

<http://abruzzi.javaeye.com>

JavaScript内核系列，主要涉及JavaScript的基础知识，以及建立在这些基础知识之上的编程风格，包括面向对象的JavaScript及函数式的JavaScript。

目 录

1. 计算机科学与技术

1.1 JavaScript内核系列 第1章 前言及概述 3

1.2 JavaScript内核系列 第2章 基本概念 19

1.3 JavaScript内核系列 第3章 对象与JSON 39

1.4 JavaScript内核系列 第4章 函数 51

1.5 JavaScript内核系列 第5章 数组 65

1.6 JavaScript内核系列 第6章 正则表达式 79

1.7 JavaScript内核系列 第7章 闭包 95

1.8 JavaScript内核系列 第8章 面向对象的JavaScript(上) 105

1.9 JavaScript内核系列 第8章 面向对象的JavaScript(下) 118

1.10 JavaScript内核系列 第9章 函数式的Javascript 132

1.1 JavaScript内核系列 第1章 前言及概述

发表时间: 2010-04-01

前言

从2006年第一次接触JavaScript至今，算来也有四年时间了。上大学的时候，有一段时间沉迷于函数式编程，而那时候对JavaScript的理解仅仅局限在这是用来做网页的一种脚本，那些很恶心的弹出窗口和不断在页面上漂浮的黄色广告就是用JavaScript做出来的。工作之后，由于我们产品本身就是基于WEB展现的，而这个时候Ajax已经复兴数年了，但是于我，则只有“Ajax可以实现页面的局部刷新”这样一个概念。在实际的应用中，我才有机会面对真正的JavaScript，早期关于函数式编程的概念开始在我的记忆力复苏，而真正将JavaScript比较熟练的应用在产品的开发中，那已经是09年年末了。

开始研究JavaScript，是从2008年的5月左右，我从当当上买了一本犀牛书，并下载到一本蝴蝶书，结合着实际工作的要求，一步一步的开始了JavaScript的探索。刚开始也只是被陷入在DOM的泥潭中，慢慢的开始考虑更进一步的东西，比如JSON作为数据交换，window对象的一些特殊的方法如eval等等，事实上仍然是陷入在语法的细节中。当终于有一天，理解了变量的作用域，理解了闭包，理解了函数的一等性，以往用来类比的来源于其他语言的所有概念则完全被颠覆了，当理解了JavaScript中的这些基本概念(虽然难以理解，但是它们的确是基本概念)之后，在回顾来时的路，则有些一览众山小的意思。

但是，这些看似基本的概念，在初学时如果混入了其他语言的概念以进行类比，则后患无穷，所以，学习JavaScript其实就一句话：“将JavaScript当成一门新的语言来学习”。鉴于我学习JavaScript的历程中，遇到了很多问题，但是最终都一一克服了，在这个过程中，有些心得，有些体会，就**分节**贴出来，做成一个系列，以方便初学JavaScript的朋友们。

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，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及其衍生的技术AJAX。

