JavaScript 内核

第 0 版

作者：邱俊涛

版权声明

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

作者信息：

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

目录

前言 6
本书组织结构 7
如何使用本书 8
致谢 9
第一章 概述 10
1.1 Javascript 简史 10
1.1.1 动态网页 10
1.1.2 浏览器之战 11
1.1.3 标准 11
1.2 JavaScript 语言特性 11
1.2.1 动态性 12
1.2.2 弱类型 12
1.2.3 解释与编译 13
1.3 Javascript 应用范围 13
1.3.1 客户端 Javascript 14
1.3.2 服务端 Javascript 16
1.3.3 其他应用中的 Javascript 19
基础部分 22
第二章 基本概念 22
2.1 数据类型 22
2.1.1 基本数据类型 22
2.1.2 对象类型 23
2.1.3 两者之间的转换 24
2.1.4 类型的判断 26
2.2 变量 27
2.2.1 基本类型和引用类型 27
2.2.2 变量的作用域 28
2.3 运算符 29
2.3.1 中括号运算符([]) 29
2.3.2 点运算符(.) 30
2.3.3 == 和 === 以及 != 和 !== 31
第三章 对象与 JSON 34

3.1 Javascript 对象 34
3.1.1 对象的属性 34
3.1.2 属性与变量 35
3.1.3 原型对象 36
3.1.4 this 指针 37
3.2 使用对象 38
3.3 JSON 及其使用 39
第四章 函数 42
4.1 函数对象 42
4.1.1 创建函数 42
4.1.2 函数的参数 43
4.2 函数作用域 45
4.2.1 词法作用域 45
4.2.2 调用对象 47
4.3 函数上下文 47
4.4 call 和 apply 47
4.5 使用函数 49
第五章 数组 52
5.1 数组的特性 52
5.2 使用数组 53
5.2.1 数组的基本方法使用 53
5.2.2 删除数组元素 57
5.2.3 遍历数组 59
第六章 正则表达式 60
6.1 正则表达式基础概念 60
6.1.1 元字符与特殊字符 60
6.1.2 范围及重复 61
6.1.3 分组与引用 63
6.2 使用正则表达式 64
6.2.1 创建正则表达式 64
6.2.2 String 中的正则表达式 66
6.3 实例：JSFilter 67

第七章 闭包 69
7.1 闭包的特性 69
7.2 闭包的用途 71
7.2.1 匿名自执行函数 71
7.2.2 缓存 72
7.2.3 实现封装 72
7.3 应该注意的问题 74
7.3.1 内存泄漏 74
7.3.2 上下文的引用 74
第八章 面向对象的 Javascript 76
8.1 原型继承 76
8.1.1 引用 78
8.1.2 new 操作符 79
8.2 封装 80
8.3 工具包 Base 81
8.4 实例：事件分发器 84
第九章 函数式的 Javascript 94
9.1 匿名函数 95
9.2 高阶函数 95
9.2.1 JavaScript 中的高阶函数 95
9.2.2 C 语言中的高阶函数 97
9.2.3 Java 中的高阶函数 98
9.3 闭包与柯里化 99
9.3.1 柯里化的概念 100
9.3.2 柯里化的应用 100
9.4 一些例子 102
9.4.1 函数式编程风格 102
9.4.2 Y-结合子 104
9.4.3 其他实例 105
后记 107

前言

大概很少有程序设计语言可以担当的起“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，这是一个复杂的对象，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 这样的浏览器中)。
- 第八章，Javascript 作为一个语言，它本身又是“可编程”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，则应该仔细读这一章，对你的编程思想大有裨益。

