔的老师,在此一并感谢。

第一章 概述

1.1 JavaScript 简史

在 20 世纪 90 年代，也就是早期的 WEB 站点上，所有的网页内容都是静态的，所谓静态是指，除了点击超链接，你无法通过任何方式同页面进行交互，比如让页面元素接受事件，修改字体等。人们于是迫切的需要一种方式来打破这个局限，于是到了 1996 年，网景 (Netscape) 公司开始研发一种新的语言 Mocha，并将其嵌入到自己的浏览器 Netscape 中，这种语言可以通过操纵 DOM (Document Object Model, 文档对象模型) 来修改页面，并加入了对鼠标事件的支持。Mocha 使用了 C 的语法，但是设计思想上主要从函数式语言 Scheme 那里取得了灵感。当 Netscape 2 发布的时候，Mocha 被改名为 LiveScript，当时可能是想让 LiveScript 为 WEB 页面注入更多的活力。后来，考虑到这个脚本语言的推广，网景采取了一种宣传策略，将 LiveScript 更名为 JavaScript，目的是为了跟当时非常流行的面向对象语言 Java 发生暧昧的关系。这种策略显然颇具成效，以至于到现在很多初学者还会为 JavaScript 和 Java 的关系而感到困惑。

JavaScript 取得成功之后，确实为页面注入了活力，微软也紧接着开发自己的浏览器脚本语言，一个是基于 BASIC 语言的 VBScript，另一个是跟 JavaScript 非常类似的 Jscript，但是由于 JavaScript 已经深入人心，所以在随后的版本中，微软的 IE 几乎是将 JavaScript 作为一个标准来实现。当然，两者仍然有不兼容的地方。1996 年后期，网景向欧洲电脑厂商协会 (ECMA) 提交了 JavaScript 的设计，以申请标准化，ECMA 去掉了一些实现，并提出了 ECMA-262 标准，并确定 JavaScript 的正式名字为 ECMAScript，但是 JavaScript 的名字已经深入人心，故本书中仍沿用 JavaScript 这个名字。

1.1.1 动态网页

WEB 页面在刚开始的时候，是不能动态修改其内容的，要改变一个页面的内容，需要先对网站上的静态 HTML 文件进行修改，然后需要刷新浏览器。后来出现的 JSP, ASP 等服务器端语言可以为页面提供动态的内容，但是如果没有 JavaScript 则无法在服务器返回之后动态的在前端修改页面，也无法有诸如鼠标移上某页面元素则高亮该元素之类的效果，因此 JavaScript 的出现大大的丰富了页面的表现，提高了用户体验。

而当 AJAX 流行起来之后，更多的非常绚丽的 WEB 应用涌现了，而且呈越来越多的趋势，如 Gmail, Google Map, Google Reader, Remember the milk, facebook 等等优秀的 WEB2.0 应用，都大量的使用了 JavaScript 以及基于 JavaScript 技术的 AJAX。

这些优秀的 Web2.0 应用提供动态的内容，客户端可以局部更新页面上的视觉元素，比如对地图的放大/缩小，新邮件到来后的提醒等等。用户体验较静态页面得到了很大的提升。事实上，后期的很多应用均建立在 B/S 架构上，因为 HTML 构筑 UI 的成本较桌面开发为低。因此基于 Web 的应用开始占有一定的份额，正在逐步替换 C/S 架构的桌面应用。

动态网页的好处在于，客户端的负载较小，只需要一个浏览器即可，主要的负担在服务器端，这就节约了客户端的开发成本。

1.1.2 浏览器之战

1994 年网景公司成立，并推出了自己的浏览器的免费版本 **Netscape**，很快就占据了浏览器市场。到了 1995 年，微软公司开始加入，并很快发布了自己的 **Internet Explorer 1.0**。在随后的几年间，网景和微软公司不停的发布新版本的浏览器，支持更多的新功能。很快，这两者的目标就不是如何做好浏览器，而是在对手擅长的方面压制对方。比如，网景的浏览器 **Netscape** 标榜速度快，**IE** 就要开发出比网景更快的浏览器，而对自身的安全漏洞，渲染能力等方面放任自流。这样纯粹为了竞争而竞争，无疑对广大的用户来说是非常不利的事情。但是一直到 1997 年，网景的浏览器 **Netscape** 份额大概在 72%，而 **IE** 只占到 18%。

但是，**IE** 在随后的版本 **IE4.0** 的时候开始支持 **W3C** 的标准，并且在网页的动态性方面加入了很大的支持。事实上，这时候的网景已经不敌慢慢崛起的微软帝国了，微软利用自己的操作系统 **Windows**，在其中捆绑了 **IE** 浏览器，而且完全免费。这样，**IE** 的市场占有率开始抽过 **Netscape**。当出现一家独大的场面之后，标准化就显得步履维艰了，开发人员开始只为 **IE** 浏览器编写代码，因为不需要在其他任何浏览器上运行，因此所有的网页都很可能只能在 **IE** 下运行，或者只能在 **IE** 下效果才可以得到保证。

1998 年，网景的 **Netscape** 开放了源码，分散在世界各地的开发人员开始贡献代码和不定，使得这个浏览器变得越来越出色，到了 2004 年，**Firefox**，作为这个项目中的一个产品，推出了 1.0 版本。这个以 **Mozilla** 为基础的浏览器才慢慢开始发展。一方面，捆绑在 **windows xp** 系统中的 **IE6.0** 中漏洞百出，大量的蠕虫病毒都会攻击 **IE** 浏览器，而 **Firefox** 则没有这方面的问题，安全且高效。因此从 2006 年到 2008 年，**Firefox** 的市场占有率开始回升，**IE** 的平均占有率大约为 85%，**Firefox** 平均占有率为 15%。而某些地区，如在欧洲，**Firefox** 的占有率高达 20%。

到了 2009 年，由于反垄断法即开源项目的影 响，**windows 7** 不再捆绑 **IE** 浏览器，这样，用户可以有权利选择自己需要的浏览器，但这并不意味着 **Firefox** 胜出，**IE** 落败。事实上，这更促进了其他的浏览器如 **Safari**，**Opera**，**Chrome** 的发展。

1.1.3 标准

1.2 JavaScript 语言特性

JavaScript 是一门动态的，弱类型，基于原型的脚本语言。在 **JavaScript** 中“一切皆对象”，在这一方面，它比其他的 **OO** 语言来的更为彻底，即使作为代码本身载体的 **function**，也是对象，数据与代码的界限在 **JavaScript** 中已经相当模糊。虽然它被广泛的应用在 **WEB** 客户端，但是其应用范围远远未局限于此。下面就这几个特点分别介绍：

1.2.1 动态性

动态性是指，在一个 **JavaScript** 对象中，要为一个属性赋值，我们不必事先创建一个字段，只需要在使用的时候做赋值操作即可，如下例：

```
//定义一个对象
var obj = new Object();

//动态创建属性name
obj.name = "an object";

//动态创建属性sayHi
obj.sayHi = function(){
    return "Hi";
}

obj.sayHi();
```

假如我们使用 Java 语言，代码可能会是这样：

```
class Obj{
    String name;
    Function sayHi;

    public Obj(String name, Function sayHi){
        this.name = name;
        this.sayHi = sayHi;
    }
}

Obj obj = new Obj("an object", new Function());
```

动态性是非常有用的，这个我们在第三章会详细讲解。

1.2.2 弱类型

与 Java, C/C++不同，JavaScript 是弱类型的，它的数据类型无需在声明时指定，解释器会根据上下文对变量进行实例化，比如：

```
//定义一个变量s，并赋值为字符串
var s = "text";
print(s);

//赋值s为整型
s = 12+5;
print(s);
```

```
//赋值s为浮点型
s = 6.3;
print(s);

//赋值s为一个对象
s = new Object();
s.name = "object";

print(s.name);
```

结果为:

```
text
17
6.3
Object
```

可见, JavaScript 的变量更像是一个容器, 类似与 Java 语言中的顶层对象 `Object`, 它可以是任何类型, 解释器会根据上下文自动对其造型。

弱类型的好处在于, 一个变量可以很大程度的进行复用, 比如 `String` 类型的 `name` 字段, 在被使用后, 可以赋值为另一个 `Number` 型的对象, 而无需重新创建一个新的变量。不过, 弱类型也有其不利的一面, 比如在开发面向对象的 JavaScript 的时候, 没有类型的判断将会是比较麻烦的问题, 不过我们可以通过别的途径来解决此问题。

1.2.3 解释与编译

通常来说, JavaScript 是一门解释型的语言, 特别是在浏览器中的 JavaScript, 所有的主流浏览器都将 JavaScript 作为一个解释型的脚本来进行解析, 然而, 这并非定则, 在 Java 版的 JavaScript 解释器 `rhino` 中, 脚本是可以被编译为 Java 字节码的。Google 的 V8 引擎则直接将 JavaScript 代码编译为本地代码, 无需解释。

解释型的语言有一定的好处, 即可以随时修改代码, 无需编译, 刷新页面即可重新解释, 可以实时看到程序的结果, 但是由于每一次都需要解释, 程序的开销较大; 而编译型的语言则仅需要编译一次, 每次都运行编译过的代码即可, 但是又丧失了动态性。

我们将在第九章和第十章对两种方式进行更深入的讨论。

1.3 JavaScript 应用范围

当 JavaScript 第一次出现的时候, 是为了给页面带来更多的动态, 使得用户可以与页面进行交互为目的的, 虽然 JavaScript 在 WEB 客户端取得了很大的成功, 但是 ECMA 标准并没有局限其应用范围。事实上, 现在的 JavaScript 大多运行与客户端, 但是仍有部分运行于服务器端, 如 `Servlet`, `ASP` 等, 当然, JavaScript 作为一个独立的语言, 同样可以运行在其他的应用程序中, 比如 Java 版的 JavaScript 引擎 `Rhino`, C 语言版的

SpiderMonkey 等，使用这些引擎，可以将 JavaScript 应用在任何应用之中。

1.3.1 客户端 JavaScript

客户端的 JavaScript 随着 AJAX 技术的复兴，越来越凸显了 JavaScript 的特点，也有越来越多的开发人员开始进行 JavaScript 的学习，使用 JavaScript，你可以使你的 WEB 页面更加生动，通过 AJAX，无刷新的更新页面内容，可以大大的提高用户体验，随着大量的 JavaScript 包如 jQuery， ExtJS， Mootools 等的涌现，越来越多的绚丽，高体验的 WEB 应用被开发出来，这些都离不开幕后的 JavaScript 的支持。



图 JavaScript 实现的一个 WEB 幻灯片

浏览器中的 JavaScript 引擎也进行了长足的发展，比如 FireFox 3，当时一个宣传的重点就是速度比 IE 要快，这个速度一方面体现在页面渲染上，另一方面则体现在 JavaScript 引擎上，而 Google 的 Chrome 的 JavaScript 引擎 V8 更是将速度发展到了极致。很难想象，如果没有 JavaScript，如今的大量的网站和 WEB 应用会成为什么样子。

我们可以看几个例子，来说明客户端的 JavaScript 的应用程度：

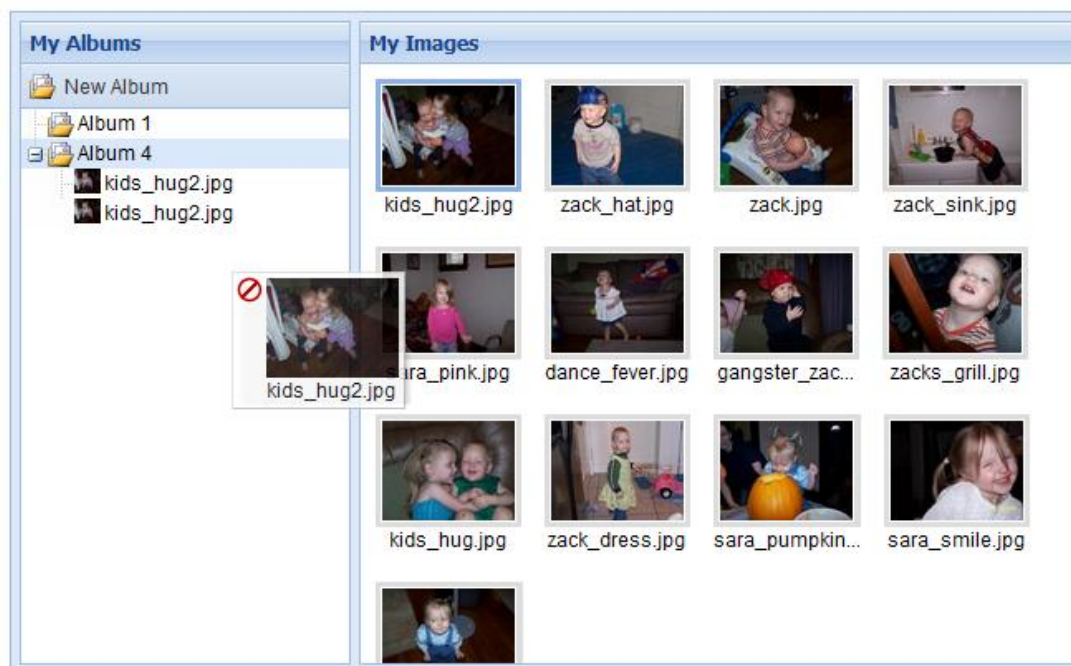
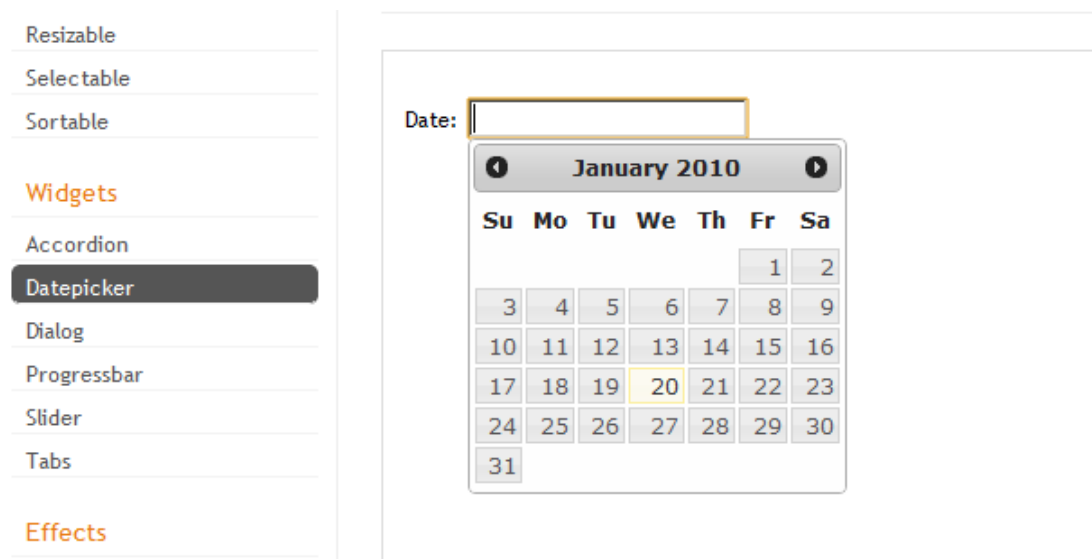


图 ExtJS 实现的一个网络相册，ExtJS 是一个非常优秀的 JavaScript 库

	Company	Price	Change	% Change
1	3m Co		0.02	0.03
2	Alcoa Inc		0.42	1.47
3	American Express Company		0.01	0.02
4	American International Group, Inc.			0.49
5	AT&T Inc.	\$31.61		-1.54
6	Caterpillar Inc.	\$67.27		1.39
7	Citigroup, Inc.	\$49.37		0.04
8	Exxon Mobil Corp	\$68.10	-0.43	-0.64
9	General Electric Company	\$34.14	-0.08	-0.23
10	General Motors Corporation	\$30.27	1.09	3.74
11	Hewlett-Packard Co.	\$36.53	-0.03	-0.08
12	Honeywell Intl Inc	\$38.77	0.05	0.13
13	Intel Corporation	\$19.88	0.31	1.58
14	Johnson & Johnson	\$64.72	0.06	0.09
15	Merck & Co., Inc.	\$40.96	0.41	1.01
16	Microsoft Corporation	\$25.84	0.14	0.54
17	The Coca-Cola Company	\$45.07	0.26	0.58

图 ExtJS 实现的一个表格，具有排序，编辑等功能

当然，客户端的 JavaScript 各有侧重，jQuery 以功能见长，通过选择器，可以完成 80% 的页面开发工作，并且提供强大的插件机制，下图为 jQuery 的 UI 插件：



总之，随着 Ajax 的复兴，客户端的 JavaScript 得到了很大的发展，网络上流行着大量的优秀的 JavaScript 库，现在有一个感性的认识即可，我们在后边的章节会择其尤要者进行详细讨论。

1.3.2 服务端 JavaScript

相对客户端而言，服务器端的 JavaScript 相对平淡很多，但是随着 JavaScript 被更多的开发人员重视，JavaScript 在服务器端也开始迅速的发展起来，Helma，Apache Sling 等等。在服务器端的 JavaScript 比客户端少了许多限制，如本地文件的访问，网络，数据库等。

一个比较有意思的服务端 JavaScript 的例子是 Aptana 的 Jaxer，Jaxer 是一个服务器端的 Ajax 框架，我们可以看这样一个例子(例子来源于 jQuery 的设计与实现这 John Resig):

```
<html>
<head>
  <script src="http://code.jquery.com/jquery.js"
runat="both"></script>
  <script>
    jQuery(function($) {
      $("form").submit(function() {
        save( $("textarea").val() );
        return false;
      });
    });
  </script>
  <script runat="server">
    function save( text ){
```

```

    Jaxer.File.write("tmp.txt", text);
  }
  save.proxy = true;

  function load(){
    $("textarea").val(
      Jaxer.File.exists("tmp.txt") ? Jaxer.File.read("tmp.txt") : "" );
  }
</script>
</head>
<body onserverload="load()" >
  <form action="" method="post">
    <textarea></textarea>
    <input type="submit"/>
  </form>
</body>
</html>

```

runat 属性说明脚本运行在客户端还是服务器端，**client** 表示运行在客户端，**server** 表示运行在服务器端，而 **both** 表示可以运行在客户端和服务器端，这个脚本可以访问文件，并将文件加载到一个 **textarea** 的 DOM 元素中，还可以将 **textarea** 的内容通过 Form 表单提交给服务器并保存。

再来看另一个例子，通过 **Jaxer** 对数据库进行访问：

```

<script runat="server">
  var rs = Jaxer.DB.execute("SELECT * FROM table");
  var field = rs.rows[0].field;
</script>

```

通过动态，灵活的语法，再加上对原生的资源(如数据库，文件，网络等)操作的支持，服务器端的 **JavaScript** 应用将会越来越广泛。

当 **Google** 的 **JavaScript** 引擎 **V8** 出现以后，有很多基于 **V8** 引擎的应用也出现了，其中最著名，最有前景的当算 **Node.js** 了，下面我们来看一下 **Node.js** 的例子：

```

var sys = require('sys'),
    http = require('http');

http.createServer(function (req, res) {
  setTimeout(function () {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.sendBody('Hello World');
    res.finish();
  }, 2000);

```

```
}).listen(8000);  
  
sys.puts('Server running at http://127.0.0.1:8000/');
```

保存这个脚本为 sayHello.js, 然后运行:

```
node sayHello.js
```

程序将会在控制台上打印:

```
Server running at http://127.0.0.1:8000/
```

访问 <http://127.0.0.1:8000>, 两秒钟之后页面会响应: Hello, World。

再来看另一个官方提供的例子:

```
var tcp = require('tcp');  
  
var server = tcp.createServer(function (socket) {  
  socket.setEncoding("utf8");  
  socket.addListener("connect", function () {  
    socket.send("hello\r\n");  
  });  
  socket.addListener("receive", function (data) {  
    socket.send(data);  
  });  
  socket.addListener("eof", function () {  
    socket.send("goodbye\r\n");  
    socket.close();  
  });  
});  
  
server.listen(7000, "localhost");
```

访问 localhost 的 7000 端口, 将建立一个 TCP 连接, 编码方式为 utf-8, 当客户端连接到来时, 程序在控制台上打印

```
hello
```

当接收到新的数据时, 会将接收到的数据原样返回给客户端, 如果客户端断开连接, 则向控制台打印:

```
goodbay
```

Node 提供了丰富的 API 来简化服务器端的网络编程，由于 Node 是基于一个 JavaScript 引擎的，因此天生的就具有动态性和可扩展性，因此在开发网络程序上，确实是一个不错的选择。

1.3.3 其他应用中的 JavaScript

通过使用 JavaScript 的引擎的独立实现，比如 Rhino, SpliderMonkey, V8 等，可以将 JavaScript 应用到几乎所有的领域，比如应用程序的插件机制，高级的配置文件分析，用户可定制功能的应用，以及一些类似与浏览器场景的比如 Mozilla 的 ThunderBrid, Mozilla 的 UI 框架 XUL，笔者开发的一个 Todo 管理器 sTodo(在第十五章详细讨论)等。



图 sTodo 一个使用 JavaScript 来提供插件机制的 Java 桌面应用

Java 版的 JavaScript 引擎原生的可以通过使用 Java 对象，那样将会大大提高 JavaScript 的应用范围，如数据库操作，服务器内部数据处理等。当然，JavaScript 这种动态语言，在 UI 方面的应用最为广泛。

著名的 Adobe reader 也支持 JavaScript 扩展，并提供 JavaScript 的 API 来访问 PDF 文档内容，可以通过 JavaScript 来定制 Adobe Reader 的界面以及功能等。

```
app.addItem({
  cName: "-",
```

```
// menu divider
cParent: "View",
// append to the View menu
cExec: "void(0);"
});

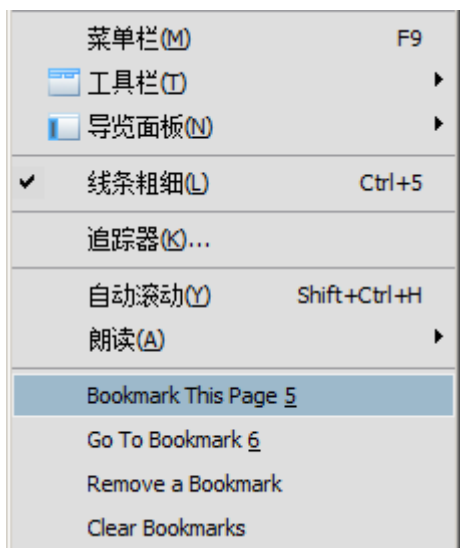
app.addItem({
  cName: "Bookmark This Page &5",
  cParent: "View",
  cExec: "AddBookmark();",
  cEnable: "event.rc= (event.target != null);"
});

app.addItem({
  cName: "Go To Bookmark &6",
  cParent: "View",
  cExec: "ShowBookmarks();",
  cEnable: "event.rc= (event.target != null);"
});

app.addItem({
  cName: "Remove a Bookmark",
  cParent: "View",
  cExec: "DropBookmark();",
  cEnable: "event.rc= (event.target != null);"
});

app.addItem({
  cName: "Clear Bookmarks",
  cParent: "View",
  cExec: "ClearBookmarks();",
  cEnable: "event.rc= true;"
});
```

为 Adobe Reader 添加了 4 个菜单项，如图：



UltraEdit 编辑器在新的版本中也加入了对脚本化的支持，使用的脚本语言正是 JavaScript，用户可以通过脚本来控制编辑器的一些公开对象，下面是一个简单的实例：

 A screenshot of the UltraEdit editor window showing a file named 'hello.js'. The code is as follows:


```

1 (function(){
2     UltraEdit.activeDocument.write("hello, world");
3 }());
  
```

 The code is highlighted in blue. The editor has a ruler at the top showing line numbers 0, 10, 20, 30, 40, 50.

UltraEdit.activeDocument 表示当前活动文档，write 方法会将字符串写入当前活动文档的缓冲区，运行这个脚本，可以得到下面的效果：

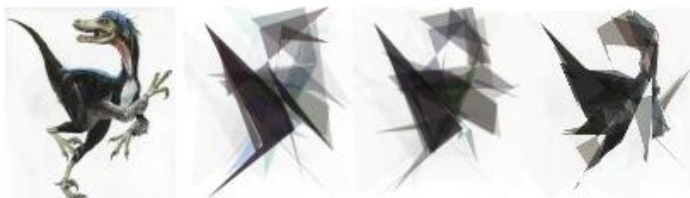
 A screenshot of the UltraEdit editor window showing the output of the JavaScript script. The text 'hello, world' is displayed on the first line of the document. The editor has a ruler at the top showing line numbers 0, 10, 20, 30. The text is highlighted in blue.

当然，这里仅是一个简单实例，关于 UltraEdit 脚本化的详细信息请参考 UltraEdit 的帮助文件或相关文档。

另一个比较有意思的 JavaScript 实例为一个在线的遗传算法的演示，给定一个图片，然后将一些多边形(各种颜色)拼成一个图片，拼图的规则为使用遗传算法，使得这些多变形组成的图片与目标图片最为相似：



- 50 polygons (6-vertex)
- 4,358 beneficial mutations
- 227,852 candidates
- 95.97% fitness
- Thanks to Quialiss.
- Images from different runs.



- 50 polygons (6-vertex)
- 718+ beneficial mutations
- 22,440+ candidates
- 95.24% fitness
- Images from different runs.



- 100 polygons (5-vertex)
- 10,490 beneficial mutations
- 2,161,018 candidates
- 95.03% fitness
- Thanks to [Asa](#), Will, Nic & Yuku.
- Images from different runs.

可见，JavaScript 在其他方面的也得到了广泛的应用。

基础部分

本部分开始正式进入 JavaScript 内核部分，包括 JavaScript 的对象，函数，数组，闭包，其中各个主题中会详细涉及到很多相关的，容易被误解的知识点，比如对象中的属性概念，函数中的匿名函数，作用域链，上下文，运行环境，JavaScript 数组与其他语言数组的区别以及其强大之处等等。

本部分为随后内容的基础，需要完全掌握，则对随后的内容可以更好的理解。此部分虽名为基础部分，实则不包含 JavaScript 的基本语法，如流控制语句，变量的声明等内容，这些部分比较基础，和其他的编程语言相差不大，如果需要可以参阅其他书籍，同时，在本部分的例子中会穿插一些基本的语法知识，如果曾经学过 C 语言或者其他任何程序设计语言都不会有阅读障碍。

第二章 基本概念

本章将聚焦于 JavaScript 中的基本概念，这些概念与传统语言有比较大的不同，因此单独列出一章来做专门描述，理解本章的概念对书中后续章节的概念，代码的行为等会有很大的帮助，读者不妨花比较大的时间在本章，即使你对 JavaScript 已经比较熟悉，也建议通读本章。

本章主要讲述 JavaScript 中的数据类型(基本类型与引用类型)，变量(包括变量的作用域)，操作符(主要是一些较为常见，但是不容易从字面上理解的操作符)。由于 JavaScript 中的“一切皆对象”，在掌握了这些基本的概念之后，读者就可以较为轻松的理解诸如作用域，调用对象，闭包，currying 等等较难理解的概念了。

2.1 数据类型

有程序设计经验的读者肯定知道，在 C 或者 Java 这样的语言中，数据是有类型的，比如用以表示用户名的属性是字符串，而一个雇员的年龄则是一个数字，表示 UI 上的一个开关按钮的数据模型则为布尔值等等，对数字可能还可以细分为浮点数，整型数，整型数又可能分为长整型和短整型，总而言之，它们都表示语言中的数据的值的类型。

JavaScript 中的数据类型分为两种：基本数据类型和对象类型，其中对象类型包含对象，数组，以及函数，基本数据类型在必要是会被隐式的转换为对象。

2.1.1 基本数据类型

在 JavaScript 中，包含六种基本的数据类型，字符串(string)，数值(number)，布尔值(boolean)，undefined，null 及对象(object)。下面是一些简单的例子：

```
var str = "Hello, world";//字符串
var i = 10;//整型数
var f = 2.3;//浮点数

var b = true;//布尔值
```

我们可以分别查看变量的值及变量的类型：

```
print(str);
print(i);
print(f);
print(b);

print(typeof str);
print(typeof i);
print(typeof f);
print(typeof b);
print(typeof x);
```

注意，在此处使用的 `print()` 函数为 `rhino` 解释器的顶层对象的方法，可以用来打印字符串，通常情况下，在客户端，程序员多使用 `alert()` 进行类似的动作，`alert()` 是浏览器中 JavaScript 解释器的顶层对象 (`window`) 的一个方法。

```
Hello, world
10
2.3
true
```

```
string
number
number
Boolean
undefined
```

在 JavaScript 中，所有的数字，不论是整型，浮点型，都属于“数字”基本类型。`typeof` 是一个一元的操作符，在本章的另外一个小节会专门讲到。

2.1.2 对象类型

这里提到的对象不是对象本身，而是指一种类型，我们在第三章会对对象进行详细的讨论，此处的对象包括，对象(属性的集合，即键值的散列表)，数组(有序列表)，函数(包含可执行的代码)。

对象类型是一种复合的数据类型，其基本元素由基本数据类型组成，当然不限于基本类

型，比如对象类型中的值可以是其他的对象类型实例，我们通过例子来说明：

```
var str = "Hello, world";
var obj = new Object();
obj.str = str;
obj.num = 2.3;

var array = new Array("foo", "bar", "zoo");

var func = function(){
    print("I am a function here");
}
```

可以看到，对象具有属性，如 `obj.str`、`obj.num`，这些属性的值可以是基本类型，事实上还可以更复杂，我们来看看他们的类型：

```
print(typeof obj);
print(typeof array);
print(typeof func);
```

//将打印出

```
object
object
function
```

读者可能会对 `print(typeof array)` 打印出 `object` 感到奇怪，事实上，对象和数组的界限并不那么明显(事实上它们是属于同一类型的)，但是他们的行为却非常不同，本书的后续章节将两个重要的数据类型做了分别介绍。

2.1.3 基本类型与对象间的转换

类似与 Java 中基本数据类型的自动装箱拆箱，JavaScript 也有类似的动作，基本数据类型在做一些运算时，会临时包装一个对象，做完运算后，又自动释放该对象。我们可以通过几个例子来说明：

```
var str = "JavaScript Kernal";
print(str.length); //打印 17
```

`str` 为一个字符串，通过 `typeof` 可知其 `type` 为“string”，而：

```
var str2 = new String("JavaScript Kernal");

print(typeof str2);
```

可知, `str2` 的 `type` 为“object”, 即这两者并不相同, 那么为什么可以使用 `str.length` 来得到 `str` 的长度呢? 事实上, 当使用 `str.length` 时, JavaScript 会自动包装一个临时的 `String` 对象, 内容为 `str` 的内容, 然后获取该对象的 `length` 属性, 最后, 这个临时的对象将被释放。

而将对象转换为基本类型则是通过这样的方式: 通过调用对象的 `valueOf()` 方法来取得对象的值, 如果和上下文的类型匹配, 则使用该值。如果 `valueOf` 取不到值的话, 则需要调用对象的 `toString()` 方法, 而如果上下文为数值型, 则又需要将此字符串转换为数值。由于 JavaScript 是弱类型的, 所以 JavaScript 引擎需要根据上下文来“猜测”对象的类型, 这就使得 JavaScript 的效率比编译型的语言要差一些。`valueOf()` 的优先级要高于 `toString()`。

`valueOf()` 的作用是, 将一个对象的值转换成一种合乎上下文需求的基本类型, `toString()` 则名副其实, 可以打印出对象对应的字符串, 当然前提是你已经“重载”了 `Object` 的 `toString()` 方法。

事实上, 这种转换规则会导致很多的问题, 比如, 所有的非空对象, 在布尔值环境下, 都会被转成 `true`, 比如:

```
function convertTest() {
    if(new Boolean(false) && new Object() &&
        new String("") && new Array()){
        print("convert to boolean")
    }
}

convertTest();//convert to Boolean
```

初学者容易被 JavaScript 中的类型转换规则搞晕掉, 很多情况下会觉得那种写法看着非常别扭, 其实只需要掌握了规则, 这些古怪的写法会大大的提高代码的性能, 我们通过例子来学习这些规则:

```
var x = 3;
var y = x + "2";// => 32
var z = x + 2;// => 5

print(y);
print(z);
```

通常可以在 JS 代码中发现这样的代码:

```
if(datamodel.item){
    //do something...
}else{
    datamodel.item = new Item();
}
```

```
}
```

这种写法事实上具有更深层次的含义：

应该注意到，`datamodel.item` 是一个对象(字符串，数字等)，而 `if` 需要一个 `boolean` 型的表达式，所以这里进行了类型转换。在 `JavaScript` 中，如果上下文需要 `boolean` 型的值，则引擎会自动将对象转换为 `boolean` 类型。转换规则为，如果该对象非空，则转换为 `true`，否则为 `false`。因此我们可以采取这种简写的形式。

而在传统的编程语言(强类型)中，我们则需要：

```
if(datamodel.item != null){
    //do something...
}else{
    datamodel.item = new Item();
}
```

2.1.4 类型的判断

前面讲到 `JavaScript` 特性的时候，我们说过，`JavaScript` 是一个弱类型的语言，但是有时我们需要知道变量在运行时的类型，比如，一个函数的参数预期为另一个函数：

```
function handleMessage(message, handle){
    return handle(message);
}
```

当调用 `handleMessage` 的函数传递的 `handle` 不是一个函数则 `JavaScript` 引擎会报错，因此我们有必要在调用之前进行判断：

```
function handleMessage(message, handle){
    if(typeof handle == "function"){
        return handle(message);
    }else{
        throw new Error("the 2nd argument should be a function");
    }
}
```

但是，`typeof` 并不总是有效的，比如下面这种情况：

```
var obj = {};
var array = ["one", "two", "three", "four"];

print(typeof obj); //object
print(typeof array); //object
```

运行结果显示，对象 `obj` 和数组 `array` 的 `typeof` 值均为“object”，这样我们就无法准确判断了，这时候，可以通过调用 `instanceof` 来进行进一步的判断：

```
print(obj instanceof Array);//false
print(array instanceof Array);//true
```

第一行代码返回 `false`，第二行则返回 `true`。因此，我们可以将 `typeof` 操作符和 `instanceof` 操作符结合起来进行判断。

2.2 变量

变量，即通过一个名字将一个值关联起来，以后通过变量就可以引用到该值，比如：

```
var str = "Hello, World";
var num = 2.345;
```

当我们下次要引用“Hello, Wrold”这个串进行某项操作时，我们只需要使用变量 `str` 即可，同样，我们可以用 `10*num` 来表示 `10*2.345`。变量的作用就是将值“存储”在这个变量上。

2.2.1 基本类型和引用类型

在上一小节，我们介绍了 JavaScript 中的数据类型，其中基本类型如数字，布尔值，它们在内存中都有固定的大小，我们通过变量来直接访问基本类型的数据。而对于引用类型，如对象，数组和函数，由于它们的大小在原则上是不受任何限制的，故我们通过对其引用的访问来访问它们本身，引用本身是一个地址，即指向真实存储复杂对象的位置。

基本类型和引用类型的区别是比较明显的，我们来看几个例子：

```
var x = 1;//数字x，基本类型
var y = x;//数字y，基本类型
print(x);
print(y);
```

```
x = 2;//修改x的值
```

```
print(x);//x的值变为2
print(y);//y的值不会变化
```

运行结果如下：

```
1  
1  
2  
1
```

这样的运行结果应该在你的意料之内，没有什么特别之处，我们再来看看引用类型的例子，由于数组的长度非固定，可以动态增删，因此数组为引用类型：

```
var array = [1,2,3,4,5];  
var arrayRef = array;
```

```
array.push(6);  
print(arrayRef);
```

引用指向的是地址，也就是说，引用不会指向引用本身，而是指向该引用所对应的实际对象。因此通过修改 `array` 指向的数组，则 `arrayRef` 指向的是同一个对象，因此运行效果如下：

```
1,2,3,4,5,6
```

2.2.2 变量的作用域

变量被定义的区域即为其作用域，全局变量具有全局作用域；局部变量，比如声明在函数内部的变量则具有局部作用域，在函数的外部是不能直接访问的。比如：

```
var variable = "out";  
  
function func(){  
    var variable = "in";  
    print(variable);//打印"in"  
}  
  
func();  
print(variable);//打印"out"
```

应该注意的是，在函数内 `var` 关键字是必须的，如果使用了变量而没有写 `var` 关键字，则默认的操作是对全局对象的，比如：

```
var variable = "out";  
  
function func(){  
    variable = "in";//注意此variable前没有var关键字  
    print(variable);
```



```
}  
  
func();  
print(variable); //全局的变量 variable 被修改
```

由于函数 `func` 中使用 `variable` 而没有关键字 `var`, 则默认是对全局对象 `variable` 属性做的操作(修改 `variable` 的值为 `in`), 因此此段代码会打印:

```
in  
in
```

关于作用域, 会引入活动对象及作用域链的概念, 这个在本书的高级主题部分专门讨论。

2.3 运算符

运算符, 通常是容易被忽略的一个内容, 但是一些比较古怪的语法现象仍然可能需要用到运算符的结合率或者其作用来进行解释, `JavaScript` 中, 运算符是一定需要注意的地方, 有很多具有 `JS` 编程经验的人仍然免不了被搞得晕头转向。

我们在这一节主要讲解这样几个运算符, 诸如算术运算, 位运算等和其他的主程序设计语言类似, 在这里不做讨论:

2.3.1 中括号运算符([])

中括号(`[]`)运算符可用在数组对象和对象上, 从数组中按下标取值:

```
var array = ["one", "two", "three", "four"];  
array[0]
```

而`[]`同样可以作用于对象, 一般而言, 对象中的属性的值是通过点(`.`)运算符来取值, 如:

```
var object = {  
  field : "self",  
  printInfo : function(){  
    print(this.field);  
  }  
}
```

```
object.field;  
object.printInfo();
```

但是考虑到这样一种情况, 我们在遍历一个对象的时候, 对其中的属性的键(`key`)是一无所知的, 我们怎么通过点(`.`)来访问呢? 这时候我们就可以使用`[]`运算符:

```
for(var key in object){
    print(key + ":" + object[key]);
}
```

运行结果如下:

```
field:slef
printInfo:function (){
    print(this.field);
}
```

2.3.2 点运算符(.)

点运算符的左边为一个对象(属性的集合), 右边为属性名, 应该注意的是右边的值除了作为左边的对象的属性外, 同时还可能是它自己的右边的值的对象:

```
var object = {
    field : "self",
    printInfo : function(){
        print(this.field);
    },
    outter:{
        inner : "inner text",
        printInnerText : function(){
            print(this.inner);
        }
    }
}
```

```
object.outter.printInnerText();
```

这个例子中, `outter` 作为 `object` 的属性, 同时又是 `printInnerText()` 的对象。

但是点(.)操作符并不总是可用的, 考虑这样一种情况, 如果一个对象的属性本身就包含点(.)的键(`self.ref`), 点操作符就无能为力了:

```
var ref = {
    id : "referencel",
    func : function(){
        return this.id;
    }
};
```

```
var obj = {
  id : "object1",
  "self.ref" : ref
};
```

当我们尝试访问 `obj` 的“`self.ref`”这个属性的时候：`obj.self.ref`，解释器会以为 `obj` 中有个名为 `self` 的属性，而 `self` 对象又有个 `ref` 的属性，这样会发生不可预知的错误，一个好的解决方法是使用中括号(`[]`)运算符来访问：

```
print(obj["self.ref"].func());
```

在这种情况下，中括号操作符成为唯一可行的方式，因此，建议在不知道对象的内部结构的时候(比如要遍历对象来获取某个属性的值)，一定要使用中括号操作符，这样可以避免一些意想不到的 `bug`。

2.3.3 相等与等同运算符

运算符 `==` 读作相等，而运算符 `===` 则读作等同。这两种运算符操作都是在 JavaScript 代码中经常见到的，但是意义则不完全相同，简而言之，相等操作符会对两边的操作数做类型转换，而等同则不会。我们还是通过例子来说明：

```
print(1 == true);
print(1 === true);
print("" == false);
print("" === false);

print(null == undefined);
print(null === undefined);
```

运行结果如下：

```
true
false
true
false
true
false
```

相等和等同运算符的规则分别如下：

相等运算符

如果操作数具有相同的类型，则判断其等同性，如果两个操作数的值相等，则返回 `true`(相等)，否则返回 `false`(不相等)。

如果操作数的类型不同，则按照这样的情况来判断：

- `null` 和 `undefined` 相等
- 其中一个是数字，另一个是字符串，则将字符串转换为数字，在做比较
- 其中一个是 `true`，先转换成 `1`(`false` 则转换为 `0`)在做比较
- 如果一个值是对象，另一个是数字/字符串，则将对象转换为原始值(通过 `toString()` 或者 `valueOf()`方法)
- 其他情况，则直接返回 `false`

等同运算符

如果操作数的类型不同，则不进行值的判断，直接返回 `false`

如果操作数的类型相同，分下列情况来判断：

- 都是数字的情况，如果值相同，则两者等同(有一个例外，就是 `NaN`，`NaN` 与其本身也不相等)，否则不等同
- 都是字符串的情况，与其他程序设计语言一样，如果串的值不等，则不等同，否则等同
- 都是布尔值，且值均为 `true/false`，则等同，否则不等同
- 如果两个操作数引用同一个对象(数组，函数)，则两者完全等同，否则不等同
- 如果两个操作数均为 `null/undefined`，则等同，否则不等同

比如：

```
var obj = {
  id : "self",
  name : "object"
};
```

```
var oa = obj;
var ob = obj;
```

```
print(oa == ob);
print(oa === ob);
```

会返回：

```
true
true
```

再来看一个对象的例子：

```
var obj1 = {
  id : "self",
  name : "object",
  toString : function(){
    return "object 1";
  }
}
```

```
var obj2 = "object 1";
```

```
print(obj1 == obj2);  
print(obj1 === obj2);
```

返回值为:

```
true  
false
```

`obj1` 是一个对象，而 `obj2` 是一个结构与之完全不同的字符串，而如果用相等操作符来判断，则两者是完全相同的，因为 `obj1` 重载了顶层对象的 `toString` 方法。

而不等(`!=`)和不等同(`!==`)，则与相等(`==`)/等同(`===`)相反。因此，在 JavaScript 中，使用相等/等同，不等/不等同的时候，一定要注意类型的转换，这里推荐使用等同/不等同来进行判断，这样可以避免一些难以调试的 `bug`。

第三章 对象与 JSON

JavaScript 对象与传统的面向对象中的对象几乎没有相似之处，传统的面向对象语言中，创建一个对象必须先有对象的模板：类，类中定义了对象的属性和操作这些属性的方法。通过实例化来构建一个对象，然后使用对象间的协作来完成一项功能，通过功能的集合来完成整个工程。而 JavaScript 中是没有类的概念的，借助 JavaScript 的动态性，我们完全可以创建一个空的对象(而不是类)，通过向对象动态的添加属性来完善对象的功能。

JSON 是 JavaScript 中对象的字面量，是对象的表示方法，通过使用 JSON，可以减少中间变量，使代码的结构更加清晰，也更加直观。使用 JSON，可以动态的构建对象，而不必通过类来进行实例化，大大的提高了编码的效率。

3.1 Javascript 对象

JavaScript 对象其实就是属性的集合(以及一个原型对象)，这里的集合与数学上的集合是等价的，即具有确定性，无序性和互异性，也就是说，给定一个 JavaScript 对象，我们可以明确的知道一个属性是不是这个对象的属性，对象中的属性是无序的，并且是各不相同的(如果有同名的，则后声明的覆盖先声明的)。

一般来说，我们声明对象的时候对象往往只是一个空的集合，不包含任何的属性，通过不断的添加属性，使得该对象成为一个有完整功能的对象，而不用通过创建一个类，然后实例化该类这种模式，这样我们的代码具有更高的灵活性，我们可以任意的增删对象的属性。

如果读者有 python 或其他类似的动态语言的经验,就可以更好的理解 JavaScript 的对象，JavaScript 对象的本身就是一个字典(dictionary)，或者 Java 语言中的 Map，或者称为关联数组，即通过键来关联一个对象，这个对象本身又可以是一个对象，根据此定义，我们可以知道 JavaScript 对象可以表示任意复杂的数据结构。

3.1.1 对象的属性

属性是由键-值对组成的，即属性的名字和属性的值。属性的名字是一个字符串，而值可以为任意的 JavaScript 对象(Javascript 中的一切皆对象，包括函数)。比如，声明一个对象：

```
//声明一个对象
var jack = new Object();
jack.name = "jack";
jack.age = 26;
jack.birthday = new Date(1984, 4, 5);

//声明另一个对象
var address = new Object();
address.street = "Huang Quan Road";
address.xno = "135";
```

```
//将addr属性赋值为对象address  
jack.addr = address;
```

这种声明对象的方式与传统的 OO 语言是截然不同的，它给了我们极大的灵活性来定制一个对象的行为。

对象属性的读取方式是通过点操作符(.)来进行的，比如上例中 `jack` 对象的 `addr` 属性，可以通过下列方式取得：

```
var ja = jack.addr;  
  
ja = jack["addr"];
```

后者是为了避免这种情况，设想对象有一个属性本身包含一个点(.)，这在 JavaScript 中是合法的，比如说名字为 `foo.bar`，当使用 `jack.foo.bar` 的时候，解释器会误以为 `foo` 属性下有一个 `bar` 的字段，因此可以使用 `jack[foo.bar]` 来进行访问。通常来说，我们在开发通用的工具包时，应该对用户可能的输入不做任何假设，通过 `[属性名]` 这种形式则总是可以保证正确性的。

3.1.2 属性与变量

在第二章，我们讲解了变量的概念，在本章中，读者可能已经注意到，这二者的行为非常相似，事实上，对象的属性和我们之前所说的变量其实是一回事。

JavaScript 引擎在初始化时，会构建一个全局对象，在客户端环境中，这个全局对象即为 `window`。如果在其他的 JavaScript 环境中需要引用这个全局对象，只需要在顶级作用域(即所有函数声明之外的作用域)中声明：

```
var global = this;
```

我们在顶级作用域中声明的变量将作为全局对象的属性被保存，从这一点上来看，变量其实就是属性。比如，在客户端，经常会出现这样的代码：

```
var v = "global";  
  
var array = ["hello", "world"];  
  
function func(id) {  
    var element = document.getElementById(id);  
    //对element做一些操作  
}
```

事实上相当于：

```
window.v = "global";

window.array = ["hello", "world"];

window.func = function(id) {
    var element = document.getElementById(id);
    //对element做一些操作
}
```

3.1.3 原型对象及原型链

原型(prototype), 是 JavaScript 特有的一个概念, 通过使用原型, JavaScript 可以建立其传统 OO 语言中的继承, 从而体现对象的层次关系。JavaScript 本身是基于原型的, 每个对象都有一个 prototype 的属性, 这个 prototype 本身也是一个对象, 因此它本身也可以有自己的原型, 这样就构成了一个链结构。

访问一个属性的时候, 解析器需要从下向上的遍历这个链结构, 直到遇到该属性, 则返回属性对应的值, 或者遇到原型为 null 的对象(JavaScript 的基对象 Object 的构造器的默认 prototype 有一个 null 原型), 如果此对象仍没有该属性, 则返回 undefined。

下面我们看一个具体的例子:

```
//声明一个对象base
function Base(name) {
    this.name = name;
    this.getName = function() {
        return this.name;
    }
}

//声明一个对象child
function Child(id) {
    this.id = id;
    this.getId = function() {
        return this.id;
    }
}

//将child的原型指向一个新的base对象
Child.prototype = new Base("base");

//实例化一个child对象
var c1 = new Child("child");
```



```
//c1本身具有getId方法
print(c1.getId());
//由于c1从原型链上"继承"到了getName方法, 因此可以访问
print(c1.getName());
```

得出结果:

```
child
base
```

由于遍历原型链的时候, 是有下而上的, 所以最先遇到的属性值最先返回, 通过这种机制可以完成继承及重载等传统的 OO 机制。

另外看一个例子并以图形的方式来说明:

```
function Person(name, age) {
  this.name = name;
  this.age = age;

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

  this.getAge = function() {
    return this.age;
  }
}

var tom = new Person("Tom", 38);
var jerry = new Person("Jerry", 6);
```