1.1.2 浏览器之战

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：

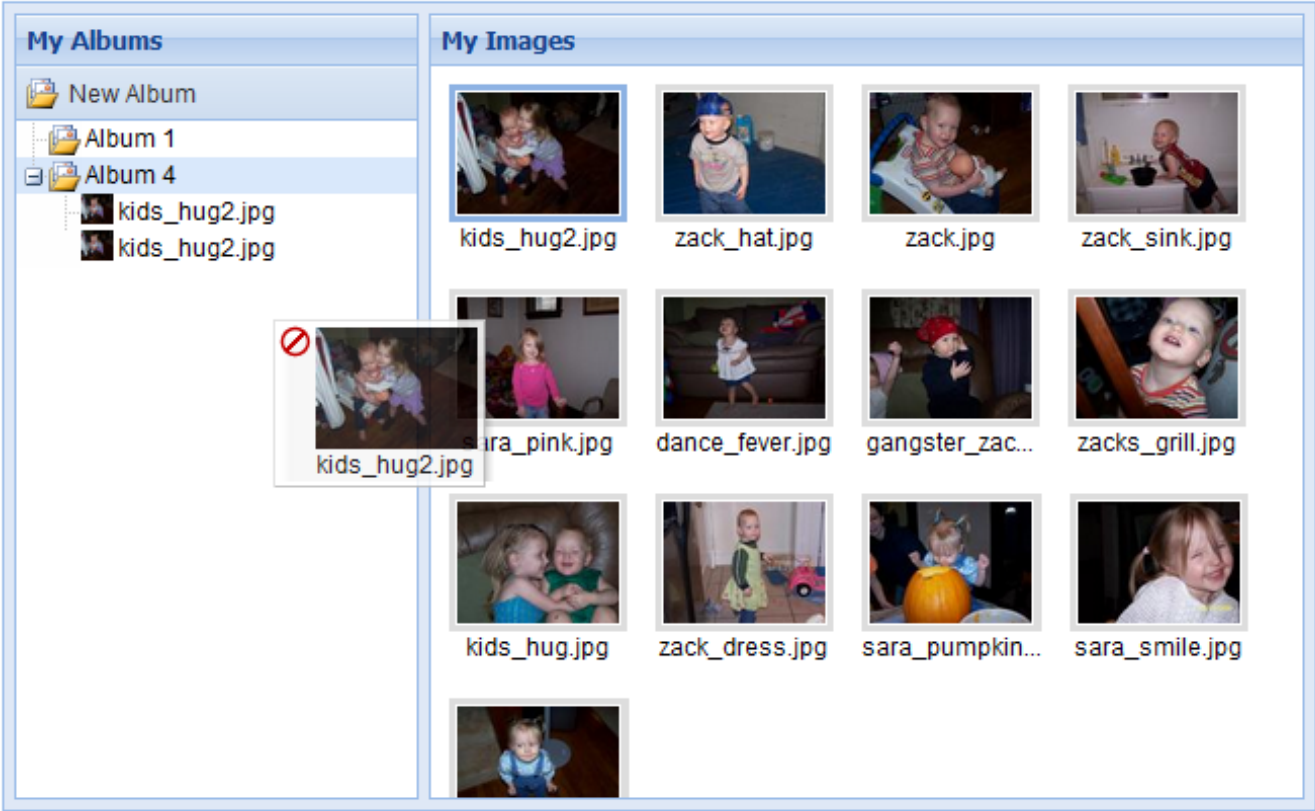
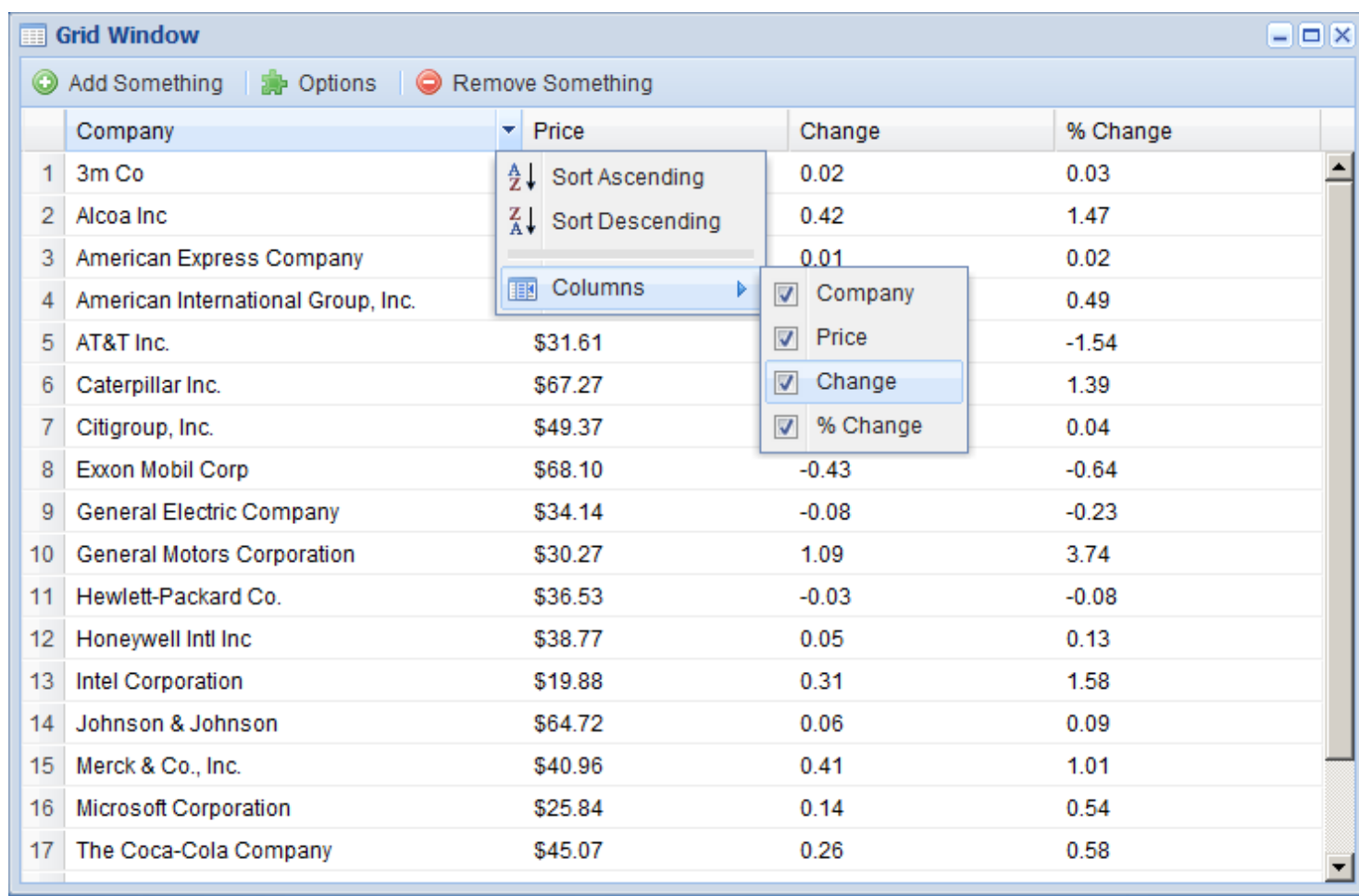