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 取得成功了之后，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 去掉了其中的一些实现，并提出了 ECMAScript-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，很快就占有了浏览器市场。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();
```

加入我们使用 Java 语言，代码可能会是这样：

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

    public Obj(Sting 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。

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，这些都离不来幕后的 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：

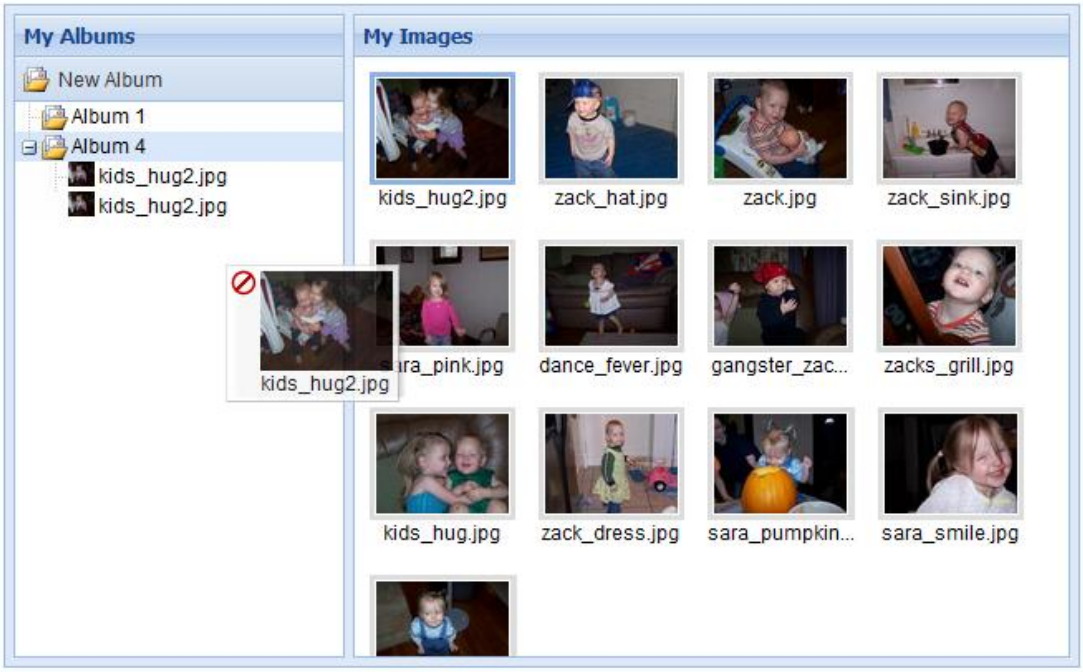


图 ExtJS 实现的一个网络相册，ExtJS 是一个非常优秀的 JavaScriipt 库