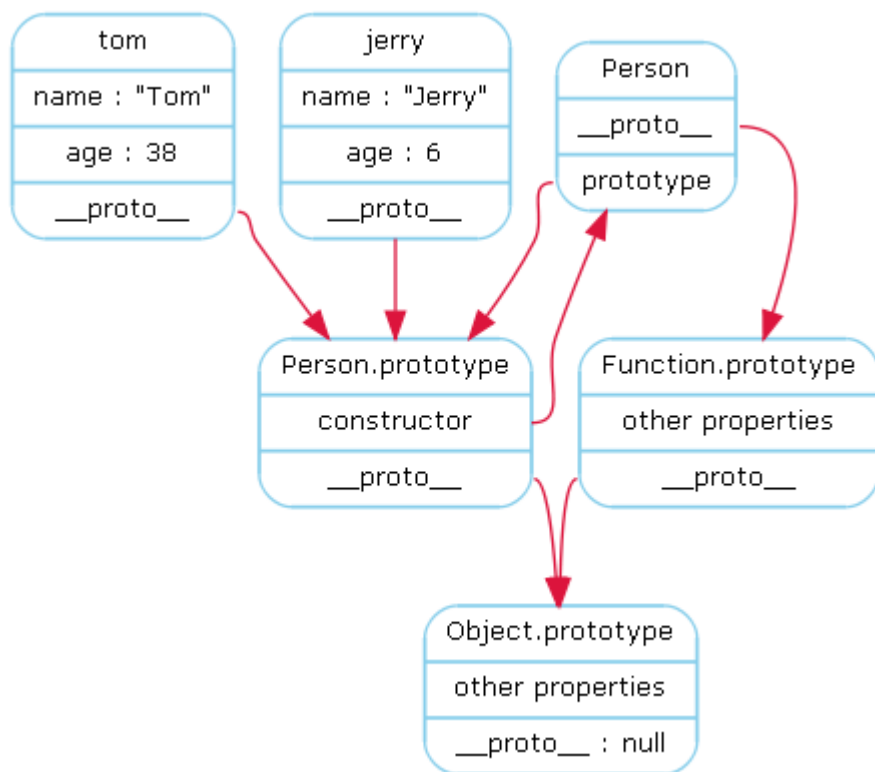


图 原型链

通过原型链，可以实现继承/重载等面向对象的 JavaScript 代码，当然这个机制并非基于类，而是基于原型，详细细节将在第 8 章讨论。

3.1.4 this 指针

JavaScript 中最容易使人迷惑的恐怕就数 this 指针了，this 指针在传统 OO 语言中，是在类中声明的，表示对象本身，而在 JavaScript 中，this 表示当前上下文，即调用者的引用。这里我们可以来看一个常见的例子：

```

//定义一个人，名字为jack
var jack = {
  name : "jack",
  age : 26
}

//定义另一个人，名字为abruzzo
var abruzzo = {
  name : "abruzzo",
  age : 26
}

//定义一个全局的函数对象

```

```
function printName() {
    return this.name;
}

//设置printName的上下文为jack, 此时的this为jack
print(printName.call(jack));
//设置printName的上下文为abruzzo, 此时的this为abruzzo
print(printName.call(abruzzo));
```

运行结果:

```
jack
Abruzzo
```

应该注意的是, `this` 的值并非函数如何被声明而确定, 而是被函数如何被调用而确定, 这一点与传统的面向对象语言截然不同, `call` 是 `Function` 内置对象上的一个方法, 详细描述在第四章。

3.2 使用对象

对象是 JavaScript 的基础, 我们使用 JavaScript 来完成编程工作就是通过使用对象来体现的, 这一小节通过一些例子来学习如何使用 JavaScript 对象:

对象的声明有三种方式:

- 通过 `new` 操作符作用于 `Object` 对象, 构造一个新的对象, 然后动态的添加属性, 从无到有的构建一个对象。
- 定义对象的“类”: 原型, 然后使用 `new` 操作符来批量的构建新的对象。
- 使用 JSON, 这个在下一节来进行详细说明。

这一节我们详细说明第二种方式, 如:

```
//定义一个"类", Address
function Address(street, xno) {
    this.street = street || 'Huang Quan Road';
    this.xno = xno || 135;
    this.toString = function() {
        return "street : " + this.street + ", No : " + this.xno;
    }
}

//定义另一个"类", Person
function Person (name, age, addr) {
    this.name = name || 'unknown';
```

```
    this.age = age;
    this.addr = addr || new Address(null, null);
    this.getName = function () {return this.name;}
    this.getAge = function(){return this.age;}
    this.getAddr = function(){return this.addr.toString();}
}

//通过new操作符来创建两个对象，注意，这两个对象是相互独立的实体
var jack = new Person('jack', 26, new Address('Qing Hai Road', 123));
var abruzzo = new Person('abruzzo', 26);

//查看结果
print(jack.getName());
print(jack.getAge());
print(jack.getAddr());

print(abruzzo.getName());
print(abruzzo.getAge());
print(abruzzo.getAddr());
```

运行结果如下：

```
jack
26
street : Qing Hai Road, No : 123
abruzzo
26
street : Huang Quan Road, No : 135
```

3.3 JSON 及其使用

JSON 全称为 JavaScript 对象表示法(JavaScript Object Notation)，即通过字面量来表示一个对象，从简单到复杂均可使用此方式。比如：

```
var obj = {
  name : "abruzzo",
  age : 26,
  birthday : new Date(1984, 4, 5),
  addr : {
    street : "Huang Quan Road",
    xno : "135"
  }
}
```

这种方式，显然比上边的例子简洁多了，没有冗余的中间变量，很清晰的表达了 `obj` 这样一个对象的结构。事实上，大多数有经验的 JavaScript 程序员更倾向与使用这种表示法，包括很多 JavaScript 的工具包如 jQuery, ExtJS 等都大量的使用了 JSON。JSON 事实上已经作为一种前端与服务器端的数据交换格式，前端程序通过 Ajax 发送 JSON 对象到后端，服务器端脚本对 JSON 进行解析，还原成服务器端对象，然后做一些处理，反馈给前端的仍然是 JSON 对象，使用同一的数据格式，可以降低出错的概率。使用 JSON 作为数据交换格式，在一定程度上比 XML 更高效，冗余更小。

而且，JSON 格式的数据本身是可以递归的，也就是说，可以表达任意复杂的数据形式。JSON 的写法很简单，即用花括号括起来的键值对，键值对通过冒号隔开，而值可以是任意的 JavaScript 对象，如简单对象 String, Boolean, Number, Null, 或者复杂对象如 Date, Object, 其他自定义的对象等。

JSON 的另一个应用场景是：当一个函数拥有多个返回值时，在传统的面向对象语言中，我们需要组织一个对象，然后返回，而 JavaScript 则完全不需要这么麻烦，比如：

```
function point(left, top){
    this.left = left;
    this.top = top;
    //handle the left and top
    return {x: this.left, y:this.top};
}
```

直接动态的构建一个新的匿名对象返回即可：

```
var pos = point(3, 4);
//pos.x = 3;
//pos.y = 4;
```

使用 JSON 返回对象，这个对象可以有任意复杂的结构，甚至可以包括函数对象。在实际的编程中，我们通常需要遍历一个 JavaScript 对象，事先我们对对象的内容一无所知。怎么做呢？JavaScript 提供了 `for..in` 形式的语法糖：

```
for(var item in json){
    //item为键
    //json[item]为值
}
```

这种模式十分有用，比如，在实际的 WEB 应用中，对一个页面元素需要设置一些属性，这些属性是事先不知道的，比如：

```
var style = {
    border:"1px solid #ccc",
    color:"blue"
};
```

然后，我们给一个 DOM 元素动态的添加这些属性：

```
for(var item in style){  
  //使用jQuery的选择器  
  $("div#element").css(item, style[item]);  
}
```

当然，jQuery 有更好的办法来做这样一件事，这里只是举例子。应该注意的是，我们在给 `$("#div#element")` 添加属性的时候，我们对 `style` 的结构是不清楚的。

另外比如我们需要收集一些用户的自定义设置，也可以通过公开一个 JSON 对象，用户将需要设置的内容填入这个 JSON，然后我们的程序对其进行处理。

```
function customize(options) {  
  this.settings = $.extend(default, options);  
}
```

第四章 函数

函数，在 C 语言之类的过程式语言中，是顶级的实体，而在 Java/C++ 之类的面向对象的语言中，则被对象包装起来，一般称为对象的方法。而在 JavaScript 中，函数本身与其他任何的内置对象在地位上是没有任何区别的，也就是说，**函数本身也是对象**。

总的来说，函数在 JavaScript 中可以：

- 被赋值给一个变量
- 被赋值为对象的属性
- 作为参数被传入别的函数
- 作为函数的结果被返回
- 用字面量来创建

4.1 函数对象

4.1.1 创建函数

创建 JavaScript 函数的一种不常用的方式(几乎没有人用)是通过 `new` 操作符来作用于 `Function` “构造器”：

```
var funcName = new Function( [argname1, [... argnameN,]] body );
```

参数列表中可以有任意多的参数，然后紧跟着是函数体，比如：

```
var add = new Function("x", "y", "return(x+y)");  
print(add(2, 4));
```

将会打印结果：

6

但是，谁会用如此难用的方式来创建一个函数呢？如果函数体比较复杂，那拼接这个 `String` 要花费很大的力气，所以 JavaScript 提供了一种语法糖，即通过字面量来创建函数：

```
function add(x, y) {  
    return x + y;  
}
```

或：

```
var add = function(x, y){
    return x + y;
}
```

事实上，这样的语法糖更容易使传统领域的程序员产生误解，`function` 关键字会调用 `Function` 来 `new` 一个对象，并将参数表和函数体准确的传递给 `Function` 的构造器。通常来说，在全局作用域(作用域将在下一节详细介绍)内声明一个对象，只不过是对一个属性赋值而已，比如上例中的 `add` 函数，事实上只是为全局对象添加了一个属性，属性名为 `add`，而属性的值是一个对象，即 `function(x, y){return x+y;}`，理解这一点很重要，这条语句在语法上跟：

```
var str = "This is a string";
```

并无二致。都是给全局对象动态的增加一个新的属性，如此而已。

为了说明函数跟其他的对象一样，都是作为一个独立的对象而存在于 JavaScript 的运行系统，我们不妨看这样一个例子：

```
function p(){
    print("invoke p by ()");
}
```

```
p.id = "func";
p.type = "function";
```

```
print(p);
print(p.id+":"+p.type);
print(p());
```

没有错，`p` 虽然引用了一个匿名函数(对象)，但是同时又可以拥有属性，完全跟其他对象一样，运行结果如下：

```
function (){
    print("invoke p by ()");
}
func:function
invoke p by ()
```

4.1.2 函数的参数

在 JavaScript 中，函数的参数是比较有意思的，比如，你可以将任意多的参数传递给一个函数，即使这个函数声明时并未制定形式参数，比如：


```
function adPrint(str, len, option){
    var s = str || "default";
    var l = len || s.length;
    var o = option || "i";

    s = s.substring(0, l);
    switch(o){
        case "u":
            s = s.toUpperCase();
            break;
        case "l":
            s = s.toLowerCase();
            break;
        default:
            break;
    }

    print(s);
}

adPrint("Hello, world");
adPrint("Hello, world", 5);
adPrint("Hello, world", 5, "l");//lower case
adPrint("Hello, world", 5, "u");//upper case
```

函数 `adPrint` 在声明时接受三个形式参数：要打印的串，要打印的长度，是否转换为大小写的标记。但是在调用的时候，我们可以按顺序传递给 `adPrint` 一个参数，两个参数，或者三个参数(甚至可以传递给它多于 3 个，没有关系)，运行结果如下：

```
Hello, world
Hello
hello
HELLO
```

事实上，JavaScript 在处理函数的参数时，与其他编译型的语言不一样，解释器传递给函数的是一个类似于数组的内部值，叫 `arguments`，这个在函数对象生成的时候就被初始化了。比如我们传递给 `adPrint` 一个参数的情况下，其他两个参数分别为 `undefined`。这样，我们可以才 `adPrint` 函数内部处理那些 `undefined` 参数，从而可以向外部公开：我们可以处理任意参数。

我们通过另一个例子来讨论这个神奇的 `arguments`：

```
function sum() {
```

```

var result = 0;
for(var i = 0, len = arguments.length; i < len; i++){
    var current = arguments[i];
    if(isNaN(current)){
        throw new Error("not a number exception");
    }else{
        result += current;
    }
}

return result;
}

print(sum(10, 20, 30, 40, 50));
print(sum(4, 8, 15, 16, 23, 42)); // 《迷失》上那串神奇的数字
print(sum("new"));

```

函数 `sum` 没有显式的形参, 而我们又可以动态的传递给它任意多的参数, 那么, 如何在 `sum` 函数中如何引用这些参数呢? 这里就需要用到 `arguments` 这个伪数组了, 运行结果如下:

```

150
108
Error: not a number exception

```

4.2 函数作用域

4.2.1 词法作用域

作用域的概念在几乎所有的主流语言中都有体现, 在 `JavaScript` 中, 则有其特殊性: `JavaScript` 中的变量作用域为函数体内有效, 而无块作用域, 我们在 `Java` 语言中, 可以这样定义 `for` 循环块中的下标变量:

```

public void method(){
    for(int i = 0; i < obj1.length; i++){
        //do something here;
    }
    //此时的i为未定义
    for(int i = 0; i < obj2.length; i++){
        //do something else;
    }
}

```

```
}
```

而在 JavaScript 中:

```
function func() {  
    for(var i = 0; i < array.length; i++){  
        //do something here.  
    }  
    //此时i仍然有值, 及i == array.length  
    print(i); //i == array.length;  
}
```

JavaScript 的函数是在局部作用域内运行的, 在局部作用域内运行的函数体可以访问其外层的(可能是全局作用域)的变量和函数。JavaScript 的作用域为**词法作用域**, 所谓词法作用域是说, 其作用域为在定义时(词法分析时)就确定下来的, 而并非在执行时确定, 如下例:

```
var str = "global";  
function scopeTest() {  
    print(str);  
    var str = "local";  
    print(str);  
}
```

```
scopeTest();
```

运行结果是什么呢? 初学者很可能得出这样的答案:

```
global  
local
```

而正确的结果应该是:

```
undefined  
local
```

因为在函数 `scopeTest` 的定义中, 预先访问了未声明的变量 `str`, 然后才对 `str` 变量进行初始化, 所以第一个 `print(str)` 会返回 `undefined` 错误。那为什么函数这个时候不去访问外部的 `str` 变量呢? 这是因为, 在词法分析结束后, 构造作用域链的时候, 会将函数内定义的 `var` 变量放入该链, 因此 `str` 在整个函数 `scopeTest` 内都是可见的(从函数体的第一行到最后一行), 由于 `str` 变量本身是未定义的, 程序顺序执行, 到第一行就会返回未定义, 第二行为 `str` 赋值, 所以第三行的 `print(str)` 将返回“local”。

4.2.2 调用对象

我们再来深入的分析一下作用域，在 JavaScript 中，在所有函数之外声明的变量为全局变量，而在函数内部声明的变量(通过 `var` 关键字)为局部变量。事实上，全局变量是全局对象的属性而已，比如在客户端的 JavaScript 中，我们声明的变量其实是 `window` 对象的属性，如此而已。

那么，局部变量又隶属于什么对象呢？就是我们要讨论的**调用对象**。在执行一个函数时，函数的参数和其局部变量会作为调用对象的属性进行存储。同时，解释器会为函数创建一个执行器上下文(`context`)，与上下文对应起来的是一个作用域链。顾名思义，作用域链是关于作用域的链，通常实现为一个链表，链表的每个项都是一个对象，在全局作用域中，该链中有且只有一个对象，即全局对象。对应的，在一个函数中，作用域链上会有两个对象，第一个(首先被访问到的)为调用对象，第二个为全局对象。

如果函数需要用到某个变量，则解释器会遍历作用域链，比如在上一小节的例子中：

```
var str = "global";
function scopeTest() {
  print(str);
  var str = "local";
  print(str);
}
```

当解释器进入 `scopeTest` 函数的时候，一个调用对象就被创建了，其中包含了 `str` 变量作为其中的一个属性并被初始化为 `undefined`，当执行到第一个 `print(str)` 时，解释器会在作用域链中查找 `str`，找到之后，打印其值为 `undefined`，然后执行赋值语句，此时调用对象的属性 `str` 会被赋值为 `local`，因此第二个 `print(str)` 语句会打印 `local`。

应该注意的是，作用域链随着嵌套函数的层次会变的很长，但是查找变量的过程依旧是遍历作用域链(链表)，一直向上查找，直到找出该值，如果遍历完作用域链仍然没有找到对应的属性，则返回 `undefined`。关于调用对象(又称活动对象)的详情在本书的高级主题部分有更详细的讨论。

4.3 函数上下文

在 Java 或者 C/C++ 等语言中，方法(函数)只能依附于对象而存在，不是独立的。而在 JavaScript 中，函数也是一种对象，并非其他任何对象的一部分，理解这一点尤为重要，特别是对理解函数式的 JavaScript 非常有用，在函数式编程语言中，函数被认为是一等的。

函数的上下文是可以变化的，因此，函数内的 `this` 也是可以变化的，函数可以作为一个对象的方法，也可以同时作为另一个对象的方法，总之，函数本身是独立的。可以通过 `Function` 对象上的 `call` 或者 `apply` 函数来修改函数的上下文：

4.4 call 和 apply

call 和 apply 通常用来修改函数的上下文，函数中的 this 指针将被替换为 call 或者 apply 的第一个参数，我们不妨来看看 2.1.3 小节的例子：

```
//定义一个人，名字为jack
var jack = {
  name : "jack",
  age : 26
}

//定义另一个人，名字为abruzzo
var abruzzo = {
  name : "abruzzo",
  age : 26
}

//定义一个全局的函数对象
function printName() {
  return this.name;
}

//设置printName的上下文为jack，此时的this为jack
print(printName.call(jack));
//设置printName的上下文为abruzzo，此时的this为abruzzo
print(printName.call(abruzzo));

print(printName.apply(jack));
print(printName.apply(abruzzo));
```

只有一个参数的时候 call 和 apply 的使用方式是一样的，如果有多个参数：

```
setName.apply(jack, ["Jack Sept."]);
print(printName.apply(jack));

setName.call(abruzzo, "John Abruzzo");
print(printName.call(abruzzo));
```

得到的结果为：

```
Jack Sept.
John Abruzzo
```

`apply` 的第二个参数为一个函数需要的参数组成的一个数组，而 `call` 则需要跟若干个参数，参数之间以逗号(,)隔开即可。

4.5 使用函数

前面已经提到，在 JavaScript 中，函数可以

- 被赋值给一个变量
- 被赋值为对象的属性
- 作为参数被传入别的函数
- 作为函数的结果被返回

我们就分别来看看这些场景：

4.5.1 赋值给一个变量

```
//声明一个函数，接受两个参数，返回其和
function add(x, y) {
    return x + y;
}

var a = 0;
a = add; //将函数赋值给一个变量
var b = a(2, 3); //调用这个新的函数a
print(b);
```

这段代码会打印“5”，因为赋值之后，变量 `a` 引用函数 `add`，也就是说，`a` 的值是一个函数对象(一个可执行代码块)，因此可以使用 `a(2, 3)` 这样的语句来进行求和操作。

4.5.2 赋值为对象的属性

```
var obj = {
    id: "obj1"
}

obj.func = add; //赋值为obj对象的属性
```

```
obj.func(2, 3); //返回5
```

事实上，这个例子与上个例子的本质上是一样的，第一个例子中的 `a` 变量，事实上是全局对象(如果在客户端环境中，表示为 `window` 对象)的一个属性。而第二个例子则为 `obj` 对象，由于我们很少直接的引用全局对象，就分开来描述。

4.5.3 作为参数传递

```
//高级打印函数的第二个版本
function adPrint2(str, handler){
    print(handler(str));
}

//将字符串转换为大写形式，并返回
function up(str){
    return str.toUpperCase();
}

//将字符串转换为小写形式，并返回
function low(str){
    return str.toLowerCase();
}

adPrint2("Hello, world", up);
adPrint2("Hello, world", low);
```

运行此片段，可以得到这样的结果：

```
HELLO, WORLD
hello, world
```

应该注意到，函数 `adPrint2` 的第二个参数，事实上是一个函数，将这个处理函数作为参数传入，在 `adPrint2` 的内部，仍然可以调用这个函数，这个特点在很多地方都是有用的，特别是，当我们想要处理一些对象，但是又不确定以何种形式来处理，则完全可以将“处理方式”作为一个抽象的粒度来进行包装(即函数)。

4.5.4 作为函数的返回值

先来看一个最简单的例子：

```
function currying() {  
  return function() {  
    print("currying");  
  }  
}
```

函数 `currying` 返回一个匿名函数，这个匿名函数会打印“currying”，简单的调用 `currying()` 会得到下面的结果：

```
function () {  
  print("currying");  
}
```

如果要调用 `currying` 返回的这个匿名函数，需要这样：

```
currying() ();
```

第一个括号操作，表示调用 `currying` 本身，此时返回值为函数，第二个括号操作符调用这个返回值，则会得到这样的结果：

currying

关于函数的应用会贯穿与本书的整个过程，本章之讲解了函数的基本概念和使用方式，其中涉及到的内容颇为繁杂，可以在读完后续的章节之后再来回顾。

第五章 数组

JavaScript 的数组也是一个比较有意思的主题，虽然名为数组(Array)，但是根据数组对象上的方法来看，更像是将很多东西混在在一起的结果。而传统的程序设计语言如 C/Java 中，数组内的元素需要具有相同的数据类型，而作为弱类型的 JavaScript，则没有这个限制，事实上，JavaScript 的同一个数组中，可以有各种完全不同类型的元素。

方法	描述
concat()	连接两个或更多的数组，并返回结果。
join()	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。
pop()	删除并返回数组的最后一个元素。
push()	向数组的末尾添加一个或更多元素，并返回新的长度。
reverse()	颠倒数组中元素的顺序。
shift()	删除并返回数组的第一个元素。
slice()	从某个已有的数组返回选定的元素。
sort()	对数组的元素进行排序。
splice()	删除元素，并向数组添加新元素。
unshift()	向数组的开头添加一个或更多元素，并返回新的长度。
valueOf()	返回数组对象的原始值。

可以看出，JavaScript 的数组对象比较复杂，包含有 pop,push 等类似与栈的操作，又有 slice, reverse, sort 这样类似与列表的操作。或许正因为如此，JavaScript 中的数组的功能非常强大。

5.1 数组的特性

数组包括一些属性和方法，其最常用的属性则为 length，length 表示数组的当前长度，与其他语言不同的是，这个变量并非只读属性，比如：

```
var array = new Array(1, 2, 3, 4, 5);
print(array.length);
array.length = 3;
print(array.length);
print(array);
```

运行结果为：

```
5
3
1,2,3
```

注意到最后的 `print` 语句的结果是“1,2,3”，原因是 `length` 属性的修改会使得数组后边的元素变得不可用(如果修改后的 `length` 比数组实际的长度小的话)，所以可以通过设置 `length` 属性来将数组元素裁减。

另一个与其他语言的数组不同的是，字符串也可以作为数组的下标，事实上，在 JavaScript 的数组中，字符串型下标和数字型的下标会被作为两个截然不同的方式来处理，一方面，如果是数字作为下标，则与其他程序设计语言中的数组一样，可以通过 `index` 来进行访问，而使用字符串作为下标，就会采用访问 JavaScript 对象的属性的方式进行，毕竟 JavaScript 内置的 `Array` 也是从 `Object` 上继承下来的。比如：

```
var stack = new Array();

stack['first'] = 3.1415926;
stack['second'] = "okay then.";
stack['third'] = new Date();

for(var item in stack){
    print(typeof stack[item]);
}
```

运行结果为：

```
number
string
object
```

在这个例子里，还可以看到不同类型的数据是如何存储在同一个数组中的，这么做有一定的好处，但是在某些场合则可能形成不便，比如我们在函数一章中讨论过的 `sum` 函数，`sum` 接受非显式的参数列表，使用这个函数，需要调用者必须为 `sum` 提供数字型的列表(当然，字符串无法做 `sum` 操作)。如果是强类型语言，则对 `sum` 传入字符串数组会被编译程序认为是非法的，而在 JavaScript 中，程序需要在运行时才能侦测到这一错误。

5.2 使用数组

5.2.1 数组的基本方法使用

数组有这样几种方式来创建：

```
var array = new Array();
var array = new Array(10); //长度
```

```
var array = new Array("apple", "borland", "cisco");
```

不过，运用最多的为字面量方式来创建，如果第三章中的JSON那样，我们完全可以这样创建数组：

```
var array = [];  
var array = ["one", "two", "three", "four"];
```

下面我们通过一些实际的小例子来说明数组的使用(主要方法的使用)：
向数组中添加元素：

```
var array = [];  
  
array.push(1);  
array.push(2);  
array.push(3);  
  
array.push("four");  
array.push("five");  
  
array.push(3.1415926);
```

前面提到过，JavaScript的数组有列表的性质，因此可以向其中push不同类型的元素，接上例：

```
var len = array.length;  
for(var i = 0; i < len; i++){  
    print(typeof array[i]);  
}
```

结果为：

```
number  
number  
number  
string  
string  
number
```

弹出数组中的元素：

```
for(var i = 0; i < len; i++){  
    print(array.pop());  
}
```

```
print(array.length);
```

运行结果如下，注意最后一个0是指array的长度为0，因为这时数组的内容已经全部弹出：

```
3.1415926
five
four
3
2
1
0
```

join，连接数组元素为一个字符串：

```
array = ["one", "two", "three", "four", "five"];

var str1 = array.join(",");
var str2 = array.join("|");

print(str1);
print(str2);
```

运行结果如下：

```
one,two,three,four,five
one|two|three|four|five
```

连接多个数组为一个数组：

```
var another = ["this", "is", "another", "array"];
var another2 = ["yet", "another", "array"];

var bigArray = array.concat(another, another2);
```

结果为：

```
one,two,three,four,five,this,is,another,array,yet,another,array
```

从数组中取出一定数量的元素，不影响数组本身：

```
print(bigArray.slice(5,9));
```

结果为：

```
this, is, another, array
```

`slice`方法的第一个参数为起始位置，第二个参数为终止位置，操作不影响数组本身。下面我们来看`splice`方法，虽然这两个方法的拼写非常相似，但是功用则完全不同，事实上，`splice`是一个相当难用的方法：

```
bigArray.splice(5, 2);
```

```
bigArray.splice(5, 0, "very", "new", "item", "here");
```

第一行代码表示，从`bigArray`数组中，从第5个元素起，删除2个元素；而第二行代码表示，从第5个元素起，删除0个元素，并把随后的所有参数插入到从第5个开始的位置，则操作结果为：

```
one, two, three, four, five, very, new, item, here, another, array, yet, another, array
```

我们再来讨论下数组的排序，JavaScript的数组的排序函数`sort`将数组按字母顺序排序，排序过程会影响源数组，比如：

```
var array = ["Cisio", "Borland", "Apple", "Dell"];
print(array);
array.sort();
print(array);
```

执行结果为：

```
Cisio, Borland, Apple, Dell
Apple, Borland, Cisio, Dell
```

这种字母序的排序方式会造成一些非你所预期的小bug，比如：

```
var array = [10, 23, 44, 58, 106, 235];
array.sort();
print(array);
```

得到的结果为：

```
10, 106, 23, 235, 44, 58
```

可以看到，`sort`不关注数组中的内容是数字还是字母，它仅仅是按照字母的字典序来进行排序，对于这种情况，JavaScript提供了另一种途径，通过给`sort`函数传递一个函数对象，按照这个函数提供的规则对数组进行排序。

```
function sorter(a, b){
    return a - b;
}

var array = [10, 23, 44, 58, 106, 235];
array.sort(sorter);
print(array);
```

函数 `sorter` 接受两个参数，返回一个数值，如果这个值大于 0，则说明第一个参数大于第二个参数，如果返回值为 0，说明两个参数相等，返回值小于 0，则第一个参数小于第二个参数，`sort` 根据这个返回值来进行最终的排序：

10,23,44,58,106,235

当然，也可以简写成这样：

```
array.sort(function(a, b){return a - b;}); //正序
array.sort(function(a, b){return b - a;}); //逆序
```

5.2.2 删除数组元素

虽然令人费解，但是 JavaScript 的数组对象上确实没有一个叫做 `delete` 或者 `remove` 的方法，这就使得我们需要自己扩展其数组对象。一般来说，我们可以扩展 JavaScript 解释器环境中内置的对象，这种方式的好处在于，扩展之后的对象可以适用于其后的任意场景，而不用每次都显式的声明。而这种做法的坏处在于，修改了内置对象，则可能产生一些难以预料的错误，比如遍历数组实例的时候，可能会产生令人费解的异常。

数组中的每个元素都是一个对象，那么，我们可以使用 `delete` 来删除元素吗？来看看下边这个小例子：

```
var array = ["one", "two", "three", "four"];
//数组中现在的内容为：
//one,two,three,four
//array.length == 4
delete array[2];
```

然后，我们再来看看这个数组的内容：

```
one, two, undefined, four
//array.length == 4
```

可以看到，`delete` 只是将数组 `array` 的第三个位置上的元素删掉了，可是数组的长度没有改变，显然这个不是我们想要的结果，不过我们可以借助数组对象自身的 `slice` 方法来做到。一个比较好的实现，是来自于 jQuery 的设计者 John Resig：

```
//Array Remove - By John Resig (MIT Licensed)
Array.prototype.remove = function(from, to) {
    var rest = this.slice((to || from) + 1 || this.length);
    this.length = from < 0 ? this.length + from : from;
    return this.push.apply(this, rest);
};
```

这个函数扩展了 JavaScript 的内置对象 `Array`，这样，我们以后的所有声明的数组都会自动的拥有 `remove` 能力，我们来看看这个方法的用法：

```
var array = ["one", "two", "three", "four", "five", "six"];
print(array);
array.remove(0); //删除第一个元素
print(array);
array.remove(-1); //删除倒数第一个元素
print(array);
array.remove(0, 2); //删除数组中下标为0-2的元素(3个)
print(array);
```

会得到这样的结果：

```
one, two, three, four, five, six
two, three, four, five, six
two, three, four, five
five
```

也就是说，`remove` 接受两个参数，第一个参数为起始下标，第二个参数为结束下标，其中第二个参数可以忽略，这种情况下会删除指定下标的元素。当然，不是每个人都希望影响整个原型链(原因在下一个小节里讨论)，因此可以考虑另一种方式：

```
//Array Remove - By John Resig (MIT Licensed)
Array.remove = function(array, from, to) {
    var rest = array.slice((to || from) + 1 || array.length);
    array.length = from < 0 ? array.length + from : from;
    return array.push.apply(array, rest);
};
```

其操作方式与前者并无二致，但是不影响全局对象，代价是你需要显式的传递需要操作的数组作为第一个参数：

```
var array = ["one", "two", "three", "four", "five", "six"];
Array.remove(array, 0, 2); //删除0, 1, 2三个元素
print(array);
```

这种方式，相当于给 JavaScript 内置的 `Array` 添加了一个静态方法。

5.2.3 遍历数组

在对象与 JSON 这一章中，我们讨论了 `for...in` 这种遍历对象的方式，这种方式同样适用于数组，比如：

```
var array = [1, 2, 3, 4];
for(var item in array){
    print(array[item]);
}
```

将会打印：

```
1
2
3
4
```

但是这种方式并不总是有效，比如我们扩展了内置对象 `Array`，如下：

```
Array.prototype.useless = function() {}
```

然后重复执行上边的代码，会得到这样的输出：

```
1
2
3
4
function() {}
```

设想这样一种情况，如果你对数组的遍历做 `sum` 操作，那么会得到一个莫名其妙的错误，毕竟函数对象不能做求和操作。幸运的是，我们可以用另一种遍历方式来取得正确的结果：

```
for(var i = 0, len = array.length; i < len; i++){
    print(array[i]);
}
```

这种 `for` 循环如其他很多语言中的写法一致，重要的是，它不会访问哪些下标不是数字的元素，如上例中的 `function`，这个 `function` 的下标为 `useless`，是一个字符串。从这个例子我们可以看出，除非必要，尽量不要对全局对象进行扩展，因为对全局对象的扩展会造成所有继承链上都带上“烙印”，而有时候这些烙印会成为滋生 `bug` 的温床。

第六章 正则表达式

正则表达式是对字符串的结构进行的形式化描述，非常简洁优美，而且功能十分强大。很多的语言都不同程度的支持正则表达式，而在很多的文本编辑器如 Emacs, vim, UE 中，都支持正则表达式来进行字符串的搜索替换工作。UNIX 下的很多命令程序，如 awk, grep, find 更是对正则表达式有良好的支持。

JavaScript 同样也对正则表达式有很好的支持，RegExp 是 JavaScript 中的内置“类”，通过使用 RegExp，用户可以自己定义模式来对字符串进行匹配。而 JavaScript 中的 String 对象的 replace 方法也支持使用正则表达式对串进行匹配，一旦匹配，还可以通过调用预设的回调函数来进行替换。

正则表达式的用途十分广泛，比如在客户端的 JavaScript 环境中的用户输入验证，判断用户输入的身份证号码是否合法，邮件地址是否合法等。另外，正则表达式可用于查找替换工作，首先应该关注的是正则表达式的基本概念。

关于正则表达式的完整内容完全是另外一个主题了，事实上，已经有很多本专著来解释这个主题，限于篇幅，我们在这里只关注 JavaScript 中的正则表达式对象。

6.1 正则表达式基础概念

本节讨论正则表达式中的基本概念，这些基本概念在很多的正则表达式实现中是一致的，当然，细节方面可能会有所不同，毕竟正则表达式是来源于数学定义的，而不是程序员。JavaScript 的正则表达式对象实现了 perl 正则表达式规范的一个子集，如果你对 perl 比较熟悉的话，可以跳过这个小节。脚本语言 perl 的正则表达式规范是目前广泛采用的一个规范，Java 中的 regex 包就是一个很好的例子，另外，如 vim 这样的应用程序中，也采用了该规范。

6.1.1 元字符与特殊字符

元字符，是一些数学符号，在正则表达式中有特定的含义，而不仅仅表示其“字面”上的含义，比如星号(*)，表示一个集合的零到多次重复，而问号(?)表示零次或一次。如果你需要使用元字符的字面意义，则需要转义。

下面是一张元字符的表：

元字符	含义
^	串的开始
\$	串的结束
*	零到多次匹配
+	一到多次匹配
?	零或一次匹配
\b	单词边界

特殊字符，主要是指注入空格，制表符，其他进制(十进制之外的编码方式)等，它们的特点是以转义字符(\)为前导。如果需要引用这些特殊字符的字面意义，同样需要转义。

下面为转移字符的一张表：

字符	含义
字符本身	匹配字符本身
\r	匹配回车
\n	匹配换行
\t	制表符
\f	换页
\x#	匹配十六进制数
\cX	匹配控制字符

6.1.2 范围及重复

我们经常会遇到要描述一个范围的例子，比如，从 0 到 3 的数字，所有的英文字母，包含数字，英文字母以及下划线等等，正则表达式规定了如何表示范围：

标志符	含义
[...]	在集合中的任一个字符
[^...]	不在集合中的任一个字符
.	出\n之外的任一个字符
\w	所有的单字，包括字母，数字及下划线
\W	不包括所有的单字，\w 的补集
\s	所有的空白字符，包括空格，制表符
\S	所有的非空白字符
\d	所有的数字
\D	所有的非数字
\b	退格字符

结合元字符和范围，我们可以定义出很强大的模式来，比如，一个简化版的匹配 Email 的正则表达是为：

```
var emailval = /^[\\w-]+(\\. [\\w-]+)*@[\\w-]+(\\. [\\w-]+)+$/;
```

```
emailval.test("kmustlinux@hotmail.com");//true
emailval.test("john.abruzzo@pl.kunming.china");//true
emailval.test("@invalid.com");//false, 不合法
```

[\\w-]表示所有的字符，数字，下划线及减号，[\\w-]+表示这个集合最少重复一次，然后紧接着的这个括号表示一个分组(分组的概念参看下一节)，这个分组的修饰符为星号

(*)，表示重复零或多次。这样就可以匹配任意字母，数字，下划线及中划线的集合，且至少重复一次。

而@符号之后的部分与前半部分唯一不同的是，后边的一个分组的修饰符为(+)，表示至少重复一次，那就意味着后半部分至少会有一个点号(.)，而且点号之后至少有一个字符。这个修饰主要是用来限制输入串中必须包含域名。

最后，脱字符(^)和美元符号(\$)限制，以……开始，且以……结束。这样，整个表达式的意义就很明显了。

再来看一个例子：在 C/Java 中，变量命名的规则为：以字母或下划线开头，变量中可以包含数字，字母以及下划线(有可能还会规定长度，我们在下一节讨论)。这个规则描述成正则表达式即为下列的定义：

```
var variable = /^[a-zA-Z_][a-zA-Z0-9_]*;/

print(variable.test("hello"));
print(variable.test("world"));
print(variable.test("_main_"));
print(variable.test("0871"));
```

将会打印：

```
true
true
true
false
```

前三个测试字符均为合法，而最后一个是数字开头，因此为非法。应该注意的是，test 方法只是测试目标串中是否有表达式匹配的**部分**，而不一定整个串都匹配。比如上例中：

```
print(variable.test("0871_hello_world")); //true
print(variable.test("@main")); //true
```

同样返回 true，这是因为，test 在查找整个串时，发现了完整匹配 variable 表达式的部分内容，同样也是匹配。为了避免这种情况，我们需要给 variable 做一些修改：

```
var variable = /^[a-zA-Z_][a-zA-Z0-9_]*$/;
```

通过**加推导(+)**，**星推导(*)**，以及谓词，我们可以灵活的对范围进行重复，但是我们仍然需要一种机制来提供诸如 4 位数字，最多 10 个字符等这样的精确的重复方式。这就需要用到下表中的标记：

标记	含义
{n}	重复 n 次
{n,}	重复 n 或更多次
{n,m}	重复至少 n 次，至多 m 次

有了精确的重复方式，我们就可以来表达如身份证号码，电话号码这样的表达式，而不用担心出做，比如：

```
var pid = /^(\d{15}|\d{18})$/; //身份证
var mphone = /\d{11}/; //手机号码
var phone = /\d{3,4}-\d{7,8}/; //电话号码

mphone.test("13893939392"); //true
phone.test("010-99392333"); //true
phone.test("0771-3993923"); //true
```

6.1.3 分组与引用

在正则表达式中，括号是一个比较特殊的操作符，它可以有三中作用，这三种都是比较常见的：

第一种情况，括号用来将子表达式标记起来，以区别于其他表达式，比如很多的命令行程序都提供帮助命令，键入 **h** 和键入 **help** 的意义是一样的，那么就会有这样的表达式：

```
h(elp)? //字符h之后的elp可有可无
```

这里的括号仅仅为了将 **elp** 自表达式与整个表达是隔离(因为 **h** 是必选的)。

第二种情况，括号用来分组，当正则表达式执行完成之后，与之匹配的文本将会按照规则填入各个分组，比如，某个数据库的主键是这样的格式：四个字符表示省份，然后是四个数字表示区号，然后是两位字符表示区县，如 **yunn0871cg** 表示云南省昆明市呈贡县(当然，看起来的确很怪，只是举个例子)，我们关心的是区号和区县的两位字符代码，怎么分离出来呢？

```
var pattern = /\w{4}(\d{4})(\w{2})/;
var result = pattern.exec("yunn0871cg");
print("city code = "+result[1]+", county code = "+result[2]);
result = pattern.exec("shax0917cc");
print("city code = "+result[1]+", county code = "+result[2]);
```

正则表达式的 **exec** 方法会返回一个数组(如果匹配成功的话)，数组的第一个元素(下标为 0)表示整个串，第一个元素为第一个分组，第二个元素为第二个分组，以此类推。因此上例的执行结果即为：

```
city code = 0871, county code = cg
city code = 0917, county code = cc
```

第三种情况，括号用来对引用起辅助作用，即在同一个表达式中，后边的式子可以引用

前边匹配的文本，我们来看一个非常常见的例子：我们在设计一个新的语言，这个语言中有字符串类型的数据，与其他的程序设计语言并无二致，比如：

```
var str = "hello, world";
var str = 'fair enough';
```

均为合法字符，我们可能会设计出这样的表达式来匹配该声明：

```
var pattern = /["']["']*["']/;
```

看来没有什么问题，但是如果用户输入：

```
var str = 'hello, world";
var str = "hello, world';
```

我们的正则表达式还是可以匹配，注意这两个字符串两侧的引号不匹配！我们需要的是，前边是单引号，则后边同样是单引号，反之亦然。因此，我们需要知道前边匹配的到底是“单”还是“双”。这里就需要用到引用，JavaScript 中的引用使用斜杠加数字来表示，如 `\1` 表示第一个分组(括号中的规则匹配的文本)，`\2` 表示第二个分组，以此类推。因此我们就设计出了这样的表达式：

```
var pattern = /(["'])["']*\\1/;
```

在我们新设计的这个语言中，为了某种原因，在单引号中我们不允许出现双引号，同样，在双引号中也不允许出现单引号，我们可以稍作修改即可完成：

```
var pattern = /(["'])["']*\\1/;
```

这样，我们的语言中对于字符串的处理就完善了。

6.2 使用正则表达式

创建一个正则表达式有两种方式，一种是借助 `RegExp` 对象来创建，另一种方式是使用正则表达式字面量来创建。在 JavaScript 内部的其他对象中，也有对正则表达式的支持，比如 `String` 对象的 `replace`，`match` 等。我们可以分别来看：

6.2.1 创建正则表达式

使用字面量：

```
var regex = /pattern/;
```

使用 `RegExp` 对象：

```
var regex = new RegExp("pattern", switches);
```

而正则表达式的一般形式描述为:

```
var regex = /pattern/[switchs];
```

这里的开关(switchs)有以下三种:

修饰符	描述
i	忽略大小写开关
g	全局搜索开关
m	多行搜索开关(重定义^与\$的意义)

比如, `/java/i` 就可以匹配 `java/Java/JAVA`, 而 `/java/` 则不可。而 `g` 开关用来匹配整个串中所有出现的子模式, 如 `/java/g` 匹配 `"javascript&java"` 中的两个 `"java"`。而 `m` 开关定义是否多行搜索, 比如:

```
var pattern = /^javascript/;
print(pattern.test("java\njavascript")); //false
pattern = /^javascript/m;
print(pattern.test("java\njavascript")); //true
```

RegExp 对象的方法:

方法名	描述
test()	测试串中是否有合乎模式的匹配
exec()	对串进行匹配
compile()	编译正则表达式

RegExp 对象的 `test` 方法用于检测字符串中是否具有匹配的模式, 而不关心匹配的结果, 通常用于测试, 如上边提到的例子:

```
var variable = /[a-zA-Z][a-zA-Z0-9_]*/;

print(variable.test("hello")); //true
print(variable.test("world")); //true
print(variable.test("_main_")); //true
print(variable.test("0871")); //false
```

而 `exec` 则通过匹配, 返回需要分组的信息, 在分组及引用小节中我们已经做过讨论, 而 `compile` 方法用来改变表达式的模式, 这个过程与重新声明一个正则表达式对象的作用相同, 在此不作深入讨论。

6.2.2 String 中的正则表达式

除了正则表达式对象及字面量外，**String** 对象中也有多个方法支持正则表达式操作，我们通过例子讨论这些方法：

方法	作用
match	匹配正则表达式，返回匹配数组
replace	替换
split	分割
search	查找，返回首次发现的位置

```
var str = "life is very much like a mirror.";
var result = str.match(/is|a/g);
print(result); //返回["is", "a"]
```

这个例子通过 **String** 的 **match** 来匹配 **str** 对象，得到返回值为["is", "a"]的一个数组。

```
var str = "<span>Welcome, John</span>";
var result = str.replace(/span/g, "div");
print(str);
print(result);
```

得到结果：

```
<span>Welcome, John</span>
<div>Welcome, John</div>
```

也就是说，**replace** 方法不会影响原始字符串，而将新的串作为返回值。如果我们在替换过程中，需要对匹配的组进行引用(正如之前的\1,\2 方式那样)，需要怎么做呢？还是上边这个例子，我们要在替换的过程中，将 **Welcome** 和 **John** 两个单词调换顺序，编程 **John, Welcome**：

```
var result = str.replace(/(\w+),\s(\w+)/g, "$2, $1");
print(result);
```

可以得到这样的结果：

```
<span>John, Welcome</span>
```

因此，我们可以通过 **\$n** 来对第 **n** 个分组进行引用。

```
var str = "john : tomorrow :remove:file";
var result = str.split(/\s*:\s*/);
```

```
print(str);
print(result);
```

得到结果:

```
john : tomorrow      :remove:file
john,tomorrow,remove,file
```

注意此处 `split` 方法的返回值 `result` 是一个数组。其中包含了 4 个元素。

```
var str = "Tomorrow is another day";
var index = str.search(/another/);
print(index); //12
```

`search` 方法会返回查找到的文本在模式中的位置，如果查找不到，返回 -1。

6.3 实例: JSFilter

本小节提供一个实例，用以展示在实际应用中正则表达的用途，当然，一个例子不可能涵盖所有内容，只是一个最常见的场景。

考虑这样一种情况，我们在 UI 上为用户提供一种快速搜索的能力，使得随着用户的键入，结果集不断的减少，直到用户找到自己需要的关键字对应的栏目。在这个过程中，用户可以选择是否区分大小写，是否全词匹配，以及高亮一个记录中的所有匹配。

显然，正则表达式可以满足这个需求，我们在这个例子中忽略掉诸如高亮，刷新结果集等部分，来看看正则表达式在实际中的应用：

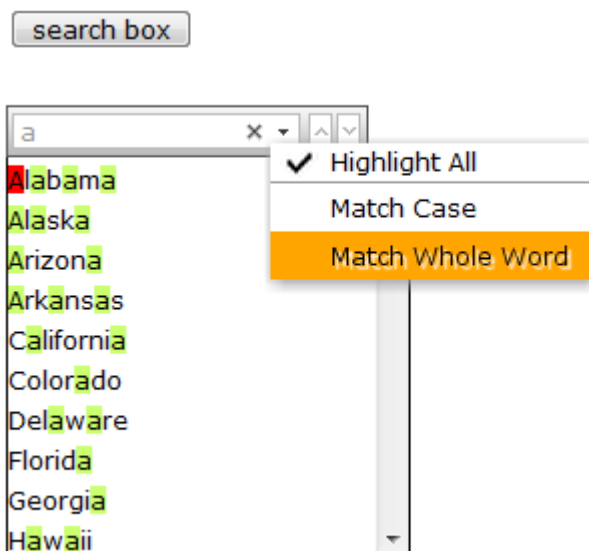


图 在列表中使用 JSFilter(结果集随用户输入而变化)

来看一个代码片段:


```

this.content.each(function() {
    var text = $(this).text();
    var pattern = new RegExp(keyword, reopts);
    if(pattern.test(text)) {
        var item = text.replace(pattern, function(t) {
            return "<span
class=\""+filterOptions.highlight+"\">"+t+"</span>";
        });
        $(this).html(item).show();
    }else{//clear previous search result
        $(this).find("span."+filterOptions.highlight).each(function() {
            $(this).replaceWith($(this).text());
        });
    }
});
});

```

其中，`content` 是结果集，是一个集合，其中的每一个项目都可能包含用户输入的关键词，`keyword` 是用户输入的关键词序列，而 `reopts` 为正则表达式的选项，可能为(i,g,m)，`each` 是 jQuery 中的遍历集合的方式，非常方便。程序的流程是这样的：

- 进入循环，取得结果集中的一个值作为当前值
- 使用正则表达式对象的 `test` 方法进行测试
- 如果测试通过，则高亮标注记录中的关键词
- 否则跳过，进行下一条的检测

遍历完所有的结果集，生成了一个新的，高亮标注的结果集，然后将其呈现给用户。而且可以很好的适应用户的需求，比如是否忽略大小写检查，是否高亮所有，是否全词匹配，如果自行编写程序进行分析，则需要耗费极大的时间和精力。

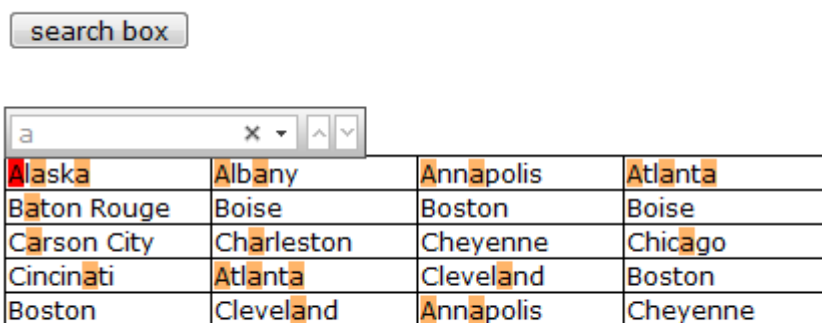


图 在表格中使用 JSFilter(不减少结果集)

这个例子来源于一个实际的项目，我对其进行了适度的简化，完整的代码可以参考附件。

第七章 闭包

闭包向来给包括 JavaScript 程序员在内的程序员以神秘，高深的感觉，事实上，闭包的概念在函数式编程语言中算不上是难以理解的知识。如果对作用域，函数为独立的对象这样的基本概念理解较好的话，理解闭包的概念并在实际的编程实践中应用则颇有水到渠成之感。

在 DOM 的事件处理方面，大多数程序员甚至自己已经在使用闭包了而不自知，在这种情况下，对于浏览器中内嵌的 JavaScript 引擎的 bug 可能造成内存泄漏这一问题姑且不论，就是程序员自己调试也常常会一头雾水。

用简单的语句来描述 JavaScript 中的闭包的概念：由于 JavaScript 中，函数是对象，对象是属性的集合，而属性的值又可以是对象，则在函数内定义函数成为理所当然，如果在函数 `func` 内部声明函数 `inner`，然后在函数外部调用 `inner`，这个过程即产生了一个闭包。