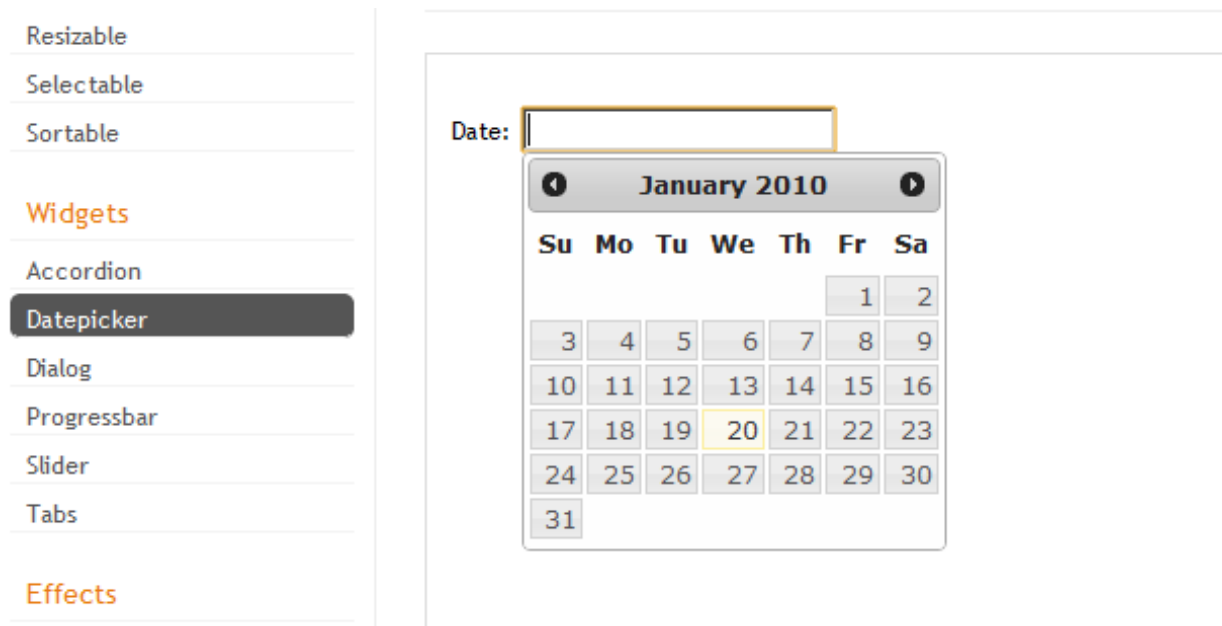
图 ExtJS实现的一个网络相册，ExtJS是一个非常优秀的JavaScript库