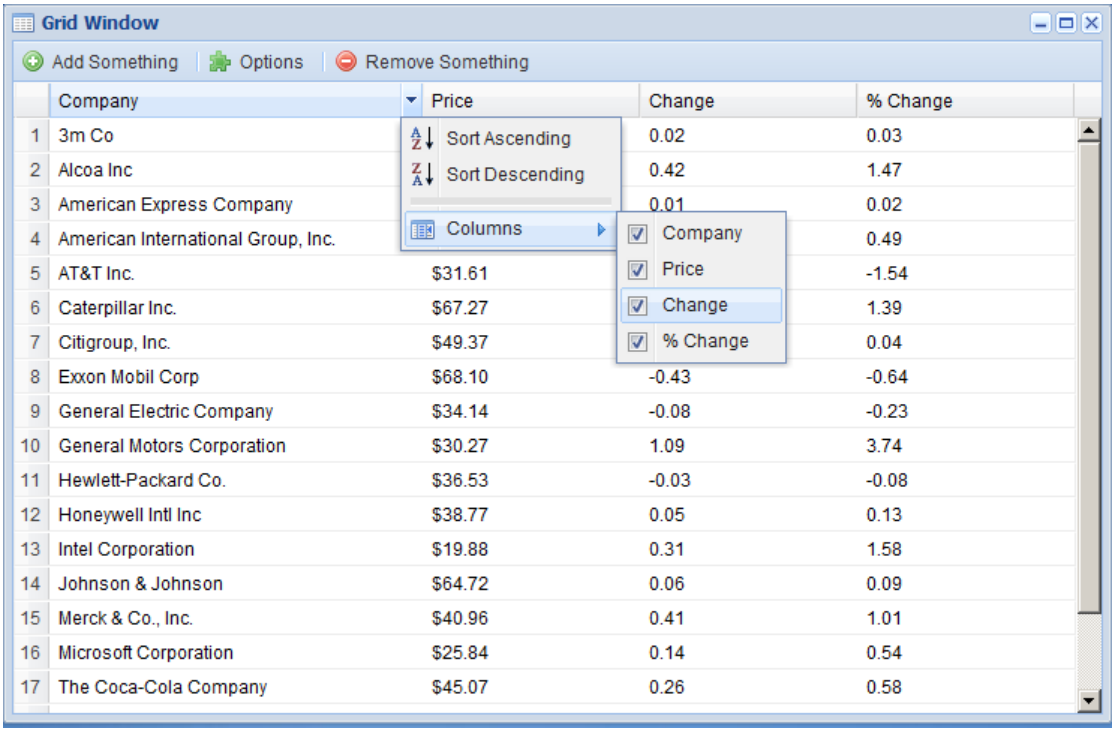
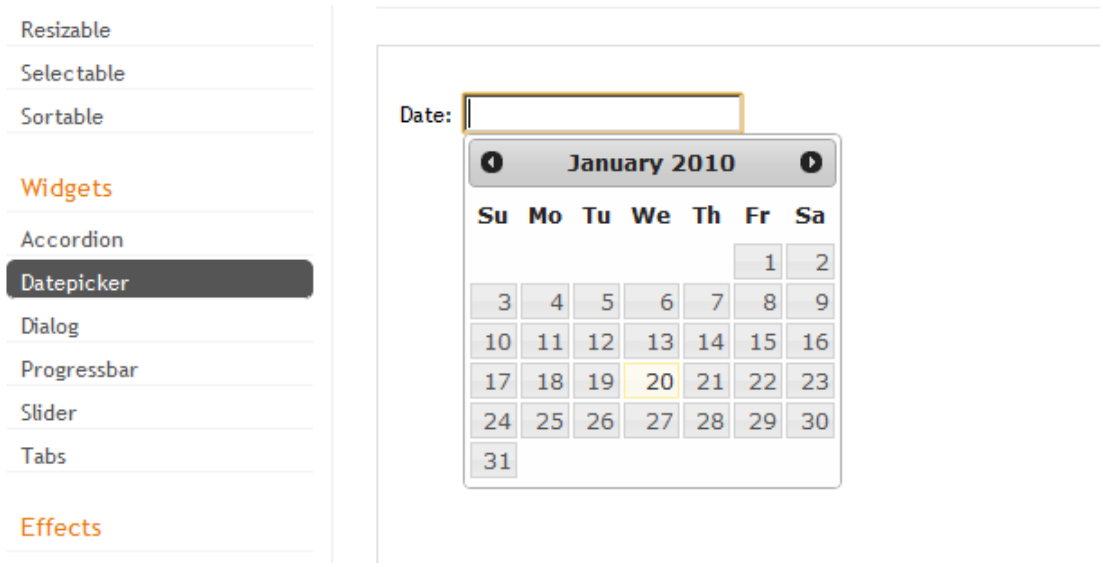


图 ExtJS 实现的一个表格，具有排序，编辑等功能

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，现在有一个感性的人是即可，我们在后边的章节会择其尤要者进行详细讲解。

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 被更多人重视，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.sendHeader(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

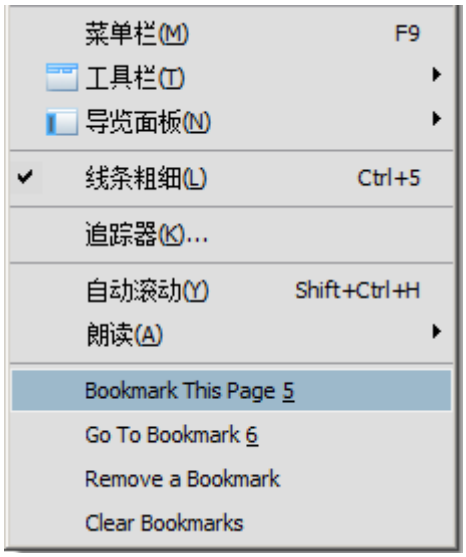
Java 版的 JavaScript 引擎原生的可以通过使用 Java 对象，那样将会大大提高 JavaScript 的应用范围，如数据库操作，服务器内部数据处理等。当然，JavaScript 这种动态语言，在 UI 方面的应用最为广泛。

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.addMenuItem({
    cName: "-",
    // menu divider
```