7.1 闭包的特性

我们先来看一个例子，如果不了解 JavaScript 的特性，很难找到原因：

```
var outter = [];  
function clouseTest () {  
    var array = ["one", "two", "three", "four"];  
    for(var i = 0; i < array.length;i++){  
        var x = {};  
        x.no = i;  
        x.text = array[i];  
        x.invoke = function () {  
            print(i);  
        }  
        outter.push(x);  
    }  
}  
  
//调用这个函数  
clouseTest();  
  
print(outter[0].invoke());  
print(outter[1].invoke());  
print(outter[2].invoke());  
print(outter[3].invoke());
```

运行的结果如何呢？很多初学者可能会得出这样的答案：

```
0
1
2
3
```

然而，运行这个程序，得到的结果为：

```
4
4
4
4
```

其实，在每次迭代的时候，这样的语句 `x.invoke = function(){print(i);}` 并没有被执行，只是构建了一个函数体为“`print(i);`”的函数对象，如此而已。而当 `i=4` 时，迭代停止，外部函数返回，当再去调用 `outter[0].invoke()` 时，`i` 的值依旧为 4，因此 `outter` 数组中的每一个元素的 `invoke` 都返回 `i` 的值：4。

如何解决这一问题呢？我们可以声明一个匿名函数，并立即执行它：

```
var outter = [];

function clouseTest2(){
    var array = ["one", "two", "three", "four"];
    for(var i = 0; i < array.length;i++){
        var x = {};
        x.no = i;
        x.text = array[i];
        x.invoke = function(no){
            return function(){
                print(no);
            }
        }(i);
        outter.push(x);
    }
}

clouseTest2();
```

这个例子中，我们为 `x.invoke` 赋值的时候，先运行一个可以返回一个函数的函数，然后立即执行之，这样，`x.invoke` 的每一次迭代器时相当与执行这样的语句：

```
//x == 0
x.invoke = function(){print(0);}
```

```
//x == 1
x.invoke = function() {print(1);}
//x == 2
x.invoke = function() {print(2);}
//x == 3
x.invoke = function() {print(3);}
```

这样就可以得到正确结果了。闭包允许你引用存在于外部函数中的变量。然而，它并不是使用该变量创建时的值，相反，它使用外部函数中该变量**最后**的值。

7.2 闭包的用途

现在，闭包的概念已经清晰了，我们来看看闭包的用途。事实上，通过使用闭包，我们可以做很多事情。比如模拟面向对象的代码风格；更优雅，更简洁的表达出代码；在某些方面提升代码的执行效率。

7.2.1 匿名自执行函数

上一节中的例子，事实上就是闭包的一种用途，根据前面讲到的内容可知，所有的变量，如果不加上 `var` 关键字，则默认会添加到全局对象的属性上去，这样的临时变量加入全局对象有很多坏处，比如：别的函数可能误用这些变量；造成全局对象过于庞大，影响访问速度(因为变量的取值是需要从原型链上遍历的)。除了每次使用变量都是用 `var` 关键字外，我们在实际情况下经常遇到这样一种情况，即有的函数只需要执行一次，其内部变量无需维护，比如 UI 的初始化，那么我们可以使用闭包：

```
var datamodel = {
  table : [],
  tree : {}
};

(function (dm) {
  for(var i = 0; i < dm.table.rows; i++){
    var row = dm.table.rows[i];
    for(var j = 0; j < row.cells; i++){
      drawCell(i, j);
    }
  }
})(datamodel);

//build dm.tree
```

我们创建了一个匿名的函数，并立即执行它，由于外部无法引用它内部的变量，因此在

执行完后很快就会被释放，最主要的是这种机制不会污染全局对象。

7.2.2 缓存

再来看一个例子，设想我们有一个处理过程很耗时的函数对象，每次调用都会花费很长时间，那么我们就需要将计算出来的值存储起来，当调用这个函数的时候，首先在缓存中查找，如果找不到，则进行计算，然后更新缓存并返回值，如果找到了，直接返回查找到的值即可。闭包正是可以做到这一点，因为它不会释放外部的引用，从而函数内部的值可以得以保留。

```
var CachedSearchBox = (function () {
    var cache = {},
        count = [];
    return {
        attachSearchBox : function (dsid) {
            if (dsid in cache) { // 如果结果在缓存中
                return cache[dsid]; // 直接返回缓存中的对象
            }
            var fsb = new uikit.webctrl.SearchBox (dsid); // 新建
            cache[dsid] = fsb; // 更新缓存
            if (count.length > 100) { // 保证缓存的大小 <= 100
                delete cache[count.shift ()];
            }
            return fsb;
        },

        clearSearchBox : function (dsid) {
            if (dsid in cache) {
                cache[dsid].clearSelection ();
            }
        }
    };
})();
```

```
CachedSearchBox.attachSearchBox ("input1");
```

这样，当我们第二次调用 `CachedSearchBox.attachSerachBox("input1")` 的时候，我们就可以从缓存中取道该对象，而不用再去创建一个新的 `searchbox` 对象。

7.2.3 实现封装

可以先来看一个关于封装的例子，在 `person` 之外的地方无法访问其内部的变量，而通过提供闭包的形式来访问：

```
var person = function() {
    //变量作用域为函数内部，外部无法访问
    var name = "default";

    return {
        getName : function() {
            return name;
        },
        setName : function(newName) {
            name = newName;
        }
    }
}();

print(person.name); //直接访问，结果为undefined
print(person.getName());
person.setName("abruzzo");
print(person.getName());
```

得到结果如下：

```
undefined
default
abruzzo
```

闭包的另一个重要用途是实现面向对象中的**对象**，传统的对象语言都提供类的模板机制，这样不同的对象(类的实例)拥有独立的成员及状态，互不干涉。虽然 JavaScript 中没有类这样的机制，但是通过使用闭包，我们可以模拟出这样的机制。还是以上边的例子来讲：

```
function Person() {
    var name = "default";

    return {
        getName : function() {
            return name;
        },
        setName : function(newName) {
            name = newName;
        }
    }
};
```

```
var john = Person();
print(john.getName());
john.setName("john");
print(john.getName());

var jack = Person();
print(jack.getName());
jack.setName("jack");
print(jack.getName());
```

运行结果如下：

```
default
john
default
jack
```

由此代码可知, `john` 和 `jack` 都可以称为是 `Person` 这个类的实例, 因为这两个实例对 `name` 这个成员的访问是独立的, 互不影响的。

事实上, 在函数式的程序设计中, 会大量的用到闭包, 我们将在第八章讨论函数式编程, 在那里我们会再次探讨闭包的作用。

7.3 应该注意的问题

7.3.1 内存泄漏

在不同的 JavaScript 解释器实现中, 由于解释器本身的缺陷, 使用闭包可能造成内存泄漏, 内存泄漏是比较严重的问题, 会严重影响浏览器的响应速度, 降低用户体验, 甚至会造成浏览器无响应等现象。

JavaScript 的解释器都具备垃圾回收机制, 一般采用的是引用计数的形式, 如果一个对象的引用计数为零, 则垃圾回收机制会将其回收, 这个过程是自动的。但是, 有了闭包的概念之后, 这个过程就变得复杂起来了, 在闭包中, 因为局部的变量可能在将来的某些时刻需要被使用, 因此垃圾回收机制不会处理这些被外部引用到的局部变量, 而如果出现循环引用, 即对象 A 引用 B, B 引用 C, 而 C 又引用到 A, 这样的情况使得垃圾回收机制得出其引用计数不为零的结论, 从而造成内存泄漏。

7.3.2 上下文的引用

关于 `this` 我们之前已经做过讨论, 它表示对调用对象的引用, 而在闭包中, 最容易出现错误的地方是误用了 `this`。在前端 JavaScript 开发中, 一个常见的错误是错将 `this` 类比为其他的外部局部变量:

```
$(function() {
    var con = $("#div#panel");
    this.id = "content";
    con.click(function() {
        alert(this.id); //panel
    });
});
```

此处的 `alert(this.id)`到底引用着什么值呢？很多开发者可能会根据闭包的概念，做出错误的判断：

content

理由是，`this.id` 显示的被赋值为 `content`，而在 `click` 回调中，形成的闭包会引用到 `this.id`，因此返回值为 `content`。然而事实上，这个 `alert` 会弹出“panel”，究其原因，就是此处的 `this`，虽然闭包可以引用局部变量，但是涉及到 `this` 的时候，情况就有些微妙了，因为调用对象的存在，使得当闭包被调用时（当这个 `panel` 的 `click` 事件发生时），此处的 `this` 引用的是 `con` 这个 jQuery 对象。而匿名函数中的 `this.id = "content"` 是对匿名函数本身做的操作。两个 `this` 引用的并非同一个对象。

如果想要在事件处理函数中访问这个值，我们必须做一些改变：

```
$(function() {
    var con = $("#div#panel");
    this.id = "content";
    var self = this;
    con.click(function() {
        alert(self.id); //content
    });
});
```

这样，我们在事件处理函数中保存的是外部的一个局部变量 `self` 的引用，而并非 `this`。这种技巧在实际应用中多有应用，我们在后边的章节里进行详细讨论。关于闭包的更多内容，我们将在第九章详细讨论，包括讨论其他命令式语言中的“闭包”，闭包在实际项目中的应用等等。

第八章 面向对象的 Javascript

面向对象编程思想在提出之后，很快就流行起来了，它将开发人员从冗长，繁复，难以调试的过程式程序中解放了出来，过程式语言如 C，代码的形式往往如此：

```
Component comp;
init_component(&comp, props);
```

而面向对象的语言如 Java，则会是这样形式：

```
Component comp;
comp.init(props);
```

可以看出，方法是对象的方法，对象是方法的对象，这样的代码形式更接近人的思维方式，因此 OO 大行其道也并非侥幸。

JavaScript 本身是**基于对象**的，而并非基于类。但是，JavaScript 的函数式语言的特性使得它本身是**可编程**的，它可以变成你想要的任何形式。我们在这一章详细讨论如何使用 JavaScript 进行 OO 风格的代码开发。

8.1 原型继承

JavaScript 中的继承可以通过原型链来实现，调用对象上的一个方法，由于方法在 JavaScript 对象中是对另一个函数对象的引用，因此解释器会在对象中查找该属性，如果没有找到，则在其内部对象 **prototype** 对象上搜索，由于 **prototype** 对象与对象本身的结构是一样的，因此这个过程会一直回溯到发现该属性，则调用该属性，否则，报告一个错误。关于原型继承，我们不妨看一个小例子：

```
function Base() {
    this.baseFunc = function() {
        print("base behavior");
    }
}

function Middle() {
    this.middleFunc = function() {
        print("middle behavior");
    }
}

Middle.prototype = new Base();
```

```
function Final() {  
    this.finalFunc = function() {  
        print("final behavior");  
    }  
}  
Final.prototype = new Middle();
```

```
function test() {  
    var obj = new Final();  
    obj.baseFunc();  
    obj.middleFunc();  
    obj.finalFunc();  
}
```

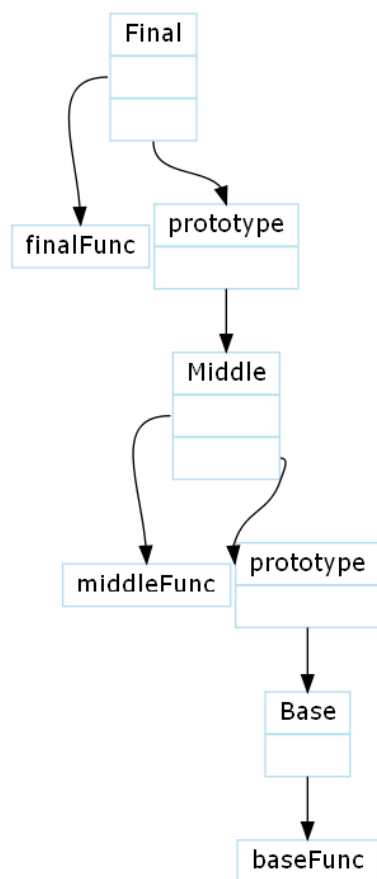


图 原型链的示意图

在 `function test` 中，我们 `new` 了一个 `Final` 对象，然后依次调用 `obj.baseFunc`，由于 `obj` 对象上并无此方法，则按照上边提到的规则，进行回溯，在其原型链上搜索，由于 `Final` 的原型链上包含 `Middle`，而 `Middle` 上又包含 `Base`，因此会执行这个方法，这样就实现了类的继承。

base behavior

middle behavior

final behavior

但是这种继承形式与传统的 OO 语言大相径庭，初学者很难适应，我们后边的章节会涉及到一个比较好的 JavaScript 的面向对象基础包 **Base**，使用 **Base** 包，虽然编码风格上会和传统的 OO 语言不同，但是读者很快就会发现这种风格的好处。

8.1.1 引用

引用是一个比较有意思的主题，JavaScript 中的引用始终指向最终的对象，而并非引用本身，我们来看一个例子：

```
var obj = {}; //空对象
var ref = obj; //引用

obj.name = "objectA";
print(ref.name); //ref跟着添加了name属性

obj = ["one", "two", "three"]; //obj指向了另一个对象(数组对象)
print(ref.name); //ref还指向原来的对象
print(obj.length); //3
print(ref.length); //undefined
```

运行结果如下：

```
objectA
objectA
3
undefined
```

`obj` 只是对一个匿名对象的引用，所以，`ref` 并非指向它，当 `obj` 指向另一个数组对象时可以看到，引用 `ref` 并未改变，而始终指向那个后来添加了 `name` 属性的“空”对象“`{}`”。理解这一点对后边的内容有很大的帮助。

再看这个例子：

```
var obj = {}; //新建一个对象，并被obj引用

var ref1 = obj; //ref1引用obj, 事实上是引用obj引用的空对象
var ref2 = obj;

obj.func = "function";
```

```
print(ref1.func);
print(ref2.func);
```

声明一个对象，然后用两个引用来引用这个对象，然后修改原始的对象，注意这两步的顺序，运行之：

```
function
function
```

根据运行结果我们可以看出，在定义了引用之后，修改原始的那个对象会影响到其引用上，这一点也应该注意。

8.1.2 new 操作符

有面向对象编程的基础有时会成为一种负担，比如看到 `new` 的时候，Java 程序员可能会认为这将会调用一个类的构造器构造一个新的对象出来，我们来看一个例子：

```
function Shape(type) {
  this.type = type || "rect";
  this.calc = function() {
    return "calc, "+this.type;
  }
}

var triangle = new Shape("triangle");
print(triangle.calc());

var circle = new Shape("circle");
print(circle.calc());
```

运行结果如下：

```
calc, triangle
calc, circle
```

Java 程序员可能会觉得 `Shape` 就是一个类，然后 `triangle`, `circle` 即是 `Shape` 对应的具体对象，而其实 JavaScript 并非如此工作的，罪魁祸首即为此 `new` 操作符。在 JavaScript 中，通过 `new` 操作符来作用与一个函数，实质上会发生这样的动作：

首先，创建一个空对象，然后用函数的 `apply` 方法，将这个空对象传入作为 `apply` 的第一个参数，及上下文参数。这样函数内部的 `this` 将会被这个空的对象所替代：

```
var triangle = new Shape("triangle");
//上一句相当于下面的代码
```

```
var triangle = {};  
Shape.apply(triangle, ["triangle"]);
```

8.2 封装

事实上，我们可以通过 JavaScript 的函数实现封装，封装的好处在于未经授权的客户代码无法访问到我们不公开的数据，我们来看这个例子：

```
function Person(name) {  
  //private variable  
  var address = "The Earth";  
  
  //public method  
  this.getAddress = function() {  
    return address;  
  }  
  
  //public variable  
  this.name = name;  
}  
  
//public  
Person.prototype.getName = function() {  
  return this.name;  
}  
  
//public  
Person.prototype.setName = function(name) {  
  this.name = name;  
}
```

首先声明一个函数，作为模板，用面向对象的术语来讲，就是一个类。用 `var` 方式声明的变量仅在类内部可见，所以 `address` 为一个私有成员，访问 `address` 的唯一方法是通过我们向外暴露的 `getAddress` 方法，而 `get/setName`，均为原型链上的方法，因此为公开的。我们可以做个测试：

```
var jack = new Person("jack");  
print(jack.name); //jack  
print(jack.getName()); //jack  
print(jack.address); //undefined  
print(jack.getAddress()); //The Earth
```

直接通过 `jack.address` 来访问 `address` 变量会得到 `undefined`。我们只能通过 `jack.getAddress` 来访问。这样，`address` 这个成员就被封装起来了。

另外需要注意的一点是，我们可以为类添加静态成员，这个过程也很简单，只需要为函数对象添加一个属性即可。比如：

```
function Person(name) {
  //private variable
  var address = "The Earth";

  //public method
  this.getAddress = function () {
    return address;
  }

  //public variable
  this.name = name;
}

Person.TAG = "javascript-core";//静态变量

print(Person.TAG);
```

也就是说，我们在访问 `Person.TAG` 时，不需要实例化 `Person` 类。这与传统的面向对象语言如 `Java` 中的静态变量是一致的。

8.3 工具包 Base

`Base` 是由 `Dean Edwards` 开发的一个 `JavaScript` 的面向对象的基础包，`Base` 本身很小，只有 140 行，但是这个很小的包对面向对象编程风格有很好的支持，支持类的定义，封装，继承，子类调用父类的方法等，代码的质量也很高，而且很多项目都在使用 `Base` 作为底层的支持。尽管如此，`JavaScript` 的面向对象风格依然非常古怪，并不可以完全和传统的 `OO` 语言对等起来。

下面我们来看几个基于 `Base` 的例子，假设我们现在在开发一个任务系统，我们需要抽象出一个类来表示任务，对应的，每个任务都可能会有一个监听器，当任务执行之后，需要通知监听器。我们首先定义一个事件监听器的类，然后定义一个任务类：

```
var EventListener = Base.extend({
  constructor : function(sense) {
    this.sense = sense;
  },
  sense : null,
  handle : function () {
    print(this.sense+" occured");
  }
});
```

```

    }
  });

  var Task = Base.extend({
    constructor : function(name) {
      this.name = name;
    },
    name : null,
    listener : null,
    execute : function() {
      print(this.name);
      this.listener.handle();
    },
    setListener : function(listener) {
      this.listener = listener;
    }
  });

```

创建类的方式很简单，需要给 `Base.extend` 方法传入一个 JSON 对象，其中可以有成员和方法。方法访问自身的成员时需要加 `this` 关键字。而每一个类都会有一个 `constructor` 的方法，即构造方法。比如事件监听器类(`EventListener`)的构造器需要传入一个字符串，而任务类(`Task`)也需要传入任务的名字来进行构造。好了，既然我们已经有了任务类和事件监听器类，我们来实例化它们：

```

var printing = new Task("printing");
var printEventListener = new EventListener("printing");
printing.setListener(printEventListener);
printing.execute();

```

首先，创建一个新的 `Task`，做打印工作，然后新建一个事件监听器，并将它注册在新建的任务上，这样，当打印发生时，会通知监听器，监听器会做出相应的判断：

```

printing
printing occurred

```

既然有了基本的框架，我们就来使用这个框架，假设我们要从 HTTP 服务器上下载一个页面，于是我们设计了一个新的任务类型，叫做 `HttpRequester`：

```

var HttpRequester = Task.extend({
  constructor : function(name, host, port) {
    this.base(name);
    this.host = host;
  }
});

```

```

        this.port = port;
    },
    host : "127.0.0.1",
    port : 9527,
    execute : function(){
        print("[ "+this.name+" ] request send to "+this.host+" of port
"+this.port);
        this.listener.handle();
    }
});

```

HttpRequester 类继承了 Task，并且重载了 Task 类的 execute 方法，setListener 方法的内容与父类一致，因此不需要重载。

```

var requester = new HttpRequester("requester1", "127.0.0.1", 8752);
var listener = new EventListener("http_request");
requester.setListener(listener);
requester.execute();

```

我们新建一个 HttpRequester 任务，然后注册上事件监听器，并执行之：

```

[requester1] request send to 127.0.0.1 of port 8752
http_request occurred

```

应该注意到 HttpRequester 类的构造器中，有这样一个语句：

```

this.base(name);

```

表示执行父类的构造器，即将 name 赋值给父类的成员变量 name，这样在 HttpRequester 的实例中，我们就可以通过 this.name 来访问这个成员了。这套机制简直与在其他传统的 OO 语言并无二致。同时，HttpRequester 类的 execute 方法覆盖了父类的 execute 方法，用面向对象的术语来讲，叫做重写。

在很多应用中，有些对象不会每次都创建新的实例，而是使用一个固有的实例，比如提供数据源的服务，报表渲染引擎，事件分发器等，每次都实例化一个会有很大的开销，因此人们设计出了单例模式，整个应用的生命周期中，始终只有顶多一个实例存在。Base 同样可以模拟出这样的能力：

```

var ReportEngine = Base.extend({
    constructor : null,
    run : function(){
        //render the report
    }
});

```

很简单，只需要将构造函数的值赋为 null 即可。好了，关于 Base 的基本用法我们已经熟

悉了，来看看用 **Base** 还能做些什么：

8.4 实例：事件分发器

这一节，我们通过学习一个面向对象的实例来对 JavaScript 的面向对象进行更深入的理解，这个例子不能太复杂，涉及到的内容也不能仅仅为继承，多态等概念，如果那样，会失去阅读的乐趣，最好是在实例中穿插一些讲解，则可以得到最好的效果。

本节要分析的实例为一个事件分发器(Event Dispatcher)，本身来自于一个实际项目，但同时又比较小巧，我对其代码做了部分修改，去掉了一些业务相关的部分。

事件分发器通常是跟 UI 联系在一起的，UI 中有多个组件，它们之间经常需要互相通信，当 UI 比较复杂，而页面元素的组织又不够清晰的时候，事件的处理会非常麻烦。在本节的例子中，事件分发器为一个对象，UI 组件发出事件到事件分发器，也可以注册自己到分发器，当自己关心的事件到达时，进行响应。如果你熟悉设计模式的话，会很快想到观察者模式，例子中的事件分发器正式使用了此模式。

```
var uikit = uikit || {};  
uikit.event = uikit.event || {};  
  
uikit.event.EventTypes = {  
  EVENT_NONE : 0,  
  EVENT_INDEX_CHANGE : 1,  
  EVENT_LIST_DATA_READY : 2,  
  EVENT_GRID_DATA_READY : 3  
};
```

定义一个名称空间 **uikit**，并声明一个静态的常量：**EventTypes**，此变量定义了目前系统所支持的事件类型。

```
uikit.event.JSEvent = Base.extend({  
  constructor : function(obj){  
    this.type = obj.type || uikit.event.EventTypes.EVENT_NONE;  
    this.object = obj.data || {};  
  },  
  
  getType : function(){  
    return this.type;  
  },  
  
  getObject : function(){  
    return this.object;  
  }  
});
```

定义事件类，事件包括类型和事件中包含的数据，通常为事件发生的点上的一些信息，比如点击一个表格的某个单元格，可能需要将该单元格所在的行号和列号包装进事件的数据。

```
uikit.event.JSEventListener = Base.extend({
  constructor : function(listener) {
    this.sense = listener.sense;
    this.handle = listener.handle || function(event) {};
  },

  getSense : function() {
    return this.sense;
  }
});
```

定义事件监听器类，事件监听器包含两个属性，及监听器所关心的事件类型 **sense** 和当该类型的事件发生后要做的动作 **handle**。

```
uikit.event.JSEventDispatcher = function() {
  if(uikit.event.JSEventDispatcher.singleton) {
    return uikit.event.JSEventDispatcher.singleton;
  }

  this.listeners = {};

  uikit.event.JSEventDispatcher.singleton = this;

  this.post = function(event) {
    var handlers = this.listeners[event.getType()];
    for(var index in handlers) {
      if(handlers[index].handle && typeof handlers[index].handle ==
"function")
        handlers[index].handle(event);
    }
  };

  this.addEventListener = function(listener) {
    var item = listener.getSense();
    var listeners = this.listeners[item];
    if(listeners) {
      this.listeners[item].push(listener);
    } else {
      var hList = new Array();
      hList.push(listener);
      this.listeners[item] = hList;
    }
  };
};
```

```

    }
  };
}

uikit.event.JSEventDispatcher.getInstance = function () {
  return new uikit.event.JSEventDispatcher();
};

```

这里定义了一个单例的事件分发器，同一个系统中的任何组件都可以向此实例注册自己，或者发送事件到此实例。事件分发器事实上需要为何这样一个数据结构：

```

var listeners = {
  eventType.foo : [
    {sense : "eventType.foo", handle : function () {doSomething();}}
    {sense : "eventType.foo", handle : function () {doSomething();}}
    {sense : "eventType.foo", handle : function () {doSomething();}}
  ],
  eventType.bar : [
    {sense : "eventType.bar", handle : function () {doSomething();}}
    {sense : "eventType.bar", handle : function () {doSomething();}}
    {sense : "eventType.bar", handle : function () {doSomething();}}
  ],
  ...
};

```

当事件发生之后，分发器会找到该事件处理器的数组，然后依次调用监听器的 `handle` 方法进行相应。好了，到此为止，我们已经有了事件分发器的基本框架了，下来，我们开始实现我们的组件(Component)。

组件要通信，则需要加入事件支持，因此可以抽取出一个类：

```

uikit.component = uikit.component || {};

uikit.component.EventSupport = Base.extend({
  constructor : function () {

  },

  raiseEvent : function (eventdef) {
    var e = new uikit.event.JSEvent (eventdef);
    uikit.event.JSEventDispatcher.getInstance().post (e);
  },

  addActionListener : function (listenerdef) {
    var l = new uikit.event.JSEventListener (listenerdef);

```

```

    uikit.event.JSEventDispatcher.getInstance().addEventListener(l);
  }
});

```

继承了这个类的类具有事件支持的能力，可以 **raise** 事件，也可以注册监听器，这个 **EventSupport** 仅仅做了一个代理，将实际的工作代理到事件分发器上。

```

uikit.component.ComponentBase = uikit.component.EventSupport.extend({
  constructor: function(canvas) {
    this.canvas = canvas;
  },

  render : function(datamodel){}
});

```

定义所有的组件的基类，一般而言，组件需要有一个画布(**canvas**)的属性，而且组件需要有展现自己的能力，因此需要实现 **render** 方法来画出自己来。

我们来看一个继承了 **ComponentBase** 的类 **JSList**:

```

uikit.component.JSList = uikit.component.ComponentBase.extend({
  constructor : function(canvas, datamodel){
    this.base(canvas);
    this.render(datamodel);
  },

  render : function(datamodel){
    var jqo = $(this.canvas);
    var text = "";
    for(var p in datamodel.items){
      text += datamodel.items[p] + ";";
    }
    var item = $("

</div>").addClass("component");
    item.text(text);
    item.click(function(){
      jqo.find("div.selected").removeClass("selected");
      $(this).addClass("selected");

      var idx = jqo.find("div").index($(".selected")[0]);
      var c = new uikit.component.ComponentBase(null);
      c.raiseEvent({
        type : uikit.event.EventTypes.EVENT_INDEX_CHANGE,
        data : {index : idx}
      });
    });
  }
});


```

```
    });

    jqo.append(item);
  },

  update : function(event){
    var jqo = $(this.canvas);
    jqo.empty();
    var dm = event.getObjcet().items;

    for(var i = 0; i < dm.length();i++){
      var entity = dm.get(i).item;
      jqo.append(this.createItem({items : entity}));
    }
  },

  createItem : function(datamodel){
    var jqo = $(this.canvas);
    var text = datamodel.items;

    var item = $("

</div>").addClass("component");
    item.text(text);
    item.click(function(){
      jqo.find("div.selected").removeClass("selected");
      $(this).addClass("selected");

      var idx = jqo.find("div").index($(".selected")[0]);
      var c = new uikit.component.ComponentBase(null);
      c.raiseEvent({
        type : uikit.event.EventTypes.EVENT_INDEX_CHANGE,
        data : {index : idx}
      });
    });

    return item;
  },

  getSelectedItemIndex : function(){
    var jqo = $(this.canvas);
    var index = jqo.find("div").index($(".selected")[0]);
    return index;
  }
});

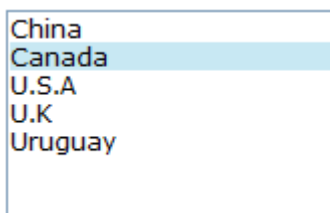

```

首先, 我们的画布其实是一个共 jQuery 选择的选择器, 选择到这个画布之后, 通过 jQuery 则可以比较容易的在画布上绘制组件。

在我们的实现中, 数据与视图是分离的, 我们通过定义这样的数据结构:

```
{items : ["China", "Canada", "U.S.A", "U.K", "Uruguay"]};
```

则可以 render 出如下图所示的 List:



好, 既然组件模型已经有了, 事件分发器的框架也有了, 相信你已经迫不及待的想要看看这些代码可以干点什么了吧, 再耐心一下, 我们还要写一点代码:

```
$(document).ready(function() {
    var ldmap = new uikit.component.ArrayLike(dataModel);

    ldmap.addActionListener({
        sense : uikit.event.EventTypes.EVENT_INDEX_CHANGE,
        handle : function(event) {
            var idx = event.getObject().index;
            uikit.component.EventGenerator.raiseEvent({
                type : uikit.event.EventTypes.EVENT_GRID_DATA_READY,
                data : {rows : ldmap.get(idx).grid}
            });
        }
    });

    var list = new uikit.component.JSList("div#componentList", []);
    var grid = new uikit.component.JSGrid("div#conditionsTable table
tbody");

    list.addActionListener({
        sense : uikit.event.EventTypes.EVENT_LIST_DATA_READY,
        handle : function(event) {
            list.update(event);
        }
    });
});
```

```
grid.addActionListener({
  sense : uikit.event.EventTypes.EVENT_GRID_DATA_READY,
  handle : function(event) {
    grid.update(event);
  }
});

uikit.component.EventGenerator.raiseEvent({
  type : uikit.event.EventTypes.EVENT_LIST_DATA_READY,
  data : {items : ldmap}
});

var colorPanel = new uikit.component.Panel("div#colorPanel");
colorPanel.addActionListener({
  sense : uikit.event.EventTypes.EVENT_INDEX_CHANGE,
  handle : function(event) {
    var idx = parseInt(10*Math.random())
    colorPanel.update(idx);
  }
});
});
```

使用 jQuery，我们在文档加载完毕之后，新建了两个对象 List 和 Grid，通过点击 List 上的条目，如果这些条目在 List 的模型上索引发生变化，则会发出 EVENT_INDEX_CHANGE 事件，接收到这个事件的组件或者 DataModel 会做出相应的响应。在本例中，ldmap 在接收到 EVENT_INDEX_CHANGE 事件后，会组织数据，并发出 EVENT_GRID_DATA_READY 事件，而 Grid 接收到这个事件后，根据事件对象上绑定的数据模型来更新自己的 UI。

上例中的类继承关系如下图：

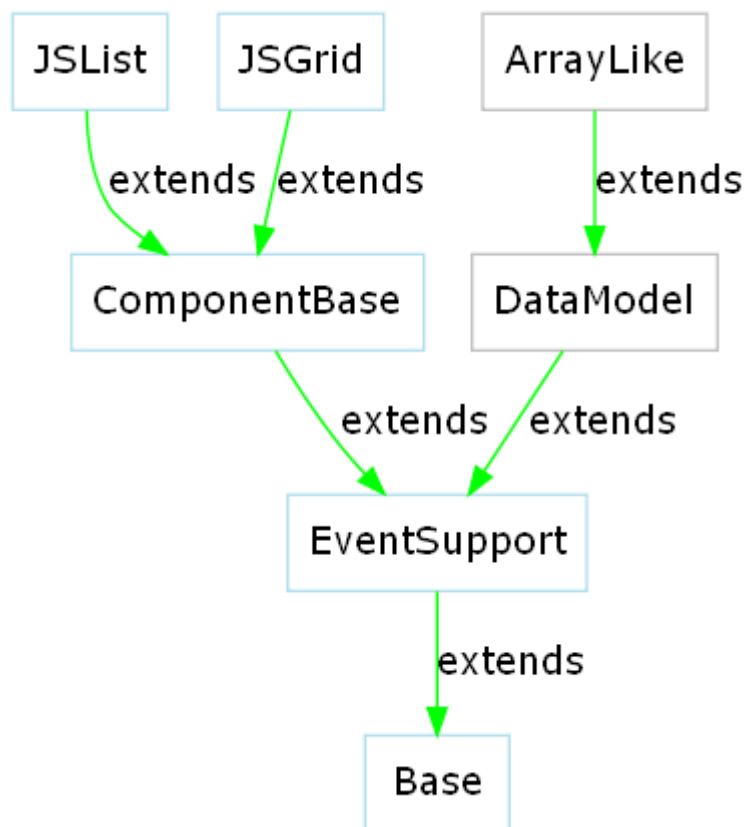


图 事件分发器类层次

应该注意的是，在绑定完监听器之后，我们手动地触发了 `EVENT_LIST_DATA_READY` 事件，来通知 List 可以绘制自身了：

```

uikit.component.EventGenerator.raiseEvent({
  type : uikit.event.EventTypes.EVENT_LIST_DATA_READY,
  data : {items : ldmapi}
});

```

在实际的应用中，这个事件可能是用户在页面上点击一个按钮，或者一个 Ajax 请求的返回，等等，一旦事件监听器注册完毕，程序就已经就绪，等待异步事件并响应。

点击 List 中的元素 China，Grid 中的数据发生变化

China	City	Product	Sales
Canada	Beijing	ProductA	1000
U.S.A	ShangHai	ProductB	23451
U.K	GuangZhou	ProductB	87652
Uruguay			

点击 Canada，Grid 中的数据同样发生相应的变化：

	City	Product	Sales
China			
Canada	Abbotsford	ProductD	56454
U.S.A	Chilliwack	ProductC	9767
U.K	Duncan	ProductX	34234
Uruguay			


由于 List 和 Grid 的数据是关联在一起的，他们的数据结构具有下列的结构：

```
var dataModel = [{
  item: "China",
  grid: [
    [{
      dname: "Beijing",
      type: "string"
    },
    {
      dname: "ProductA",
      type: "string"
    },
    {
      dname: 1000,
      type: "number"
    }
  ]],
  [{
    dname: "ShangHai",
    type: "string"
  },
  {
    dname: "ProductB",
    type: "string"
  },
  {
    dname: 23451,
    type: "number"
  }
  ]],
  [{
    dname: "GuangZhou",
    type: "string"
  },
  {
    dname: "ProductB",
    type: "string"
  }
  ],
}
```

```
{
  dname: 87652,
  type: "number"
}]
]
}, ...
];
```

一个组件可以发出多种事件，同时也可以监听多种事件，所以我们可以为 **List** 的下标改变事件注册另一个监听器，监听器为一个简单组件 **Panel**，当接收到这个事件后，该 **Panel** 会根据一个随机的颜色来重置自身的背景色(注意在 **List** 和 **Grid** 下面的灰色 **Panel**):

China	City	Product	Sales
Canada	Birmingham	ProductC	23451
U.S.A	Landon	ProductB	32445
U.K	Manchester	ProductC	87652
Uruguay			



第九章 函数式的 Javascript

要说 JavaScript 和其他较为常用的语言最大的不同是什么，那无疑就是 JavaScript 是函数式的语言，函数式语言的特点如下：

函数为第一等的元素，即人们常说的一等公民。就是说，在函数式编程中，函数是不依赖于其他对象而独立存在的(对比与 Java，函数必须依赖对象，方法是对象的方法)。

函数可以保持自己内部的数据，函数的运算对外部无副作用(修改了外部的全局变量的状态等)，关于函数可以保持自己内部的数据这一特性，称之为闭包。我们可以来看一个简单的例子：

```
var outter = function () {  
    var x = 0;  
    return function () {  
        return x++;  
    }  
}
```

```
var a = outter();  
print(a());  
print(a());
```

```
var b = outter();  
print(b());  
print(b());
```

运行结果为：

```
0  
1  
0  
1
```

变量 **a** 通过闭包引用 **outter** 的一个内部变量，每次调用 **a()** 就会改变此内部变量，应该注意的是，当调用 **a** 时，函数 **outter** 已经返回了，但是内部变量 **x** 的值仍然被保持。而变量 **b** 也引用了 **outter**，但是是一个不同的闭包，所以 **b** 开始引用的 **x** 值不会随着 **a()** 被调用而改变，两者有不同的实例，这就相当于面向对象中的不同实例拥有不同的私有属性，互不干涉。

由于 JavaScript 支持函数式编程，我们随后会发现 JavaScript 许多优美而强大的能力，这些能力得力于以下主题：匿名函数，高阶函数，闭包及柯里化等。熟悉命令式语言的开发人员可能对此感到陌生，但是使用 **lisp**, **scheme** 等函数式语言的开发人员则觉得非常亲切。

9.1 匿名函数

匿名函数在函数式编程语言中，术语成为 **lambda** 表达式。顾名思义，匿名函数就是没有名字的函数，这个是与日常开发中使用的语言有很大不同的，比如在 **C/Java** 中，函数和方法必须有名字才可以被调用。在 **JavaScript** 中，函数可以没有名字，而且这一个特点有着非凡的意义：

```
function func(){
    //do something
}

var func = function(){
    //do something
}
```

这两个语句的意义是一样的，它们都表示，为全局对象添加一个属性 **func**，属性 **func** 的值为一个函数对象，而这个函数对象是匿名的。匿名函数的用途非常广泛，在 **JavaScript** 代码中，我们经常可以看到这样的代码：

```
var mapped = [1, 2, 3, 4, 5].map(function(x){return x * 2});
print(mapped);
```

应该注意的是，**map** 这个函数的参数是一个匿名函数，你不需要显式的声明一个函数，然后将其作为参数传入，你只需要临时声明一个匿名的函数，这个函数被使用之后就别释放了。在高阶函数这一节中更可以看到这一点。

9.2 高阶函数

通常，以一个或多个函数为参数的函数称之为高阶函数。高阶函数在命令式编程语言中有对应的实现，比如 **C** 语言中的函数指针，**Java** 中的匿名类等，但是这些实现相对于命令式编程语言的其他概念，显得更为复杂。

9.2.1 JavaScript 中的高阶函数

Lisp 中，对列表有一个 **map** 操作，**map** 接受一个函数作为参数，**map** 对列表中的所有元素应用该函数，最后返回处理后的列表(有的实现则会修改原列表)，我们在这一小节中分别用 **JavaScript/C/Java** 来对 **map** 操作进行实现，并对这些实现方式进行对比：

```
Array.prototype.map = function(func /*, obj */){
    var len = this.length;
    //check the argument
```

```
if(typeof func !== "function"){
    throw new Error("argument should be a function!");
}

var res = [];
var obj = arguments[1];
for(var i = 0; i < len; i++){
    //func.call(), apply the func to this[i]
    res[i] = func.call(obj, this[i], i, this);
}

return res;
}
```

我们对 JavaScript 的原生对象 **Array** 的原型进行扩展, 函数 **map** 接受一个函数作为参数, 然后对数组的每一个元素都应用该函数, 最后返回一个新的数组, 而不影响原数组。由于 **map** 函数接受的是一个函数作为参数, 因此 **map** 是一个高阶函数。我们进行测试如下:

```
function double(x){
    return x * 2;
}
```

```
[1, 2, 3, 4, 5].map(double); //return [2, 4, 6, 8, 10]
```

应该注意的是 **double** 是一个函数。根据上一节中提到的匿名函数, 我们可以为 **map** 传递一个匿名函数:

```
var mapped = [1, 2, 3, 4, 5].map(function(x){return x * 2});
print(mapped);
```

这个示例的代码与上例的作用是一样的, 不过我们不需要显式的定义一个 **double** 函数, 只需要为 **map** 函数传递一个“可以将传入参数乘 2 并返回”的代码块即可。再来看一个例子:

```
[
    {id : "item1"},
    {id : "item2"},
    {id : "item3"}
].map(function(current){
    print(current.id);
});
```

将会打印:

item1

```
item2
```

```
item3
```

也就是说，这个 `map` 的作用是将传入的参数(处理器)应用在数组中的每个元素上，而不关注数组元素的数据类型，数组的长度，以及处理函数的具体内容。

9.2.2 C 语言中的高阶函数

C 语言中的函数指针，很容易实现一个高阶函数。我们还以 `map` 为例，说明在 C 语言中如何实现：

```
//prototype of function
void map(int* array, int length, int (*func)(int));
```

`map` 函数的第三个参数为一个函数指针，接受一个整型的参数，返回一个整型参数，我们来看看其实现：

```
//implement of function map
void map(int* array, int length, int (*func)(int)){
    int i = 0;
    for(i = 0; i < length; i++){
        array[i] = func(array[i]);
    }
}
```

我们在这里实现两个小函数，分别计算传入参数的乘 2 的值，和乘 3 的值，然后进行测试：

```
int twice(int num) { return num * 2; }
int triple(int num){ return num * 3; }
```

```
//function main
int main(int argc, char** argv){
    int array[5] = {1, 2, 3, 4, 5};
    int i = 0;
    int len = 5;

    //print the original array
    printArray(array, len);

    //mapped by twice
    map(array, len, twice);
    printArray(array, len);
}
```

```
//mapped by twice, then triple
map(array, len, triple);
printArray(array, len);

return 0;
}
```

运行结果如下:

```
1 2 3 4 5
2 4 6 8 10
6 12 18 24 30
```

应该注意的是 `map` 的使用方法, 如 `map(array, len, twice)` 中, 最后的参数为 `twice`, 而 `twice` 为一个函数。因为 C 语言中, 函数的定义不能嵌套, 因此不能采用诸如 JavaScript 中的匿名函数那样的简洁写法。

虽然在 C 语言中可以通过函数指针的方式来实现高阶函数, 但是随着高阶函数的“阶”的增高, 指针层次势必要跟着变得很复杂, 那样会增加代码的复杂度, 而且由于 C 语言是强类型的, 因此在数据类型方面必然有很大的限制。

9.2.3 Java 中的高阶函数

Java 中的匿名类, 事实上可以理解成一个笨重的闭包(可执行单元), 我们可以通过 Java 的匿名类来实现上述的 `map` 操作, 首先, 我们需要一个对函数的抽象:

```
interface Function{
    int execute(int x);
}
```

我们假设 `Function` 接口中有一个方法 `execute`, 接受一个整型参数, 返回一个整型参数, 然后我们在类 `List` 中, 实现 `map` 操作:

```
private int[] array;

public List(int[] array){
    this.array = array;
}

public void map(Function func){
    for(int i = 0, len = this.array.length; i < len; i++){
        this.array[i] = func.execute(this.array[i]);
    }
}
```

`map` 接受一个实现了 `Function` 接口的类的实例，并调用这个对象上的 `execute` 方法来处理数组中的每一个元素。我们这里直接修改了私有成员 `array`，而并没有创建一个新的数组。好了，我们来做个测试：

```
public static void main(String[] args){
    List list = new List(new int[]{1, 2, 3, 4, 5});
    list.print();
    list.map(new Function(){
        public int execute(int x){
            return x * 2;
        }
    });
    list.print();

    list.map(new Function(){
        public int execute(int x){
            return x * 3;
        }
    });
    list.print();
}
```

同前边的两个例子一样，这个程序会打印：

```
1 2 3 4 5
2 4 6 8 10
6 12 18 24 30
```

灰色背景色的部分即为创建一个匿名类，从而实现高阶函数。很明显，我们需要传递给 `map` 的是一个可以执行 `execute` 方法的代码。而由于 `Java` 是命令式的编程语言，函数并非第一位的，函数必须依赖于对象，附属于对象，因此我们不得不创建一个匿名类来包装这个 `execute` 方法。而在 `JavaScript` 中，我们只需要传递函数本身即可，这样完全合法，而且代码更容易被人理解。

9.3 闭包与柯里化

闭包和柯里化都是 `JavaScript` 经常用到而且比较高级的技巧，所有的函数式编程语言都支持这两个概念，因此，我们想要充分发挥出 `JavaScript` 中的函数式编程特征，就需要深入的了解这两个概念，我们在第七章中详细的讨论了闭包及其特征，闭包事实上更是柯里化所不可缺少的基础。

9.3.1 柯里化的概念

闭包的我们之前已经接触到，先说说柯里化。柯里化就是预先将函数的某些参数传入，得到一个简单的函数，但是预先传入的参数被保存在闭包中，因此会有一些奇特的特性。比如：

```
var adder = function(num) {
    return function(y) {
        return num + y;
    }
}
```

```
var inc = adder(1);
var dec = adder(-1);
```

这里的 `inc/dec` 两个变量事实上是两个新的函数，可以通过括号来调用，比如下例中的用法：

```
//inc, dec现在是两个新的函数，作用是将传入的参数值(+/-)1
print(inc(99)); //100
print(dec(101)); //100

print(adder(100)(2)); //102
print(adder(2)(100)); //102
```

9.3.2 柯里化的应用

根据柯里化的特性，我们可以写出更有意思的代码，比如在前端开发中经常会遇到这样的情况，当请求从服务端返回后，我们需要更新一些特定的页面元素，也就是局部刷新的概念。使用局部刷新非常简单，但是代码很容易写成一团乱麻。而如果使用柯里化，则可以很大程度上美化我们的代码，使之更容易维护。我们来看一个例子：

```
//update会返回一个函数，这个函数可以设置id属性为item的web元素的内容
function update(item) {
    return function(text) {
        $("div#" + item).html(text);
    }
}

//Ajax请求，当成功是调用参数callback
function refresh(url, callback) {
    var params = {
```

```
    type : "echo",
    data : ""
  };

$.ajax({
  type:"post",
  url:url,
  cache:false,
  async:true,
  dataType:"json",
  data:params,

  //当异步请求成功时调用
  success: function(data, status){
    callback(data);
  },

  //当请求出现错误时调用
  error: function(err){
    alert("error : "+err);
  }
});
}

refresh("action.do?target=news", update("newsPanel"));
refresh("action.do?target=articles", update("articlePanel"));
refresh("action.do?target=pictures", update("picturePanel"));
```

其中，`update` 函数即为柯里化的一个实例，它会返回一个函数，即：

```
update("newsPanel") = function(text){
  $("div#newsPanel").html(text);
}
```

由于 `update("newsPanel")` 的返回值为一个函数，需要的参数为一个字符串，因此在 `refresh` 的 Ajax 调用中，当 `success` 时，会给 `callback` 传入服务器端返回的数据信息，从而实现 `newsPanel` 面板的刷新，其他的文章面板 `articlePanel`，图片面板 `picturePanel` 的刷新均采取这种方式，这样，代码的可读性，可维护性均得到了提高。

9.4 一些例子

9.4.1 函数式编程风格

通常来讲，函数式编程的谓词(关系运算符，如大于，小于，等于的判断等)，以及运算(如加减乘数等)都会以函数的形式出现，比如：

```
a > b
```

通常表示为：

```
gt(a, b)//great than
```

因此，可以首先对这些常见的操作进行一些包装，以便于我们的代码更具有“函数式”风格：

```
function abs(x){ return x>0?x:-x; }
function add(a, b){ return a+b; }
function sub(a, b){ return a-b; }
function mul(a, b){ return a*b; }
function div(a, b){ return a/b; }
function rem(a, b){ return a%b; }
function inc(x){ return x + 1; }
function dec(x){ return x - 1; }
function equal(a, b){ return a==b; }
function great(a, b){ return a>b; }
function less(a, b){ return a<b; }
function negative(x){ return x<0; }
function positive(x){ return x>0; }
function sin(x){ return Math.sin(x); }
function cos(x){ return Math.cos(x); }
```

如果我们之前的编码风格是这样：

```
// n*(n-1)*(n-2)*...*3*2*1
function factorial(n){
  if(n == 1){
    return 1;
  }else{
    return n * factorial(n - 1);
  }
}
```

在函数式风格下，就应该是这样了：

```
function factorial(n) {
  if(equal(n, 1)){
    return 1;
  }else{
    return mul(n, factorial(dec(n)));
  }
}
```

函数式编程的特点当然不在于编码风格的转变，而是由更深层次的意义。比如，下面是另外一个版本的阶乘实现：

```
/*
 * product <- counter * product
 * counter <- counter + 1
 * */

function factorial(n) {
  function fact_iter(product, counter, max) {
    if(great(counter, max)) {
      return product;
    }else {
      fact_iter(mul(counter, product), inc(counter), max);
    }
  }

  return fact_iter(1, 1, n);
}
```

虽然代码中已经没有诸如+/-/*//之类的操作符，也没有>,<==,之类的谓词，但是，这个函数仍然算不上具有函数式编程风格，我们可以改进一下：

```
function factorial(n) {
  return (function factiter(product, counter, max) {
    if(great(counter, max)) {
      return product;
    }else {
      return factiter(mul(counter, product), inc(counter), max);
    }
  })(1, 1, n);
}

factorial(10);
```