	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

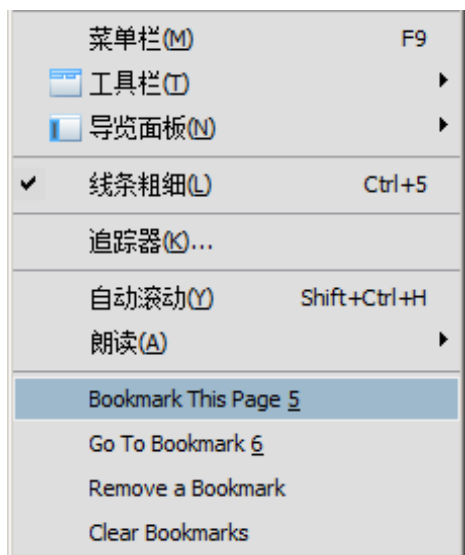
Java版的JavaScript引擎原生的可以通过使用Java对象，那样将会大大提高JavaScript的应用范围，如数据库操作，服务器内部数据处理等。当然，JavaScript这种动态语言，在UI方面的应用最为广泛。

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。

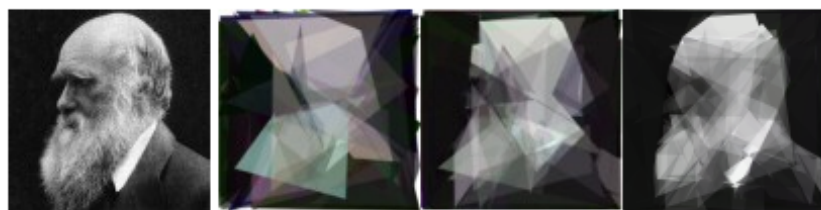
```
app.addItem({
  cName: "-",
  // menu divider
  cParent: "View",
  // append to the View menu
  cExec: "void(0);"
});
```

```
app.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;"
});
```

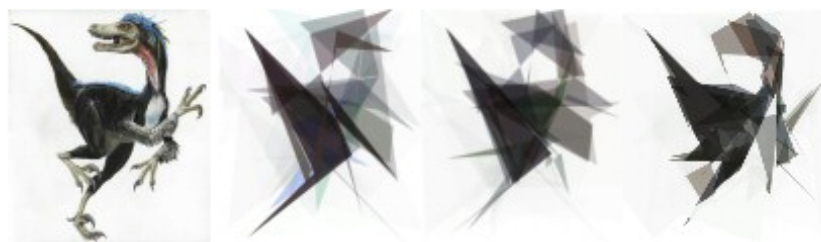
为Adobe Reader 添加了4个菜单项，如图：



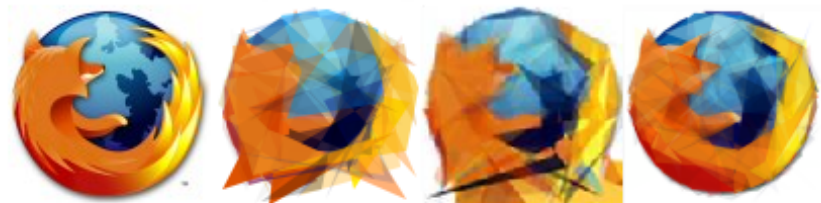
另一个比较有意思的JavaScript实例为一个在线的遗传算法的演示，给定一个图片，然后将一些多边形(各种颜色)拼成一个图片，拼图的规则为使用遗传算法，使得这些多边形组成的图片与目标图片最为相似：