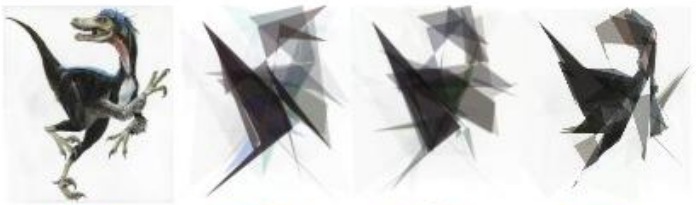
```
    cParent: "View",
    // append to the View menu
    cExec: "void(0);"
});

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

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

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

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

为 Adobe Reader 添加了 4 个菜单项，如图：



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 benefitial mutations
- 227,852 candidates
- 95.97% fitness
- Thanks to Quialiss.
- Images from different runs.



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

- 100 polygons (5-vertex)
- 10,490 benefitial 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，很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。

2.1.1 基本数据类型

在 JavaScript 中，包含三种基本的数据类型，字符串(String)，数值(Number)，布尔值(boolean)，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);
```

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
```

在 JavaScript 中，所有的数字，不论是整型浮点，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(有序的列表)，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 两者之间的转换

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

来的到 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;
```

当我们下一次要引用”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
```

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：

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 : "reference1",
    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 == 和 === 以及 != 和 !==

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。通过实例化来构筑一个对象，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)，通过像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，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elemen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elemen做一些操作
}
```

3.1.3 原型对象

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 的属性来，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重载的机制。

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abruzzi
var abruzzi = {
    name : "abruzzi",
    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abruzzi,此时的this为abruzzi
print(printName.call(abruzzi));
```

运行结果：

```
jack
Abruzzi
```

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 上的一个函数，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 操作符作用域 Object 对象，构造一个新的对象，然后动态的添加属性，从无到有的构筑一个对象。
- 定义对象的“类”:原型，然后使用 new 操作符来批量的构筑新的对象。
- 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 abruzzi = new Person('abruzzi', 26);
```

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

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

运行结果如下：

```
jack
26
street : Qing Hai Road, No : 123
abruzzi
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 : "abruzzi",
    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。

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低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 函数的一种不长用的方式(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其任意多的参数，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 undifined 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。

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abruzzi
var abruzzi = {
    name : "abruzzi",
    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abruzzi,此时的this为abruzzi
print(printName.call(abruzzi));

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

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(abruzzi, "John Abruzzi");
print(printName.call(abruzzi));
```

得到的结果为：

```
Jack Sept.
John Abruzzi
```

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：

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。

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。

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)。

作为函数的返回值：
先来看一个最简单的例子：

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

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

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

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
```

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"，原因是对 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 下的很多命令行程序，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。JavaScriipt 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.abruzzi@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]*\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", switchs);
```

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，我们来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式的用途，当然，一个例子不可能涵盖所有的内容，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：

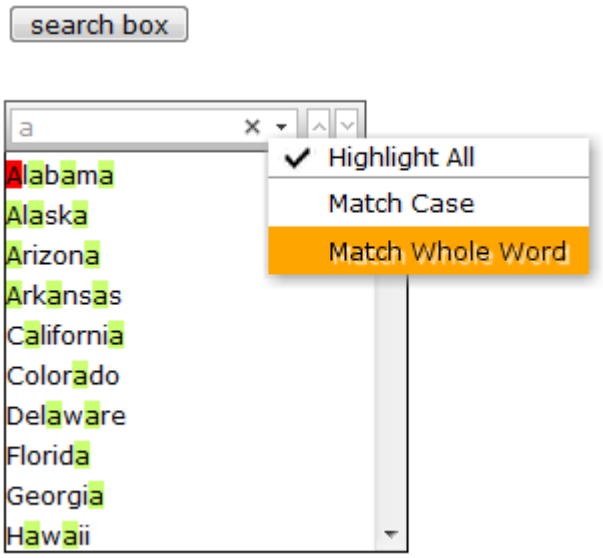


图 在列表中使用 JSFilter(结果集随用户输入而变化)