通过一个立即运行的函数 **factiter**，将外部的 **n** 传递进去，并立即参与计算，最终返回运算结果。

9.4.2 Y-结合子

提到递归，函数式语言中还有一个很有意思的主题，即：如果一个函数是匿名函数，能不能进行递归操作呢？如何可以，怎么做？我们还是来看阶乘的例子：

```
function factorial(x) {
  return x == 0 ? 1 : x * factorial(x-1);
}
```

factorial 函数中，如果 **x** 值为 **0**，则返回 **1**，否则递归调用 **factorial**，参数为 **x** 减 **1**，最后当 **x** 等于 **0** 时进行规约，最终得到函数值(事实上，命令式程序语言中的递归的概念最早即来源于函数式编程中)。现在考虑：将 **factorial** 定义为一个匿名函数，那么在函数内部，在代码 **x*factorial(x-1)** 的地方，这个 **factorial** 用什么来替代呢？

lambda 演算的先驱们，天才的发明了一个神奇的函数，成为 **Y-结合子**。使用 **Y-结合子**，可以做到对匿名函数使用递归。关于 **Y-结合子** 的发现及推导过程的讨论已经超出了本部分的范围，有兴趣的读者可以参考附录中的资料。我们来看看这个神奇的 **Y-结合子**：

```
var Y = function(f) {
  return (function(g) {
    return g(g);
  })(function(h) {
    return function() {
      return f(h(h)).apply(null, arguments);
    };
  });
};
```

我们来看看如何运用 **Y-结合子**，依旧是阶乘这个例子：

```
var factorial = Y(function(func) {
  return function(x) {
    return x == 0 ? 1 : x * func(x-1);
  }
});
```

```
factorial(10);
```

或者：

```
Y(function(func) {
```

```

    return function(x) {
        return x == 0 ? 1 : x * func(x-1);
    }
}) (10);

```

不要被上边提到的 Y-结合子的表达式吓到，事实上，在 JavaScript 中，我们有一种简单的方法来实现 Y-结合子：

```

var fact = function(x) {
    return x == 0 ? 1 : x * arguments.callee(x-1);
}

fact(10);

```

或者：

```

(function(x) {
    return x == 0 ? 1 : x * arguments.callee(x-1);
}) (10); //3628800

```

其中，`arguments.callee` 表示函数自身，而 `arguments.caller` 表示函数调用者，因此省去了很多复杂的步骤。

9.4.3 其他实例

下面的代码则颇有些“开发智力”之功效：

```

//函数的不动点
function fixedPoint(fx, first){
    var tolerance = 0.00001;
    function closeEnough(x, y){return less(abs(sub(x, y)), tolerance)};
    function Try(guess){//try 是javascript中的关键字，因此这个函数名为大写
        var next = fx(guess);
        //print(next+" "+guess);
        if(closeEnough(guess, next)){
            return next;
        }else{
            return Try(next);
        }
    };
    return Try(first);
}

```

```
// 数层嵌套函数，
function sqrt(x) {
  return fixedPoint(
    function(y) {
      return function(a, b) { return div(add(a, b), 2); }(y, div(x, y));
    },
    1.0);
}

print(sqrt(100));
```

`fixedPoint` 求函数的不动点，而 `sqrt` 计算数值的平方根。这些例子来源于《计算机程序的构造和解释》，其中列举了大量的计算实例，不过该书使用的是 `scheme` 语言，在本书中，例子均被翻译为 JavaScript。

高级主题

第十章 核心概念深入

在前半部分章节中，涉及到一些重要的概念，在当时章节上下文中，限于内容，没有展开讨论，这些内容可能较难理解，因此都集中在这个章节进行讨论。具体涉及到的内容有原型链，执行期上下文，活动对象，作用域链以及 **this** 值。这部分内容可以结合之前章节中相关部分一起参考。

10.1 原型链

10.1.1 原型对象与原型链

正如第三章提到的，JavaScript 对象是一个属性的集合，另外有一个隐式的对象：原型对象。原型的值可以是一个对象或者 **null**。一般的引擎实现中，JS 对象会包含若干个隐藏属性，对象的原型由这些隐藏属性之一引用，我们在本文中讨论时，将假定这个属性的名称为 `__proto__`（事实上，SpiderMonkey 内部正是使用了这个名称，但是规范中并未做要求，因此这个名称依赖于实现）。

由于原型对象本身也是对象，根据上边的定义，它也有自己的原型，而它自己的原型对象又可以有自己的原型，这样就组成了一条链，这个链就是原型链。

JavaScript 引擎在访问对象的属性时，如果在对象本身中没有找到，则会去原型链中查找，如果找到，直接返回值，如果整个链都遍历且没有找到属性，则返回 **undefined**。原型链一般实现为一个链表，这样就可以按照一定的顺序来查找。

结合下边的例子：

```
var base = {
  name : "base",
  getInfo : function(){
    return this.name;
  }
}
```

```
var ext1 = {
  id : 0,
  __proto__ : base
}
```



```
var ext2 = {  
  id : 9,  
  __proto__ : base  
}  
  
print(ext1.id);  
print(ext1.getInfo());  
print(ext2.id);  
print(ext2.getInfo());
```

可以得到:

```
0  
base  
9  
base
```

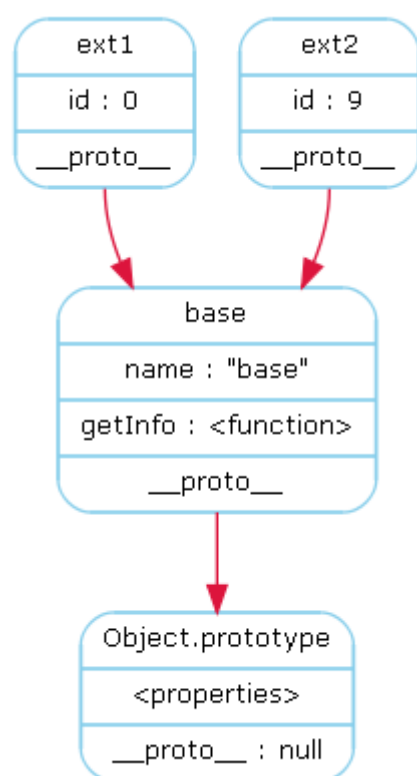


图 上例中对象的原型链

可以看到, 当执行 `ext1.id` 时, 引擎在 `ext1` 对象本身中就找到了 `id` 属性, 因此返回其值 `0`, 当执行 `ext1.getInfo` 时, `ext1` 对象中没有找到, 因此在其原型对象 `base` 中查找, 找到之后, 执行这个函数, 得到输出“base”。

我们将上例中的 `ext1` 对象稍加修改, 为 `ext1` 对象加上 `name` 属性:

```
var base = {
  name : "base",
  getInfo : function(){
    return this.name;
  }
}
```

```
var ext1 = {
  id : 0,
  name : "ext1",
  __proto__ : base
}
```

```
print(ext1.id);
print(ext1.getInfo());
```

可以看到:

```
0
ext1
```

这个运行效果同样验证了原型链的运行机制: 从对象本身出发, 沿着__proto__查找, 直到找到属性名称相同的值(没有找到, 则返回 undefined)。

我们对上例再做一点修改, 来更好的演示原型链的工作方式:

```
var base = {
  name : "base",
  getInfo : function(){
    return this.id + ":" + this.name;
  }
}
```

```
var ext1 = {
  id : 0,
  __proto__ : base
}
```

```
print(ext1.getInfo());
```

我们在 `getInfo` 函数中加入 `this.id`, 这个 `id` 在 `base` 对象中没有定义。同时, 删掉了 `ext1` 对象中的 `name` 属性, 执行结果如下:

```
0:base
```

应该注意的是，`getInfo` 函数中的 `this` 表示原始的对象，而并非原型对象。上例中的 `id` 属性来自于 `ext1` 对象，而 `name` 来自于 `base` 对象。这个特性的机制在 10.3 小节再做讨论。如果对象没有显式的声明自己的“`__proto__`”属性，这个值默认的设置是 `Object.prototype`，而 `Object.prototype` 的“`__proto__`”属性的值为“`null`”，标志着原型链的终结。

10.1.2 构造器

我们在来讨论一下构造器，除了上边提到的直接操作对象的 `__proto__` 属性的指向以外，JavaScript 还支持构造器形式的对象创建。构造器会自动的为新创建的对象设置原型对象，此时的原型对象通过构造器的 `prototype` 属性来引用。

我们以例子来说明，将 `Task` 函数作为构造器，然后创建两个实例 `task1`, `task2`:

```
function Task(id) {
    this.id = id;
}

Task.prototype.status = "STOPPED";
Task.prototype.execute = function(args) {
    return "execute task_" + this.id + "[" + this.status + "]: " + args;
}

var task1 = new Task(1);
var task2 = new Task(2);

task1.status = "ACTIVE";
task2.status = "STARTING";

print(task1.execute("task1"));
print(task2.execute("task2"));
```

运行结果如下：

```
execute task_1[ACTIVE]:task1
execute task_2[STARTING]:task2
```

构造器会自动为 `task1`, `task2` 两个对象设置原型对象 `Task.prototype`，这个对象被 `Task`（在此最为构造器）的 `prototype` 属性引用，参看下图中的箭头指向。

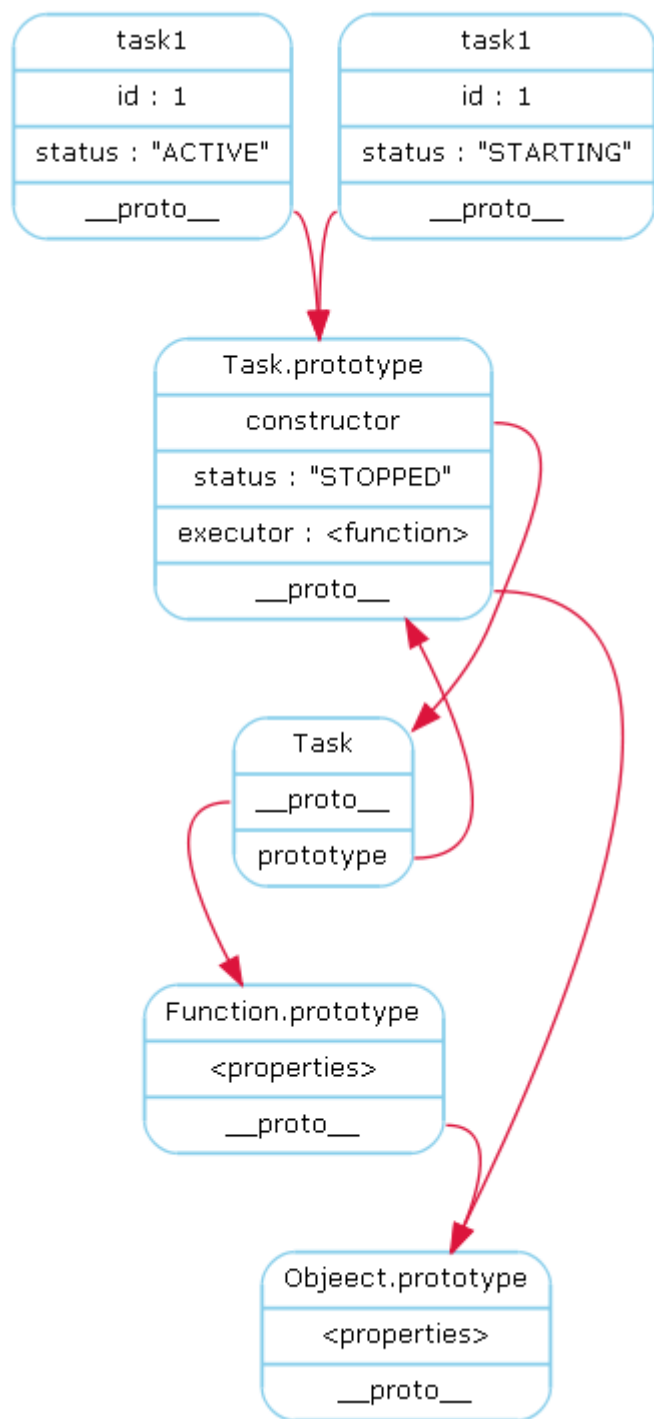


图 构造器方式的原型链

由于 `Task` 本身仍旧是函数，因此其“`__proto__`”属性为 `Function.prototype`，而内建的函数原型对象的“`__proto__`”属性则为 `Object.prototype` 对象。最后 `Object.prototype` 的“`__proto__`”值为 `null`。

10.2 执行期上下文

执行器上下文的概念贯穿于 JavaScript 引擎解释代码的全过程，这个概念是一个运行期的概念，执行器上下文一般实现为一个栈。按照 ECMAScript 的规范，一共有三种类型的代码，全局代码(游离于任何函数体之外)，函数代码，以及 eval 代码(eval 接受字符串，并对这个字符串求值，通常来讲，函数式编程都会提供这个函数或者类似的机制)。这三种代码均在自身的执行期上下文中求值。全局上下文仅有一个，函数上下文和 eval 上下文则可能有多个。

引擎在调用一个函数时，进入该函数上下文，并执行函数体，与其他程序设计语言类似，函数体内可以有递归，也可以调用其他函数(进入另外一个上下文，此时调用者被阻塞，直至返回)。调用 eval 会有类似的情况。

引擎在初始化之后，将 global 的上下文对象压入栈：

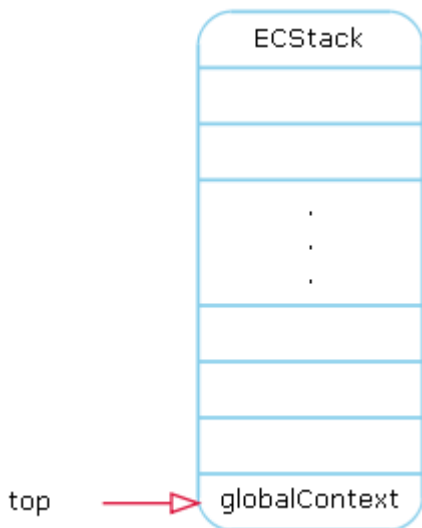


图 执行期上下文栈(初始状态)

```
(function (name) {  
    print("hello, "+name);  
})("jack");
```

执行上边代码的时候，进入函数执行期上下文，将匿名函数执行期上下文压入栈中：

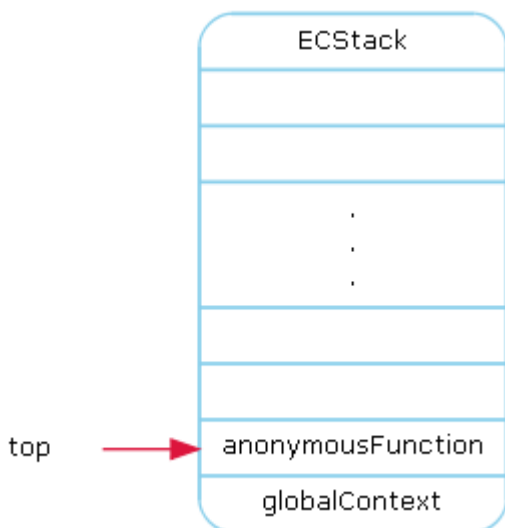


图 进入函数执行期上下文

执行完成之后，弹出该上下文对象。既然执行期上下文栈中存放的是执行期上下文对象，那么我们来详细看看这个对象的结构。上文提到，ECMAScript 代码有三类，对应的执行期上下文对象也有三类，每个上下文对象都有一些必须的属性用以为执行于其上的代码服务，以及记录/跟踪代码执行状态等。

一个典型的上下文对象的结构如下：

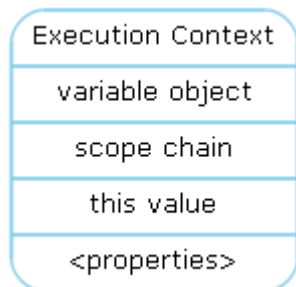


图 上下文对象结构

当然，根据不同的实现，这个对象可以包含任意其他的属性(上图中的<properties>部分)。每个上下文对象所需要包含的有变量对象，作用域链以及 **this**。这三个属性在不同类型的上下文对象中意义可能不同。

变量对象只是一个抽象概念，在全局上下文中，变量对象是全局变量自身。而在函数上下文中，变量对象表现为活动对象，活动对象将在下一小节展开，对于 **eval** 上下文，**eval** 可能使用全局的变量对象，也可能使用函数的变量对象，这取决于其调用的位置。因此可以说，变量对象有两类，全局的变量对象(全局变量 **global** 本身或者活动对象，**eval** 使用这两者之一)。

结合执行器上下文栈和原型对象，我们可以得到下列的示意图：

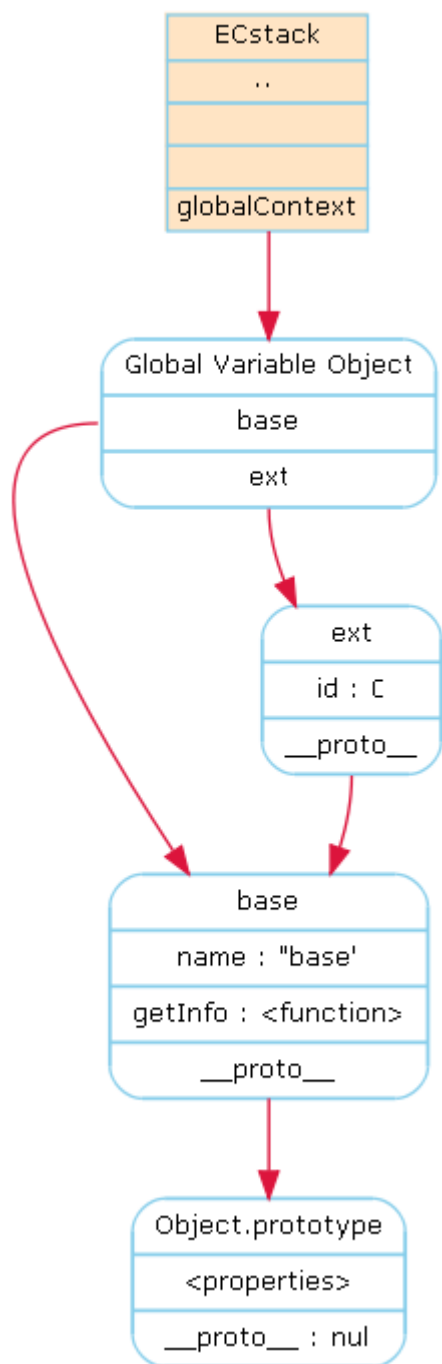


图 执行器上下文栈及变量对象，原型链示意图

10.3 活动对象

在 JavaScript 中，当一个函数被调用的时候，就会产生一个特殊的对象：活动对象。这个对象中包含了参数列表和 **arguments** 对象等属性。由于活动对象是变量对象的特例，因此它包含变量对象所有的属性如变量定义，函数定义等。

我们来看一个实例：

```
function func(handle, message){
  var id = 0;
  function doNothing(x){
    return x;
  }
  handle(message);
}

func(print, "hello");
```

当代码执行到 `func(print, "hello")` 时，活动对象被创建，这个活动对象的图形示意如下：

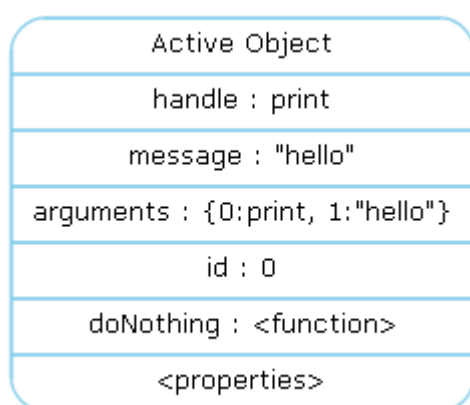


图 上例中函数调用时的活动对象

10.4 作用域链

作用域链与原型链类似，也是一个对象组成的链，用以在上下文中查找标识符(变量，函数等)。查找时也与原型链类似，如果激活对象本身具有该变量，则直接使用变量的值，否则向上层搜索，一次类推，知道查找到或者返回 `undefined`。作用域链的主要作用是用以查找自由变量，所谓自由变量是指，在函数中使用的，非函数内部局部变量，也非函数内部定义的函数名，也非形式参数的变量。这些变量通常来自于函数的“外层”或者全局作用域，比如，我们在函数内部使用的 `window` 对象及其属性。

关于作用域链及自由变量，我们可以来看下面一个例子：

```
var topone = "top-level";

(function outter(){
  var middle = "mid-level";

  (function inner(){
```



```

var bottom = "bot-level";

print(topone+">" + middle+">" + bottom);
}) ();
}) ();

```

在函数 `inner` 之中，`print` 语句中出现的 `topone`, `middle` 变量就是自由变量。

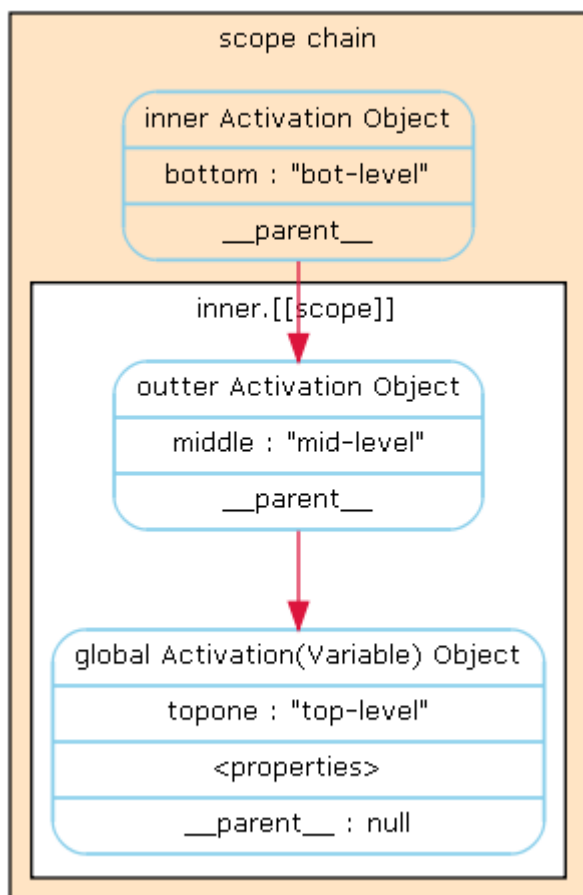


图 上例中的作用域链

根据上图我们可以看出，内部函数的作用域链，由两部分：内部函数自身的活动对象，内部函数的一个属性“`[[scope]]`”，而“`[[scope]]`”的值为其外部函数 `outter` 的活动对象，其更外部的全局 `global` 对象的变量对象。这样，如果在 `inner` 中要使用外部的自由变量，显然可以很方便的沿着作用域链上溯。

事实上，函数的属性“`[[scope]]`”会在函数对象创建的时候被创建，这个特性在下一小节中讨论，而不论函数的嵌套层次有多深，它的“`[[scope]]`”总会引用所有的位于其外层的上下文中的变量对象(在函数中，为活动对象)。

10.5 this 值

`this` 在之前的章节中做过讨论，在 ECMAScript 的规范中对 `this` 的定义为：`this` 是一个特殊的对象，与执行期上下文相关，因此可以称之为上下文对象。`this` 是执行期上下文对象的一个属性，参见本章 10.2 小节的图。

由于 `this` 是执行期上下文对象的属性，因此在代码中使用 `this`，其值直接从上下文对象那个中获得，而无需查找作用域链，其值在进入上下文的那个时刻被确定。

在全局上下文中，`this` 是全局对象本身：

```
var attribute = "attribute";

print(attribute);
print(this.attribute);
```

执行结果为：

```
attribute
attribute
```

而在函数上下文中，不同的调用方式可以有不同的值。

10.5.1 词法作用域

在 JavaScript 中，函数对象的创建和函数本身的执行是完全不同的两个过程：

```
function func() {
    var x = 0;
    print("function func");
}
```

是为函数的创建，而下面这条语句：

```
func();
```

才是函数的执行。

所谓词法作用域(静态作用域)是指，在函数对象的创建时，作用域“[[scope]]”就已经建立，而并非到执行时，因为函数创建后可能永远不会被执行，但是作用域是始终存在的。

比如在上例中，如果在程序中使用没有调用 `func()`，那么，`func` 对象仍旧是存在的，在内存

的结构可能是这样的:

```
func.[ "[scope]" ] = global.[ "variable object" ];
```

而当函数执行时, 进入函数执行期上下文, 函数的活动对象被创建, 此时的作用域链是活动对象和 "[scope]" 属性的合成。

10.5.2 this 的上下文

this 值是执行期上下文对象的一个属性(执行期上下文对象包括变量对象, 作用域链以及 **this**)。执行期上下文对象有三类, 当进入不同的上下文时, **this** 的值会确定下来, 并且 **this** 的值不能更改。结合前面小节讨论的内容, 在执行全局代码时, 控制流会进入全局执行期上下文, 而在执行函数时, 又会有函数执行期上下文。我们来看下面一个例子:

```
var global = this;
var tom = {
  name : "Tom",
  home : "desine",
  getInfo : function(){
    print(this.name + ", from "+this.home);
  }
};
```

```
tom.getInfo();
```

```
var jerry = {
  name : "Jerry",
  getInfo : tom.getInfo
}
```

```
jerry.getInfo();
```

```
global.getInfo = tom.getInfo;
global.getInfo();
```

执行结果为:

```
Tom, from desine
Jerry, from undefined
undefined, from undefined
```

`tom` 对象本身具有 `name` 和 `home` 属性，因此在执行 `tom.getInfo` 时，会打印 `tom` 对象上的这两个属性值。当将 `global.getInfo` 属性设置为 `tom.getInfo` 时，`getInfo` 中的 `this` 值，在运行时，事实上是 `global` 对象(还记得在全局执行期上下文对象中，`global` 的变量对象的 `this` 值吗?)的变量对象中的 `this` 就是自身。而 `global.name` 和 `global.home` 都没有定义，因此会得到上边的结果。

从上例中还可以看到，在函数 `getInfo` 调用时，在 `getInfo` 之前的对象 (`tom,jerry,global`)会被作为 `this` 来执行。当然 `global` 可以省略，这时仍然是全局对象作为 `this`。应该记住的是，`this` 的值取决于调用函数的方式(当然这里的 `this` 指函数上下文中的 `this`，全局上下文的我们已经讨论过了)。

第十一章 客户端的 JavaScript

毫无疑问，到目前为止，JavaScript 应用最为广泛，也最为成功的领域就是客户端，或者称为浏览器上的 JavaScript。JavaScript 为页面开发注入了活力，如与服务器交互形成的局部刷新，鼠标事件的响应，动态的页面风格变换等等，都直接依靠与 JavaScript 的支持。

11.1 客户端 JavaScript 执行环境

我们在基础部分提到过，JavaScript 代码都有一个执行环境，所有的 JavaScript 代码均被包含在一个全局对象中，比如在所有函数之外声明：

```
var x = 3;
var str = "global";
```

这两个变量声明语句事实上是为全局对象添加了两个属性 `x` 和 `str`。可以将全局对象想象成一个大的匿名自执行函数：

```
(function() {
  var x = 3;
  var str = "global";
  //...
})();
```

在浏览器端，这个全局的对象称为 `window`，`window` 是所有 JavaScript 对象的根，我们可以通过 `window` 对象的属性 `document` 来访问页面本身，也可以调用 `window` 的一些方法来与用户交互，比如：

```
window.alert("This is a message");
alert("This is a message");
```

这两个语句事实上的效果是相同的，在引用全局对象 `window` 的方法时，我们可以忽略前缀，只使用方法名本身。

`window` 对象是对浏览器当前窗口的引用，因为浏览器会将 `window` 与自身绘制出来的窗口绑定起来，我们对 `window` 的操作事实上会映射到浏览器窗口上。正是浏览器本身提供了这种脚本化的能力，我们才有机会通过 JavaScript 代码来完成诸如改变页面标题，弹出警告框，修改页面内容(通过对 `window.document` 的修改)等操作。

11.2 文档对象模型(DOM)

DOM 即文档对象模型，它是一个平台，提供语言无关的 API，允许程序访问并更改文档的内容，结构以及样式。HTML 文档是一个树形的结构，与浏览器中的页面中的结构一一对应。

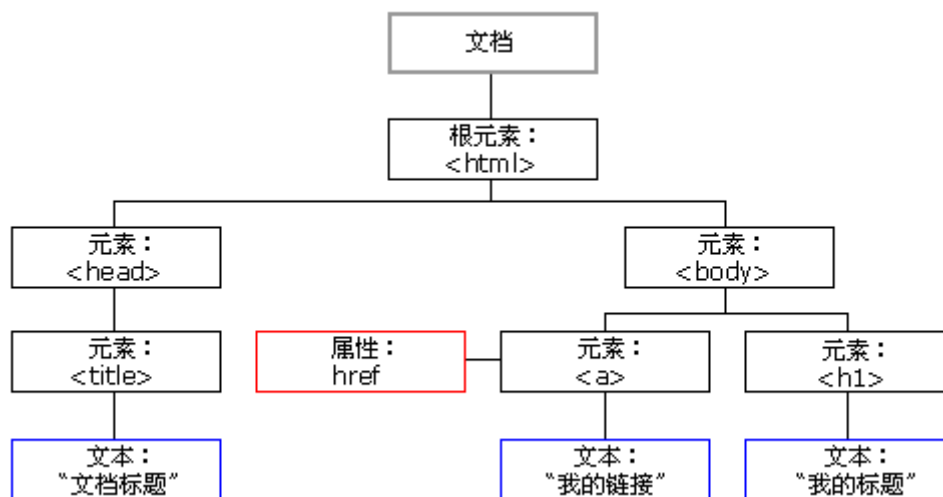


图 W3C 站点中的一个 HTML 结构示例

JavaScript 通过修改/遍历 DOM，即可映射到对 WEB 页面的操作上，比如一个简单的页面如下：

```

<html>
  <head>
  </head>
  <body>
    <div id="con">
    </div>
  </body>
</html>

```

通过 JavaScript 操作 DOM:

```
var con = document.getElementById("con");
```

即可对应到 WEB 页面中的 id 为“con”的 div 标签，可以为该标签设置背景色，或者绑定 click 事件的处理函数等等。JavaScript 可以通过一些浏览器内置的方法来对 DOM 进行遍历，增加，删除 DOM 的子节点，访问 DOM 上的 Form, Frame, Image 等节点，修改他们的 CSS 样式，注册/注销事件处理函数，从而使页面“活动”起来。

11.3 事件驱动模型

由于客户端 JavaScript 的开发属于用户界面开发的范畴，因此使用事件驱动模型就显得非常自然了，事件驱动的特点在于：代码块的运行与否与程序流程无关。而传统的流式代码的特点是，从函数的入口进入，依次调用各个子模块的处理函数，最后退出。这种编程模式主要适用于与 UI 关系不大的场合，很少有异步的过程，这些特点可能要追溯到计算机程序的最初模型：批处理。

而事件驱动模型主要是面向用户的，你在实现无法知道用户会如何使用你的程序，因此就只能通过回调，事件监听等方式，当用户做出某个动作时才会触发之前已经注册好的监听器，从而执行相关代码，而不是顺序的执行。甚至在一次运行中，部分代码始终没有被触发，也就根本不会被执行到。

比如在页面中，我们为按钮的点击事件注册了事件监听器，当点击的时候弹出一个对话框，但是用户打开页面后，浏览完内容后就直接关闭了，没有点击按钮，那么相关的代码就没有被触发，我们来看一个简单的示例：

我们有一个页面，内容如下：

```
<html>
<head>
<style>
body{
  margin: 20px;
  font-family : "Verdana";
  font-size : normal;
  background-color : #eee;
}
</style>
</head>
<body onload="init()" >
  <p>
    <label for="toclick">Please click the button:</label>
    <input id="toclick" type="button" value="Click me" />
  </p>
</body>
</html>
```

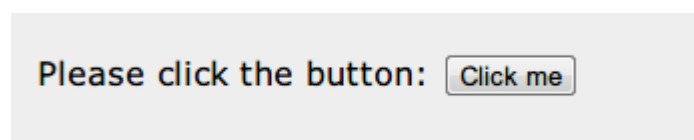


图 页面 click-me 的展示

可以看到 `body` 标签的 `onload` 属性被赋予一个值 `"init()"`，这是一个对 JavaScript 代码的

调用, 时机发生在当页面加载完成之后, 也就是浏览器已经创建了所有 **body** 中的 **DOM** 对象之后。

我们来看看这个 **init** 函数的内容:

```
<script type="text/javascript" language="javascript">
function init(){
    var button = document.getElementById('tclick');
    button.onclick = function(){
        alert('button is clicked');
    }
}
</script>
```

这段代码先从 **document** 中获取 id 为“tclick”的元素(也就是我们刚才在 **html** 中声明的 **button**), 然后为其 **click** 事件绑定一个函数, 当 **click** 事件被触发时, 弹出一个警告框。

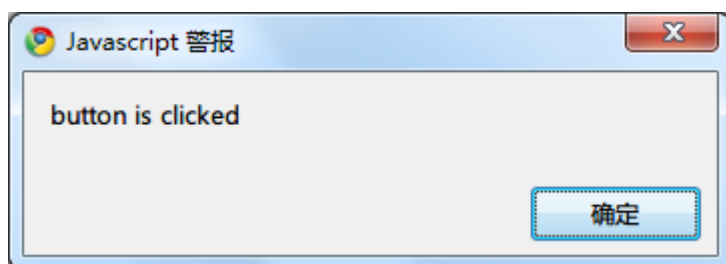


图 警告框效果

11.4 与服务器端交互(Ajax)

Ajax 本身被作为前端技术, 提出的时间是比较早的, 但是由于种种原因, 没有引起人们的普遍关注, 而当 **Google** 的很多产品如 **Google Map**, **GMail** 等横空出世之后, **Ajax** 才被越来越多的公司所接受并引入到自己的产品中。现在几乎所有的网站都有不同层次的 **Ajax** 交互, 纯静态的页面已经非常少了。

简而言之, **Ajax** 表示异步 **JavaScript** 与 **XML**, 事实上, **Ajax** 与 **XML** 几乎没有任何关系, 因为是异步交互, 所以用户的页面不用刷新, 在同一个页面中, 客户端请求服务数据, 当服务数据返回时, 通过 **JavaScript** 将数据片段填充到页面的某个部分, 即实现了局部刷新, 这个过程对用户来说实际上是透明的。

Ajax 对服务器端没有限制, 一个 **Ajax** 请求和普通的请求一样, 发送数据(**GET/POST**) 到一个服务端 **URL** 上。服务端的实现可以是 **php**, 可以是 **ASP** 或者 **JSP** 等。以 **J2EE** 为例, 数据发送到 **servlet**, **servlet** 对请求进行处理, 最后向输出流写入数据, 浏览器得到这些数据之后, 会按照 **Ajax** 调用时的注册情况来回调 **JavaScript** 函数, **JavaScript** 在操作 **DOM** 上具有天生的优势, 因此可以很方便的在无刷新的情况下更新页面的部分或者全部。

下面我们来看一个具体的实例，通过异步的向后端的 **php** 脚本发送请求，页面的内容得到了局部更新，而无需重刷整个页面。首先编写一个简单的 **php** 脚本：

```
<?php

$type = $_REQUEST['type'];
$string = $_REQUEST['string'];

if($type == 0){
    echo strtoupper($string);
}else{
    echo strtolower($string);
}

?>
```

php 的逻辑很简单，请求中包含两个参数，**type** 和 **string**，如果 **type** 为 0，则将第二个参数 **string** 转换为大写返回，否则转换为小写返回。

XMLHttpRequest 对象是进行 **Ajax** 的核心，所有的主流浏览器都已各自不同的方式进行了实现，如果不使用客户端 **JavaScript** 框架，那么使用起来会有所差异：

```
function initxhr(){
    if(window.XMLHttpRequest){
        xhr = new XMLHttpRequest();
    }else if(window.ActiveXObject){
        xhr = new ActiveXObject("Msxml2.XMLHTTP");
    }else{
        throw new Error("xhr is not supported");
    }
}
```

创建了 **XMLHttpRequest** 对象之后，我们即可使用它来进行与服务端的异步通信：

```
function doajax(url, panelId){
    if(xhr == null){
        initxhr();
    }

    if(xhr != null){
        xhr.open("GET", url, true);
        xhr.onreadystatechange = updatePanel(panelId);
        xhr.send(null);
    }
}
```

```
    }else{  
        throw new Error("xhr is not initied");  
    }  
}  
}
```

通过设置 XMLHttpRequest 对象的 onreadystatechange 属性，当服务端产生响应时，浏览器会回调此处注册的函数：

```
function updatePanel (panelId) {  
    return function () {  
        if(xhr.readyState == 4) {  
            var response = xhr.responseText;  
            alert(response);  
            document.getElementById(panelId).innerHTML = response;  
        }  
    }  
}
```

运行结果如下：

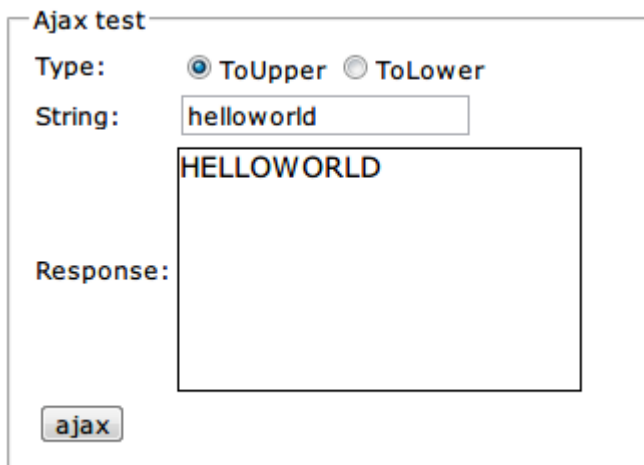


图 Ajax 示例

11.5 调试

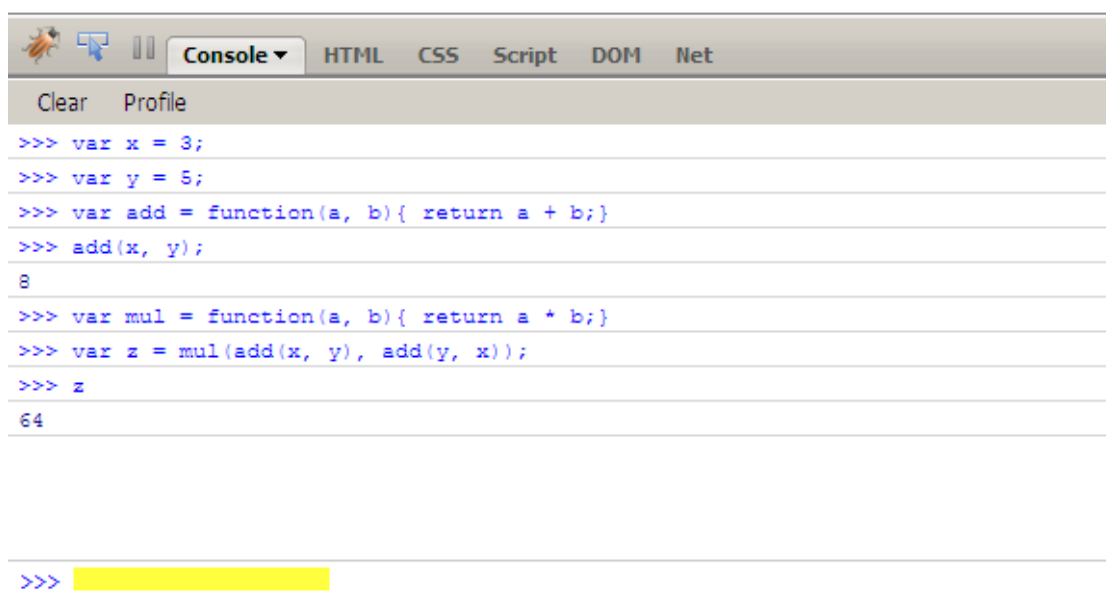
在实际的开发环境中，不可能做到代码没有 bug，变量的拼写错误，使用了未经初始化的变量，死循环等等，这些都是比较容易解决的错误，但是如果有隐藏的比較深的 bug，时隐时现，我们就需要一个调试环境了。脚本语言的调试尤其艰难，因为一切都是运行时决定的，因此不可能预先知道代码有错误(不包括显式的词法错误)。

我们这一节中介绍集中浏览器中的调试 JavaScript 的方式：

11.5.1 FireFox

Firefox 以速度，可扩展性，标准性深受广大开发人员的喜爱。Firefox 具有强大的插件机制，用户可以自己为 Firefox 增添新的功能(很多插件只需要有 JavaScript/CSS/XUL 经验即可)，而 WEB 开发人员最喜爱的插件之一即为：FireBug。FireBug 是一个 Firefox 的插件，使用它，使前端开发人员丢弃了老旧的 `alert`，FireBug 提供一个控制台，开发人员可以直接像在 windows 的控制台那样，通过简单的命令来查看变量的值，状态等。

作为一个调试工具，FireBug 提供如其他 IDE 中那样的基本功能，断点，步进，watch 等等功能，下面我们来详细介绍 FireBug：



```
Clear Profile
>>> var x = 3;
>>> var y = 5;
>>> var add = function(a, b){ return a + b;}
>>> add(x, y);
8
>>> var mul = function(a, b){ return a * b;}
>>> var z = mul(add(x, y), add(y, x));
>>> z
64

>>>
```

通过在提示符“>>>”之后键入 JavaScript 代码即可执行这些代码，由图可见 FireBug 有 6 个标签：控制台，HTML 查看器，CSS 查看器，脚本查看器，DOM 查看器，以及网络性能监控。

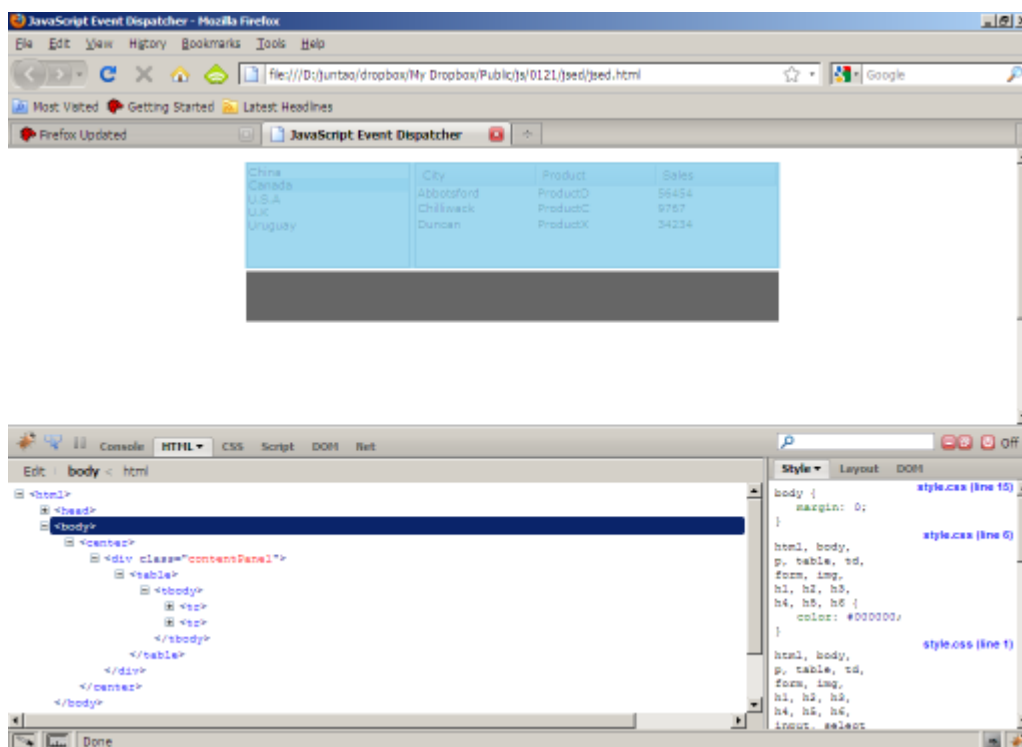


图 HTML 查看器标签

在 FireBug 处于“查看”状态时，通过鼠标在页面上移动，页面的一部分区域会高亮，点击此高亮区域即进入查看状态，注意此时 FireBug 的 HTML 查看窗口中的内容：它会高亮显示当前选中区域的 HTML 代码，而且你可以动态的编辑这些标签的属性和样式，对于页面大小，风格的微调十分有用。

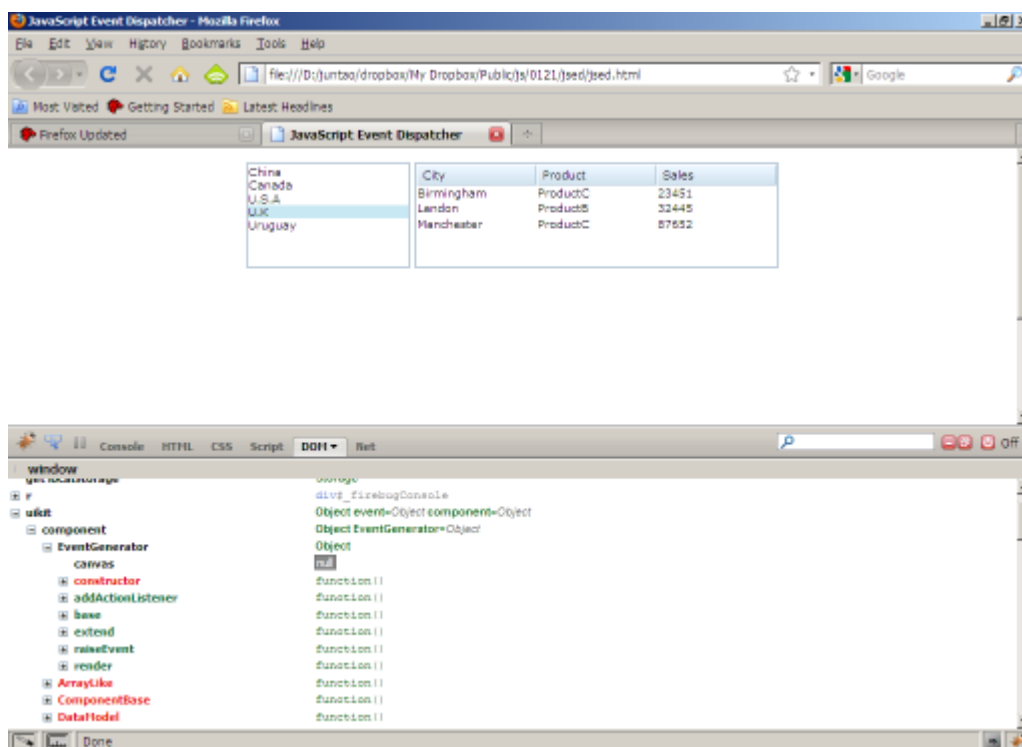


图 FireBug DOM 查看器

通过 DOM 查看器，我们可以清晰的看到页面中所有 JavaScript 对象，而且是以树的形式，可以逐层点开，查看详情，这对于调试页面非常有用。

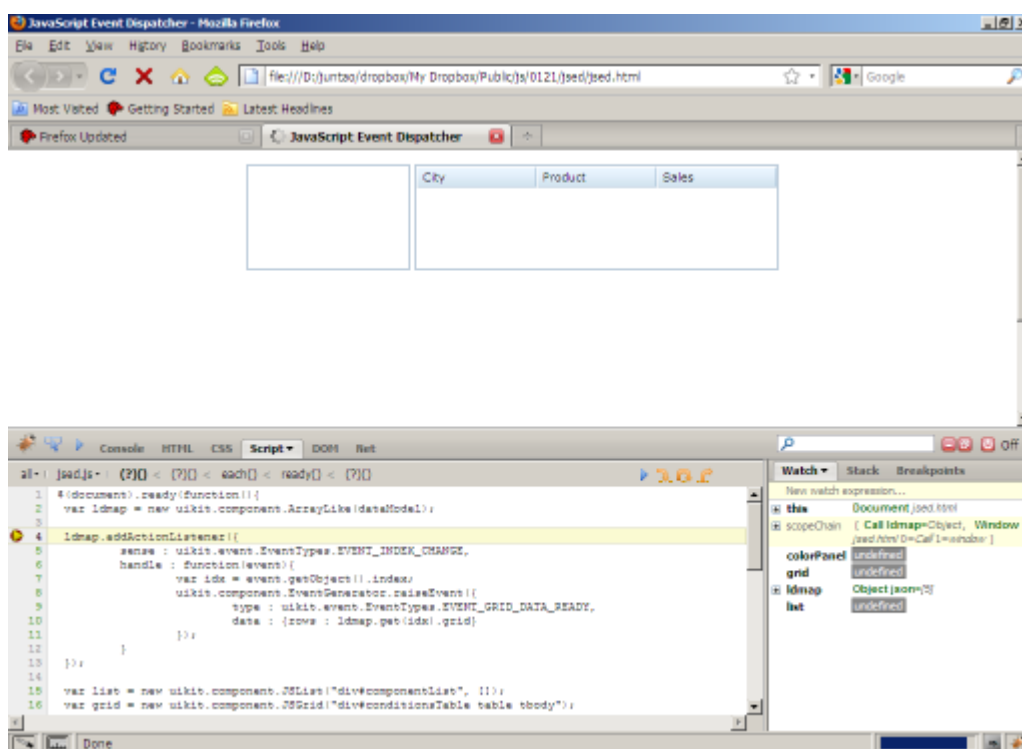


图 FireBug 的调试界面

通过使用 FireBug，我们就可以丢弃 alert 方式的调试了，FireBug 的调试与其他的 IDE 的使用非常类似，通过在代码左侧的行号旁边单击打断点，然后刷新页面进入调试环境，可以单步执行，直接返回等。右侧的面板上包含 watch，栈情况，以及关于所有断点的信息，调试起来非常方便。