- 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。

附：由于作者本身水平有限，文中难免有纰漏错误等，或者语言本身有不妥当之处，欢迎及时指正，提出建议，参与讨论，谢谢大家！

1.2 JavaScript内核系列 第2章 基本概念

发表时间: 2010-04-03

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(事实上，函数，数组等也都是对象，这个在后边的章节详述)。

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运算符可知其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([])

[]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。

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

附：由于作者本身水平有限，文中难免有纰漏错误等，或者语言本身有不妥当之处，欢迎及时指正，提出建议，参与讨论，谢谢大家！

1.3 JavaScript内核系列 第3章 对象与JSON

发表时间: 2010-04-12

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

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prototype属性即为null), 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：

- Ø 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。
- Ø 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。
- Ø 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);  
}
```

附：由于作者本身水平有限，文中难免有纰漏错误等，或者语言本身有不妥当之处，欢迎及时指正，提出建议，参与讨论，谢谢大家！

1.4 JavaScript内核系列 第4章 函数

发表时间: 2010-04-18

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

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”。

4.3 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
```

附：由于作者本身水平有限，文中难免有纰漏错误等，或者语言本身有不妥当之处，欢迎及时指正，提出建议，参与讨论，谢谢大家！

1.5 JavaScript内核系列 第5章 数组

发表时间: 2010-04-24

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
<code>concat()</code>	连接两个或更多的数组，并返回结果。
<code>join()</code>	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。
<code>pop()</code>	删除并返回数组的最后一个元素。
<code>push()</code>	向数组的末尾添加一个或更多元素，并返回新的长度。
<code>reverse()</code>	颠倒数组中元素的顺序。
<code>shift()</code>	删除并返回数组的第一个元素。
<code>slice()</code>	从某个已有的数组返回选定的元素。
<code>sort()</code>	对数组的元素进行排序。
<code>splice()</code>	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);
```

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，数字下标最终会被解释器转化为字符串，也就是说，所谓的数字下标只不过是看着像数字而实际上是字符串的属性名)，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。

附：由于作者本身水平有限，文中难免有纰漏错误等，或者语言本身有不妥当之处，欢迎及时指正，提出建议，参与讨论，谢谢大家！

1.6 JavaScript内核系列 第6章 正则表达式

发表时间: 2010-04-27

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。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
<code>^</code>	串的开始
<code>\$</code>	串的结束
<code>*</code>	零到多次匹配
<code>+</code>	一到多次匹配
<code>?</code>	零或一次匹配
<code>\b</code>	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
<code>\r</code>	匹配回车
<code>\n</code>	匹配换行
<code>\t</code>	制表符
<code>\f</code>	换页

<code>\x#</code>	匹配十六进制数
<code>\cX</code>	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
<code>[...]</code>	在集合中的任一个字符
<code>[^...]</code>	不在集合中的任一个字符
<code>.</code>	出 <code>\n</code> 之外的任一个字符
<code>\w</code>	所有的单字，包括字母，数字及下划线
<code>\W</code>	不包括所有的单字， <code>\w</code> 的补集
<code>\s</code>	所有的空白字符，包括空格，制表符
<code>\S</code>	所有的非空白字符
<code>\d</code>	所有的数字
<code>\D</code>	所有的非数字
<code>\b</code>	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，而不用担心出错，比如：

```
var pid = /^[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式隔离(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：