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字，keyword 是用户输入的关键字序列，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字
- 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。

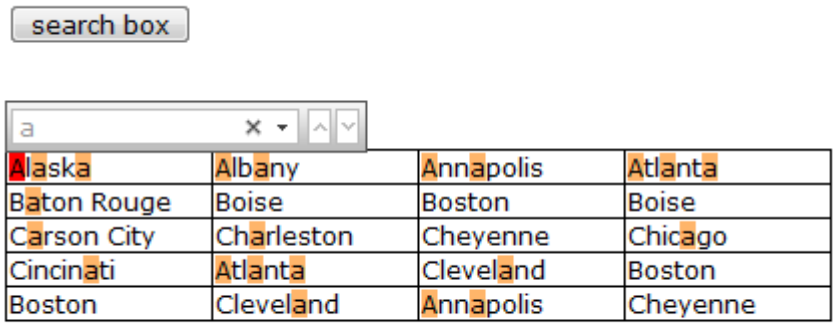


图 在表格中使用 JSFilter(不减少结果集)

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，则默认的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);
        }
    }

    //build dm.tree
})(datamodel);
```

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，关键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){//保正缓存的大小<=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("abruzzi");
print(person.getName());
```

得到结果如下：

```
undefined
default
abruzzi
```

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，则会是这种形式：

```
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();
}
```