FireBug 的控制台中常用的一些命令：

记录日志，一般用于调试时将 JavaScript 字符串打印出来

```
console.log(object [, object, ...])
```

打印消息，用于调试

```
console.info(obj)
```

用于打印一个对象，将 JavaScript 对象的各个属性树以直观的形式展现。

```
console.dir(obj)
```

```
console.trace()
```

11.5.2 Chrome

Chrome 是 Google 的浏览器，渲染引擎为苹果公司(Apple)的开源项目 WebKit，而 JavaScript 引擎为 V8，根据作者经验，Chrome 为当前浏览器界综合性能最好的一款浏览器，无论是速度，资源的占用，响应时间，以及界面，可扩展性等方面，远远的超过了同侪。

Chrome 一开始就附有开发人员工具，前端开发人员可以使用这个工具进行类似与 FireBug 那样对代码进行调试，同时可以动态增删 HTML 属性，查看页面响应时间，以便找出系统的瓶颈等。

像 FireBug 一样，Chrome 的内置开发人员工具也可以查看 JavaScript 的运行时结构，我们可以通过下面的一些图例来看 Chrome 的调试工具的用法：

在 chrome 中，CTRL-SHIFT-I 启动开发人员工具：

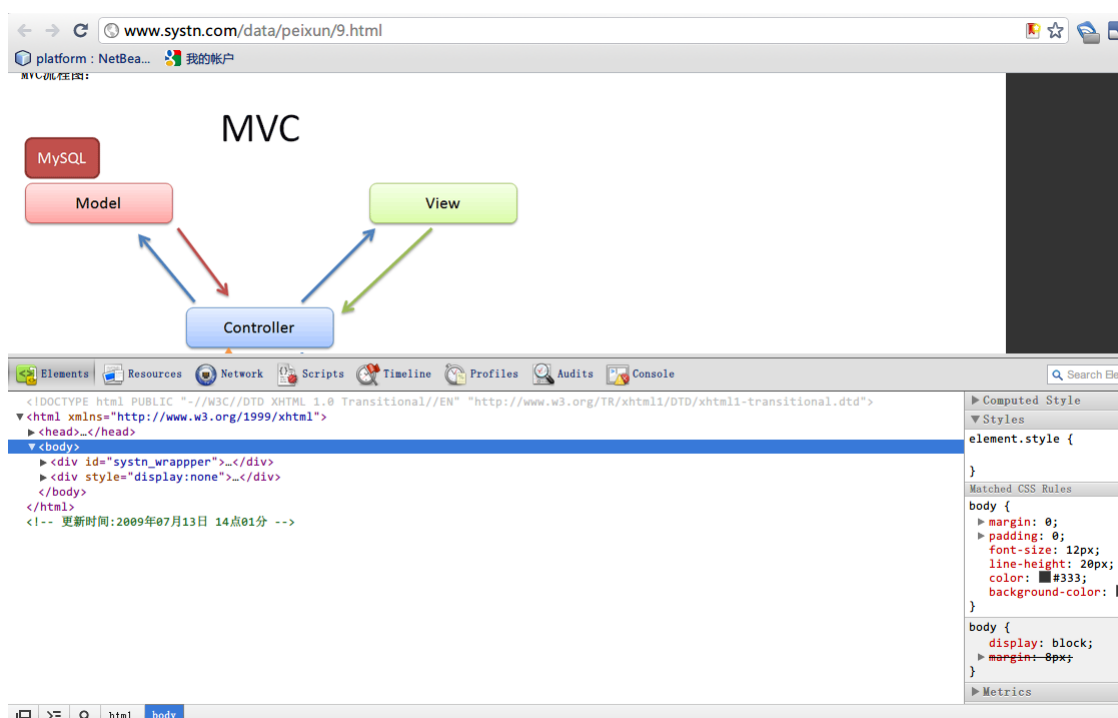


图 chrome 开发人员工具

在 console 标签中，可以进行一些脚本的测试：

```
> var a = 5;
undefined
> console.log(a)
5
< undefined
> var func = function(a, b){ return a+b;}
undefined
> var str = func("hello", " world");
undefined
> str
"hello world"
>
```

图 chrome 的 console 界面

在实际的项目中，console 系列的方法非常有用，特别是 web 容器如 tomcat 等返回的是复杂的 JavaScript 对象时，console.dir(object) 可以直观的让开发人员看到对象的结构，非常便于调试。

11.6 客户端的 MVC

MVC 模型是在实际应用中使用的较多的一种模式，它在很大程度上降低了整个应用开发的复杂度。在 J2EE 应用中，使用了 MVC 的框架不计其数，比如 struts, JSF 等等。当然 MVC 作为一种应用程序的模型并不限制其使用的场景，比较著名的 Swing 工具包也是建立在 MVC 模型上的。

在 MVC 模型中，应用被分为三个功能块，M 表示模型(Model)，通常为后台的数据；V 表示视图(View)，表示数据的展现，如 Web 页面或者 2D 库渲染出来的其他 UI 组件，而 C 表示控制器(Controller)，负责逻辑部分的控制，协调模型和视图的关系。使用这个模型，可以降低个层次之间的耦合度，使得软件可以较大程度上得到重用。

一般而言，MVC 是构建在整个系统中的，比如应用的数据来自于远程数据库或本地的文件系统，视图则在前端，直接展示给用户，控制器部分则运行在容器端。我们这个小节要讨论的并不是这个结构，而是纯粹建立在前端的 MVC。

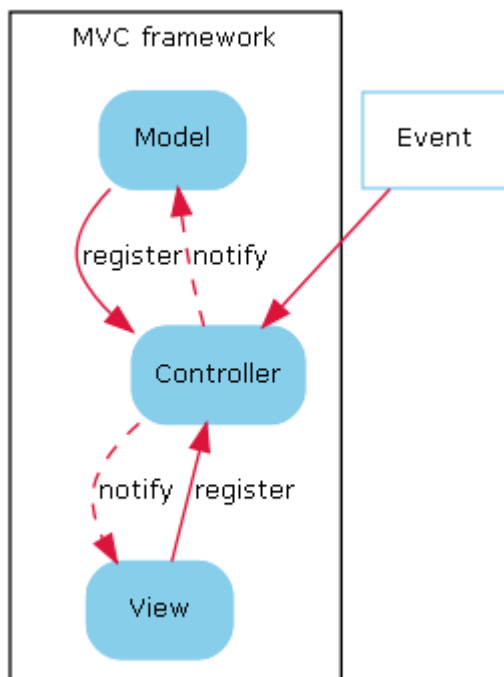


图 MVC 模型示意图

既然 MVC 模型的目的是降低层次间的耦合，降低开发的复杂度，使得软件尽可能的重用，我们不妨建立这样一套规则：

- 用 HTML 表示模型
- 用 CSS 来负责渲染视图
- 用 JavaScript 负责控制前两者

在 HTML 中，只是将文档的结构和内容组织起来，通过 CSS 进行渲染，布局等工作，而为用户交互的部分则由 JavaScript 来控制，我们来看这样一个例子：我们有一个页面，其中有一个 panel，当点击这个 panel 时其背景色会发生变化。

这个 HTML 文件看起来应该是这样的：

```
<html>
  <head>
    <script src="jquery-1.3.2.js" type="text/javascript" ></script>
    <script src="controller.js" type="text/javascript"></script>
    <link rel="stylesheet" href="style.css" type="text/css" />
  </head>
  <body>
    <div class="contentGray" id="content">
      This is the content of a panel
    </div>
  </body>
</html>
```

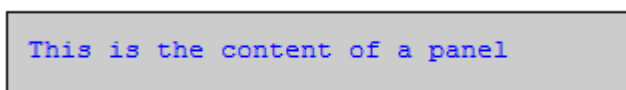

它引入了两个脚本文件(控制器)和一个样式表文件(视图风格), 而 HTML 本身只不过定义了 `div` 的结构和其位于 `body` 中的层次关系。

我们的视图渲染部分: 样式表的内容是这样的:

```
.contentGray{
  background : #ccc;
  color : blue;
  border : 1px solid black;
  font: 13px 'Courier New';
  width : 300px;
  height : 30px;
  padding-top : 10px;
  padding-left : 10px;
}

.contentDark{
  background : #666;
  color : white;
  border : 1px solid black;
  font : 13px 'Courier New';
  width : 300px;
  height : 30px;
  padding-top : 10px;
  padding-left : 10px;
}
```

CSS 文件定义了类(`contentGray`)的样式及另外一个类(`contentDark`)的样式, 这样 HTML 文件就可以看起来比较漂亮:



我们的需求是, 当我们点击这个 `panel` 的时候, 它的背景色加深, 文字变成白色, 当再次点击的时候又恢复之前的颜色和背景色。这种“动态”的就交给 JavaScript 来处理了:

```
$(document).ready(function() {
  $("#div#content").click(function() {
    $(this).toggleClass("contentDark");
  })
});
```

使用 jQuery 可以为我们带来很多便利, 我们选择 id 为 `content` 的 `div`, 然后注册事件处

理函数，当 click 事件发生的时候，我们就切换 CSS 类 contentDark:

```
This is the content of a panel
```

当然，这个只是一个简单的例子，如果你的应用所涉及的页面比较多，而且控制器部分 (JavaScript 脚本)的工作量比较大，那么你会发现，客户端的 MVC 模型对你有很大的帮助，不但是界面的统一风格，而且 JavaScript 代码会更加模块化，从而可能一定程度上提高应用程序的效率，同时降低维护的难度。

11.7 Javascript/Ajax 框架

随着富客户端的流行，基于 WEB 的应用越来越多，人们在开发过程中不再满足于 DOM 提供的简单 API，特别是 DOM 对页面的操作比较繁琐，而且容易出错。当页面越来越华丽，页面 UI 越来越复杂(事件处理，特效处理)的时候，就有大量第三方 JavaScript 框架被开发出来了，比如较早的 prototype, dojo, 以及 yahoo 的 YUI, 后来的 jQuery, 以及 jQuery 的 UI 插件 jQuery-UI, 最早基于 YUI 而后来又进行了重构的 ExtJs, 号称纯 OO 的 Mootools 等等。

这些 JavaScript 框架的开发，大大的简化了页面的开发速度，也提高了开发效率，同时比较注重用户体验，这里列举出的框架几乎都是完全免费，所以应用十分广泛。我们在随后的两章中将列举两个最流行的框架做一些介绍，以期读者可以有一些感性的认识，关于 jQuery 和 ExtJS 的深入的研究已经大大的超出了本书的范围(事实上每一个框架都足以写一本书)，有兴趣的读者可以参考相关的书籍。

第十二章 前端 JavaScript 框架:jQuery

12.1 jQuery 简介

jQuery 是目前应用最为广泛,最为优秀的 Ajax/JavaScript 开源框架之一,有数以千万记的用户,更有多不胜数的技术文档与之相关,在一定程度上, jQuery 如其所宣扬的那样,改变了人们编写 JavaScript 的方式。

jQuery 通过提供 CSS 标准的选择器来对页面元素进行选择,然后对这些元素组成的一个列表进行某些操作,参与过页面开发的人员都知道,基于 WEB 的 UI 实际上要做的事情就是:

- 找到页面上的某个/某些元素
- 改变这些元素的属性或者样式
- 绑定一些事件处理程序在这些元素上

为了对开发者更友好, jQuery 使用了独特的链式操作,使得可以使用尽可能的代码来完成尽可能多的任务。就个人而言, jQuery 是我个人最喜欢的 JavaScript 框架。我们可以通过一些例子来看看 jQuery 是怎样工作的。

假设我们有一个 table,如果给 table 加上斑马线的话,用户可以更清晰的看清楚每一行,整个页面也更有层次感,但是问题是我们的页面已经做好了,我不想再对页面做修改!

City	Product	Sales
Birmingham	ProductC	23451
Landon	ProductB	32445
Manchester	ProductC	87652
Landon	ProductB	32445
Manchester	ProductC	87652
Landon	ProductB	32445
Manchester	ProductC	87652
Landon	ProductB	32445
Manchester	ProductC	87652

图 修改前的表格

这样单调的一种颜色很容易使用户的视觉产生疲劳,我们应该为偶数行添加浅绿色的背景,像这样:

City	Product	Sales
Birmingham	ProductC	23451
Landon	ProductB	32445
Manchester	ProductC	87652
Landon	ProductB	32445
Manchester	ProductC	87652
Landon	ProductB	32445
Manchester	ProductC	87652
Landon	ProductB	32445
Manchester	ProductC	87652

图 修改后的表格

这样效果就好多了，要做成这种效果，用 jQuery 需要多少代码呢？一行！我们来看看如何用一行代码完成这样的效果：首先，我们找到这个 `table`，然后告诉 jQuery，给这个 `table` 中的所有偶数行都添加一个 `css` 伪类：

```
$("#div#informationTable table tr:nth-child(even)")
.addClass("striped");
```

`"div#informationTable table tr:nth-child(even)"` 为一个 CSS 标准的选择器，表示在一个 `id` 为 `informationTable` 的 `div` 的孩子中，找到所有的 `table`，`tr` 是 `table` 的孩子，并使行，`nth-child(even)` 表示为偶数的孩子。

完整的代码如下：

```
$(document).ready(function() {
    $("#div#informationTable table tr:nth-child(even)")
        .addClass("striped");
});
```

这段代码表示，当文档加载完成后 (`$(document).ready()`)，调用一个匿名的函数，这个函数的函数体为我们上面分析过的，找到 `table` 的所有偶数列，为这些列添加背景色 (通过使用 `css` 类 `"striped"`)。

事实上，jQuery 深受开发人员青睐的更深层次的原因可能要归功于贯穿于其中的编程思想，如果仔细审视 jQuery 的代码，你应该会发现，其中的集合的概念以及对集合的各种操作，与函数式语言 `lisp` 是不谋而合的，比如 `map`, `filter`, 以及 `grep` 等等。

我们可以来看看这样几个简单示例：

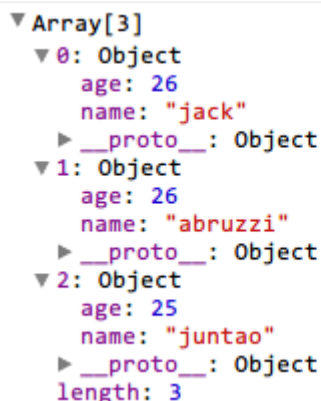
`jQuery.grep` 对列表进行过滤，并返回过滤后的列表。我们来看一个例子，首先定义一个人员列表，每个条目包含 `name` 和 `age` 字段，现在要找出所有 `age` 大于 24 岁的人员，并以列表的形式返回：

```
$(document).ready(function(){
    var person = [
        {name : "jack", age : 26},
        {name : "johb", age : 23},
        {name : "smith", age : 20},
        {name : "abruzzo", age : 26},
        {name : "juntao", age : 25},
        {name : "jim", age : 24},
        {name : "bob", age : 24}
    ];

    var gt23s = $.grep(person, function(item){
        return item.age > 24;
    });

    console.dir(gt23s);
});
```

jQuery 的工具方法定义在 jQuery 对象上，类似于 Java 中的静态方法。console.dir 是 Firefox 或者 Chrome 的调试助手，在 chrome 下的结果如下：



```
▼ Array[3]
  ▼ 0: Object
    age: 26
    name: "jack"
    ▶ __proto__: Object
  ▼ 1: Object
    age: 26
    name: "abruzzo"
    ▶ __proto__: Object
  ▼ 2: Object
    age: 25
    name: "juntao"
    ▶ __proto__: Object
  length: 3
```

图 chrome 开发工具中 console.dir 的效果示意(\$.grep)

再来看一下 map 的例子：

```
$(document).ready(function(){
    var person = [
        {name : "jack", age : 26},
        {name : "johb", age : 23},
        {name : "smith", age : 20},
        {name : "abruzzo", age : 26},
        {name : "juntao", age : 25},
        {name : "jim", age : 24},
        {name : "bob", age : 24}
    ];
```

```

];

var mapped = $.map(person, function(item) {
    return item.name = item.name.toUpperCase();
});

console.dir(person); //原始的person列表已被修改
});

```

我们将 `person` 列表中的每一个元素的 `name` 字段的值转换为大写。Map 的修改是直接体现在原始列表上的，结果如下：

```

▼ Array[7]
  ▼ 0: Object
    age: 26
    name: "JACK"
    ▶ __proto__: Object
  ▼ 1: Object
    age: 23
    name: "JOHN"
    ▶ __proto__: Object
  ▼ 2: Object
    age: 20
    name: "SMITH"
    ▶ __proto__: Object
  ▶ 3: Object
  ▶ 4: Object
  ▶ 5: Object
  ▶ 6: Object
  length: 7

```

图 chrome 开发人员工具中 `console.dir` 的效果示意(`$.map`)

jQuery 本身不如 ExtJS 那样可以轻松而快速的开发出华丽的 UI，但是 jQuery 本身提供的插件机制为使用 jQuery 方式快速开发华丽的 UI 提供了可能，比如 jQuery-UI, EasyUI 等插件的出现，使得用户可以向使用 jQuery 那样，快速的生成 UI，提高开发速度。

12.2 jQuery 基础

12.2.1 jQuery 选择器

jQuery 最强大易用的即是它提供的选择器，它支持 CSS 选择器及其扩展，很方便已经熟悉传统 web 开发模式的用户快速过渡到 jQuery 上来。比如下面这些常用的 CSS 选择器：

- `div.highlight` 选择 CSS 类 `highlight` 的所有 `div` 元素
- `table#tabid` 选择 ID 为 `tabid` 的 `table` 元素
- `a#aid.aclass` 选择 ID 为 `aid`，CSS 类为 `aclass` 的链接元素

这些选择器均可直接在 jQuery 中使用，只需要将选择器包装在 `$()` 中即可。jQuery 的选择器完全兼容 CSS3 选择器。这为跨浏览器的 web 应用提供了极大的便利。

除了这些基本的 CSS 选择器外，jQuery 提供了更丰富的选择器，如通过位置选择：

选择器	作用
<code>:first</code>	选择第一个匹配
<code>:last</code>	选择最后一个匹配
<code>:first-child</code>	选择第一个子元素
<code>:last-child</code>	选择最后一个子元素
<code>:nth-child(n)</code>	选择第 n 个子元素
<code>:even</code> 及 <code>:odd</code>	选择偶数/奇数子元素集
<code>:eq(n)</code>	第 n 个元素(从 0 开始)
<code>:gt(n)</code>	选择第 n 个元素之后的元素集
<code>:lt(n)</code>	选择第 n 个元素之前的元素集

比如：

- `li a:first` 匹配在列表(``)元素下的第一个链接(`<a>`)元素。
- `table tbody td:nth-child(5)` 返回 `table` 的第 6 列元素集。

自定义选择器：

选择器	作用
<code>:button</code>	选择按钮
<code>:checkbox</code>	选择复选框
<code>:checked</code>	选择已经选中的复选框/单选框
<code>:hidden</code>	选择属性为隐藏的元素
<code>:enable</code>	选择启用的元素
<code>:disable</code>	选择禁用的元素
<code>:image</code>	选择图片
<code>:input</code>	选择输入框
<code>:radio</code>	选择单选框
<code>:not(filter)</code>	反向选择器

应该注意的是，这些选择器可以组合使用，这样会给我们提供极大的方便，比如：

- `:input:enable` 选择已经启用的文本框元素
- `:checkbox:checked` 选择已经选择的复选框元素

12.2.2 对 DOM 的操作

在实际应用中，经常需要操作 DOM 元素，比如插入一段 HTML 到指定位置，删除某些被选择的 DOM 段，修改某些元素的内容等。

例如有一个 HTML 页面:

```
<html>
  <head>
    <link rel="stylesheet" href="style.css" type="text/css" />
    <script type="text/javascript" src="jquery-1.3.2.js"></script>
    <script type="text/javascript" src="selector.js"></script>
  </head>
  <body>
    <div id="container"></div>
  </body>
</html>
```

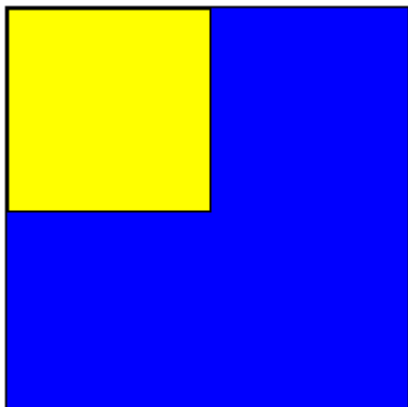
样式表为:

```
div#container{
  background:blue;
  border:1px solid black;
  width:200px;
  height:200px;
}
```

```
div#child{
  background:yellow;
  border:1px solid black;
  width:100px;
  height:100px;
}
```

container 是一个蓝色的 200x200 的方框, 我们要动态的为这个 div 添加一个子元素, 子元素的 ID 为 child:

```
$(function(){
  var container = $("#div#container");
  $("#<div id='child'></div>").appendTo(container);
});
```

运行结果如上图所示。

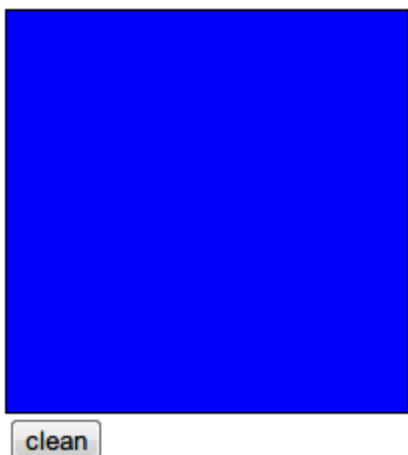
再进一步，我们为页面添加一个按钮(clean)，点击此按钮将清除新添加的黄色 child 方框。

```
<input type="button" id="clean" value="clean" />
```

并添加 JavaScript 代码:

```
$(function() {  
    var container = $("#div#container");  
    $("#<div id='child'></div>").appendTo(container);  
  
    $("#input#clean").click(function() {  
        container.find("#child").remove();  
    });  
});
```

单击 clean 按钮之后，将会移除新添加的 child 框。



12.2.3 对 CSS 的操作

使用 jQuery，可以很方便的对 CSS 类进行添加/删除/toggle 等操作，我们来看一个简单的示例：

首先定义三个 CSS 类：base, red-region, yellow-region:

```
.base{
  background:white;
  border:1px solid black;
  width:200px;
  height:200px;
}
```

```
.red-region{
  background:red;
  border:1px solid blue;
  width:200px;
  height:200px;
}
```

```
.yellow-region{
  background:yellow;
  border:1px solid green;
  width:200px;
  height:200px;
}
```

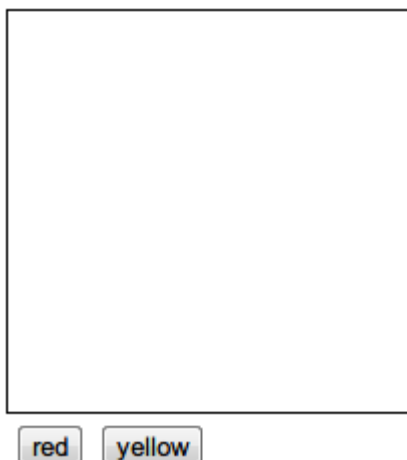
定义一个 ID 为 base 的面板，两个按钮：red 和 yellow。当点击 red 时，判断 base 是否已经被 yellow 修饰过，如果已经被修饰过，则移除 CSS 类 yellow-region。点击 yellow 时情形类似：

```
var base = $("#div#base");
$("#input#red").click(function() {
  if(base.hasClass("yellow-region")) {
    base.removeClass("yellow-region");
  }
  $("#div#base").addClass("red-region");
});

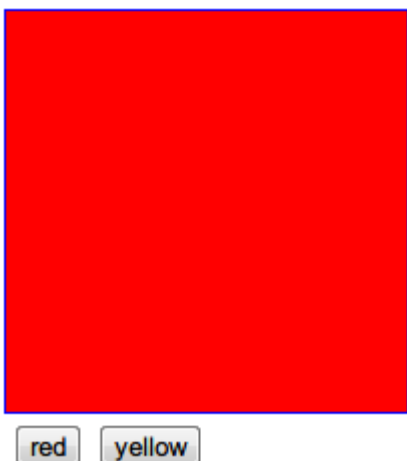
$("#input#yellow").click(function() {
  if(base.hasClass("red-region")) {
    base.removeClass("red-region");
  }
});
```

```
$("#div#base").addClass("yellow-region");
});
```

页面效果如下：



点击 red 按钮之后，base 添加了 red-region 的 CSS 类，变为红色：



在使用 jQuery 选择器选择到预期的元素集之后，我们可以修改器 CSS，来完成页面的动态化。动态修改 CSS 非常简单：

```
var base = $("#div#base");
base.css('width', '300px');
base.css({
    'width' : '300px',
    'height' : '300px',
    'background' : 'green'
});
```

使用 css 函数，可以进行一个值的修改，同样可以传入一个集合，整体进行修改。

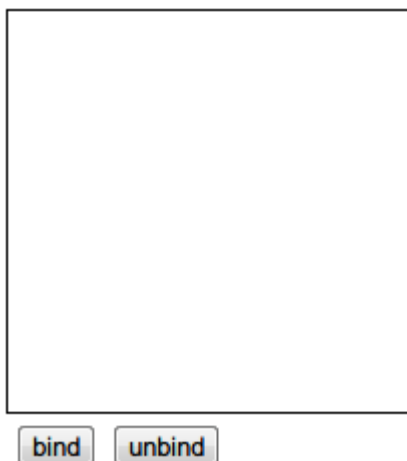
12.2.4 事件处理

事实上，在上边的例子中很多地方已经涉及到 jQuery 事件处理部分了。jQuery 不但提供基本的 `bind/unbind` 来负责注册及删除事件处理器，同时还提供很多更方便 web 开发的助手函数，如 `toggle/hover` 等。

注册一个事件处理器非常容易：

```
var base = $("#div#base");
base.bind('click', function(event) {
    alert($("#this").width()+"", "+$("#this").height());
});
```

当鼠标单击 ID 为 `base` 的 `div` 时，触发该事件。使用 `unbind` 将事件处理器删除。我们来看一个小例子：

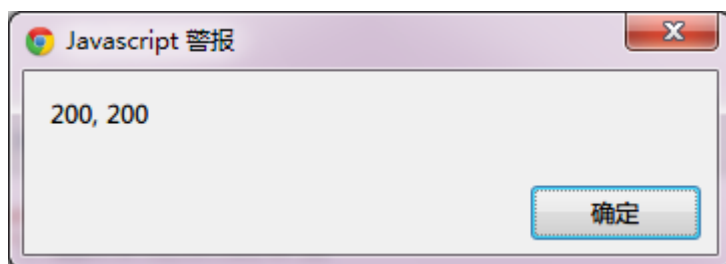


单击 `bind` 按钮时，我们为按钮上方的 `div` 注册 `click` 事件处理器，单击 `unbind` 时，移除该事件处理器：

```
$("#input#bind").click(function() {
    base.bind('click', function(event) {
        alert($("#this").width()+"", "+$("#this").height());
    });
});

$("#input#unbind").click(function() {
    base.unbind('click');
});
```

这样，单击 `bind` 之后，单击 `div` 则会弹出一个对话框：



点击 `unbind` 之后，`div` 将不会再处理 `click` 事件。有时候，我们会需要为单击的次数为奇数和偶数时注册不同的事件处理器，如第一次单击时将 `panel` 的背景色变为红色，再次单击则将背景色变为黄色，这时候我们可以使用 `toggle` 函数：

```
var base = $("#div#base");

base.toggle(
  function () {
    $(this).css('background', 'red');
  },
  function () {
    $(this).css('background', 'yellow');
  }
);
```

当然，更多的是处理鼠标移入/移出事件的 `hover`，当用户在页面上移动鼠标，将展现不同的视觉效果：

```
base.hover(
  function (event) {
    $(this).css('background', 'red');
  },
  function (event) {
    $(this).css('background', 'yellow');
  }
);
```

12.2.5 实用函数

jQuery 除了提供对 DOM 操作的 API 之外，还提供了操作普通 JavaScript 对象的一些函数，这些函数均已“`$.`”开头，非常方便易用。这些实用函数包括：对字符串操作的函数，遍历对象的函数，过滤数组中元素等。

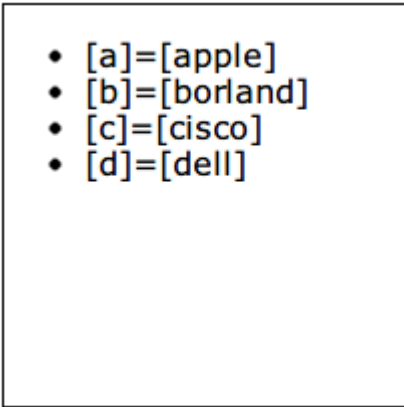
我们来看一些小例子：

```
var obj = {
  a : 'apple',
  b : 'borland',
  c : 'cisco',
  d : 'dell'
};

$.each(obj, function(name, value){
  var li = $("- </li>");
  li.html([""+name+"=["+value+"]");
  li.appendTo(base);
});

```

遍历对象 `obj`,然后将其中的键值对拼装成一个字符串, 添加到一个 `panel` 上:



- [a]=[apple]
- [b]=[borland]
- [c]=[cisco]
- [d]=[dell]

`$.grep/$.map` 两个实用函数已经在第一小节做过基本的讲解, 这里仅列举出两个函数的原型:

```
/**
 * array    : 要过滤的数组对象
 * callback : 过滤条件
 * invert   : 是否启用反转, 如果启用, 则符合callback的将被过滤
 */
$.grep(array, callback, invert);

/**
 * array    : 需要做转换的数组对象
 * callback : 对数组中元素的映射函数
 */
$.map(array, callback);
```

有时候, 我们可能需要合并数个对象为一个对象, 覆盖其中重复的项等:

```
var obj1 = {
```

```
    name : 'juntao',
    last : 'qiu',
  };

  var obj2 = {
    addr : 'unknown',
    title : 'unknown'
  };

  var obj3 = {
    addr : 'KunMing, Yunnan, China'
  };

  result = $.extend({}, obj1, obj2, obj3);

  $.each(result, function(name, value){
    var li = $("<li></li>");
    li.html("[" + name + "] = [" + value + "]");
    li.appendTo(base);
  });
```

obj1, obj2, obj3 的属性被合并在一起，并且 obj3 中的 addr 属性覆盖了 obj2 中的 addr 属性。

-
- [name]=[juntao]
 - [last]=[qiu]
 - [addr]=[KunMing, Yunnan, China]
 - [title]=[unknown]

总而言之，jQuery 是一个小巧，实用，易用且功能强大的框架。使用它，可以将原本复杂难懂的 JavaScript 代码压缩至很小，而且更容易维护，代码更加优美。jQuery 可以称得上是 web 上的 lisp。

12.3 jQuery 实例

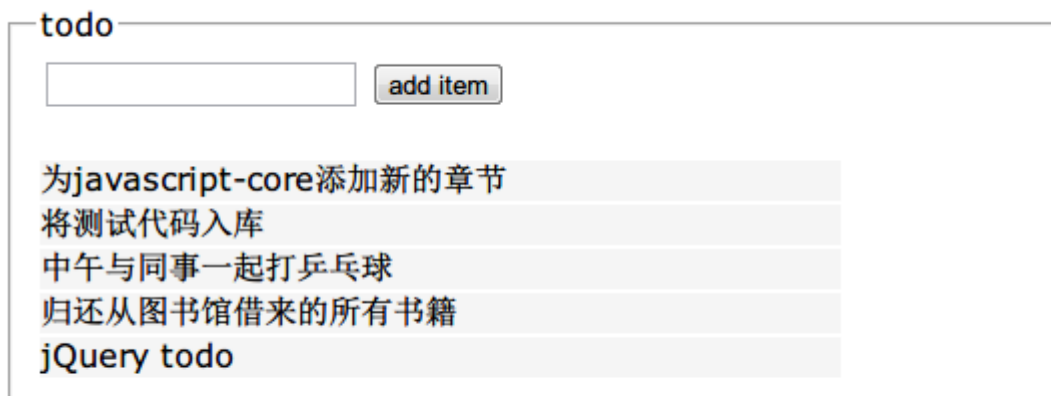
在这一小节，我们将使用 jQuery 开发一个简单的 todo 管理器 jqtodo。jqtodo 使用

httpd+php 脚本作为后台,数据库使用小巧的 `sqlite`。`jqtodo` 简单到仅支持新建一个 `todo` 及对之前所有 `todo` 的查询操作。

页面布局上,有一个输入框和一个按钮,用户在输入框中填写待办事项,然后点击按钮,即可将这条待办事项添加到数据库中,并同时 will 页面的待办事项列表更新:



简单起见,这里没有对用户的输入做任何校验,如果插入成功,则展示在列表中:



5:Fri Feb 03 2012 06:08:34 GMT+0800 (中国标准时间)

我们需要用 `jQuery` 做的事情如下:

- 当点击 `add item` 时,将文本框中的内容取出,并发送给服务器
- 当服务器完成并响应时,我们需要及时的更新列表
- 当用户首次进入此页面时,需要将历史中的待办事项列出来

在页面上定义一个 `div`,其 `id` 为 `itemlist`,则当进入页面时,可以通过 `jQuery.ajax` 来获取数据库信息,并填充页面:

```
var list = $("div#itemlist");

$.ajax({
  url : 'queryitems.php',
  type : 'GET',
  error : function(xhr) {
    alert(xhr);
  },
  success : function(obj) {
    obj = eval('(' + obj + ')');
    var dataset = obj.dataset;
    for(var i = 0; i < dataset.length; i++){
      var current = dataset[i];
      var newItem = $("<div></div>").text(current.desc)
```



```

        .attr({
            "id" : current.itemid,
            "time" : current.ctime
        })
        .addClass("item");

        newitem.appendTo(list);
    }
}
});

```

后台提供一个 `queryitems.php` 的页面，该页面负责查询数据库，并将结构及作为 `json` 数组的格式返回，并将数据集存放在 `"dataset"` 属性中，当成功时，我们可以遍历这个数组，并动态的创建条目，为条目添加属性及 `CSS` 类，最后将其添加到 `id` 为 `itemlist` 的 `div` 上展现。

类似的，页面上有一个 `id` 为 `add` 的按钮，单击该按钮将触发以下事件：

```

$("input#add").click(function(){
    var item = $("input#item").val();
    if(!item || item.length == 0){
        alert("please set the item description");
        return false;
    }else{
        additem(item);
    }
});

```

首先获取文本框中的字符串，并做一下简单的非空校验。通过校验后则调用 `additem` 函数进行查询及页面更新：

```

function additem(item){
    var dat = "item="+item;

    $.ajax({
        url : 'additem.php',
        type : 'POST',
        dataType : 'text',
        data : dat,
        error : function(xhr){
            alert(xhr);
        },
        success : function(obj){
            obj = eval('(' +obj+')');
            var newitem = $("<div></div>").text(obj.desc)

```

```
.attr({
    "time" : obj.ctime
})
.addClass("item");

newitem.appendTo(list);
}
});
}
```

在这个函数中，只是简单的组织了需要 POST 的数据，然后使用 `jQuery.ajax` 异步的更新页面中的待办事项列表。

使用 `jQuery`，可以在很短小的代码量中完成很多工作，一般而言，简洁的代码更容易维护和扩展。哪怕仅仅只是从代码的可读性和美学的意义上来讲，`jQuery` 也非常值得一试。

第十三章 JavaScript 引擎

详细讨论一个脚本引擎的实现，已经大大超出了本书的范围，如果读者需要深入这方面，可以参考其他书籍。本章仅仅讨论 SpiderMonkey 引擎的部分细节，对 v8 及 rhino 引擎只是简单的做些分析。在本章的最后，会有一些相应的示例。

13.1 使用 SpiderMonkey

此文最早在 IBM Developerworks 中国发表，地址为：[基于 C 语言的 JavaScript 引擎](#)。

13.1.1 SpiderMonkey 简介

和其他的 JavaScript 引擎一样，SpiderMonkey 不直接提供像 DOM 这样的对象，而是提供解析，执行 JavaScript 代码，垃圾回收等机制。SpiderMonkey 是一个在 Mozilla 之下的开源项目，要使用 SpiderMonkey，需要下载其源码，然后编译为静态/动态库使用。

要在自己的应用程序中使用 SpiderMonkey，首先需要了解以下三个核心概念：

运行时环境 运行时环境是所有 JavaScript 变量，对象，脚本以及代码的上下文所存在的空间。每一个上下文对象，以及所有的对象均存在于此。一般应用仅需要一个运行时即可。

上下文 上下文即脚本执行的环境，在 SpiderMonkey 中，上下文可以编译执行脚本，可以存取对象的属性，调用 JavaScript 的函数，转换类型，创建/维护对象等。几乎所有的 SpiderMonkey 函数都需要上下文作为其第一个参数(JSContext *)。

上下文与线程密不可分，一般来讲，单线程应用可以使用一个上下文来完成所有的操作，每一个上下文每次只能完成一个操作，所有在多线程应用中，同一时刻只能有一个线程来使用上下文对象。一般而言，多线程应用中，每个线程对应一个上下文。

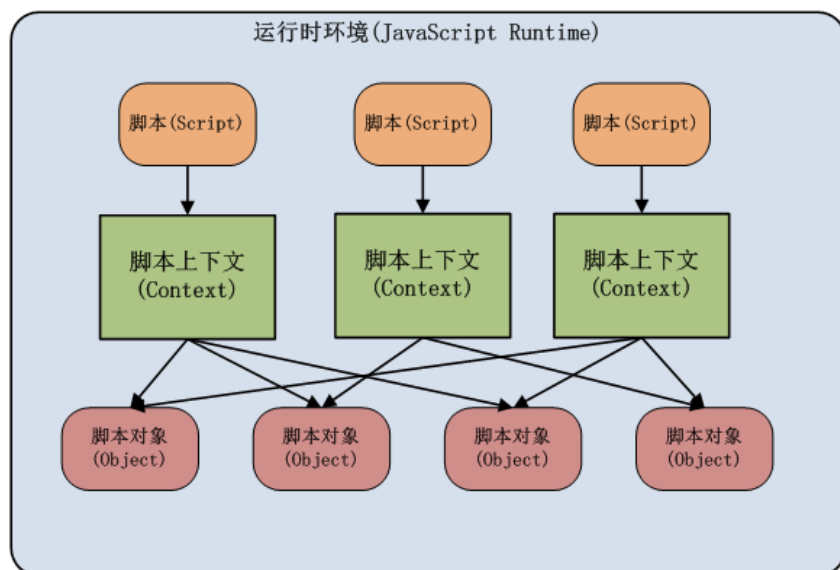
全局对象 全局对象包含 JavaScript 代码所用到的所有类，函数，变量。在 DOM 操作中，我们使用的：

```
alter("something");
```

事实上使用的是全局变量 window 的一个属性 alter(这个属性正好是一个函数)，事实上上边的语句在执行时会别解释为：

```
window.alter("something");
```

三者的关系如下图所示：



13.1.2 JavaScript 对象与 C 对象间的转换关系

JavaScript 是一门弱类型的语言，变量的值的类型在运行时才确定，而且可以在运行时被修改为其他类型的变量；而 C 语言，是一门静态类型的语言，变量类型在编译时就已经确定。因此，这两者之间变量的互访就有了一定的难度，SpiderMonkey 提供了一个通用的数据类型 `jsval` 来完成两者之间的交互。

事实上，在 C 代码中定义的 `jsval` 类型的变量可以是 JavaScript 中的字符串，数字，对象，布尔值，以及 `null` 或者 `undefined`。基于这个类型，SpiderMonkey 提供了大量的类型判断及类型转换的宏和函数。可以参看下表：

JavaScript t 类型	jsval 类型判断	jsval 常量	jsval 转化
<code>null</code>	<code>JSVAL_IS_NULL(v)</code>	<code>JSVAL_NULL</code>	
<code>Undefined</code>	<code>JSVAL_IS_VOID(v)</code>	<code>JSVAL_VOID</code>	
<code>Boolean</code>	<code>JSVAL_IS_BOOLEAN(v)</code>	<code>JSVAL_TRUE,</code> <code>JSVAL_FALSE,</code> <code>BOOLEAN_TO_JSVAL(b)</code>	<code>JSVAL_TO_BOOLEAN(v)</code>
<code>number</code>	<code>JSVAL_IS_NUMBER(v),</code> <code>JSVAL_IS_INT(v),</code> <code>JSVAL_IS_DOUBLE(v)</code>	<code>INT_TO_JSVAL(i),</code> <code>DOUBLE_TO_JSVAL(d)</code>	<code>JSVAL_TO_INT(v),</code> <code>JSVAL_TO_DOUBLE(v)</code>
<code>string</code>	<code>JSVAL_IS_STRING(v)</code>	<code>STRING_TO_JSVAL(s)</code>	<code>JSVAL_TO_STRING(v),</code> <code>JS_GetStringChars(s)</code>

), JS_GetStringLength(s)
object	JSVAL_IS_OBJECT(v)) && JSVAL_IS_NULL(v)	OBJECT_TO_JSVAL(o))	JSVAL_TO_OBJECT(v)

应该注意的是，`jsval` 有一定的缺陷：

- `jsval` 并非完全的类型安全，在进行类型转换之前，你需要明确被转换的对象的真正类型，比如一个变量的值为 `number` 类型，而对其做向字符串的转化，则可能引起程序崩溃。解决方法是，在转换之前，先做判断。
- `jsval` 是 `SpiderMonkey` 垃圾回收机制的主要目标，如果 `jsval` 引用一个 `JavaScript` 对象，但是垃圾收集器无法得知这一点，一旦该对象被释放，`jsval` 就会引用到一个悬空指针。这样很容易使得程序崩溃。解决方法是，在引用了 `JavaScript` 对象之后，需要显式的告知垃圾收集器，不引用时，再次通知垃圾收集器。
-

13.1.3 基本代码模板

13.1.3.1 基本流程

使用 `SpiderMonkey`，一般来讲会使用以下流程：

- 创建运行时环境
- 创建一个/多个上下文对象
- 初始化全局对象
- 执行脚本，处理结果
- 释放引擎资源

在下一小节详细说明每个流程

13.1.3.2 代码模板

使用 `SpiderMonkey`，有部分代码是几乎每个应用程序都会使用的，比如错误报告，初始化运行时环境，上下文，全局变量，实例化全局变量等操作。这里是一个典型的模板：

```
#include "jsapi.h"
```

引入 `jsapi.h`，声明引擎中的所用到的记号，结构体，函数签名等，这是使用 `SpiderMonkey`

所需的唯一一个接口文件(当然, `jsapi.h` 中不可能定义所有的接口, 这些文件在 `jsapi.h` 头部引入 `jsapi.h`, 如果对 C 语言的接口, 头文件引入方式不熟悉的读者, 请参阅相关资料)。

```
/* 全局变量的类声明 */
static JSClass global_class = {
    "global",
    JSCCLASS_GLOBAL_FLAGS,
    JS_PropertyStub,
    JS_PropertyStub,
    JS_PropertyStub,
    JS_PropertyStub,
    JS_EnumerateStub,
    JS_ResolveStub,
    JS_ConvertStub,
    JS_FinalizeStub,
    JSCCLASS_NO_OPTIONAL_MEMBERS
};
```

`JSClass` 是一个较为重要的数据结构, 定义了 JavaScript 对象的基本结构---“类”, 这个类可以通过 SpiderMonkey 引擎来实例化为对象。`JS_PropertyStub` 是 `JS_PropertyOp` 类型的变量, 这里的 `JS_PropertyStub` 是为了提供一个默认值。`JS_PropertyOp` 可以用做对象的 `setter/getter` 等的, 这些内容我们将在后边的章节详细讨论。

```
/* 错误处理函数, 用于回调, 打印详细信息 */
void report_error(JSContext *cx, const char *message, JSErrorReport
*report) {
    fprintf(stderr, "%s:%u:%s\n",
        report->filename ? report->filename : "<no filename>",
        (unsigned int) report->lineno,
        message);
}
```

定义好这些结构之后, 我们需要实例化这些结构, 使之成为内存对象, 流程如下:

```
int main(int argc, char *argv[]) {
    JSRuntime *runtime;
    JSContext *context;
    JSObject *global;

    //创建新的运行时8M
    runtime = JS_NewRuntime(8L * 1024L * 1024L);
    if(runtime == NULL) {
        return -1;
    }
}
```

```
//创建新的上下文
context = JS_NewContext(runtime, 8*1024);
if(context == NULL){
    return -1;
}

//
JS_SetOptions(context, JSOPTION_VAROBJFIX);
//设置错误回调函数, report_error函数定义如上
JS_SetErrorReporter(context, report_error);

//创建一个新的JavaScript对象
global = JS_NewObject(context, &global_class, NULL, NULL);
if (global == NULL){
    return -1;
}

//实例化global,加入对象,数组等支持
if (!JS_InitStandardClasses(context, global)){
    return -1;
}

//
//使用global, context等来完成其他操作,用户定制代码由此开始
//

//释放上下文对象
JS_DestroyContext(context);
//释放运行时环境
JS_DestroyRuntime(runtime);
//停止JS虚拟机
JS_ShutDown();

return 0;
}
```

用户自己的代码从上边代码中部注释部分开始,用户代码可以使用此处的 `context` 对象及预设过一定属性,方法的 `global` 对象。

13.1.4 执行 JavaScript 代码

13.1.4.1 执行 JavaScript 代码片段

执行 JS 最简单的方式，是将脚本作为字符串交给引擎来解释执行，执行完成之后释放临时的脚本对象等。SpiderMonkey 提供一个 `JS_EvaluateScript` 函数，原型如下：

```
JSBool JS_EvaluateScript(JSContext *cx, JSObject *obj,
    const char *src, uintN length, const char *filename,
    uintN lineno, jsval *rval);
```

使用这个函数，需要提供上下文，全局变量，字符串形式的脚本，脚本长度及返回值指针，脚本名和行号参数可以填空值(分别为 `NULL` 和 `0`)。如果函数返回 `JS_TRUE`，表示执行成功，执行结果存放在 `rval` 参数中，否则执行失败，`rval` 中的值为 `undefined`。我们可以具体来看一个例子：

```
char *script = "(function(a, b){return a * b;})(15, 6)";
jsval rval;

status = JS_EvaluateScript(context, global, script, strlen(script)\
    , NULL, 0, &rval);

if(status == JS_TRUE){
    jsdouble d;
    JS_ValueToNumber(context, rval, &d);
    printf("eval result = %f\n", d);
}
```

执行结果为：

```
eval result = 90.000000
```

13.1.4.2 编译 JavaScript 代码

通常，我们可能会多次执行一段脚本，SpiderMonkey 可以将脚本编译成 `JSScript` 对象，然后可以供后续的多次调用。现在来看一个例子，使用 C 代码编译一个 JavaScript 脚本，然后运行这个脚本。

```
JSBool evalScriptFromFile(JSContext *ccontext, const char *file){
    JSScript *script;
    JSString *jss;
```



```

JSBool status;
jsval value;

//get the global object
JSObject *global = JS_GetGlobalObject(context);

//compile the script for further using
script = JS_CompilFile(context, global, file);

if(script == NULL){
    return JS_FALSE;
}

//execute it once
status = JS_ExecuteScript(context, global, script, &value);
jss = JS_ValueToString(context, value);
printf("eval script result is : %s\n", JS_GetStringBytes(jss));

//destory the script object
JS_DestroyScript(context, script);

return status;
}

```

这里传递给函数 **evalScriptFromFile** 的 JSContext* 参数为外部创建好的 Context 对象，创建的方法参看上一节。

```

JSBool status = evalScriptFromFile(context, "jstest.js");
if(status == JS_FALSE){
    fprintf(stderr, "error while evaluate the script\n");
}

```

假设我们将如下脚本内容保存进一个脚本 **jstest.js**:

```

var Person = function(name){
    var _name_ = name;
    this.getName = function(){
        return _name_;
    }

    this.setName = function(newname){
        _name_ = newname;
    }
}

```

```
var jack = new Person("jack");
jack.setName("john");
//最后一句将作为脚本的执行结果返回给C代码
jack.getName();
```

jack 对象的名字现在设置为了“john”，脚本的最后一条语句的值将作为脚本的返回值返回到 C 代码处，并打印出来：

```
eval script result is : john
```

13.1.5 C 程序调用 JavaScript 函数

由于两者的数据类型上有较大的差异，因此无法直接从 C 代码中调用 JavaScript 代码，需要通过一定的转化，将 C 的变量转换为 JavaScript 可以识别的变量类型，然后进行参数的传递，返回值的处理也同样要经过转换。

我们在 JavaScript 中定义一个函数 add，这个函数接受两个参数然后返回传入的两个参数的和。定义如下：

```
function add(x, y){
    return x + y;
}
```