写道

```
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", 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()	测试模式是否匹配
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
<code>match</code>	匹配正则表达式，返回匹配数组
<code>replace</code>	替换
<code>split</code>	分割
<code>search</code>	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：

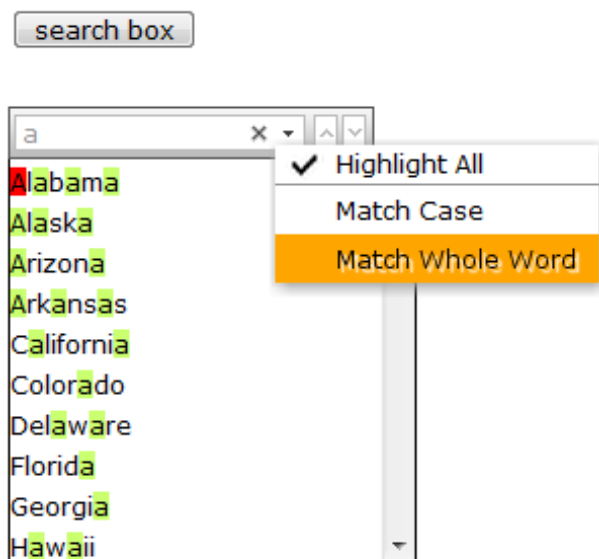


图1 在列表中使用JSFilter(结果集随用户输入而变化)

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。

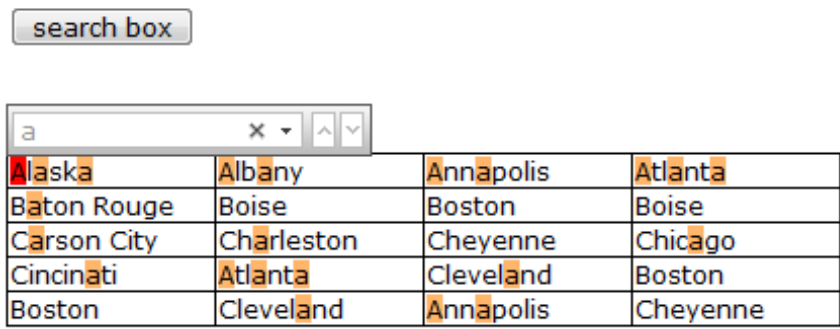


图2 在表格中使用JSFilter(不减少结果集)

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。

附：由于作者本身水平有限，文中难免有纰漏错误等，或者语言本身有不妥当之处，欢迎及时指正，提出建议，参与讨论，谢谢大家！

附件下载:

- scrolling.zip (100.7 KB)
- dl.javaeye.com/topics/download/93ffdc7c-f960-3f8e-8b60-b3465a308c5d

1.7 JavaScript内核系列 第7章 闭包

发表时间: 2010-05-04

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(outer[0].invoke());  
print(outer[1].invoke());  
print(outer[2].invoke());  
print(outer[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`outer[0].invoke()`时，`i`的值依旧为4，因此`outer`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 outer = [];
```



```
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("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。

附：由于作者本身水平有限，文中难免有纰漏错误等，或者语言本身有不妥当之处，欢迎及时指正，提出建议，参与讨论，谢谢大家！

1.8 JavaScript内核系列 第8章 面向对象的JavaScript(上)

发表时间: 2010-05-06

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();
}
```

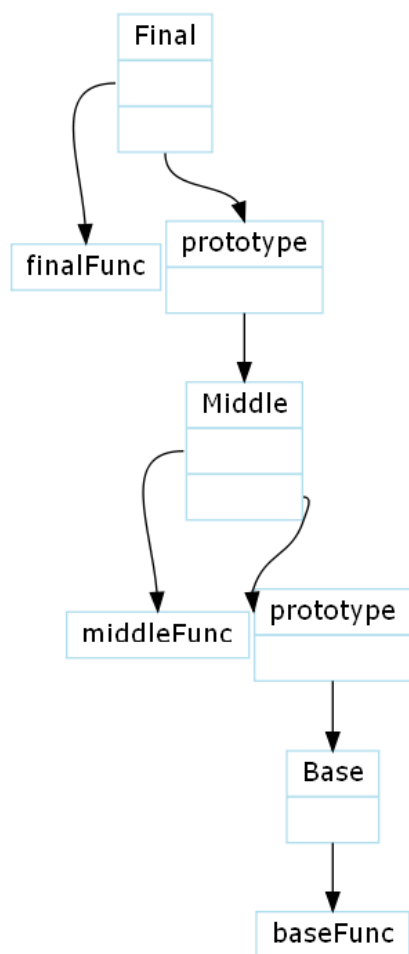


图 原型链的示意图

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，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+" occurred");
  }
});