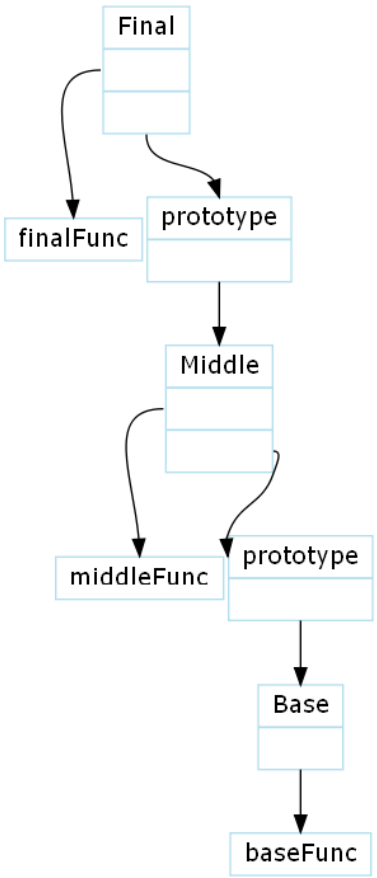


图 原型链的示意图

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，跟其他的语言不同的是，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，而始终指向这那个后来添加了 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 occured
```

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，叫做重载。

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 还能做点什么：

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.singlton){
        return uikit.event.JSEventDispatcher.singlton;
    }

    this.listeners = {};

    uikit.event.JSEventDispatcher.singlton = 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></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.getObject().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></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_CHAGE 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：

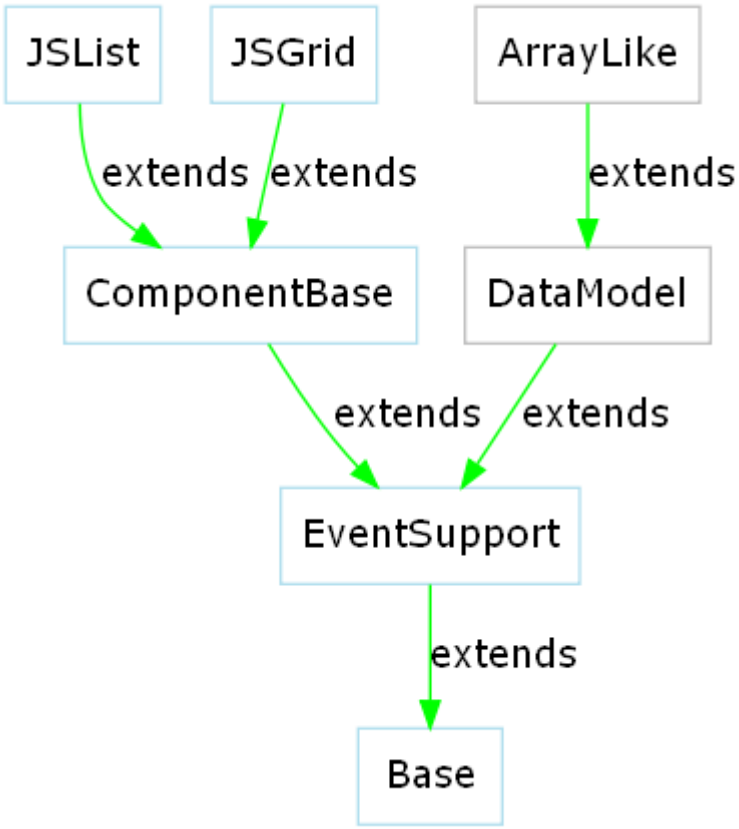


图 事件分发器类层次

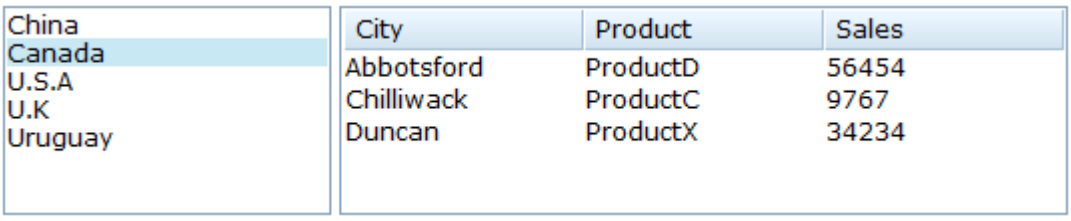
应该注意的是，在绑定完监听器之后，我们手动的触发了 EVENT_LIST_DATA_READY 事件，来通知 List 可以绘制自身了：

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

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

点击 Canada，Grid 中的数据同样发生相应的变化：

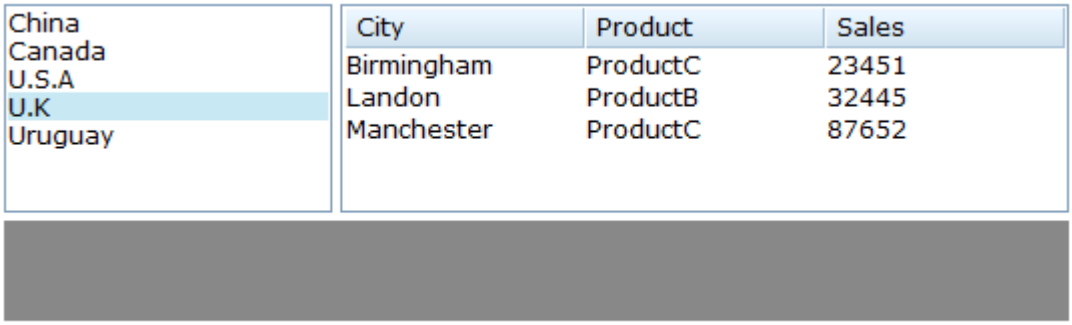


由于 List 和 Grid 的数据是关联在一起的，他们的数据结构具有下列的结构：

```
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)：



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 orignal 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教笨重的闭包(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 表示函数的调用者，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));
```

fiexedPoint 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。

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，单元这个版本不算太晚。

邱俊涛
2011 年 1 月 25 日于昆明