然后，我们在 C 语言中根据名称调用这个 JS 函数：

```
JSBool func_test(JSContext *context){
    jsval res;
    JSObject *global = JS_GetGlobalObject(context);
    jsval argv[2];

    //new 2 number to pass into the function "add" in script
    JS_NewNumberValue(context, 18.5, &res);
    argv[0] = res;
    JS_NewNumberValue(context, 23.1, &res);
    argv[1] = res;

    JS_CallFunctionName(context, global, "add", 2, argv, &res);

    jsdouble d;

    //convert the result to jsdouble
    JS_ValueToNumber(context, res, &d);
```

```
printf("add result = %f\n", d);

return JS_TRUE;
}
```

这里需要注意的是，JS_CallFunctionName 函数的参数列表：

```
JSType JS_CallFunctionName(JSContext *cx, JSObject *obj,
    const char *name, uintN argc, jsval *argv, jsval *rval);
```

名称	类型	类型描述
cx	JSContext *	上下文定义
obj	JSObject *	调用该方法的对象
name	const char *	函数名
argc	uintN	函数参数个数
argv	jsval *	函数实际参数形成的数组
rval	jsval *	返回值

参数中的 argv 是一个 jsval 形成的数组，如果直接传递 C 类型的值，则很容易出现 core dump(Linux 下的段错误所导致)，因此，需要 JS_NewNumberValue 函数转换 C 语言的 double 到 number(原因见对象转换小节)。

13.1.6 JavaScript 程序调用 C 函数

从 JS 中调用 C 函数较上一节为复杂，我们来看一个较为有趣的例子：SpiderMonkey 中原生的 JavaScript 的全局变量中没有 print 函数，我们可以使用 C 的 printf 来实现这个功能。我们定义了一个函数 print，print 使用 logging 函数，而 logging 函数是定义在 C 语言中的，接受一个字符串作为参数，打印这个字符串到标准输出上：

```
//log user log in information
logging("user jack login on 2010/7/6");

//user do nothing else
nothing();

//log user log out information
logging("user jack logout on 2010/7/7");

function print() {
    for(var i = 0; i < arguments.length; i++){
        logging(arguments[i]);
    }
}
```

```
print("hello", "all", "my", "friend");
```

在 C 语言中，我们定义 `logging` 函数和 `nothing` 函数的原型如下：

```
/**
 * define an exposed function to be used in scripts
 * print out all the incoming arguments as string.
 */
static JSBool logging(JSContext *context, JSObject *object, uintN argc,
    jsval *argv, jsval *value){
    int i = 0;
    JSString *jss;

    for(i = 0; i < argc; i++){
        jss = JS_ValueToString(context, argv[i]);
        printf("message from script environment : %s\n", \
            JS_GetStringBytes(jss));
    }
    return JS_TRUE;
}

/**
 * define an exposed function to be used in scripts
 * do nothing but print out a single line.
 */
static JSBool nothing(JSContext *context, JSObject *object, uintN argc,
    jsval *argv, jsval *value){
    printf("got nothing to do at all\n");
    return JS_TRUE;
}
```

从函数的签名上可以看出，C 中暴露给 JS 使用的函数，参数的个数，及对应位置上的类型，返回值都是固定的。所有的从 C 中暴露给 JS 的函数都需要“实现这个接口”。

定义好了函数之后，还需要一些设置才能在 JS 中使用这些函数。首先定义一个 `JSFunctionSpec` 类型的数组，然后通过 `JS_DefineFunctions` 将这些函数放到 `global` 对象上，然后在 JS 代码中就可以访问上边列出的 C 函数了。具体步骤如下：

```
static JSFunctionSpec functions[] = {
    {"logging", logging, LOG_MINARGS, 0, 0},
    {"nothing", nothing, NOT_MINARGS, 0, 0},
    {0, 0, 0, 0, 0}
```

```
};

//define function list here
if(!JS_DefineFunctions(context, global, functions)){
    return -1;
}
```

运行结果如下:

```
message from script environment : user jack login on 2010/7/6
got nothing to do at all
message from script environment : user jack logout on 2010/7/7
message from script environment : hello
message from script environment : all
message from script environment : my
message from script environment : friend
```

13.1.7 在 C 程序中定义 JavaScript 对象

在 SpiderMonkey 中, 在 JavaScript 中使用由 C 语言定义的对象较为复杂, 一旦我们可以定义对象, 使得两个世界通过 JS 交互就变得非常简单而有趣, 很容易使用这样的方式来定制我们的应用, 在系统发布之后仍然可以轻松的修改系统的行为。

首先, 我们要定义好基本的数据结构, 即我们要暴露给 JS 世界的对象的属性, 结构; 然后, 使用 JSAPI 定义这个对象的属性; 然后, 使用 JSAPI 定义对象的方法; 最后, 创佳这个对象, 并绑定其属性表和方法列表, 放入全局对象。

假设我们有这样一个数据结构, 用来表述一个人的简单信息:

```
typedef struct{
    char name[32];
    char addr[128];
}PersonInfo;
```

定义属性表为枚举类型:

```
enum person{
    NAME,
    ADDRESS
};
```

我们需要将 C 语言原生的数据结构定义为 JSAPI 所识别的那样:

```
//define person properties
JSPropertySpec pprops[] = {
    {"name", NAME, JSPROP_ENUMERATE},
    {"address", ADDRESS, JSPROP_ENUMERATE},
    {0}
};

//define person methods
JSFunctionSpec pfuncs[] = {
    {"print", printing, 0},
    {"getName", getName, 0},
    {"setName", setName, 0},
    {"getAddress", getAddress, 0},
    {"setAddress", setAddress, 0},
    {0}
};

//define person class (JSClass)
JSClass pclass = {
    "person", 0,
    JS_PropertyStub, JS_PropertyStub, JS_PropertyStub, JS_PropertyStub,
    JS_EnumerateStub, JS_ResolveStub, JS_ConvertStub, JS_FinalizeStub
};
```

一旦这些基本信息定义好(pfuncs 数组中的 **getter/setter** 的实现比较简单, 这里由于篇幅不列出代码, 感兴趣的朋友可以参看附录), 我们就可以实例化它, 并将其放入上下文中, 使得 JS 代码可以访问。

```
JSObject *person;

//define the object
person = JS_DefineObject(\
    context, global, "person", &pclass, 0, JSPROP_ENUMERATE);

//install the properties and methods on the person object
JS_DefineProperties(context, person, pprops);
JS_DefineFunctions(context, person, pfuncs);
```

这样, 在 JavaScript 代码中, 我们就可以通过 **person** 这个标识符来访问 **person** 这个对象了:

```
//undefined of course
person.print();

//person.name = "abruzzi";
//person.address = "Huang Quan Road";

person.setName("Desmond");
person.setAddress("HuangQuan Road");

//print is global function, access properties directly
print("person name = " + person.name);
print("person address = " + person.address);

person.print();

(function(){
    //using getter/setter to access properties
    return person.getName() + " : " + person.getAddress();
})();
```

对运行结果如下:

```
name : undefined
address : undefined
person name = Desmond
person address = HuangQuan Road
name : Desmond
address : HuangQuan Road
eval script result is : Desmond : HuangQuan Road
```

13.2 SpiderMonkey 的实现

SpiderMonkey 是一个 C 语言实现的 JavaScript 引擎，著名的 Firefox 浏览器使用此引擎解释 JavaScript。

13.2.1 虚拟机概述

一般而言，编程语言的虚拟机是针对某种指令的解释器。比如 JVM 会解释.class 中的指令集，python 的虚拟机会解释.pyc 中的指令集。指令通常由几个字节来表示，字节中的某些位表示操作符，其余部分表示操作数，指令又可能分为定长和不定长。这个和实际的 CPU 指令非常类似。虚拟机运行起来之后，它会逐条的解释指令，遇到跳转或者分支，则按照操

作数中的地址进行跳转，直至遇到停机指令，然后退出。

13.2.2 SpiderMonkey 体系结构

SpiderMonkey 引擎是一个快速的解释器，也就是一个虚拟机。同时，它还包括两个 JIT 编译器，垃圾回收器，以及一些通用的库。SpiderMonkey 实现了 ECMA 262-3 标准，并提供了一些扩展。

解释器部分与其他的虚拟机一样，主体部分具有一个 `switch` 语句，根据不同的操作码来进行分发，并调用各自的处理函数，直到遇到停机指令。由于代码中会有这种场景：JavaScript 代码调用 C 代码，被调用的 C 代码里又调用其他的 JavaScript 代码，这就要求解释器必须是可重入的。

编译器部分用以分析 JavaScript 代码，生成字节码，代码注释，常量表等。编译器由三部分组成：一个词法分析器，一个递归向下的分析器(生成抽象语法树)，一个代码生成器(遍历语法树产生代码)。编译器会做一些代码优化，并将源码的注解附加在生成的脚本中，以便后期的反编译。反编译器中实现了 `Function.toSource()`，用以重建函数的源码，并将后缀形式(指令集中的格式)转换成中缀(程序员可读)的格式。

垃圾回收器使用标记-清除方式，SpiderMonkey 的垃圾回收采用精确方式，一般情况下，SpiderMonkey 的 API 应采用 `JS_GC` 及 `JS_MaybeGC` 来进行显式的垃圾回收。由于采用精确方式，使用此引擎时，应确保所有的活动对象，字符串，对象等都是垃圾回收器可访问的。

正如第三章提到的，JavaScript 对象是一个属性的集合，并且有一个原型对象。原型可以是一个对象或者 `null`。一般的引擎实现中，JS 对象会包含若干个隐藏属性，对象的原型由这些隐藏属性之一引用，我们在本文中讨论时，将假定这个属性的名称为"`__proto__`"(事实上，SpiderMonkey 内部正是使用了这个名称，但是规范中并未做要求，因此这个名称依赖于实现)。

13.2.3 jsval 类型

`jsval` 类型是一个有符号的机器字，可能是一个有符号的整数(如果低位被设置)，或者带有类型标记的指针，或者特殊值(如果低位被清空)。标记过的指针引用 GC 堆中，以 8 个字节对齐的对象。

特殊值是 `JVAL_NULL`, `JVAL_VOID(undefined)`, `JVAL_TRUE`, `JVAL_FALSE`. 另一个特殊值：`JVAL_HOLE`，只限于内部使用(用于表示已经被删除的数组元素)。这个值没有在 JSAPI 中公开，可以不理睬。

`jsval` 可以与宿主对象互相转换，如将 C 的字符串(`char *`)转换为 JavaScript 的字符串，以及浮点数，布尔值等的转换。

13.2.4 对象

对象，由一个可以共享的结构描述(`map` 或者叫 `scope`)，及向量形式的不可共享的属性值(叫做 `slot`)组成。每一个属性都有一个 `id`，可能是一个非负整数或者一个原子(不相等的串)，标记指针被解码为 `jsval`。

13.3 V8 引擎概览

V8 引擎是 Google 公司的开源的 JavaScript 引擎，由 C++ 语言编写。与 Google 的浏览器 Chrome 同时发布，Chrome 浏览器内部使用了 V8 引擎。V8 实现了 ECMA-262 的第三版，并可以运行在 windows, linux, mac 等操作系统中，V8 的精确内存回收算法是其性能的一个重要因素。

使用 V8 引擎，可以很容易的完成 C++ 应用程序中的宿主对象与 JavaScript 脚本的交互，如变量访问，函数调用，对象传递等，这样可以很容易的完成 C++ 应用的脚本化。

13.3.1 V8 引擎基本概念

下面我们来看一看 V8 引擎中的基本概念，熟悉这些基本概念，有助于理解后续小节中的示例：

- `handle`，`handle` 是指向对象的指针，在 V8 中，所有的对象都通过 `handle` 来引用，`handle` 主要用于 V8 的垃圾回收机制。
- `scope`，`scope` 是 `handle` 的集合，可以包含若干个 `handle`，这样就无需将每个 `handle` 逐次释放，而是直接释放整个 `scope`。
- `context`，`context` 是一个执行器环境，使用 `context` 可以将相互分离的 JavaScript 脚本在同一个 V8 实例中运行，而互不干涉。在运行 JavaScript 脚本是，需要显式的指定 `context` 对象。

在 V8 中，`handle` 分为两种：持久化(`Persistent`)`handle` 和本地(`Local`)`handle`，持久化 `handle` 存放在堆上，而本地 `handle` 存放在栈上。这个与 C/C++ 中的堆和栈的意义相同。持久化 `handle` 与本地 `handle` 都是 `Handle` 的子类。在 V8 中，所有数据访问均需要通过 `handle`。需要注意的是，使用持久化 `handle` 之后，需要显式的调用 `Dispose()` 来通知垃圾回收机制。

在使用本地 `handle` 时，需要声明一个 `HandleScope` 的实例，`scope` 是 `handle` 的容器，使用 `scope`，则无需依次释放 `handle`。

```
HandleScope handle_scope;  
Local<ObjectTemplate> temp;
```

由于 C++ 原生数据类型与 JavaScript 中数据类型有很大差异，因此 V8 提供了 `Data` 类，从 JavaScript 到 C++，从 C++ 到 JavaScript 都会用到这个类及其子类，比如：

```
Handle<Value> Add(const Arguments& args) {
    int a = args[0]->Uint32Value();
    int b = args[1]->Uint32Value();

    return Integer::New(a+b);
}
```

Integer 即为 Data 的一个子类。

V8 中，有两个模板(Template)类(并非 C++中的模板类)：对象模板(ObjectTemplate)和函数模板(FunctionTemplate)，这两个模板类用以定义 JavaScript 对象和 JavaScript 函数。我们在后续的小节部分将会接触到模板类的实例。通过使用 ObjectTemplate，可以将 C++中的对象暴露给脚本环境，类似的，FunctionTemplate 用以将 C++函数暴露给脚本环境，以供使用。

初始化 context 是使用 V8 引擎所必需的过程，代码非常简单：

```
Persistent<Context> context = Context::New();
```

有了上面所述的基本概念之后，我们来看一下一个使用 V8 引擎的应用程序的基本流程：

1. 创建 HandleScope 实例
2. 创建一个持久化的 Context
3. 进入 Context
4. 创建脚本字符串
5. 创建 Script 对象，通过 Script::Compile()
6. 执行脚本对象的 Run 方法
7. 获取/处理结果
8. 显式的调用 Context 的 Dispose 方法

13.3.2 V8 引擎使用示例

我们来看一下 V8 引擎的使用，首先看一个 V8 版的 HelloWorld:

```
#include <v8.h>

using namespace v8;

int main(int argc, char *argv[]) {
    // 创建一个句柄作用域(在栈上)
    HandleScope handle_scope;
```

```
// 创建一个新的上下文对象
Persistent<Context> context = Context::New();

// 进入上一步创建的上下文，用于编译执行helloworld
Context::Scope context_scope(context);

// 创建一个字符串对象，值为'Hello, World!', 字符串对象被JS引擎
// 求值后，结果为'Hello, World!'
Handle<String> source = String::New("'Hello' + ', World!'");

// 编译字符串对象为脚本对象
Handle<Script> script = Script::Compile(source);

// 执行脚本，获取结果
Handle <Value> result = script->Run();

// 释放上下文资源
context.Dispose();

// 转换结果为字符串
String::AsciiValue ascii(result);

printf("%s\n", *ascii);

return 0;
}
```

13.3.3 使用 C++变量

在 JavaScript 与 V8 间共享变量事实上是很容易的，基本模板如下：

```
static type xxx;

static Handle<Value> xxxGetter(
    Local<String> name,
    const AccessorInfo& info) {

    //code about get xxx
}

static void xxxSetter(
    Local<String> name,
```

```

Local<Value> value,
const AccessorInfo& info){

    //code about set xxx
}

```

首先在 C++ 中定义数据，并以约定的方式定义 `getter/setter` 函数，然后将 `getter/setter` 通过下列机制公开给脚本：

```
global->SetAccessor(String::New("xxx"), xxxGetter, xxxSetter);
```

其中，`global` 对象为一个全局对象的模板：

```
Handle<ObjectTemplate> global = ObjectTemplate::New();
```

下面我们来看一个实例：

```

static char sname[512] = {0};

static Handle<Value> NameGetter(Local<String> name,
    const AccessorInfo& info) {
    return String::New((char*)&sname, strlen((char*)&sname));
}

static void NameSetter(Local<String> name,
    Local<Value> value,
    const AccessorInfo& info) {
    Local<String> str = value->ToString();
    str->WriteAscii((char*)&sname);
}

```

定义了 `NameGetter`, `NameSetter` 之后，在 `main` 函数中，将其注册在 `global` 上：

```

// Create a template for the global object.
Handle<ObjectTemplate> global = ObjectTemplate::New();

//public the name variable to script
global->SetAccessor(String::New("name"), NameGetter, NameSetter);

```

在 C++ 中，将 `sname` 的值设置为“cpp”：

```

//set sname to "cpp" in cpp program
strncpy(sname, "cpp", sizeof(sname));

```

然后在 JavaScript 中访问该变量，并修改：

```
print(name);

//set the variable `name` to "js"
name='js';
print(name);
```

运行结果如下：

```
cpp
js
```

13.3.4 调用 C++函数

在 JavaScript 中调用 C++函数是脚本化最常见的方式，通过使用 C++函数，可以极大程度的增强 JavaScript 脚本的能力，如文件读写，网络/数据库访问，图形/图像处理等等，而在 V8 中，调用 C++函数也非常的方便。

在 C++代码中，定义以下原型的函数：

```
Handle<Value> function(const Arguments& args){
    //return something
}
```

然后，再将其公开给脚本：

```
global->Set(String::New("function"),FunctionTemplate::New(function));
```

同样，我们来看两个示例：

```
Handle<Value> Add(const Arguments& args){
    int a = args[0]->Uint32Value();
    int b = args[1]->Uint32Value();

    return Integer::New(a+b);
}
```

```
Handle<Value> Print(const Arguments& args) {
    bool first = true;
    for (int i = 0; i < args.Length(); i++) {
        HandleScope handle_scope;
        if (first) {
```

```
        first = false;
    } else {
        printf(" ");
    }
    String::Utf8Value str(args[i]);
    const char* cstr = ToCString(str);
    printf("%s", cstr);
}
printf("\n");
fflush(stdout);
return Undefined();
}
```

函数 `Add` 将两个参数相加，并返回和。函数 `Print` 接受任意多个参数，然后将参数转换为字符串输出，最后输出换行。

```
global->Set(String::New("print"), FunctionTemplate::New(Print));
global->Set(String::New("add"), FunctionTemplate::New(Add));
```

我们定义以下脚本：

```
var x = (function(a, b){
    return a + b;
})(12, 7);

print(x);

//invoke function add defined in cpp
var y = add(43, 9);
print(y);
```

运行结果如下：

```
19
52
```

13.3.5 使用 C++ 类

如果从面向对象的视角来分析，最合理的方式是将 C++ 类公开给 JavaScript，这样可以将 JavaScript 内置的对象数量大大增加，从而尽可能少的使用宿主语言，而更大的利用动态语言的灵活性和扩展性。事实上，C++ 语言概念众多，内容繁复，学习曲线较 JavaScript

远为陡峭。最好的应用场景是：既有脚本语言的灵活性，又有 C/C++ 等系统语言的效率。使用 V8 引擎，可以很方便的将 C++ 类“包装”成可供 JavaScript 使用的资源。

我们这里举一个较为简单的例子，定义一个 `Person` 类，然后将这个类包装并暴露给 JavaScript 脚本，在脚本中新建 `Person` 类的对象，使用 `Person` 对象的方法。

首先，我们在 C++ 中定义好类 `Person`：

```
class Person {
private:
    unsigned int age;
    char name[512];

public:
    Person(unsigned int age, char *name) {
        this->age = age;
        strncpy(this->name, name, sizeof(this->name));
    }

    unsigned int getAge() {
        return this->age;
    }

    void setAge(unsigned int nage) {
        this->age = nage;
    }

    char *getName() {
        return this->name;
    }

    void setName(char *nname) {
        strncpy(this->name, nname, sizeof(this->name));
    }
};
```

`Person` 类的结构很简单，只包含两个字段 `age` 和 `name`，并定义了各自的 `getter/setter`。然后我们来定义构造器的包装：

```
Handle<Value> PersonConstructor(const Arguments& args) {
    Handle<Object> object = args.This();
    HandleScope handle_scope;
    int age = args[0]->Uint32Value();

    String::Utf8Value str(args[1]);
    char* name = ToCString(str);
```

```

    Person *person = new Person(age, name);
    object->SetInternalField(0, External::New(person));
    return object;
}

```

从函数原型上可以看出，构造器的包装与上一小节中，函数的包装是一致的，因为构造函数在 V8 看来，也是一个函数。需要注意的是，从 `args` 中获取参数并转换为合适的类型之后，我们根据此参数来调用 `Person` 类实际的构造函数，并将其设置在 `object` 的内部字段中。紧接着，我们需要包装 `Person` 类的 `getter/setter`：

```

Handle<Value> PersonGetAge(const Arguments& args){
    Local<Object> self = args.Holder();
    Local<External> wrap =
Local<External>::Cast(self->GetInternalField(0));

    void *ptr = wrap->Value();

    return Integer::New(static_cast<Person*>(ptr)->getAge());
}

```

```

Handle<Value> PersonSetAge(const Arguments& args)
{
    Local<Object> self = args.Holder();
    Local<External> wrap =
Local<External>::Cast(self->GetInternalField(0));

    void* ptr = wrap->Value();

    static_cast<Person*>(ptr)->setAge(args[0]->Uint32Value());
    return Undefined();
}

```

而 `getName` 和 `setName` 的与上例类似。在对函数包装完成之后，需要将 `Person` 类暴露给脚本环境：

首先，创建一个新的函数模板，将其与字符串“Person”绑定，并放入 `global`：

```

Handle<FunctionTemplate> person_template =
FunctionTemplate::New(PersonConstructor);
person_template->SetClassName(String::New("Person"));
global->Set(String::New("Person"), person_template);

```

然后定义原型模板：


```
Handle<ObjectTemplate> person_proto =
person_template->PrototypeTemplate();

person_proto->Set("getAge", FunctionTemplate::New(PersonGetAge));
person_proto->Set("setAge", FunctionTemplate::New(PersonSetAge));

person_proto->Set("getName", FunctionTemplate::New(PersonGetName));
person_proto->Set("setName", FunctionTemplate::New(PersonSetName));
```

最后设置实例模板:

```
Handle<ObjectTemplate> person_inst =
person_template->InstanceTemplate();
person_inst->SetInternalFieldCount(1);
```

随后, 创建一个用以测试的脚本:

```
//global function to print out detail info of person
function printPerson(person) {
    print(person.getAge()+" "+person.getName());
}

//new a person object
var person = new Person(26, "juntao");

//print it out
printPerson(person);

//set new value
person.setAge(28);
person.setName("juntao.qiu");

//print it out
printPerson(person);
```

运行得到以下结果:

```
26:juntao
28:juntao.qiu
```

第十四章 Java 应用中的 JavaScript

虽然 JavaScript 在设计之初是为了网页的动态化，但是随着这个语言被人们普遍接受之后，它就被多种其他编程语言所实现，比如 C 语言的 SpiderMonkey，Java 语言的 Rhino，以及 Google 的浏览器 Chrome 中使用的 V8 引擎(该引擎的执行速度是现在所有的引擎中最快的)。将 JavaScript 嵌入 C/C++/Java 应用有很多的好处，比如更好的实现用户配置，提供应用程序的扩展性，控制变化快速的需求等等。

14.1 脚本化基础

一般而言，脚本语言有这样的好处：

- 弱类型，动态数据类型，便于变量的复用
- 开发速度较编译型语言为快，因为大部分脚本语言为解释执行
- 应用程序的高可配置性

开发应用程序的语言成为宿主环境，而嵌入到宿主环境的即为脚本语言，比如在浏览器 Firefox 中，嵌入其内的 JavaScript 可以定制 UI，这为 Firefox 提供了大量可用的插件机制。而著名的编辑环境 vim/Emacs 更是提供了难以计数的插件，这些插件可以自由的引用宿主环境中的一些对象，比如 UI 组件，UI 组件的内部模型等等。从而使得软件的定制性更高，比如几乎每一个 Emacs 用户的快捷键，窗口数目，窗口的背景色都是不同的。

我们在本书中，当然是以 JavaScript 语言为本，我们在本节来讨论如何使用 JavaScript 来脚本化 Java 应用，首先需要了解的是执行 JavaScript 代码的基础：JavaScript 引擎。

14.1.1 脚本化框架

Rhino JavaScript 引擎本来由 Mozilla 开发，后来在 JDK6 的时候加入了 JDK，因此如果你使用的 JDK 版本为 6 或更高，则无需任何配置，即可使用这个脚本化框架。事实上，JDK6 中带的这个脚本化框架是与脚本无关的一个框架，用户可以使用已经存在的脚本引擎，甚至实现自己的脚本引擎。目前，已经可以使用的脚本包括 python, Groovy 等，当然也有 JavaScript。

使用脚本化框架，可以在 Java 代码中执行 JavaScript 脚本中的函数，可以引用 JavaScript 变量，而 JavaScript 可以充分利用 Java 中的大量可用的工具包，创建 Java 对象，调用 Java 中对象的方法，使用 Java 代码中共有的属性等等。通过脚本化框架，我们可以使用 Java 开发出宿主环境的结构，然后使用 JavaScript 来定制用户界面布局,流程控制等，从而实现脚本化。

JDK6 中引入了 javax.script 包，其中

14.2 使用 Rhino 引擎

在本章中，假设读者的计算机中都有 JDK6 或以上的版本，如果没有，可以更新安装或者参考其他资料。在 JDK6 的 `JAVA_HOME\bin` 下，有一个名为 `jrscript` 的脚本，运行的时候会进入一个命令环境，可以运行 JavaScript 片段，加载外部 JS 文件等，我们以这个工具为例来说明如何让 Java 代码与 JS 交互的一些基本概念。

14.2.1 直接对脚本求值

脚本化的 Hello,world 版本：

```
import javax.script.*;

public class HelloScript {

    public static void main(String args[]){
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("javascript");
        try {
            engine.eval("print('Hello, world');");
        } catch (ScriptException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

首先，创建 `ScriptEngineManager` 对象，然后根据引擎名称“`javascript`”来获取一个 JavaScript 引擎的实例，然后调用引擎实例的 `eval` 方法。应该注意的是，`print` 函数是 rhino 引擎内建的全局函数，在浏览器上的 JS 环境中是无法执行的。

`eval` 方法可以接受一个字符串，然后直接求值，也可以接受一个 `Reader` 对象，将一个脚本文件完整求值。另外，`eval` 还接受第二个参数，以指定运行时的上下文对象，如果不指定，则按照全局上下文来求值。

14.2.2 传递 Java 对象

前面提到，脚本化技术允许我们在 Java 中访问 JavaScript 变量，调用 JavaScript 函数，以及反过来，从 JavaScript 中访问 Java 对象的属性，方法。

我们先创建一个 **Person** 的简单类，其中包含 **name** 和 **age** 属性，以及 **getter/setter**，然后通过脚本来访问这个对象。

```
package com.rc.scripting;

public class Person{
    String name;
    int age;

    public Person(String name, int age){
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

有了这个 **Person** 类之后，我们在来建立一个脚本文件，命名为 **script.js**:

```
print(jack);
print('\n');
print(jack.getName());
```

然后，我们将 **HelloScript** 类的 **main** 做一些简单修改，创建新的 **Person** 对象 **jack**，并将其 **put** 到引擎中，然后去 **eval** 刚才建立的脚本文件 **script.js**:

```
public static void main(String args[]){
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine engine = manager.getEngineByName("javascript");
```

```
Person jack = new Person("jack", 28);

engine.put("jack", jack);

try {
    engine.eval(new java.io.FileReader("scripts/script.js"));
} catch (ScriptException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
} catch (FileNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
}
```

可以得到如下结果:

```
com.rc.scripting.Person@18ac738
jack
```

14.2.3 调用脚本内的函数

再来看看在 Java 中调用 JavaScript 函数，首先在脚本 script.js 中定义函数 sayHello:

```
function sayHello(name) {
    print("hello, "+name+"!");
}
```

然后在 Java 代码中，将引擎转换为可调用(Invocable)引擎，应该注意的是，Invocable 接口是一个可选的接口，如果要编写自己的脚本引擎，可以不实现此接口。rhino 引擎实现了该接口，因此我们可以很方便的做一个转换，然后调用 `invokeFunction` 方法来调用脚本中的函数:

```
public static void main(String args[]){
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine engine = manager.getEngineByName("javascript");

    try {
        engine.eval(new java.io.FileReader("scripts/script.js"));
    } catch (ScriptException e) {
        e.printStackTrace();
    } catch (FileNotFoundException e) {
```

```
        e.printStackTrace();
    }

    Invocable invEngine = (Invocable)engine;

    try {
        invEngine.invokeFunction("sayHello", "jack");
    } catch (ScriptException e) {
        e.printStackTrace();
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    }
}
```

运行结果为:

hello, jack!

由于 JavaScript 是“面向对象”的语言，因此我们可以调用一个对象的方法，我们将全局函数 `sayHello` 包装在对象 `object` 上，作为 `object` 的一个属性：

```
var object = new Object();

object.sayHello = function(name){
    print("hello, "+name+"!");
}
```

然后在 Java 代码中通过 `invokeMethod` 方法来调用，首先从脚本引擎中取道这个对象的实例，通过 `engine.get(name)` 方法来实现：

```
public static void main(String args[]){
    ScriptEngineManager manager = new ScriptEngineManager();
    ScriptEngine engine = manager.getEngineByName("javascript");

    try {
        engine.eval(new java.io.FileReader("scripts/script.js"));
    } catch (ScriptException e) {
        e.printStackTrace();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }

    Invocable invEngine = (Invocable)engine;
```

```
Object obj = engine.get("object");
try {
    invEngine.invokeMethod(obj, "sayHello", "jack again");
} catch (ScriptException e) {
    e.printStackTrace();
} catch (NoSuchMethodException e) {
    e.printStackTrace();
}
}
```

14.2.4 在脚本中使用 Java 资源

`javax.script` 提供的最值得一提的功能就是可以在脚本中使用所有的 `java` 资源, 由于 `Java` 语言已经发展的极为成熟, 有大量可用的工具包可供使用, 而很多情况下, 限于 `Java` 语言本身的功能限制, 我们可能需要具有其他特征如动态性, 弱类型, 嵌套函数等脚本特性, 将两者结合起来, 一方面使得编程工作更为有趣, 更为高效快速, 又不必担心代码的质量或运行效率。

在 `JDK6` 中的脚本化框架中, 我们可以在脚本中导入 `Java` 包, `Java` 类, 实现 `Java` 接口(在下一小节中详细讨论), 这给开发人员带来极大的便利, `javax.script` 提供了两个内置函数 `importPackage` 和 `importClass` 分别用以导入 `Java` 包和类。

比如下列语句:

```
importPackage(java.awt, java.awt.event)
importPackage(Packages.javax.swing)
importPackage(java.io)
importClass(java.lang.System)
```

我们来看一个小例子, 通过导入 `javax.swing` 工具包, 绘制一个小窗口:

```
importPackage(java.awt, java.awt.event);
importPackage(Packages.javax.swing);

(function(title){
    var frame = new JFrame(title);
    frame.setSize(300, 150);

    var label = new JLabel("I'am a label");
    frame.add(label);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    frame.setVisible(true);
});
```

```
})( "Script frame" );
```

得到下图这个小窗体:

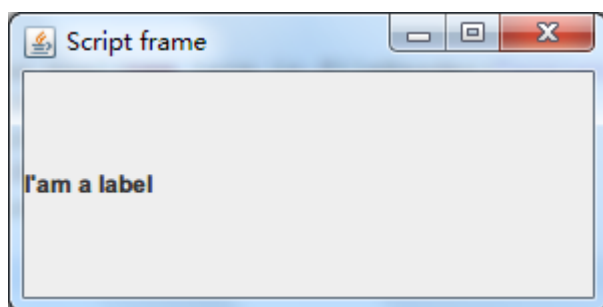


图 JavaScript 通过 swing 绘制小窗体

14.2.5 实现 Java 接口

通过接口的方式, 可以更好的分离宿主环境与脚本之间的依赖, **Java** 暴露给脚本以接口, 脚本实现接口, 然后在 **Java** 中可以在完全不知道脚本是如何实现的前提下进行调用, 我们来看一个简单的示例:

```
var task = {
  run : function () {
    print("start execute...");
  }
}
```

脚本实现了 `java.lang.Runnable` 接口, `Runnable` 接口仅有一个方法: `run`。然后在 **Java** 代码中, 我们从引擎对象 `engine` 中获得此对象 `task`, 然后创建一个新的线程以使用 `Runnable` 接口的实现:

```
public static void main(String args[]) {
  ScriptEngineManager manager = new ScriptEngineManager();
  ScriptEngine engine = manager.getEngineByName("javascript");

  try {
    engine.eval(new java.io.FileReader("scripts/script.js"));
  } catch (ScriptException e) {
    e.printStackTrace();
  } catch (FileNotFoundException e) {
    e.printStackTrace();
  }

  Invocable invEngine = (Invocable)engine;
```



```
Object obj = engine.get("task");

Runnable r = invEngine.getInterface(obj, Runnable.class);

Thread thread = new Thread(r);
thread.start();
}
```

通过引擎的 `getInterface(object, class)` 来获得此接口的实现，由于 JavaScript 的弱类型，Java 端只能通过判断对象是否与接口的定义匹配来确定其是否实现了指定的接口，因此 `getInterface` 的第二个参数是一个“类”类型。

另一个相关的例子是，在 JavaScript 代码中实现 Java 接口，所有代码均属于 JavaScript 代码：

```
var task = new java.lang.Runnable() {
  run : function() {
    print("start execute...");
  }
};

var thread = new java.lang.Thread(task);
thread.start();
```

与上例相同，会得到下面的执行结果：

```
start execute...
```

14.3 实例：sTodo

14.3.1 sTodo 简介

sTodo 是一个简单的待办事项管理工具，使用 Java 的 Swing 工具包作为 UI 展现，而内部使用一个嵌入式的数据库 SQLite 作为数据源，并使用了 JavaScript 来进行界面风格的定制，功能的扩展等。

运行时的界面如下图：

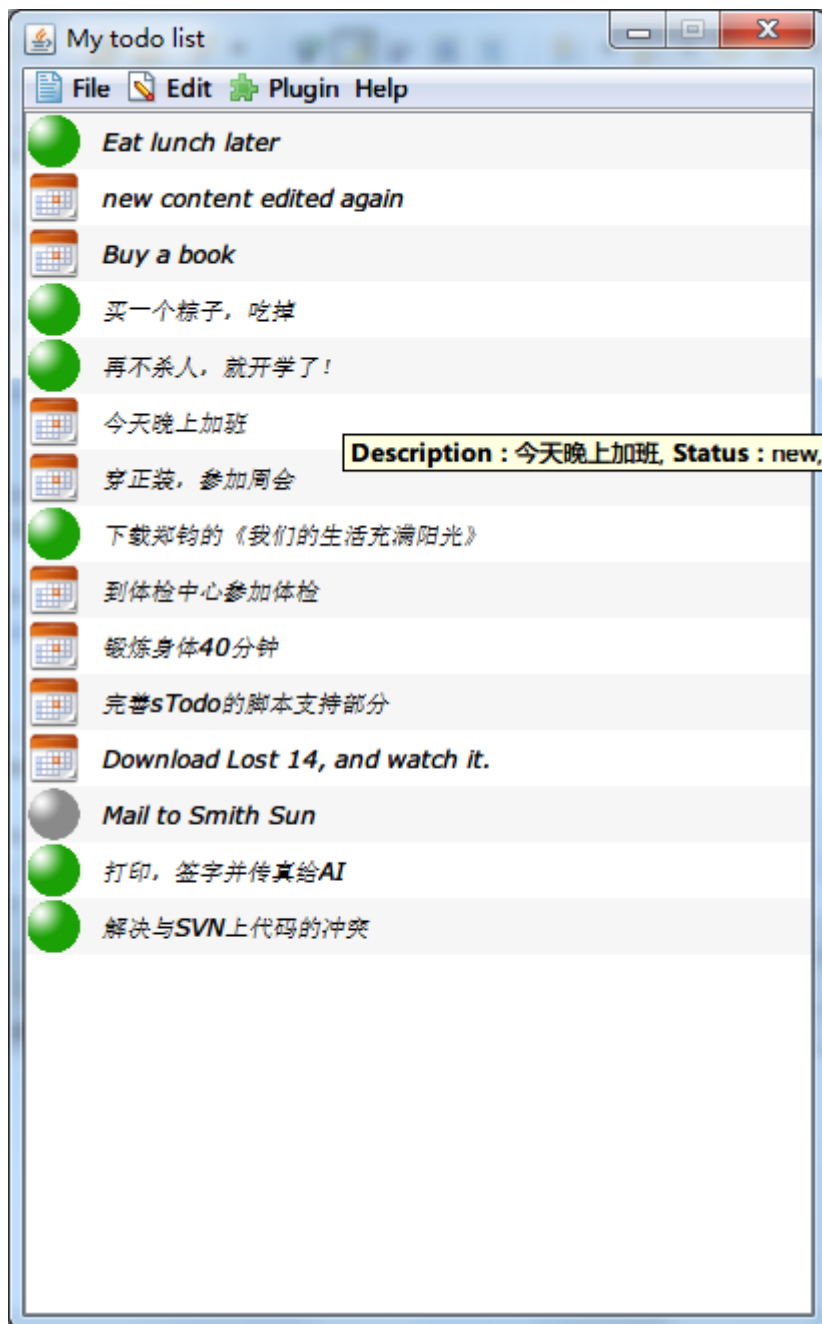


图 sTodo 运行界面

事实上，上图中的菜单栏中的 `plugin` 和 `help` 都是通过脚本添加的，sTodo 原始的界面如下：

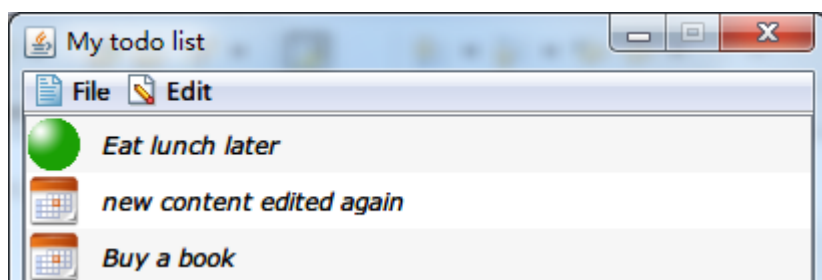


图 sTodo 原始界面

sTodo 中有一个对 `javax.script` 的浅包装，作为 sTodo 中的插件机制。插件机制我们在下一小节详细讨论。sTodo 的其他模块，比如数据源模块，邮件发送模块等与本章的主题无关，这里不做讨论，sTodo 托管在 `google code` 上，是一个完全开源的小项目，有兴趣的读者可以自行研究。

14.3.2 sTodo 的插件机制

sTodo 的插件机制是对 `javax.script` 做的一个浅包装。插件在物理上是一个脚本文件，sTodo 的 Java 端代码公开一些对象供 JavaScript 访问，比如菜单栏，这样 JavaScript 代码则可以创建新的菜单，提供新的功能给 Java。同样的，在 JavaScript 端，公开一个入口，供 Java 端的代码调用。

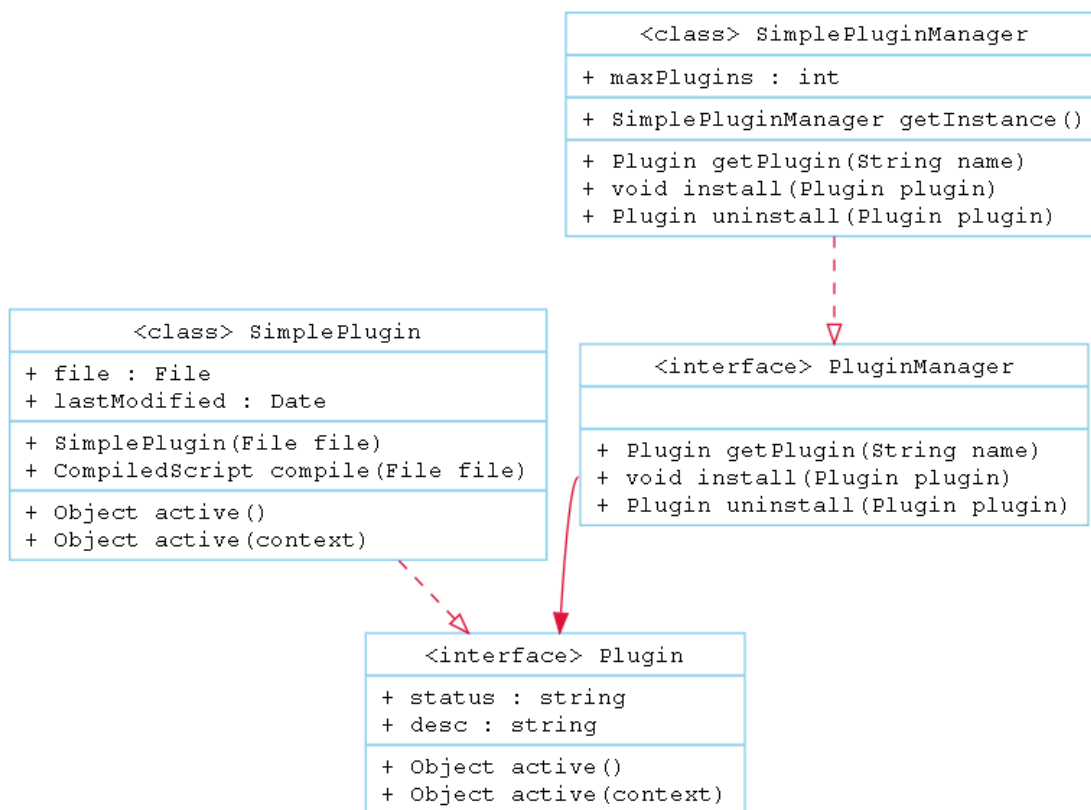


图 sTodo 插件机制结构图

插件具有状态，并被安装在插件管理器中，可以被激活。sTodo 的生命周期中，有一个单例的插件管理器，可以在一个模块内安装插件，并在另外一个模块中使用，为了保证访问插件的时刻必须在安装之后，我们可以在软件初始化的时候将依赖关系调整好，然后将异步的响应动作放到 UI 组件的监听器上。

我们来详细查看一下 sTodo 的工作机制：

在 `sTodo` 的初始化中:

```
public void initEnv(){
    PluginManager pManager = TodoPluginManager.getInstance();

    Plugin system =
        new TodoPlugin("scripts/system.js", "system", "system
initialize");
    pManager.install(system);
}
```

我们会安装一个 `system.js` 的插件, 这个插件的功能是初始化脚本环境, 加载其他插件 (也就是脚本)。然后在 `main` 方法中:

```
public static void main(String[] args){
    STodo sTodo = new STodo(new MainFrame("My todo list"));
    sTodo.initEnv();

    Plugin system =
TodoPluginManager.getInstance().getPlugin("system");
    system.putValueToContext("Application", sTodo);
    system.putValueToContext("DataModel", new DataModel());
    system.putValueToContext("Util", new Util());

    system.execute("main", new Object());
}
```

将一些公用的组件暴露给脚本环境, 如 `Application`, `DataModel`, `Util` 等, 以便脚本环境可以直接访问 `Java` 端的 `swing` 组件及数据模型。最后, 执行 `system.execute`。也就是执行 `main` 函数, 这个 `main` 函数定义在插件 `system` 中。

最后, 还有一个方法用以启动 `sTodo` 的 `Java` 端:

```
public void launch(){
    SwingUtilities.invokeLater(new Runnable(){
        public void run() {
            mainFrame.initUI();
        }
    });
}
```

这个方法通过 `JavaScript` 端(`system` 插件对应的脚本)来完成。流程是这样: `Java` 端执行 `main` 方法, 初始化环境, 加载脚本, 然后想底层的引擎中放入公有对象(`Application` 等), 然后调用 `JavaScript` 端的 `main` 函数, `JavaScript` 端开始初始化其自身的环境, 与共有对

象交互，最后反过来调用 Java 端的 launch 启动 UI。

14.3.3 sTodo 中的脚本

脚本入口 main 方法:

```
function main() {
    var app = Application;
    var ui = app.getUI();

    //set look and feel to windows
    ui.setLookAndFeel("windows");

    //load some new scripts
    app.activePlugin("scripts/json.js");
    app.activePlugin("scripts/date.js");
    app.activePlugin("scripts/util.js");
    app.activePlugin("scripts/menuBar.js");
    app.activePlugin("scripts/misc.js");

    app.launch();
    //loadTodosFromFile("todos.txt");
}
```

这里的 main 函数首先获取 Java 端暴露的 Application 对象，然后设置 L&F 为 Windows 平台，然后依次加载一些脚本，最后调用 Application 上的方法 launch 启动 Java 端。

另外一个暴露给 Java 端的函数是初始化菜单栏的函数:

```
//this function will be invoked from java code, MainFrame...
function _customizeMenuBar_(menuBar) {
    menuBar.add(buildPluginMenu());
    menuBar.add(buildHelpMenu());
}
```

menubar 参数通过 Java 传递过来，然后依次调用 buildPluginMenu 和 buildHelpMenu 函数，创建新的菜单项，注册事件监听器，我们这里进讨论 buildPluginMenu 函数:

```
function buildPluginMenu() {
    var menuPlugin = new JMenu();
    menuPlugin.setText("Plugin");
    menuPlugin.setIcon(new ImageIcon("imgs/plugin.png"));
}
```

```
var menuItemListPlugin = new JMenuItem();
menuItemListPlugin.setText("list plugins");
menuItemListPlugin.addActionListener(
    new JavaAdapter(
        ActionListener, {
            actionPerformed : function(event){
                var plFrame = new JFrame("plugins list");
                var epNote = new JEditorPane();
                var s = "";
                pluginList = Application.getPluginList();
                for(var i = 0; i<pluginList.size();i++){
                    var pi = pluginList.get(i);
                    s += pi.getName()+":"+pi.getDescription()+"\n";
                }
                epNote.setText(s);
                epNote.setEditable(false);
                plFrame.add(epNote, BorderLayout.CENTER);
                plFrame.setSize(200,200);
                plFrame.setLocationRelativeTo(null);
                plFrame.setVisible(true);
            }
        }
    ));

menuPlugin.add(menuItemListPlugin);

return menuPlugin;
}
```

`buildPluginMenu` 函数创建一个新的 `JMenu` 对象，并为其添加一个 `JMenuItem`，当点击该菜单项时，创建一个新的 `JFrame`，列出当前软件已经安装的插件(通过调用 `Application.getPluginList()`获取)，运行效果如下图：

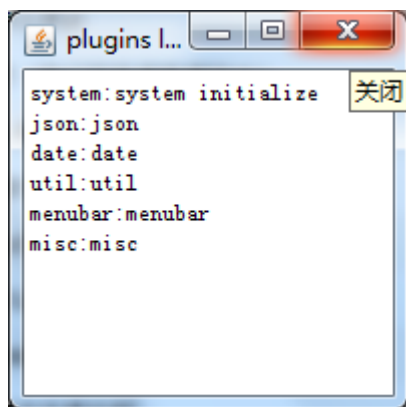


图 列出当前已经安装的插件列表

当然，sTodo 中还有很多地方使用了插件，感兴趣的朋友可以自行参考，比如通过脚本来建立新的待办事项，发送待办事项到指定邮箱，日期格式的分析等。

14.4 实例：可编程计算器 phoc

所谓可编程计算机是指，phoc 本身可以通过程序来进行扩展，当然这里的程序是指 JavaScript 脚本，利用第三方的优秀开源工具 jmathtool 的绘制功能，phoc 可以绘制出 2d 及 3d 的函数图像。phoc 事实上是建立在 javax.script 基础上的一个包装，计算功能，函数定义功能，变量功能都是直接使用脚本引擎提供的能力。

14.4.1 phoc 简介

我们先来看看 phoc 的界面：

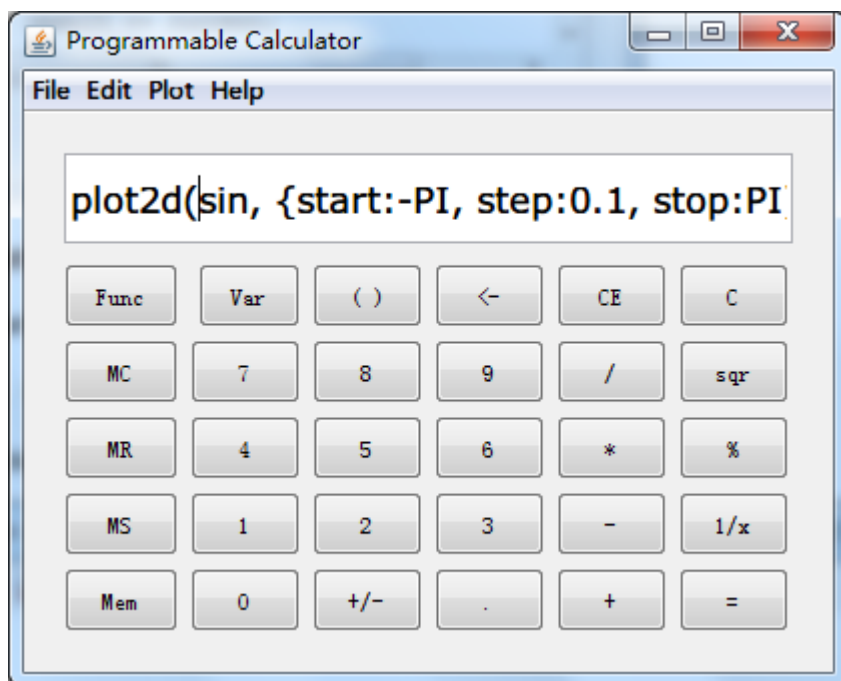


图 phoc 主界面

和其他的计算器界面并无二致，但是读者可能已经注意到，有两个一般不会出现在计算器上的按钮，Func 和 Var，正如其名字显示的那样，这两个按钮分别为：函数定义和变量定义，函数定义可以定义自己的函数，如：