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：

由于本篇篇幅较长，因此分为**两部分**发布！

附：由于作者本身水平有限，文中难免有纰漏错误等，或者语言本身有不妥当之处，欢迎及时指正，提出建议，参与讨论，谢谢大家！

1.9 JavaScript内核系列 第8章 面向对象的JavaScript(下)

发表时间: 2010-05-06

接上篇：[JavaScript内核系列 第8章 面向对象的JavaScript\(上\)](#)

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.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>").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;
},

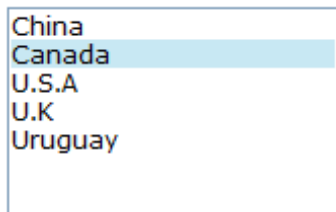
getSelectedItemIndex : function(){
    var jqo = $(this.canvas);
    var index = jqo.find("div").index($(".selected")[0]);
    return index;
}
});
```

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：

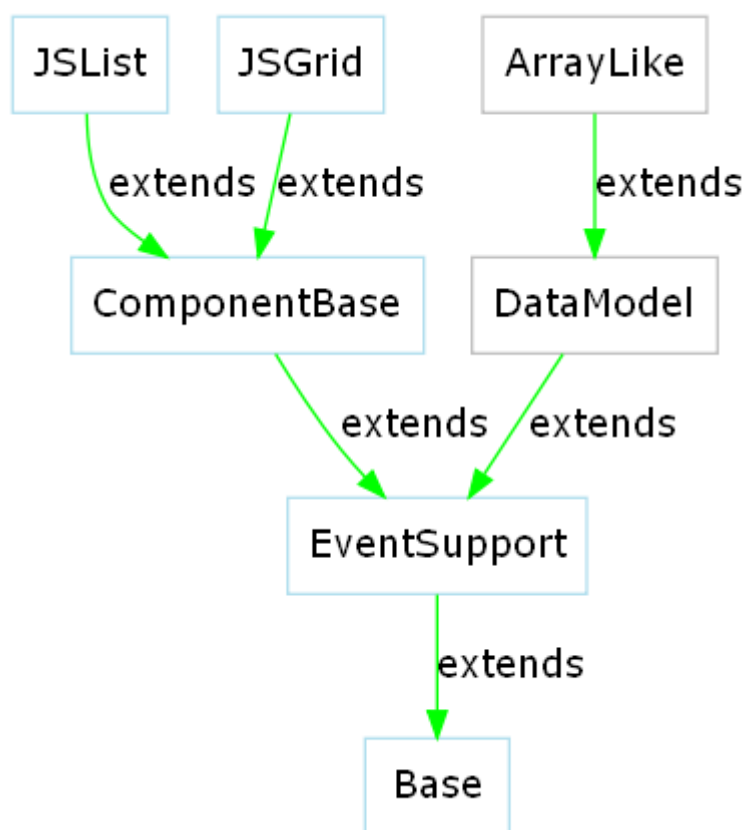


图 事件分发器类层次

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}
});
```



```
        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			

附：由于作者本身水平有限，文中难免有纰漏错误等，或者语言本身有不妥当之处，欢迎及时 指正，提出建议，参与讨论，谢谢大家！

附件下载:

- [jsed.zip \(43.7 KB\)](#)
- dl.javaeye.com/topics/download/de98bf4d-5877-32c9-8b47-32b5fee04285

[1.10 JavaScript内核系列 第9章 函数式的Javascript](#)

发表时间: 2010-05-13

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));
```

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。

附：由于作者本身水平有限，文中难免有纰漏错误等，或者语言本身有不妥当之处，欢迎及时 指正，提出建议，参与讨论，谢谢大家！