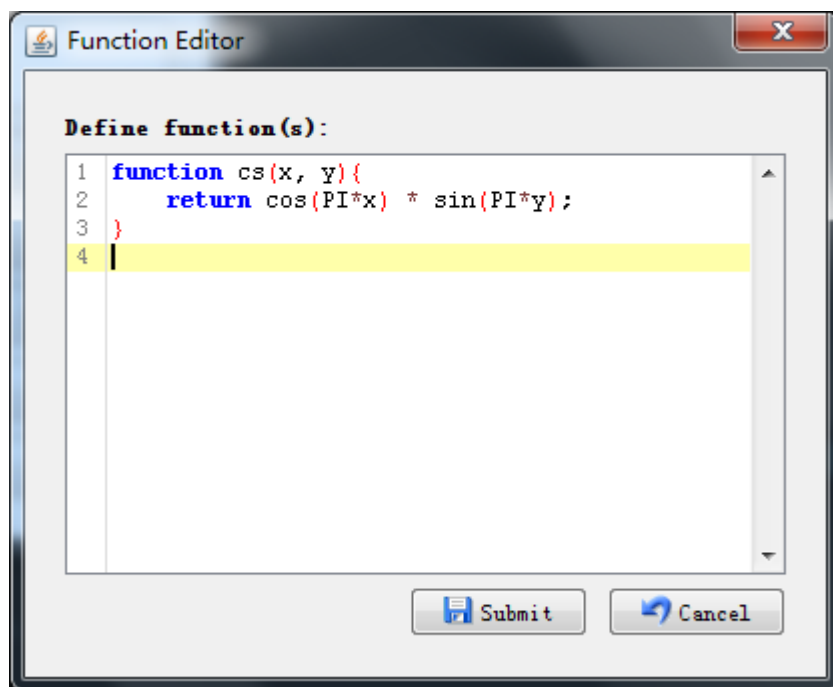


图 函数定义面板

定义之后的函数可以直接在输入框中使用。由于使用了 JavaScript 语言作为解释器，因此这个计算器甚至具有计算对象，字符串，布尔值等的功能。

下面是用 phoc 绘制的 2d 的 sin 函数在 $-\pi$ 到 π 之间的图形：

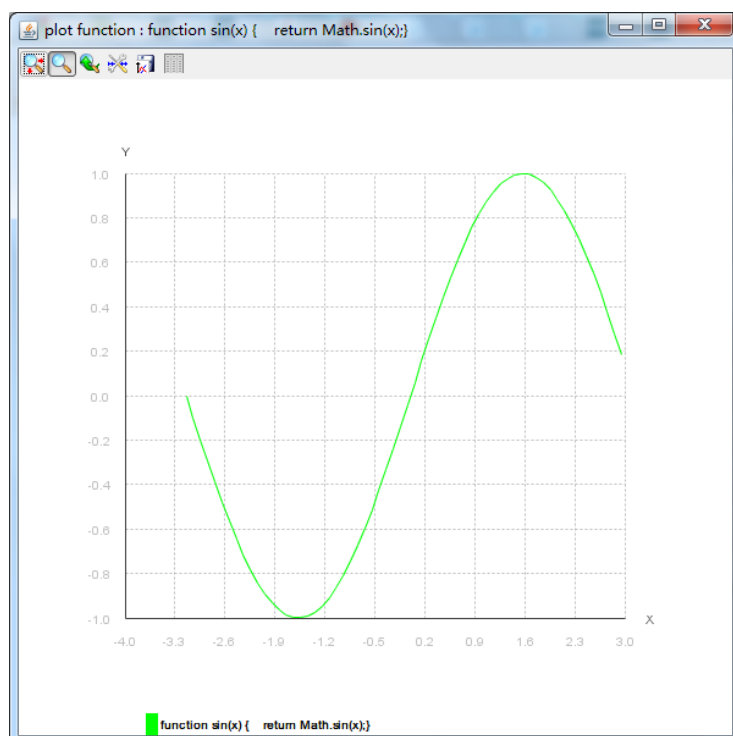


图 phoc 2d 绘图示例

下图为 phoc 绘制的 3d 图形：

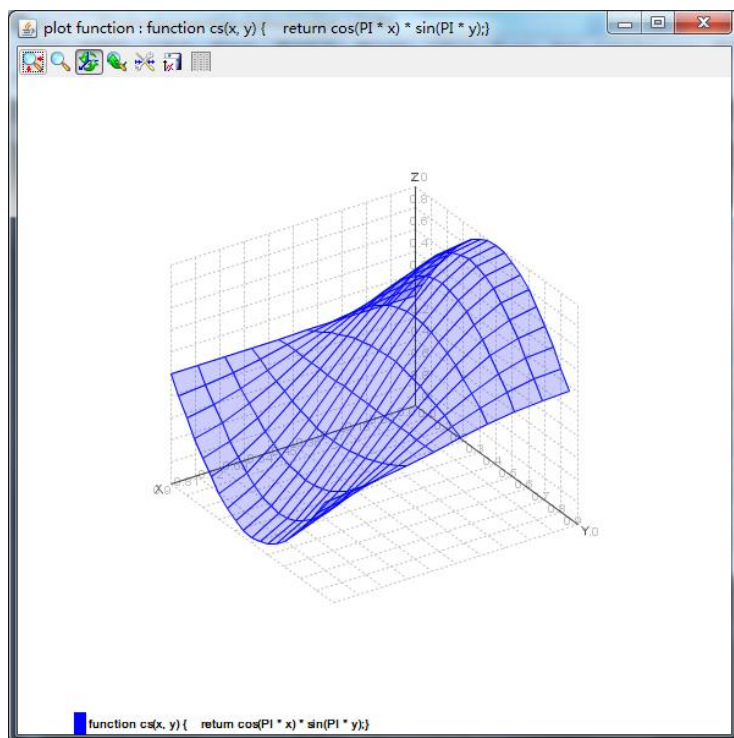


图 phoc 3d 绘图示例

14.4.2 phoc 中的脚本

phoc 中包含两个脚本文件，分别为 `math.js` 和 `plot.js`。`math.js` 中对 JavaScript 的内建 `Math` 函数做了一个重定义和包装，`plot.js` 则对 `jmathtool` 工具包做了 Java-JavaScript 的转换，使得在 phoc 中使用 JavaScript 函数来绘制 2d 和 3d 的图形。

```
var PI = Math.PI;
var E = Math.E;

function sqrt(x) {
    return Math.sqrt(x);
}

function sin(x) {
    return Math.sin(x);
}

function cos(x) {
    return Math.cos(x);
}
```

```

function sum() {
  var result = 0;
  for(var i = 0, len = arguments.length; i < len; i++){
    var current = arguments[i];
    if(isNaN(current)){
      throw new Error("not a number exception");
    }else{
      result += current;
    }
  }

  return result;
}

```

`math.js` 的部分代码，目的仅为在 `phoc` 的函数定义器及输入框中减少输入量。这个脚本无需展开讨论，下面看一下 `plot.js`：

```

importPackage(Packages.javax.swing)
importPackage(java.awt)
importClass(org.math.array.DoubleArray)
importClass(org.math.plot.Plot2DPanel)
importClass(org.math.plot.Plot3DPanel)

/**
 * a little helper for plot
 */
function increment(start, step, stop){
  var x = [];
  for(start; start+step < stop; start += step){
    x.push(start);
  }
}

```

首先导入“`org.math.array.DoubleArray`”及“`org.math.plot.*`”，`jmathtool` 的绘图接口包装在 `Plot2DPanel` 和 `Plot3DPanel` 中。

```

/**
 * i.e.
 * plot(function(x){return Math.sin(x);}, {
 *   start : -3.0,
 *   step : 0.1,
 *   stop : 3.0
 * });
 */

```

```
function plot2d(func, range){
    var x = DoubleArray.increment(range.start, range.step, range.stop);
    var y = new Array(x.length);

    for(var i = 0, len = x.length; i < len; i++){
        y[i] = func(x[i]);
    }

    var plot = new Plot2DPanel();
    plot.addLegend("SOUTH");

    var name = func.toString();
    plot.addLinePlot(name, Color.GREEN, x, y);

    var frame = new JFrame("plot function : "+name);
    frame.setSize(600, 600);
    frame.setContentPane(plot);
    frame.setVisible(true);
}

/**
 * plot function 3d
 */
function plot3d(func, xrange, yrange){
    var x = DoubleArray.increment(xrange.start, xrange.step,
xrange.stop);
    var y = DoubleArray.increment(yrange.start, yrange.step,
yrange.stop);

    var name = func.toString();

    var z = (function(x, y){
        var dims = new Array(y.length, x.length);
        var r =
java.lang.reflect.Array.newInstance(java.lang.Double.TYPE, dims);
        for(var i = 0, len = x.length; i < len; i++){
            for(var j = 0, len2 = y.length; j < len2; j++){
                r[j][i] = func(x[i], y[j]);
            }
        }
        return r;
    })(x, y);

    var plot = new Plot3DPanel("SOUTH");
```

```
plot.addGridPlot(name, x, y, z);

var frame = new JFrame("plot function : "+name);
frame.setSize(600, 600);
frame.setContentPane(plot);
frame.setVisible(true);
}
```

使用 `jmathtool` 的 `Plot*DPanel` 产生新的 `Panel`, 然后创建新的 `JFrame` 对象, 并将 `Panel` 添加在 `JFrame` 上管理。

第十五章 服务器端的 JavaScript

15.1 node.js

15.1.1 node.js 简介

Node.js 是一个基于 Google V8 JavaScript 引擎的框架，它的设计目标为提供一个构建快速，可伸缩网络应用程序的平台。Node.js 是一个基于事件，非阻塞 I/O 的框架，可以很好的处理高并发的业务。这个在每个平台，每个语言中均有类似的对应，如 C 语言中的 libevent/libev，python 语言中的 twisted 等。

基于事件+非阻塞是保证 Node 保持高速的关键所在，与客户端的 JavaScript 类似，通过注册事件处理器来确保当关心的事件发生时进行回调，这样主线程就进入了 MainLoop。这样可以仅通过一个线程就处理大量的并发。但是这种模式的限制在于，每个事件的处理器不能太耗时，否则其他的事件要么排队等待，要么会丢失。

基于事件+非阻塞 IO 的流程大致如下图，当有数据可读的时候，select 将返回可读/写的描述符，然后才开始真正的 read/write 操作：

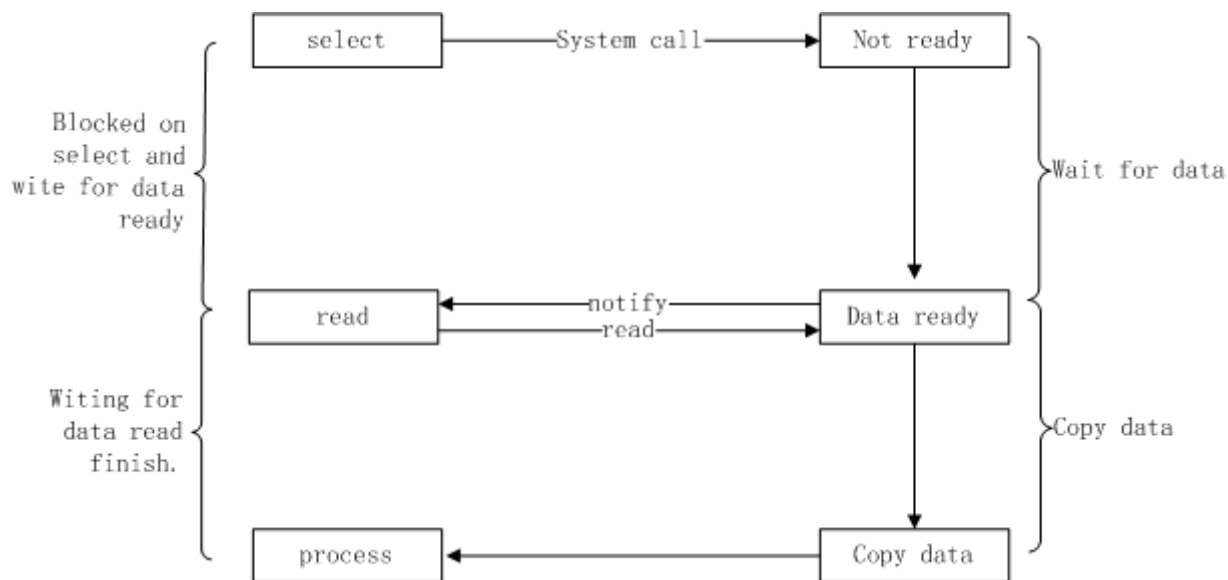


图 多路复用模式示意图

这种模型在监听多个客户端描述符时，会非常的高效，且占用的系统资源非常少。node.js 支持扩展。由于实现了 CommonJS 的一些规范，node.js 支持模块的 require 和 exports，从而使得代码更加模块化，更易于管理。

15.1.2 node.js 使用示例

我们可以通过一些简单的实例来查看 Node 如何提高开发服务器的效率,事实上,使用 Node 来编写一个简单的 echo 服务器,仅需要 6 行代码:

```
var net = require("net");

var server = net.createServer(function(socket) {
  socket.write("Echo server\n");
  socket.pipe(socket);
});

server.listen(8580, "10.111.43.117");
```

我们在 8580 端口启动一个 socket 服务器,它将客户端的请求原封不动的返回:

```
[juntao@rd117 ~]$ telnet 10.111.43.117 8580
Trying 10.111.43.117...
Connected to rd117 (10.111.43.117).
Escape character is '^]'.
Echo server
hello, echo
hello, echo
Are you really node?
Are you really node?
bye
bye
^]
telnet> quit
Connection closed.
```

加粗的为服务器端返回的数据。require 是 node 中的包管理机制, net 是一个内置的模块,用以提供原生的 socket 访问。net 的 createServer 接受一个参数,其类型为一个函数,当有连接到达时, node 会调用这个函数。

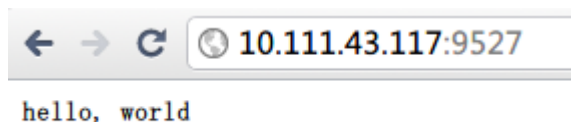
下面是一个简单的 http 服务器,当接收到客户端请求之后,返回 200 及一个字符串"hello,world"。

```
var http = require("http");

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type":"text/plain"});
  response.end("hello, world\n");
});
```

```
}).listen(9527, "10.111.43.117");  
  
console.log("http server running at http://10.111.43.117:9527");
```

运行效果如下:



`console` 是一个全局的对象, `console.log` 用以在控制台上打印日志信息。通过 `log`, 我们可以在后台监控客户端传递的数据。

最后来看一个传输文件的 Node 脚本:

```
var http = require("http"),  
    url = require("url"),  
    path = require("path"),  
    fs = require("fs");  
  
http.createServer(function(request, response) {  
  var uri = url.parse(request.url).pathname;  
  var filename = path.join(process.cwd(), uri);  
  
  path.exists(filename, function(exists) {  
    if(!exists) {  
      response.writeHead(404, {"Content-Type": "text/plain"});  
      response.write("404 Not Found\n");  
      return;  
    }  
  
    fs.readFile(filename, "binary", function(err, file) {  
      if(err) {  
        response.writeHead(500, {"Content-Type": "text/plain"});  
        response.write(err + "\n");  
        return;  
      }  
  
      response.writeHead(200);  
      response.write(file, "binary");  
      response.end();  
    });  
  });  
}).listen(8080);
```

```
console.log("File Server running at http://localhost:8080/");
```

其中使用到了很多 Node 的模块: `url`, `path`, `http`, `fs`。如果文件不存在, 则返回 404, 如果读文件出错, 则返回 500, 否则将文件内容写入 `response`, 并终止连接。这个 `http` 服务器的根为 `process.cwd()`, 即进程启动的目录, 这样我们可以在另一个终端中, 通过 `curl` 来请求上例中的 `echo.js`:

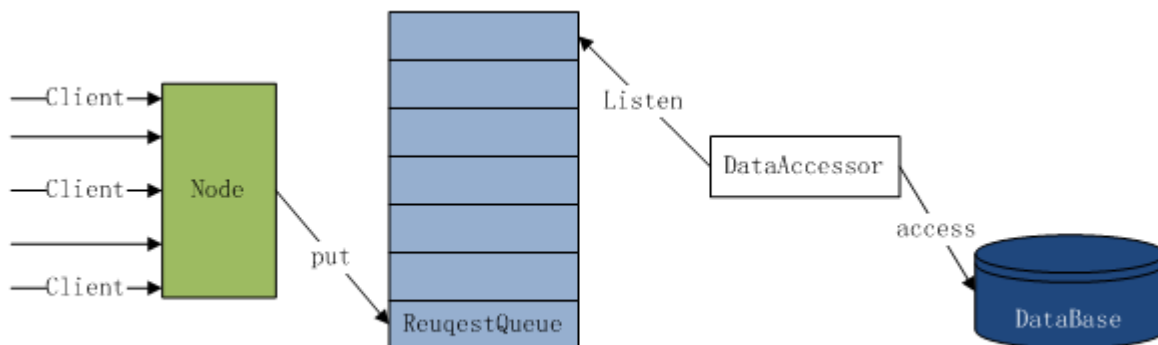
```
[juntao@rd117 ~]$ curl http://10.111.43.117:8080/echo.js
var net = require("net");

var server = net.createServer(function(socket) {
    socket.write("Echo server\n");
    socket.pipe(socket);
});

server.listen(8580, "10.111.43.117");
```

事实上, 使用 Node 作为一个间接层, 可以极大的提高服务器的并发能力, 将耗时的操作(数据库, I/O 等)作为独立的进程。然后使用 Node 的高并发能力将数据存入一个队列, 并通知该独立进程进行耗时的操作。这种模式事实上是将单线程-事件模型-非阻塞与多线程结合起来, 实践证明, 这种模式是处理高并发, 频繁业务很好的方式。

下图是一个典型的场景示意图:



再来看一个 `node.js` 中模块的使用, 比如我们建立了一个模块, 名叫 `producter.js`, 其中定义了一个变量和一个函数, 然后想要在另一个模块 `customer.js` 中访问这个变量和函数。`producter.js` 中, 定义了 `pname` 和 `pfunc`, 然后通过 `node.js` 提供的 `exports` 来将其公开:

```
var pname = "javascript-core";

function pfunc() {
```



```
    return "this is javascript-core";  
}
```

```
exports.pname = pname;  
exports.pfunc = pfunc;
```

然后在 `customer.js` 中，使用 `require` 来将这个模块导入：

```
var producer = require("./producer.js");  
  
console.log(producer.pname);  
var s = producer.pfunc();  
console.log(s);
```

运行结果如下：

```
javascript-core  
this is javascript-core
```

15.1.3 node.js 实例

在这个小节中，我们将实现一个使用 `node.js` 开发的 HTTP 服务器，这个服务器提供 REST 方式的 API 来对客户端的数据进行处理，简单起见，这个服务器仅处理 POST 的数据，同时，这个服务器仅处理三种字符串处理操作：

- 回显(echo)，将客户端发送的字符串原封不动的返回
- 转大写(upper)，将客户端发送的字符串转换为大写并返回
- 转小写(lower)，将客户端发送的字符串转换为小写并返回

访问方式为：

```
curl -X POST http://10.111.43.117:8080/upper -d "hello, world"
```

该服务器将返回“HELLO, WORLD”。

我们建立两个模块，`httpserver.js` 负责接受请求，并根据请求类型分发；而 `actions.js` 则负责实际的请求处理(字符串转换)。

`httpserver.js` 的原型如下，加载 `node.js` 内置的 `http` 模块，然后调用 `createServer`，并在 8080 端口监听：

```
var http = require("http");  
  
http.createServer(function(request, response){
```

```
    response.writeHead(200, {"Content-Type":"text/plain"});
    response.write("httpserver");
    response.end();
  }).listen(8080);
```

我们可以将 `createServer` 中的匿名函数抽取出来，使得代码更为简洁：

```
var http = require("http");

function handler(request, response){
  response.writeHead(200, {"Content-Type":"text/plain"});
  response.write("httpserver");
  response.end();
}

http.createServer(handler).listen(8080);
```

然后我们就可以放心的改造 `handler` 函数了，首先我们需要接受客户端发送的请求，这个在 `node.js` 中很容易实现，为 `request` 添加两个监听器：对于数据到达的事件，`node` 会触发“`data`”事件，当数据传输完成之后，会触发“`end`”事件。

```
function handler(request, response){
  var data = "";

  request.addListener("data", function(chunk){
    data += chunk;
  });

  request.addListener("end", function(){
    //invoke the real action handler
  });
}
```

当 `data` 事件触发后，陆续的将接收到的数据拼接在 `data` 变量中，当 `end` 事件触发时，我们就可以将数据传递给真实的处理函数来完成了。

客户端请求的 URL 格式为：`http://host:ip/path`，我们可以通过 `node.js` 的 `url` 模块来解析这个 `path`，这样我们就得到的操作类型。

```
var url = require("url");

function handler(request, response){
  var data = "";
  var path = url.parse(request.url).pathname;
```

```
request.addListener("data", function(chunk) {
  data += chunk;
});

request.addListener("end", function() {
  //RESUful
  if(path == "") {
    //do something
  }else if(path == ""){
    //do something else
  }else{
    //error
  }
});
}
```

然后，在 `end` 事件触发时，我们可以根据请求名称来进行分发。但是这样的写法并不好，如果请求类型变得很大的时候，代码中会有很多的 `if-else-if` 判断，不但代码会很丑陋，而且效率会降低。于是我们将定义一个请求名称和请求处理的映射表，当客户端的请求和映射表中的键匹配时，就调用这个键对应的值(这个值是一个函数)：

```
var error = function(path, response){}
var echo = function(data, response){}
var upper = function(data, response){}
var lower = function(data, response){}

var map = {
  "/echo" : echo,
  "/upper" : upper,
  "/lower" : lower,
  "error" : error
};
```

我们将这个映射表放在 `actions.js` 模块中，并将 `map` 暴露出来，然后在 `httpserver.js` 中 `require` 它即可。这样在 `end` 事件的处理函数中，就变得非常简单了：

```
request.addListener("end", function() {
  if(path in map){
    map[path](data, response);
  }else{
    map["error"](path, response);
  }
});
```

这时，我们的 `handler` 就有点名不副实了，它不负责处理具体的请求，仅仅是做分发，因此更名为 `dispatch`，这样我们的 `httpserver.js` 就已经开发完成了：

```
var http = require("http");
var url = require("url");
var actions = require("./actions");

map = actions.map;

function dispatch(request, response){
  var data = "";
  var path = url.parse(request.url).pathname;

  request.addListener("data", function(chunk){
    data += chunk;
  });

  request.addListener("end", function(){
    //RESUful
    if(path in map){
      map[path](data, response);
    }else{
      map["error"](path, response);
    }
  });
}

http.createServer(dispatch).listen(8080);
```

而 `actions.js` 中，我们的业务极为简单，只是对字符串进行大小写转换：

```
var error = function(path, response){
  response.writeHead(500, {"Content-Type" : "text/plain"});
  response.write("no such action handler for "+path);
  response.end();
}

var echo = function(data, response){
  response.writeHead(200, {"Content-Type" : "text/plain"});
  response.write(data);
  response.end();
}

var upper = function(data, response){
```

```
response.writeHead(200, {"Content-Type" : "text/plain"});
response.write(data.toUpper());
response.end();
}

var lower = function(data, response){
  response.writeHead(200, {"Content-Type" : "text/plain"});
  response.write(data.toLowerCase());
  response.end();
}

var map = {
  "echo" : echo,
  "upper" : upper,
  "lower" : lower,
  "error" : error
};

exports.map = map;
```

运行服务器:

```
node httpserver.js
```

然后使用 `curl` 进行测试:

```
$ curl -X POST http://10.111.43.117:8080/upper -d "hello, world"
HELLO, WORLD
```

我们可以很容易的对这个小型的服务器进行扩展，以支持更多的 `action`。

15.2 CouchDB

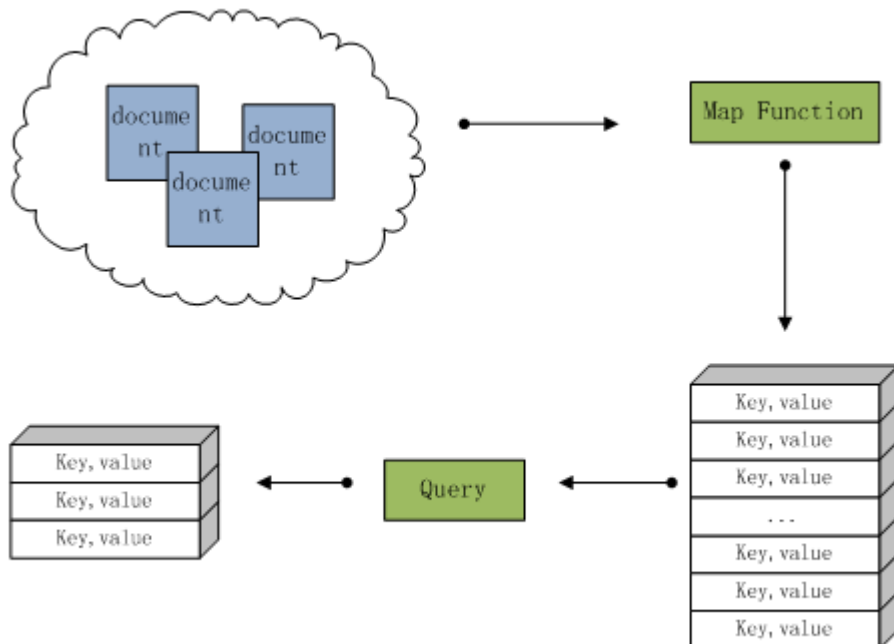
15.2.1 CouchDB 简介

CouchDB 是一个 Apache 的开源项目，它是一个面向文档的数据库系统。CouchDB 提供 REST 形式的 API 的访问，并以 JSON 作为数据交换格式。这一点使得 CouchDB 很容易被使用。事实上，支持 HTTP 的编程语言都可以作为 CouchDB 的客户端。它使用 MapReduce 方式来进行数据的访问和索引，在这一点上，函数式编程无疑更加合适：因为 map/reduce 本身就是函数式编程中的概念。

CouchDB 并无意取代关系数据库系统，而是作为一种支持方案呈现。使用关系型数据库，可以进行规整而复杂的关系运算。而在 CouchDB 中，并不存在模式(schema)的概念。比如一些消息系统，内容管理等，如果使用关系型数据库，当表结构变动时，可能会非常复杂，

而且容易出错。但是采用 CouchDB 的松散的 key/value 对来存放，则处理起来会容易的多。一般来说，在 CouchDB 中，文档之间是不应该存在关系的，每个文档都是独立的实体，但是可能具有某些共性，这些共性可以通过 map/reduce 来抽取。

CouchDB 的工作方式大致如下：



云朵表示一个数据库，其中蓝色的块表示文档，这些文档之间并无直接关联。通过 map 函数，将其中的某些项抽取为一些 key/value 对，然后根据 query 找到用户需要的结果集。由于 key/value 都可以是复杂的 JSON，因此它的表现力十分强大。

CouchDB 的文档采用 JSON 来表示，一个典型的文档示例如下：

```
{
  "_id": "46efa16cc220f6548696a6e6fc004463",
  "_rev": "1-37802625d560be4af0dc18155422fd2b",
  "language": [
    "Chinese",
    "English"
  ],
  "name": "Juntao"
}
```

每个文档都会带有两个隐式的属性“_id”和“_rev”。“_id”如果在创建文档时不显式的指定，则系统会分配一个 GUID 作为其值。“_rev”是一个内部的版本号，CouchDB 中包含一个版本控制系统，关于版本控制系统的细节，此处不做讨论。

15.2.1 CouchDB 使用

为了使用 JavaScript 来访问 CouchDB 提供的 REST 形式的 API, 我们使用 Rhino 引擎, 实现了一个简单的 `xmlhttprequest` 来与 CouchDB 的服务器进行通信, 当然使用其他语言如 C/Python 等也可以做到, 此处我们使用 JavaScript 本身。

另一个常用的工具是 `curl`, `curl` 是基于命令行的, 功能非常强大, 支持多种网络协议。CouchDB 的官方示例中, 很多次的用到了 `curl`, 但是我们只是在简单的测试时才会使用它。

这个小节中, 我们主要讨论如何通过 HTTP 请求完成下列动作:

- 创建数据库
- 创建文档
- 创建设计文档
- 过滤数据

创建数据库非常容易, 仅需要发送 PUT 请求到下列形式的 URL 即可:

```
http://localhost:5984/database_name
```

即可创建一个数据库, 名称为 `database_name`, 后续的操作即以此作为基准。

创建文档时, 需要指定数据库, 然后将文档(JSON 字符串)通过 HTTP 的 POST 方式发送到下列形式的 URL 上:

```
http://localhost:5984/database_name
```

需要发送的文档的形式如下:

```
{
  "name" : "Juntao",
  "language" : ["Chinese", "English"]
}
```

请求成功之后, 服务器会为此文档创建 id 及版本信息。可以通过 GET 下列形式的 URL 来获取文档:

```
http://localhost:5984/contacts/46efa16cc220f6548696a6e6fc004463
```

此处的 `46efa16cc220f6548696a6e6fc004463` 即为 CouchDB 为此文档生成的 GUID, 结果如下:

```
{
  "_id": "46efa16cc220f6548696a6e6fc004463",
  "_rev": "1-37802625d560be4af0dc18155422fd2b",
  "language": [
```

```

    "Chinese",
    "English"
  ],
  "name": "Juntao"
}

```

设计文档是一种特别的文档，用以表现应用程序的逻辑部分，其中包含视图(map/reduce 对)，附件定义等。CouchDB 是基于文档的，因此将应用程序作为文档也就不足为奇了。下面是一个设计文档的示例：

```

{
  "_id": "_design/filters",
  "_rev": "2-85428a15c29f7dc5b31bela0a3a217c9",
  "views": {
    "view_gt": {
      "map": "function(doc){...}",
      "reduce" : "function(keys, values, rereduce){}"
    },
    "view_lt": {
      "map": "function(doc){...}",
      "reduce" : "function(keys, values, rereduce){}"
    },
    "view_company": {
      "map": "function(doc){...}",
      "reduce" : "function(keys, values, rereduce){}"
    }
  },
  "shows" : {
    "show_sth" : "function(doc, req){}"
  },

  "_attachments" : {

  }
}

```

其中，**views** 中包含 **map/reduce** 的定义，用以对数据进行抽取，转换。**_attachments** 中包含文档的附件的定义。设计文档的路径均以 `"/database_name/_design"` 开头。我们这里着重讨论设计文档的 **views** 节：

每个 **view** 中包含一个 **map** 函数和一个可选的 **reduce** 函数。**map** 负责在所有的文档中收集信息，并生成 **key/value** 对，这个动作是并行的，可能同时有多个 **map** 函数在工作。所有的 **map** 函数都是用同一种 **hash** 算法，从而使得具有相同 **hash** 值的结果会被存储到一起。而 **reduce** 则负责将相同 **hash** 值的结果进行处理，并最终产生结果。**map** 函数相当于关系数据库中聚合查询的 **group-by** 子句，而 **reduce** 函数则相当于聚合函数，如统计，

求平均值等。

比如，我们可以将 `gt10` 这个 `view` 的 `map/reduce` 定义如下：

```
function (doc) {
  if (doc.name && doc.age) {
    if (doc.age > 10) {
      emit(doc.name, doc.age);
    }
  }
}

function (keys, values, rereduce) {
  return sum(values);
}
```

这个 `map` 将产生 `key` 为联系人名字，`value` 为联系人年龄的 `key/value` 对，而 `reduce` 将符合 `map` 的所有联系人的年龄加在一起。

15.2.3 CouchDB 实例

在这个小节中，我们将开发一个简单的 `CouchDB` 的应用，这个应用用以保存个人的联系人名单，然后创建一个设计文档，其中包含 3 个 `view`，分别用以查询联系人中年龄大于 10 岁的，小于 5 岁的，以及联系人所在的公司为“`Infonet`”的。

我们通过 `JavaScript` 来进行所有的这些操作，因此我们需要封装一些常用的操作：

- 创建数据库
- 添加联系人(文档)
- 添加视图
- 查询数据

创建数据库非常容易，以 `PUT` 方式访问需要建立的数据库的 `URL` 即可：

```
//PUT http://localhost:5984/contacts
function create_couch_db(host, port, dbname, handler) {
  host = host || "localhost";
  port = port || "5984";

  if (!dbname) {
    return false;
  }

  var xhr = new XMLHttpRequest();
```

```

var url = "http://" + host + ":" + port + "/" + dbname;

println(url);

xhr.open("PUT", url, false);
xhr.send(null);

function complete() {
    if(xhr.readyState == 4) {
        handler(xhr);
    }
}

xhr.onreadystatechange = complete;
}

```

此处的 XMLHttpRequest 是一个用 Java 做的简单实现，不支持 POST 数据。对于 POST 的请求，我们在随后再做介绍。create_couch_db 函数的最后一个参数是一个函数，当回调发生时调用此函数。

比如我们要在本机上部署的 CouchDB 创建一个名为“contacts”的数据库，当完成时我们将服务端返回的 JSON 解析并打印出其中包含的信息：

```

function complete(xhr) {
    var obj = JSON.parse(xhr.responseText);
    for(var item in obj) {
        if(typeof obj[item] == "object") {
            println(JSON.stringify(obj[item]));
        }
        println("key = " + item + ", value=" + obj[item]);
    }
}

create_couch_db("localhost", "5984", "contacts", complete);

```

调用之后，我们得到以下结果：

```

key = id, value=3e1a1b09536ee010352d79fd89000961
key = ok, value=true
key = rev, value=1-122ac58db939abb4aaf227accf5405b7

```

我们再来添加文档，首先定义几个联系人的信息：

```
var contacts = [
```

```
{
  name : "John",
  age : 28,
  address : "Somewhere in Kunming",
  interests : ["trevaling", "reading"],
  phone : "(871)-1234567"
},

{
  name : "Nelson",
  age : 35,
  company : "Infonet",
  address : "Eest Black-bridge"
},

{
  name : "Boycott",
  company : "Infonet"
},

{
  name : "Smith",
  age : 27,
  company : "Infonet",
},

{
  name : "Jim",
  address : "West Mountain",
  phone : "(871)-7654321"
},

{
  name : "SiJing",
  address : "HeBei",
  age : 3
},
];
```

然后在一个循环中调用添加文档的接口, 前面的那个 XMLHttpRequest 对象不支持 POST, 我们只好自行开发一个可以发送 HTTP 请求的 JavaScript 脚本:

```
for(var i = 0; i < contacts.length; i++){
  add_couch_doc_raw("http://localhost:5984", "contacts", \
```

```
        contacts[i], complete_raw);
    }
```

`add_couch_doc_raw` 函数将用以创建文档，并通过 `complete_raw` 函数将结果呈现出来：

```
function add_couch_doc_raw(uri, dbname, doc, handler){
    var url = new java.net.URL(uri+"/"+dbname);
    var data = JSON.stringify(doc);

    var con = url.openConnection();
    con.setDoOutput(true);
    con.setRequestProperty("Content-Type", "application/json");
    var writer = new java.io.OutputStreamWriter(con.getOutputStream());
    writer.write(data);
    writer.flush();
    writer.close();

    var reader = new java.io.BufferedReader(
        new java.io.InputStreamReader(con.getInputStream()));

    var line, text='';
    while((line = reader.readLine()) != null){
        text += line;
    }

    reader.close();

    if(handler){
        handler(text);
    }
}
```

完成之后的 `complete` 接收服务端的 JSON 字符串：

```
function complete_raw(raw){
    var obj = JSON.parse(raw);
    for(var item in obj){
        println("key = "+item+", value="+obj[item]);
    }
}
```

运行之后，服务器将返回结果：

```

key = id, value=46efa16cc220f6548696a6e6fc0044f4
key = ok, value=true
key = rev, value=1-485b14c13e90cfad44d539a3860dc1f3
key = id, value=46efa16cc220f6548696a6e6fc004f83
key = ok, value=true
key = rev, value=1-a22f80d27046f1b46ae218788747ffdb
key = id, value=46efa16cc220f6548696a6e6fc005855
key = ok, value=true
key = rev, value=1-fc222c18f52f463c44ac48328b10385c

```

好了，现在我们可以通过 CouchDB 的 web 界面来查看刚才创建的数据库以及数据库中的文档了：

The screenshot shows the CouchDB web interface for a database named 'contacts'. The interface includes a navigation bar with 'Overview' and 'contacts', and a toolbar with options like 'New Document', 'Security...', 'Compact & Cleanup...', and 'Delete Database...'. A table displays the following documents:

Key ▲	Value
"3e1a1b09536ee010352d79fd89000961" ID: 3e1a1b09536ee010352d79fd89000961	{rev: "1-122ac58db939abb4aaf227accf5405b7"}
"46efa16cc220f6548696a6e6fc004463" ID: 46efa16cc220f6548696a6e6fc004463	{rev: "1-37802625d560be4af0dc18155422fd2b"}
"46efa16cc220f6548696a6e6fc0044f4" ID: 46efa16cc220f6548696a6e6fc0044f4	{rev: "1-485b14c13e90cfad44d539a3860dc1f3"}
"46efa16cc220f6548696a6e6fc004f83" ID: 46efa16cc220f6548696a6e6fc004f83	{rev: "1-a22f80d27046f1b46ae218788747ffdb"}
"46efa16cc220f6548696a6e6fc005855" ID: 46efa16cc220f6548696a6e6fc005855	{rev: "1-fc222c18f52f463c44ac48328b10385c"}
"46efa16cc220f6548696a6e6fc00594f" ID: 46efa16cc220f6548696a6e6fc00594f	{rev: "1-13dbbee75759a603409dba106b1b7b5e"}
"46efa16cc220f6548696a6e6fc0065c5" ID: 46efa16cc220f6548696a6e6fc0065c5	{rev: "1-57c3c47a16eb318cf8c1c88b72acd3c5"}
"46efa16cc220f6548696a6e6fc0071bf" ID: 46efa16cc220f6548696a6e6fc0071bf	{rev: "1-6d0cef5685983ff287978ffd4bf85bd6"}
"_design/filters" ID: _design/filters	{rev: "2-85428a15c29f7dc5b31be1a0a3a217c9"}

下面我们来尝试创建一个设计文档，并在设计文档中添加一些 view：

```

{
  "_id" : "_design/filters",
  "views" : {
    "gt10" : {
      "map" : "function(doc) {
        if(doc.name && doc.age){
          if(doc.age > 10){
            emit(doc.name, doc.age);
          }
        }
      }
    }
  }
}

```

```
        }"
    },
    "lt5" : {
        "map" : "function(doc) {
            if(doc.name && doc.age){
                if(doc.age < 5 ){
                    emit(doc.name, doc.age);
                }
            }
        }"
    },
    "infonet" : {
        "map" : "function(doc) {
            if(doc.name && doc.company){
                if(doc.company == 'Infonet'){
                    emit(doc.name, doc.company)
                }
            }
        }"
    }
}
}
```

我们为设计文档添加了三个 **view**，其中的 **map** 将用作过滤器，它将对所有的文档进行抽取转换。将这个设计文档保存为文本，并通过 **curl** 上传该文档：

```
$ curl -X PUT http://localhost:5984/contacts/_design/filters \
-d @contacts-filter.js
```

这个设计文档的名称叫做 **filters**，因此完整的路径为 `/contacts/_design/filters`。

现在我们就可以使用 **view** 来进行数据的过滤了，比如我们要找出所有联系人中，在 **Infonet** 公司工作的人：

Key ▲	Value
"Boycott" ID: 46efa16cc220f6548696a6e6fc005855	"Infonet"
"Nelson" ID: 46efa16cc220f6548696a6e6fc004f83	"Infonet"
"Smith" ID: 46efa16cc220f6548696a6e6fc00594f	"Infonet"

Showing 1-3 of 3 rows ← Previous Page | Rows per page: 10 | Next Page →

当然，我们可以通过自己编写 JavaScript 来访问这些数据：

```
function filter_couch_data(host, port, dbname, doc, view, handler){
    host = host || "localhost";
    port = port || "5984";

    if(!dbname || !doc || !view){
        return false;
    }

    var xhr = new XMLHttpRequest();
    var url =
"http://" + host + ":" + port + "/" + dbname + "/" + doc + "/" + view;

    xhr.open("GET", url, false);

    function complete(){
        if(xhr.readyState == 4){
            handler(xhr);
        }
    }

    xhr.onreadystatechange = complete;

    xhr.send(null);
}
```

访问 `http://localhost:5984/contacts/_design/filters/_view/infonet` 即可：

```
filter_couch_data("localhost", "5984", "contacts",
```

```
"filters", "infonet", complete);
```

当插入新的数据时，`map` 可以自动的将符合条件的 `key/value` 对放入 `view`，比如：

```
var newguy = {  
  age : 30,  
  name : "Fire"  
};
```

```
add_couch_doc_raw("http://localhost:5984", "contacts", newguy,  
complete_raw);
```

```
filter_couch_data("localhost", "5984", "contacts",  
  "filters", "gt10", complete);
```

我们新添加了一个 `newguy`，他的年龄为 `30`，然后我们访问 `gt10` 这个 `view`，可以看到，符合这个条件的记录多了一条。这一点与关系数据库中的 `query` 的效果相同。

附录一 一些 JavaScript 技巧

创建对象

在 JavaScript 中, 存在一种强大而灵活的机制用以创建各种对象, 这就是对象字面量, 我们应该尽量避免使用 `new` 操作符来创建 `Object`, `Array` 等对象, 字面量更方便, 也更小巧。除非你需要从原型链上继承一些方法和属性, 那样的话, 使用 `new` 操作符则不可避免, 幸运的是, 这种情况已经不是很常见, 至少在 Web 开发上。

比如, 我们可以用 `{}` 来替代 `new Object()` 操作, 用 `[]` 代替 `new Array()`, 同样可以用 `/pattern/` 代替 `new RegExp("pattern")` 等等。为什么我们提倡这么做呢? 事实上, `new` 关键字在 JavaScript 表示的含义与在传统的面向对象的语言中的含义完全不同。与其留着这样一个可以表达两种意思的操作符存在, 不如我们干脆不用它, 设想你的代码交给一个只有 OO 编程经验的同事来读, 结果除了让他误入歧途外几乎没什么好处。

访问对象的属性

`eval` 是 JavaScript 提供给程序员的一个很有用的函数, 它也是 JavaScript 作为函数式语言存在的一个标志。比如, 我们使用 `Ajax` 从后台请求了一部分数据, 这些数据有一部分是 HTML 代码, 用于补充页面上的一个面板的内容, 另一部分包含一些 JavaScript 代码, 用来控制这些 HTML 代码应该如何展示。不幸的是, 从后台直接返回的代码不会自动被执行, 这些代码以字符串的形式存在, 因此我们需要对这些字符串求值(`evaluate`), 这里就可以使用 `eval` 函数, 将 JavaScript 代码串传入即可:

```
eval('(' + jscode + ')');
```

这样我们就可以动态的执行“字符串”了。

遍历对象

JavaScript 可以很容易对对象进行遍历(数组也是一个特殊的对象), 一般而言, 对于数组, 可以使用:

```
var array = ['apple', 'borland', 'cisco', 'dell'];

for(var i = 0; i < array.length; i++){
    print(array[i]);
}
```

来进行遍历, 而对于对象, 则使用 `for..in` 进行遍历:

```
var obj = {
  a : 'apple',
  b : 'borland',
  c : 'cisco',
  d : 'dell'
}

for(item in obj){
  print(item+ " = "+obj[item]);
}
```

虽然 `for..in` 同样可以用以遍历数组，但是不推荐使用。因为数组对象可能被添加了附加的属性，使用 `for..in` 会将所有的元素和属性都遍历到。这可能并不是我们所需要的：

```
var array = ["one", "two", "there", "four"];

array.extern = "external";

for(item in array){
  print(item+ " = "+array[item]);
}
```

会将 `array` 上的 `extern` 属性打印：

```
0 = one
1 = two
2 = there
3 = four
extern = external
```

名称空间

为了使得我们的代码模块尽可能的和其他人的代码发生冲突(变量命名冲突，函数命名冲突等等)，我们需要引入名称空间的概念，虽然 `JavaScript` 本身没有定义名称空间，但是使用 `JavaScript` 的对象本身的一些特性，我们可以模拟出名称空间来。

```
var jscore = jscore || {};
jscore.util = jscore.util || {};
jscore.util.common = {
  raiseError : function(message) {
    throw new Error(message);
  },
  showMessage : function(info) {
```

```

    print(info);
  }
}

```

我们建立了一个 `jscore` 对象，`jscore` 有个 `util` 属性，同样 `util` 也是一个对象，而 `common` 对象则为 `util` 的一个属性。当我们想要访问 `raiseError` 或者 `showMessage` 两个函数时，必须加上 `jscore.util.common` 前缀才可以。

```

jscore.util.common.raiseError("An I/O Error occured");
jscore.util.common.showMessage("This is a message for you");

```

否则解释器会提示，函数未定义的错误。通过这种方式，我们可以尽可能的避免别人使用了我们的变量名或者相反，而且这种记法与 `Java/python` 这些语言中的包概念很相似。所以建议大家使用这种规则来管理自己的包。

当然，你可能发现这种方式虽然较好，但是写起来比较麻烦，这一点你需要仔细权衡，因为当你的项目变大比较大，而且模块划分越来越细，你就会发现这样的麻烦比起维护变量冲突来说，简直不算什么。

我们将 `namespace` 的过程再简化一下，提供一个方法出来，这样用户可以使用字符串来创建一个名字空间，这样不是更方便吗？

```

var __global__ = this;

var jscore = jscore || {};
jscore.util = jscore.util || {};
jscore.util.namespace = function(name){
  var parent = __global__;
  var array = name.split(".");

  for(var i = 0, len = array.length; i < len; i++){
    parent[array[i]] = parent[array[i]] || {};
    parent = parent[array[i]];
  }
}

```

有了 `jscore.util.namespace` 方法，我们可以很快速的定义一个新的名字空间，比如一个与 `UI` 相关的工具包，名叫 `uikit`，`uikit` 下有个名叫 `ui` 的子包，用以封装真是的绘画逻辑：

```

jscore.util.namespace("uikit.ui");//定义新的名称空间

uikit.ui.showMessage = function(msg){
  var tag = new Date().toString();
  print("[ "+tag+" ] "+msg);
} //该名称空间下的一个方法

```

```
uikit.ui.showMessage("initializing the net-work envirnoment...");  
uikit.ui.showMessage("initializing the graphcis envirnoment...");
```

测试之:

```
[Apr 25 2010 17:08:25] initializing the net-work envirnoment...  
[Apr 25 2010 17:08:25] initializing the graphcis envirnoment...
```

附录二 使用 graphviz 绘图

graphviz 简介

本文介绍一个高效而简洁的绘图工具 graphviz。graphviz 是贝尔实验室开发的一个开源的工具包，它使用一个特定的 DSL(领域特定语言):dot 作为脚本语言，然后使用布局引擎来解析此脚本，并完成自动布局。graphviz 提供丰富的导出格式，如常用的图片格式，SVG，PDF 格式等。

graphviz 中包含了众多的布局器：

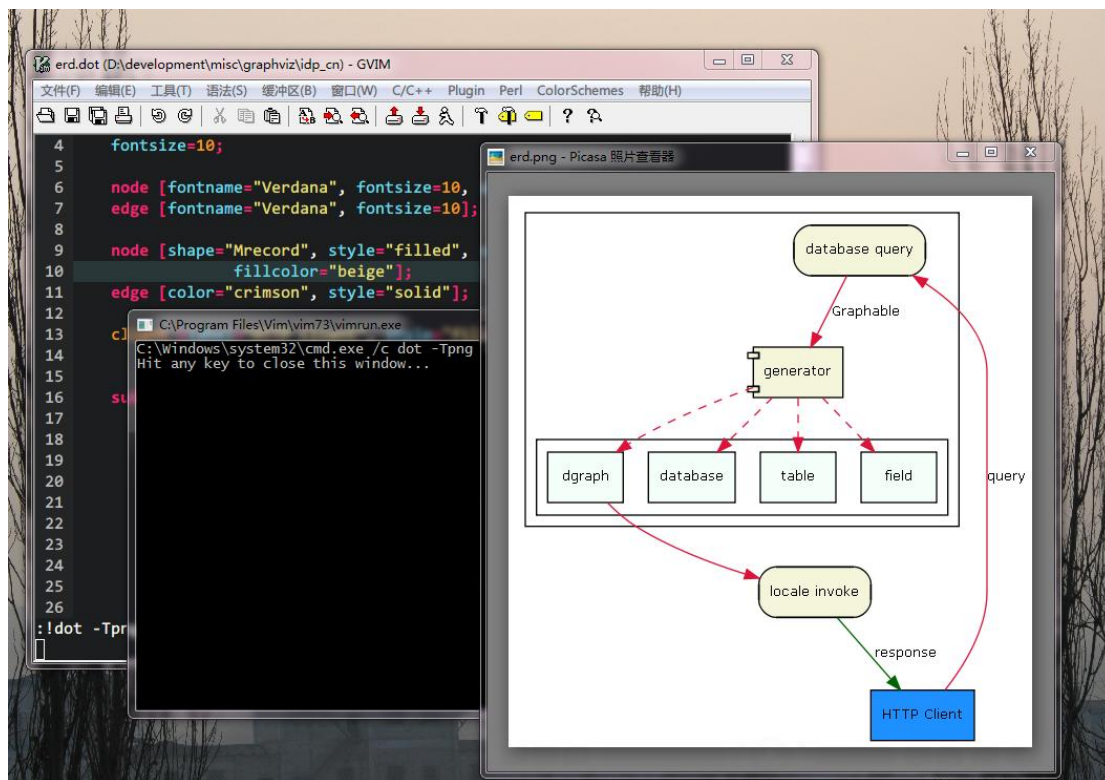
1. dot 默认布局方式，主要用于有向图
2. neato 基于 spring-model(又称 force-based)算法
3. twopi 径向布局
4. circo 圆环布局
5. fdp 用于无向图

graphviz 的设计初衷是对有向图/无向图等进行自动布局，开发人员使用 dot 脚本定义图形元素，然后选择算法进行布局，最终导出结果。

首先，在 dot 脚本中定义图的顶点和边，顶点和边都具有各自的属性，比如形状，颜色，填充模式，字体，样式等。然后使用合适的布局算法进行布局。布局算法除了绘制各个顶点和边之外，需要尽可能的将顶点均匀的分布在画布上，并且尽可能的减少边的交叉(如果交叉过多，就很难看清楚顶点之间的关系了)。所以使用 graphviz 的一般流程为：

1. 定义一个图，并向图中添加需要的顶点和边
2. 为顶点和边添加样式
3. 使用布局引擎进行绘制

一旦熟悉这种开发模式，就可以快速的将你的想法绘制出来。配合一个好的编辑器(vim/emacs)等，可以极大的提高开发效率，与常见的 GUI 应用的所见即所得模式对应，此模式称为所思即所得。比如在我的机器上，使用 vim 编辑 dot 脚本，然后将 F8 映射为调用 dot 引擎去绘制当前脚本，并打开一个新的窗口来显示运行结果：



对于开发人员而言，经常会用到的图形绘制可能包括：函数调用关系，一个复杂的数据结构，系统的模块组成，抽象语法树等。

基础知识

graphviz 包含 3 中元素，图，顶点和边。每个元素都可以具有各自的属性，用来定义字体，样式，颜色，形状等。下面是一些简单的示例，可以帮助我们快速的了解 graphviz 的基本用法。

第一个 graphviz 图

比如，要绘制一个有向图，包含 4 个节点 a,b,c,d。其中 a 指向 b，b 和 c 指向 d。可以定义下列脚本：

```
digraph abc{
  a;
  b;
  c;
  d;

  a -> b;
  b -> d;
  c -> d;
}
```

```
c -> d;  
}
```

使用 dot 布局方式，绘制出来的效果如下：

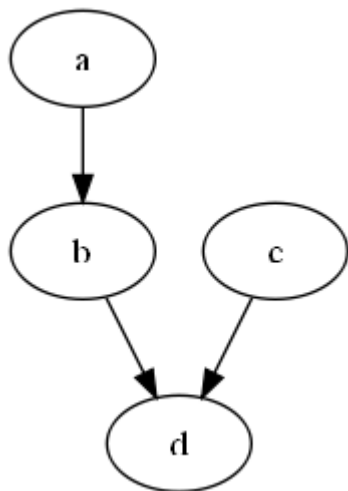


图 1

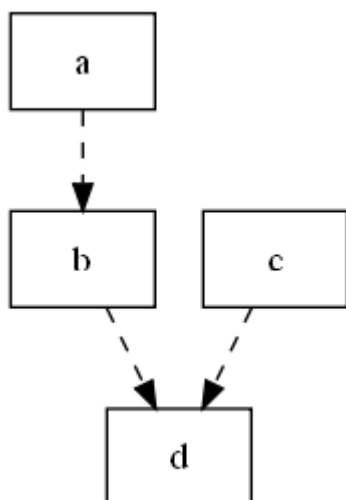
默认的顶点中的文字为定义顶点变量的名称，形状为椭圆。边的默认样式为黑色实线箭头，我们可以在脚本中做一下修改，将顶点改为方形，边改为虚线。

定义顶点和边的样式

在 `digraph` 的花括号内，添加顶点和边的新定义：

```
node [shape="record"];  
edge [style="dashed"];
```

则绘制的效果如下：



进一步修改顶点和边样式

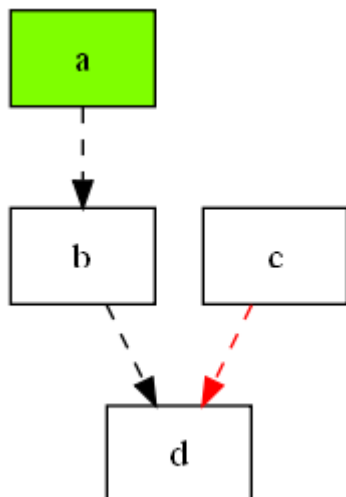
进一步，我们将顶点 **a** 的颜色改为淡绿色，并将 **c** 到 **d** 的边改为红色，脚本如下：

```
digraph abc{
  node [shape="record"];
  edge [style="dashed"];

  a [style="filled", color="black", fillcolor="chartreuse"];
  b;
  c;
  d;

  a -> b;
  b -> d;
  c -> d [color="red"];
}
```

绘制的结果如下：



应当注意到，顶点和边都接受属性的定义，形式为在顶点和边的定义之后加上一个由方括号括起来的 **key-value** 列表，每个 **key-value** 对由逗号隔开。如果图中顶点和边采用统一的风格，则可以在图定义的首部定义 **node**, **edge** 的属性。比如上图中，定义所有的顶点为方框，所有的边为虚线，在具体的顶点和边之后定义的属性将覆盖此全局属性。如特定与 **a** 的绿色，**c** 到 **d** 的边的红色。

子图的绘制

graphviz 支持子图，即图中的部分节点和边相对对立(软件的模块划分经常如此)。比如，我们可以将顶点 **c** 和 **d** 归为一个子图：

```

digraph abc{
  node [shape="record"];
  edge [style="dashed"];

  a [style="filled", color="black", fillcolor="chartreuse"];
  b;

  subgraph cluster_cd{
    label="c and d";
    bgcolor="mintcream";
    c;
    d;
  }

  a -> b;
  b -> d;
  c -> d [color="red"];
}

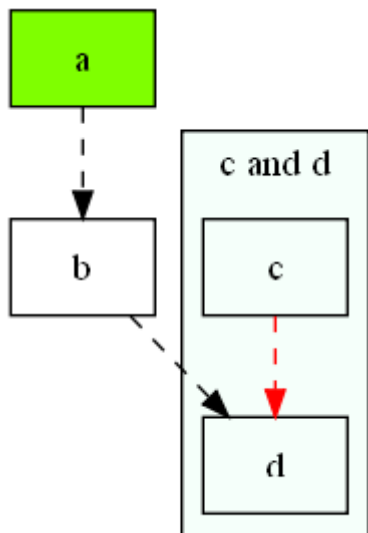
```

```

}

```

将 c 和 d 划分到 `cluster_cd` 这个子图中，标签为“c and d”，并添加背景色，以方便与主图区分开，绘制结果如下：



应该注意的是，子图的名称必须以 `cluster` 开头，否则 `graphviz` 无法识别。

数据结构的可视化

实际开发中，经常用到的是对复杂数据结构的描述，`graphviz` 提供完善的机制来绘制此类图形。

一个 hash 表的数据结构

比如一个 hash 表的内容，可能具有下列结构：

```

struct st_hash_type {
    int (*compare) ();
    int (*hash) ();
};

struct st_table_entry {
    unsigned int hash;
    char *key;
    char *record;
    st_table_entry *next;
};

```

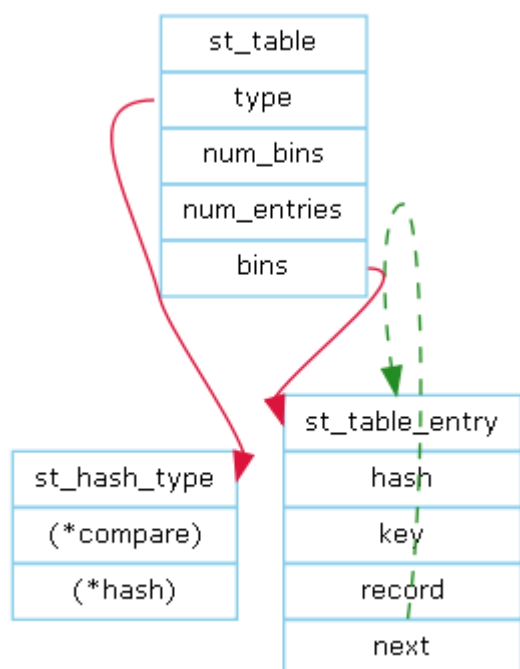
```

struct st_table {
    struct st_hash_type *type;
    int num_bins;           /* slot count */
    int num_entries;      /* total number of entries */
    struct st_table_entry **bins; /* slot */
};

```

绘制 hash 表的数据结构

从代码上看，由于结构体存在引用关系，不够清晰，如果层次较多，则很难以记住各个结构之间的关系，我们可以通过下图来更清楚的展示：



脚本如下：

```

digraph st2{
    fontname = "Verdana";
    fontsize = 10;

    rankdir=TB;

    node [fontname = "Verdana", fontsize = 10, \
color="skyblue", shape="record"];
    edge [fontname = "Verdana", fontsize = 10, \

```

```

color="crimson", style="solid"];

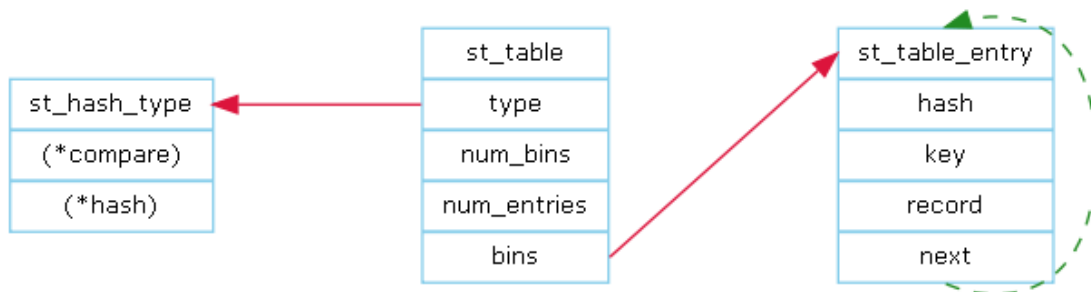
st_hash_type \
[label="{<head>st_hash_type|(*compare)|(*hash)}"];
st_table_entry \
[label="{<head>st_table_entry|hash|key|record|<next>next}"];
st_table \
[label="{st_table|<type>type|num_bins|num_entries|<bins>bins}"];

st_table:bins -> st_table_entry:head;
st_table:type -> st_hash_type:head;
st_table_entry:next -> st_table_entry:head \
[style="dashed", color="forestgreen"];
}

```

应该注意到，在顶点的形状为“record”的时候，label 属性的语法比较奇怪，但是使用起来非常灵活。比如，用竖线“|”隔开的串会在绘制出来的节点中展现为一条分隔符。用“<>”括起来的串称为锚点，当一个节点具有多个锚点的时候，这个特性会非常有用，比如节点 `st_table` 的 `type` 属性指向 `st_hash_type`，第 4 个属性指向 `st_table_entry` 等，都是通过锚点来实现的。

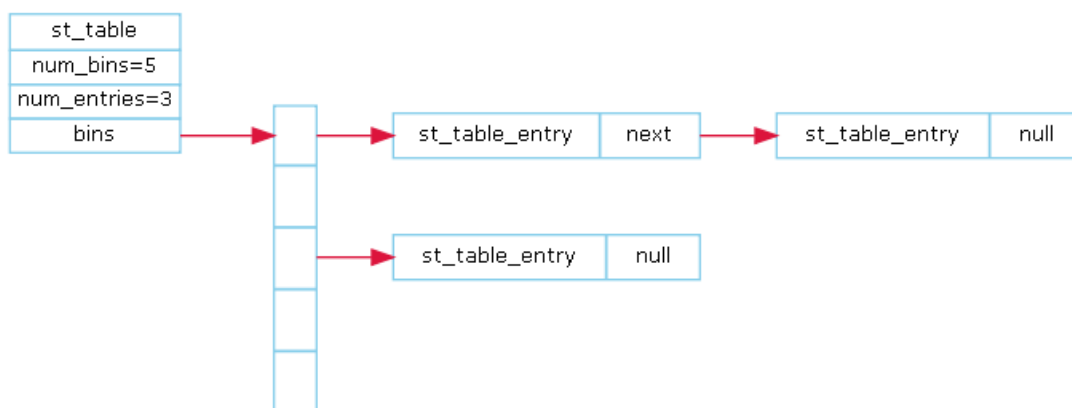
我们发现，使用默认的 dot 布局后，绿色的这条边覆盖了数据结构 `st_table_entry`，并不美观，因此可以使用别的布局方式来重新布局，如使用 circo 算法：



则可以得到更加合理的布局结果。

hash 表的实例

另外，这个 hash 表的一个实例如下：



脚本如下:

```
digraph st{
  fontname = "Verdana";
  fontsize = 10;

  rankdir=LR;
  rotate=90;

  node [ shape="record", width=.1, height=.1];

  node [fontname = "Verdana", fontsize = 10, \
  color="skyblue", shape="record"];
  edge [fontname = "Verdana", fontsize = 10, \
  color="crimson", style="solid"];

  node [shape="plaintext"];
  st_table [label=<
    <table border="0" cellborder="1" cellspacing="0" align="left">
      <tr>
        <td>st_table</td>
      </tr>
      <tr>
        <td>num_bins=5</td>
      </tr>
      <tr>
        <td>num_entries=3</td>
      </tr>
      <tr>
        <td port="bins">bins</td>
      </tr>
    </table>
  >];
```

```

node [shape="record"];
num_bins [label=" <b1> | <b2> | <b3> | <b4> | <b5> ", height=2];

node[ width=2 ];
entry_1 [label="{<e>st_table_entry|<next>next}"];
entry_2 [label="{<e>st_table_entry|<next>null}"];
entry_3 [label="{<e>st_table_entry|<next>null}"];

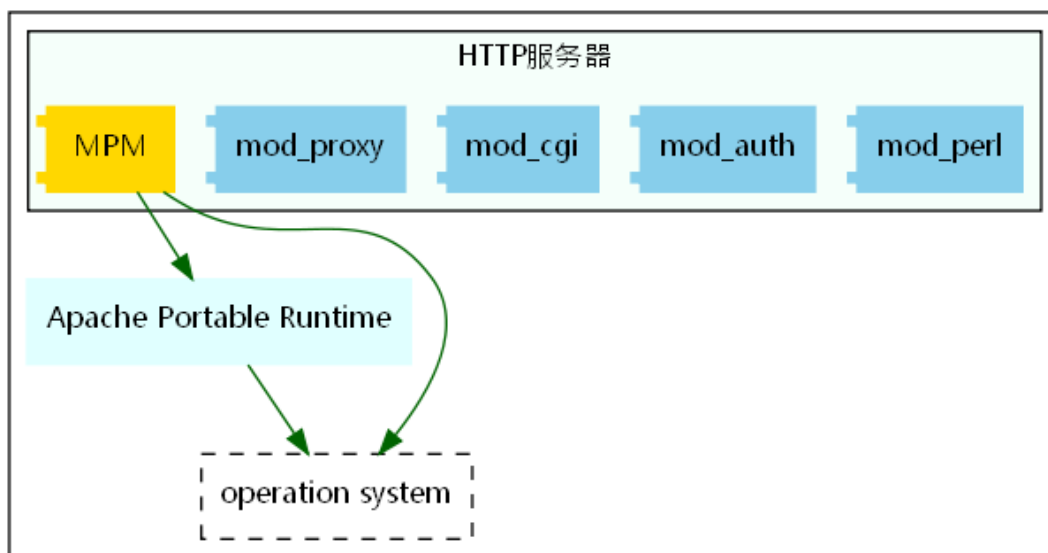
st_table:bins -> num_bins:b1;
num_bins:b1 -> entry_1:e;
entry_1:next -> entry_2:e;
num_bins:b3 -> entry_3:e;
}

```

上例中可以看到，节点的label属性支持类似于HTML语言中的TABLE形式的定义，通过行列的数目来定义节点的形状，从而使得节点的组成更加灵活。

软件模块组成图

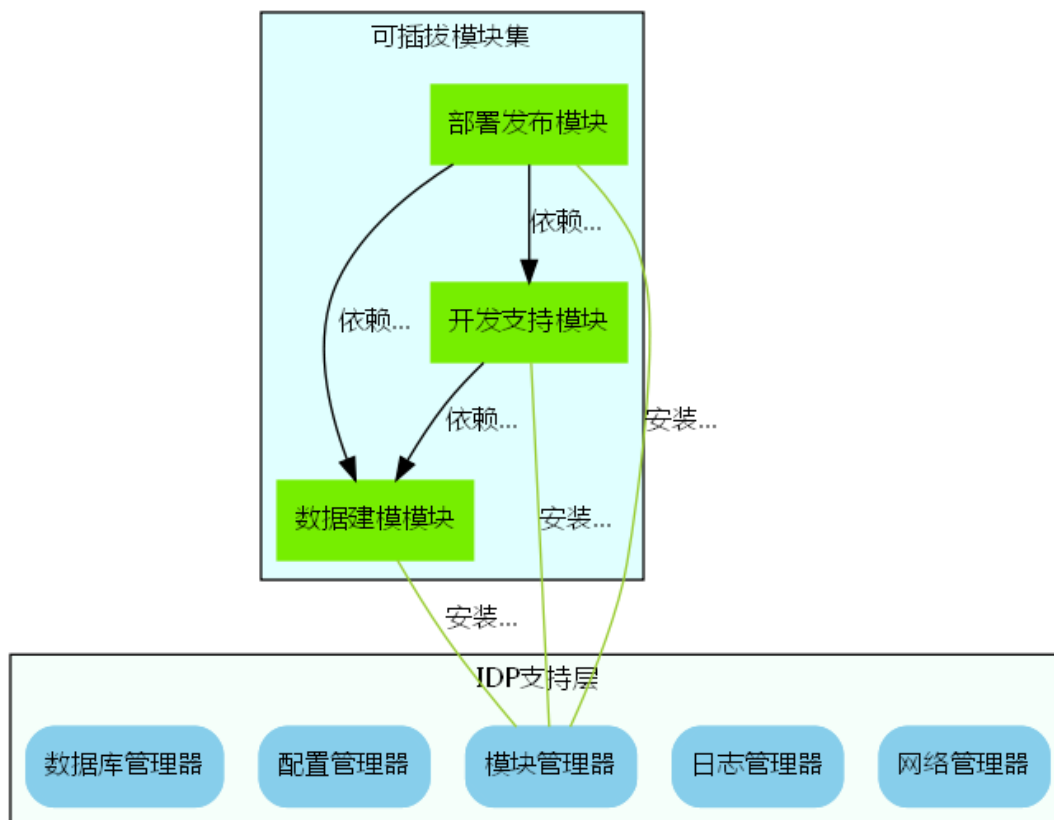
Apache httpd 模块关系



模块组成关系

在实际的开发中，随着系统功能的完善，软件整体的结构会越来越复杂，通常开发人员会将软件划分为可理解的多个子模块，各个子模块通过协作，完成各种各样的需求。

下面有个例子，是在一个框架设计时的一个草稿：



IDP 支持层为一个相对独立的子系统，其中包括如数据库管理器，配置信息管理等模块，另外为了提供更大的灵活性，将很多其他的模块抽取出来作为外部模块，而支持层提供一个模块管理器，来负责加载/卸载这些外部的模块集合。

这些模块间的关系较为复杂，并且有部分模块关系密切，应归类为一个子系统中，上图对应的 dot 脚本为：

```
digraph idp_modules{
    rankdir=TB;
    fontname = "Microsoft YaHei";
    fontsize = 12;

    node [ fontname = "Microsoft YaHei", fontsize = 12, \
    shape = "record" ];
    edge [ fontname = "Microsoft YaHei", fontsize = 12 ];
```

```
subgraph cluster_sl{
    label="IDP支持层";
    bgcolor="mintcream";
    node [shape="Mrecord", color="skyblue", style="filled"];
    network_mgr [label="网络管理器"];
    log_mgr [label="日志管理器"];
    module_mgr [label="模块管理器"];
    conf_mgr [label="配置管理器"];
    db_mgr [label="数据库管理器"];
};

subgraph cluster_md{
    label="可插拔模块集";
    bgcolor="lightcyan";
    node [color="chartreuse2", style="filled"];
    mod_dev [label="开发支持模块"];
    mod_dm [label="数据建模模块"];
    mod_dp [label="部署发布模块"];
};

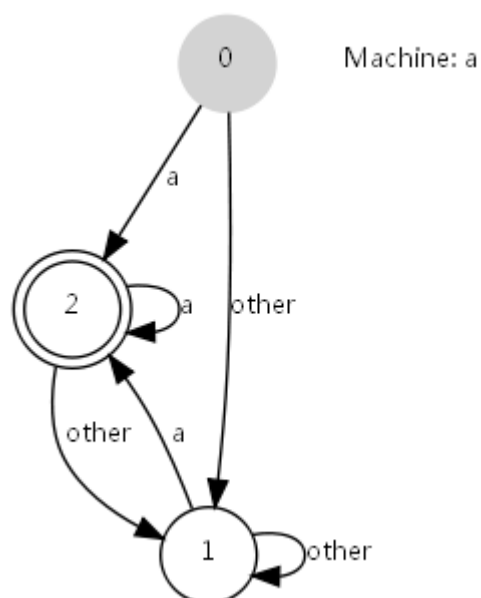
mod_dp -> mod_dev [label="依赖..."];
mod_dp -> mod_dm [label="依赖..."];
mod_dp -> module_mgr [label="安装...", \
color="yellowgreen", arrowhead="none"];

mod_dev -> mod_dm [label="依赖..."];
mod_dev -> module_mgr [label="安装...", \
color="yellowgreen", arrowhead="none"];

mod_dm -> module_mgr [label="安装...", \
color="yellowgreen", arrowhead="none"];
}
```


状态图

有限自动机示意图



上图是一个简易有限自动机，接受 **a** 及 **a** 结尾的任意长度的串。其脚本定义如下：

```
digraph automata_0 {
    size = "8.5, 11";
    fontname = "Microsoft YaHei";
    fontsize = 10;

    node [shape = circle, fontname = "Microsoft YaHei", fontsize = 10];
    edge [fontname = "Microsoft YaHei", fontsize = 10];

    0 [ style = filled, color=lightgrey ];
    2 [ shape = doublecircle ];

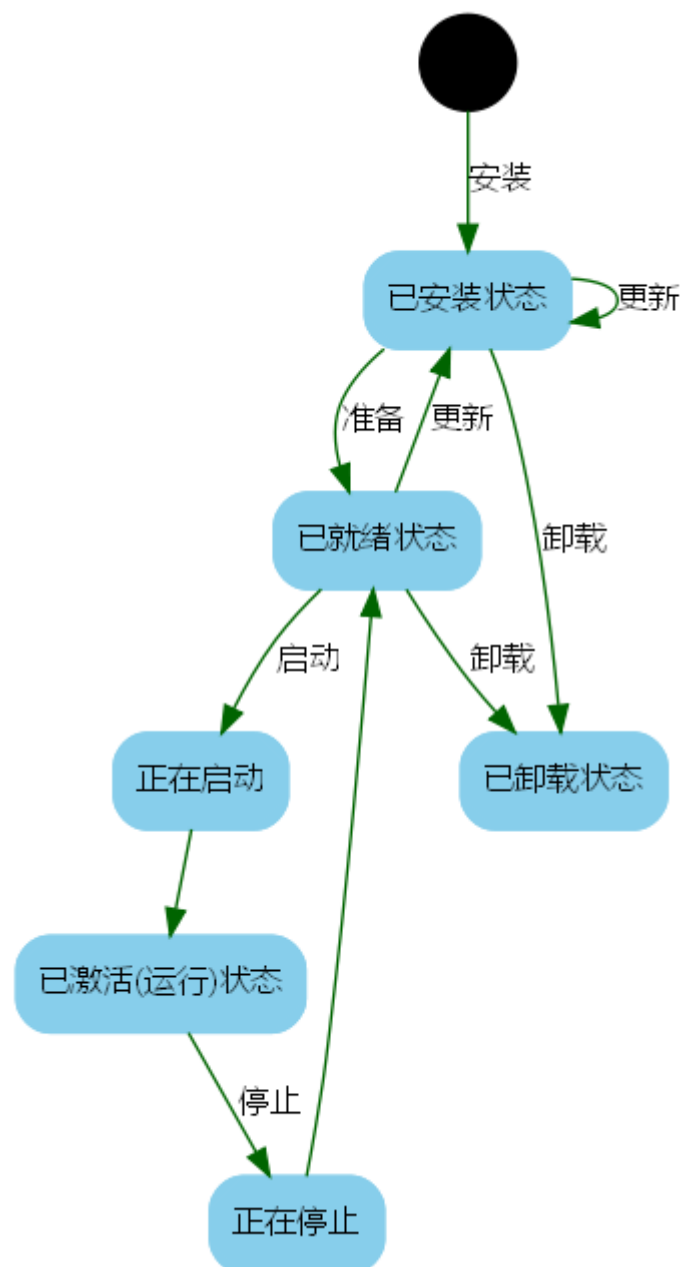
    0 -> 2 [ label = "a " ];
    0 -> 1 [ label = "other " ];
    1 -> 2 [ label = "a " ];
    1 -> 1 [ label = "other " ];
    2 -> 2 [ label = "a " ];
    2 -> 1 [ label = "other " ];

    "Machine: a" [ shape = plaintext ];
}
```

形状值为 `plaintext` 的表示不用绘制边框，仅展示纯文本内容，这个在绘图中，绘制指示性的文本时很有用，如上图中的“Machine: a”。

OSGi 中模块的生命周期图

OSGi 中，模块具有生命周期，从安装到卸载，可能的状态具有已安装，已就绪，正在启动，已启动，正在停止，已卸载等。如下图所示：



对应的脚本如下：

```
digraph module_lc{
  rankdir=TB;
  fontname = "Microsoft YaHei";
  fontsize = 12;

  node [fontname = "Microsoft YaHei", fontsize = 12, \
    shape = "Mrecord", color="skyblue", style="filled"];
  edge [fontname = "Microsoft YaHei", fontsize = 12, color="darkgreen" ];

  installed [label="已安装状态"];
  resolved [label="已就绪状态"];
  uninstalled [label="已卸载状态"];
  starting [label="正在启动"];
  active [label="已激活(运行)状态"];
  stopping [label="正在停止"];

  start [label="", shape="circle", width=0.5, fixedsize=true, \
    style="filled", color="black"];

  start -> installed [label="安装"];
  installed -> uninstalled [label="卸载"];
  installed -> resolved [label="准备"];
  installed -> installed [label="更新"];

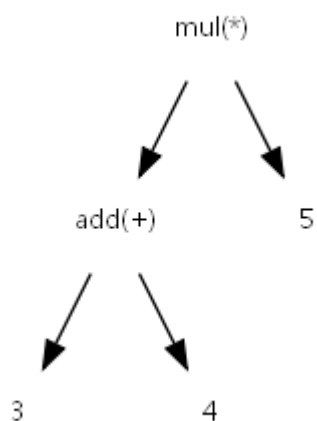
  resolved -> installed [label="更新"];
  resolved -> uninstalled [label="卸载"];
  resolved -> starting [label="启动"];

  starting -> active [label=""];
  active -> stopping [label="停止"];
  stopping -> resolved [label=""];
}
```

其他实例

一棵简单的抽象语法树(AST)

表达式 $(3+4)*5$ 在编译时期，会形成一棵语法树，一边在计算时，先计算 $3+4$ 的值，最后与 5 相乘。



对应的脚本如下：

```
digraph ast{
  fontname = "Microsoft YaHei";
  fontsize = 10;

  node [shape = circle, fontname = "Microsoft YaHei", fontsize = 10];
  edge [fontname = "Microsoft YaHei", fontsize = 10];

  node [shape="plaintext"];

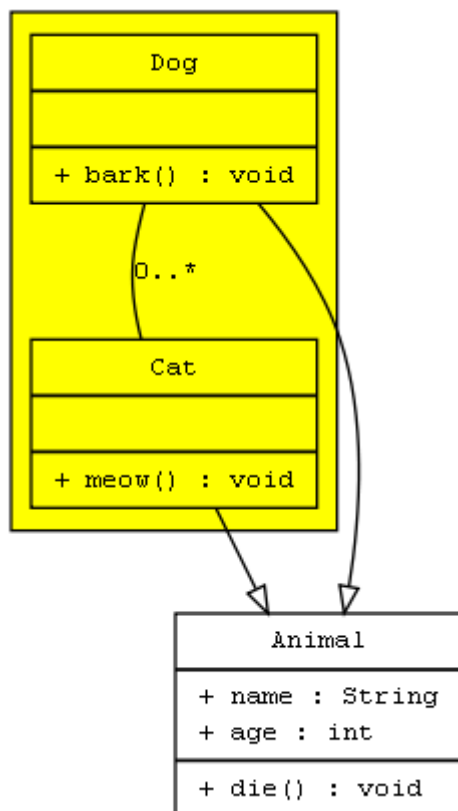
  mul [label="mul (*)"];
  add [label="add (+)"];

  add -> 3
  add -> 4;

  mul -> add;
  mul -> 5;
}
```

简单的 UML 类图

下面是一简单的 UML 类图，Dog 和 Cat 都是 Animal 的子类，Dog 和 Cat 同属一个包，且有可能有联系(0..n)。



脚本:

```

digraph G{
    fontname = "Courier New"
    fontsize = 10

    node [ fontname = "Courier New", fontsize = 10, \
        shape = "record" ];

    edge [ fontname = "Courier New", fontsize = 10 ];

    Animal [ label = "{Animal |+ name : String\n|+ age : int\n|+ die() : \
void\n|}" ];

    subgraph clusterAnimalImpl{
        bgcolor="yellow"
        Dog [ label = "{Dog||+ bark() : void\n|}" ];
        Cat [ label = "{Cat||+ meow() : void\n|}" ];
    };

    edge [ arrowhead = "empty" ];

    Dog->Animal;
  
```

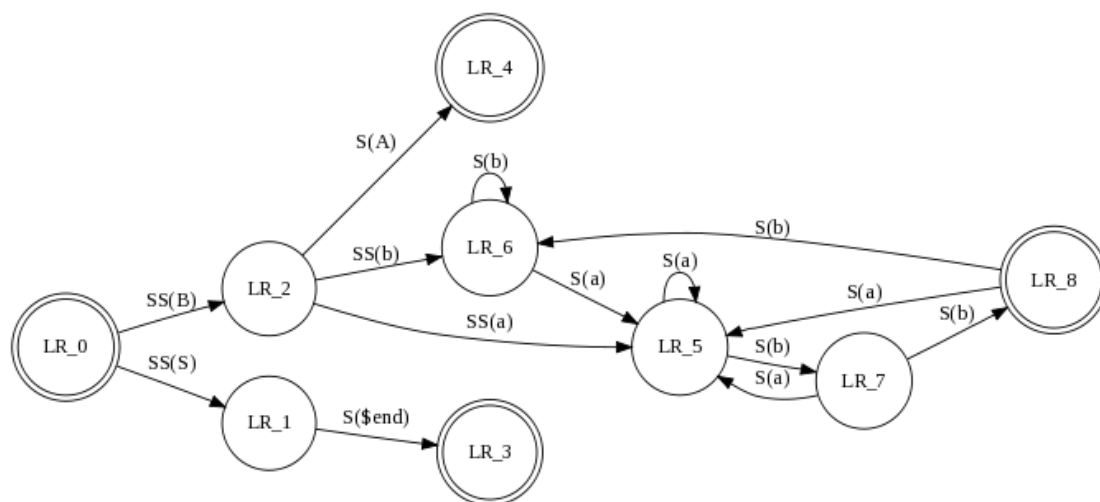
```

Cat->Animal;

Dog->Cat [arrowhead="none", label="0..*"];
}

```

状态图



脚本:

```

digraph finite_state_machine {
    rankdir=LR;
    size="8,5"

    node [shape = doublecircle];
    LR_0 LR_3 LR_4 LR_8;

    node [shape = circle];
    LR_0 -> LR_2 [ label = "SS(B)" ];
    LR_0 -> LR_1 [ label = "SS(S)" ];
    LR_1 -> LR_3 [ label = "S($end)" ];
    LR_2 -> LR_6 [ label = "SS(b)" ];
    LR_2 -> LR_5 [ label = "SS(a)" ];
    LR_2 -> LR_4 [ label = "S(A)" ];
    LR_5 -> LR_7 [ label = "S(b)" ];
    LR_5 -> LR_5 [ label = "S(a)" ];
    LR_6 -> LR_6 [ label = "S(b)" ];
    LR_6 -> LR_5 [ label = "S(a)" ];
    LR_7 -> LR_8 [ label = "S(b)" ];
    LR_7 -> LR_5 [ label = "S(a)" ];
}

```

```

LR_8 -> LR_6 [ label = "S(b)" ];
LR_8 -> LR_5 [ label = "S(a)" ];
}

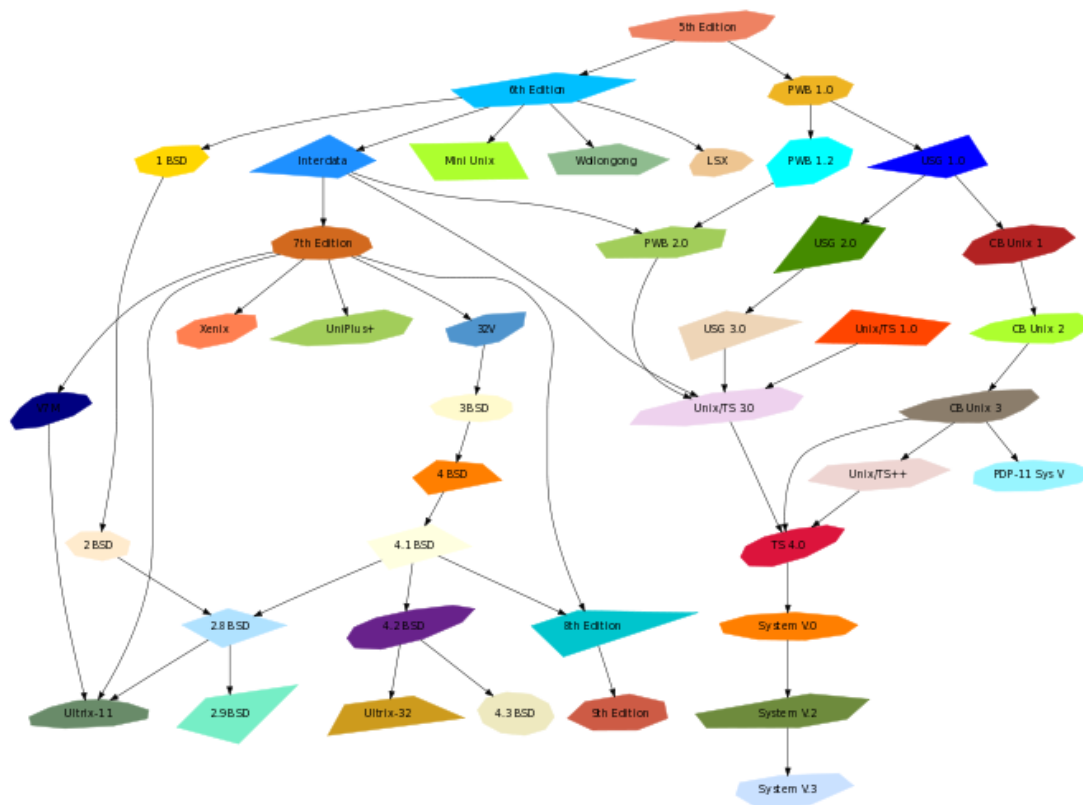
```

附录

事实上，从 `dot` 的语法及上述的示例中，很容易看出，`dot` 脚本很容易被其他语言生成。比如，使用一些简单的数据库查询就可以生成数据库中的 ER 图的 `dot` 脚本。

如果你追求高效的开发速度，并希望快速的将自己的想法画出来，那么 `graphviz` 是一个不错的选择。

当然，`graphviz` 也有一定的局限，比如绘制时序图(序列图)就很难实现。`graphviz` 的节点出现在画布上的位置事实上是不确定的，依赖于所使用的布局算法，而不是在脚本中出现的位置，这可能使刚开始接触 `graphviz` 的开发人员有点不适应。`graphviz` 的强项在于自动布局，当图中的顶点和边的数目变得很多的时候，才能很好的体会这一特性的好处：



Object Oriented Graphs
Stephen North, 3/19/93

比如上图，或者较上图更复杂的图，如果采用手工绘制显然是不可能的，只能通过 `graphviz`

提供的自动布局引擎来完成。如果仅用于展示模块间的关系，子模块与子模块间通信的方式，模块的逻辑位置等，**graphviz** 完全可以胜任，但是如果图中对象的物理位置必须是准确的，如节点 **A** 必须位于左上角，节点 **B** 必须与 **A** 相邻等特性，使用 **graphviz** 则很难做到。毕竟，它的强项是自动布局，事实上，所有的节点对与布局引擎而言，权重在初始时都是相同的，只是在渲染之后，节点的大小，形状等特性才会影响权重。

本文只是初步介绍了 **graphviz** 的简单应用，如图的定义，顶点/边的属性定义，如果运行等，事实上还有很多的属性，如画布的大小，字体的选择，颜色列表等，大家可以通过 **graphviz** 的官网来找到更详细的资料。

附录三 ExtJs 简介及示例

ExtJs 简介

jQuery 大大的方便了开发人员，以很少的代码，用很简洁，优雅的方式就可以使得页面开发简直就是一种乐趣，而 ExtJS 则更侧重于构建华丽而强健的 UI，它提供完整的 UI 框架，包括对 DOM 事件的在包装，WEB 元素组件化，动画，特效，当然也包含 DOM 元素的操作，当然相对于 jQuery，ExtJS 是一个重量级的框架。

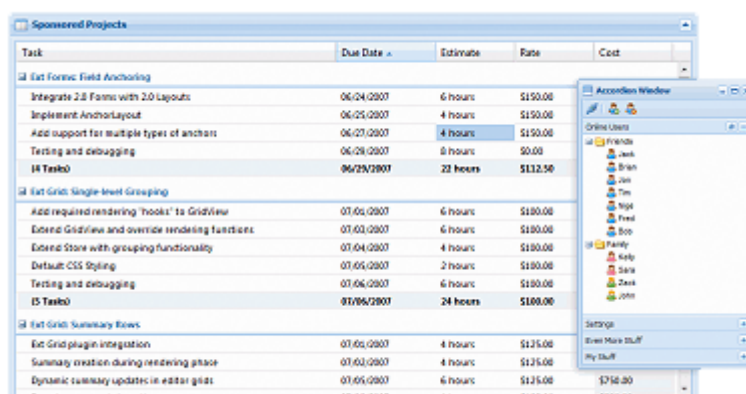


图 ExtJS 示例 1

ExtJS 提供完整的 Demo，读者可以在 ExtJS 的官方网站上下载到最新的包，其中包括 Demo 以及详细的文档，还包括 ExtJS 本身的代码。在本地安装后，部署到服务器上(比如，tomcat, IIS 等)，即可看到运行效果：

Ext JS: Cross-Browser Rich Internet Application Framework

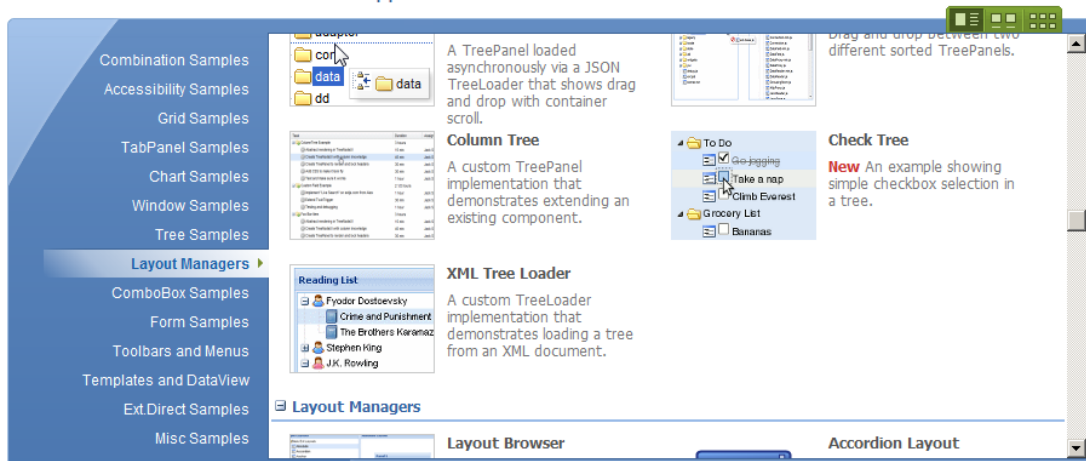


图 ExtJS 示例 2

点击每一个链接均可进入一个演示，通过这些演示，可以很快的学会 EXTJS 的基本应

用，另外，EXTJS 的社区也非常活跃，读者可以与别的爱好者一起讨论，学习。我们这里就简单的列举两个例子来说明 ExtJS 开发的便捷性和其强大的 UI 封装。

ExtJS 示例

我们结合 ExtJS 官方的一个教程来示例：

```
//当DOM加载就绪时执行
Ext.onReady(function() {
    //将一个匿名函数赋值给一个变量
    var paragraphClicked = function(e) {
        var paragraph = Ext.get(e.target);
        paragraph.highlight();
        //新建一个MessageBox, 并显示
        Ext.MessageBox.show({
            title: 'Paragraph Clicked',
            msg: paragraph.dom.innerHTML,
            width:400,
            buttons: Ext.MessageBox.OK,
            animEl: paragraph
        });
    }
    //为所有的p元素绑定click事件的响应函数
    Ext.select('p').on('click', paragraphClicked);
});
```

短短的不到 20 行代码，就可以为页面上的每个端楼添加事件处理器，当点击每个段落时，都会弹出一个消息框，消息框的内容为段落的内容。而且这个消息框的 UI 非常漂亮，可以四处移动，并且确实浮在页面上层：

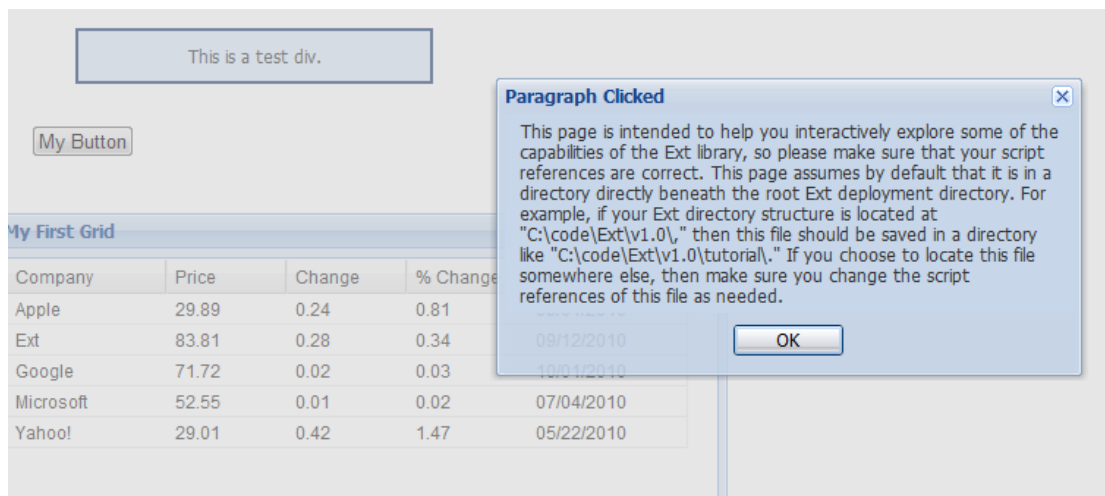


图 ExtJS 消息框示例

ExtJS 中封装了大量的 UI 组件,使得开发 RIA(富客户端应用)非常方便,较常见的有 Tree, Grid, 我们来看看 Grid 组件的使用:

```
Ext.onReady(function() {

    //定义数据源, 在应用中可能来自服务器端
    var myData = [
        ['Apple', 29.89, 0.24, 0.81, '9/1 12:00am'],
        ['Ext', 83.81, 0.28, 0.34, '9/12 12:00am'],
        ['Google', 71.72, 0.02, 0.03, '10/1 12:00am'],
        ['Microsoft', 52.55, 0.01, 0.02, '7/4 12:00am'],
        ['Yahoo!', 29.01, 0.42, 1.47, '5/22 12:00am']
    ];

    //定义reader, 指定为ArrayReader, 用于格式化
    var myReader = new Ext.data.ArrayReader({}, [
        {name: 'company'},
        {name: 'price', type: 'float'},
        {name: 'change', type: 'float'},
        {name: 'pctChange', type: 'float'},
        {name: 'lastChange', type: 'date', dateFormat: 'n/j h:ia'}
    ]);

    //创建Ext.grid.GridPanel
    var grid = new Ext.grid.GridPanel({
        store: new Ext.data.Store({
            data: myData,
            reader: myReader
        }),
        columns: [
            {header: "Company", width: 120,
                sortable: true, dataIndex: 'company'},
            {header: "Price", width: 90,
                sortable: true, dataIndex: 'price'},
            {header: "Change", width: 90,
                sortable: true, dataIndex: 'change'},
            {header: "% Change", width: 90,
                sortable: true, dataIndex: 'pctChange'},
            {header: "Last Updated", width: 120,
                sortable: true,
                renderer: Ext.util.Format.dateRenderer('m/d/Y'),
                dataIndex: 'lastChange'}
        ],
```

```
viewConfig: {
    forceFit: true
},
renderTo: 'grid',
title: 'My First Grid',
width: 500,
height: 250,
frame: true
});

//指定选择模式, 选中第一行
grid.getSelectionModel().selectFirstRow();
});
```

同样, 通过寥寥数行代码, 定义了数据源及数据源的格式之后, 很容易就可以构建出非常漂亮且标准的 Grid 来:

Company	Price	Change	% Change	Last Updated
Apple	29.89			09/01/2010
Ext	83.81			09/12/2010
Google	71.72			10/01/2010
Microsoft	52.55			
Yahoo!	29.01	0.42	1.47	

图 ExtJS Grid 示例

这个 Grid 中, 包括了列的排序支持(正序/逆序), 对需要显式的列的过滤等等, 通过使用 ExtJS, 可以大大提高 B/S 架构下页面 UI 开发的效率。

后记

这个系列(JavaScript 内核)最初是发布在网络上的(Javaeye 论坛),陆续整理出来之后,读者朋友非常积极的就一些问题与我讨论,在时间充裕的情况下,我都做过解答。后来由于工作上和其他方面的一些原因,很难有闲暇时间在做进一步的整理,因此在 2010 年后半年的很多问题都没有及时回答,这点向读者朋友们道歉,也请大家谅解。

进入 2011 年之后,工作的任务告一段落之后,我得以有时间,有机会来为这个《JavaScript 内核》系列做一个收尾工作。之前的计划是:在基础部分讲解完成之后,尽量找一些实例,特别是 JavaScript 在服务端的应用实例来做一些讨论,或者加入一定的脚本引擎工作机制等方面的讨论,现在不知道今年还有没有足够的时间和精力。原则上来说,如果时间精力不够,我则尽可能的不动笔,否则可能陷入以其昏昏,使人昭昭的尴尬境地。后半部分是否有能力来做暂不讨论,那我就先讲之前的版本整理出来,也有很多朋友向我索要过完整的电子版,不过当时陷于项目开发中,没有时间来做,但愿这个版本不算太晚。

邱俊涛

2011 年 1 月 25 日于昆明