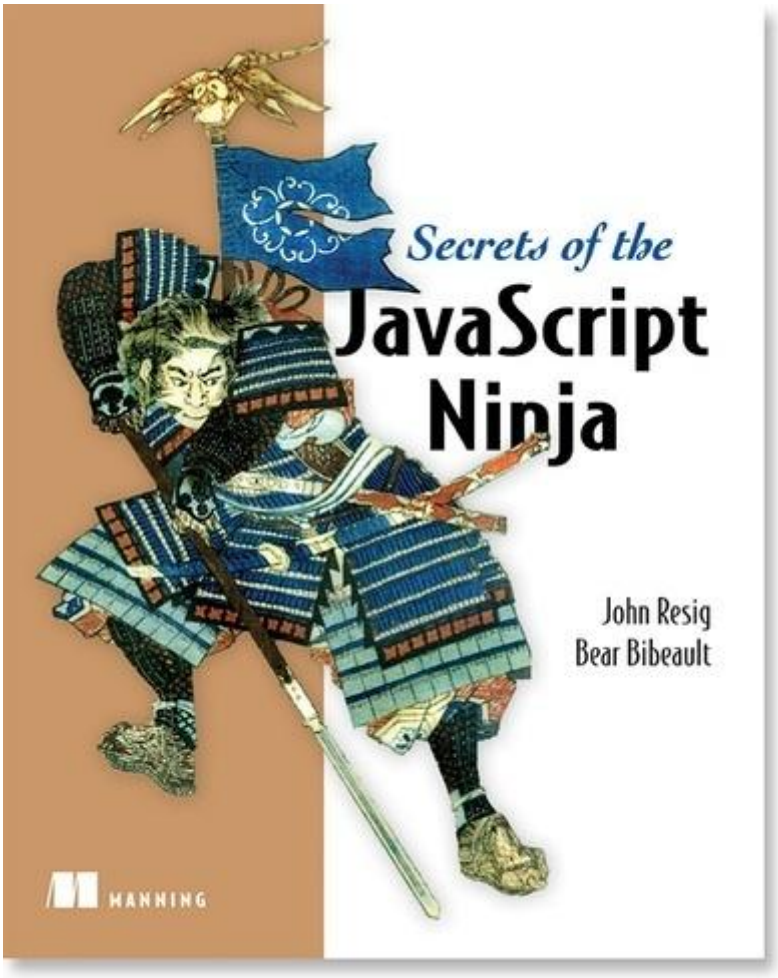


JavaScript 忍者禁术

(Secrets of the JavaScript Ninja 中文版)

版本：1.0

原书: Secrets of the JavaScript Ninja

作者: John Resig (ejohn.org)

译者: Yann (yannhe.com)

邮箱: yannhack@163.com

GitHub: github.com/yannhack/jsninja

第一阶段：准备阶段

1.进入忍者的世界(Enter the ninja) [完成]

2.测试和 debug(Testing and debugging)[完成]]

第二阶段：学徒阶段

3.函数是根基(Functions are fundamental)[完成]

4.挥舞函数(Wielding functions)[完成]

5.闭包 (Closing in on closures)[完成]

6.原型与面向对象(Object-orientation with prototypes)[完成]

第三阶段：忍者阶段

第四阶段：大师阶段

第一章.进入忍者的世界(1.Enter the ninja)[完成]

翻译 Secrets of the JavaScript Ninja (JavaScript 忍者禁术)

第一章 进入忍者的世界(1.Enter the ninja)

本章重点：

1.介绍本书的目的和结构

2.我们将要关注的类库，我们将要讲解的比较 cool 的亮点都会在这些类库中找到具体的实现。

3.什么是 JavaScript 的高级编程(世界级的开发者用什么方式来编写 JavaScript)

4.什么是跨浏览器编程

5.展示测试组件(如何测试你的代码)

目录链接：http://yannhe.com/secrets-of-the-javascript-ninja-javascript

本文链接：http://yannhe.com/1-enter-the-ninja

如果你在阅读本书，你一定使用过 JavaScript，你一定会感到编写有效并且可以跨浏览器的 JavaScript 代码并不轻松。

我们的挑战，除了要写出干净的代码，还要考虑不同浏览器之间的差异。为了可以解决这些复杂问题，我们想到将可重用的功能以类库的形式实现。

这些类库，虽然彼此之间很不同，但是他们的统一原则是：

它们要很容易的被使用，构建的开支要尽量的小，并且可以在所有的浏览器中正常运行。

通过分析这些类库的构造，我们可以学习到很多非常牛逼的经验，如果你可以学以致用，你一样可以构建出同样牛逼的代码。

本书将那些世界级的大牛写的代码融合在一起，目的就是想为你开启探索 JavaScript 精髓的大门。

1.1 JavaScript 类库

在本书中，到处都是一些有趣的技术和代码示例，目的是想通过这些技术和示例让你体会到其背后的思想和理念。

那些世界级的类库，正是基于这些思想构建和理念构建而成。

它们是：

·Prototype（http://prototypejs.org/)：2005 年由现代 JavaScript 库教父 SamStephenson 创建并发布的。封装了 DOM，Ajax 和 event 事件功能，此外还有涉及到面向对象编程技术、面向方面编程技术和函数式编程技术。

·jQuery（http://jquery.com)：2006 年 1 月由 John Resig 发布，通过 CSS 选择器来匹配 DOM 这种技术也因此库而流行起来。还包括：DOM，Ajax，event，等灵活的功能。

现在，这两个库统治者 JavaScript 类库市场，被无数 web 站点和个人使用。

除了这两个类库，我们还会提到两个类库：

·Yahoo UI（http://developer.yahoo.com/yui/):由雅虎公司在 2006 年 2 月公开发布，是其公司内部开发的 JavaScript 框架。包含 DOM、Ajax、事件和动画处理能力，除此之外还有一系列预先构建好的小部件（日历、网格、手风琴折叠控件）

·base2（http://code.google.com/p/base2/)：由 Dean Edwards 创建并于 2007 年 3 月发布，支持 DOM 和事件功能。它名声鹊起的原因是它试图用一种通用的、跨浏览器的方式实现各种 W3C 规范。

所有这些库，都是世界级的，构建精良，历经沉淀。因此，这些库都是值得深入学习的。

但是，这些技术并不只是为了用于构建大型的类库，而是适用于任何规模的 JavaScript 代码。

构建 JavaScript 类库需要重点关注三个方面：

1.如何用一种卓越的方式编写 JavaScript

2.全面实用的跨浏览器代码

3.如何将所有涉及到的技术整合在一起

1.2 理解 JavaScript 这门语言

很多 JavaScript 开发者(包括那些有很多年工作经验的 JavaScript coder),都对自己的 JavaScript 的编程水平没有一个清醒的认知。

他们以为牛逼的 JavaScript 代码就是要用很多 JavaScript 的特色元素，例如用很多的对象和函数，甚至是匿名函数。

但是真实的情况是，这些技巧并不是所谓的 JavaScript 高级编程，这些

都是花拳绣腿的假把式。

还有一点值得一提的是，他们对闭包(closures)这个概念的理解极其的匮乏，包括闭包的目的和实现。

其实在 JavaScript 这门语言中，闭包是个十分重要的概念。

由于闭包的存在，函数才能在 JavaScript 中显得如此基本和重要。

在 JavaScript 中，对象(objects)，函数(functions)－在 JavaScript 中函数是自然类型的元素(first class elements)和闭包(closures)之间拥有十分紧密的关系。

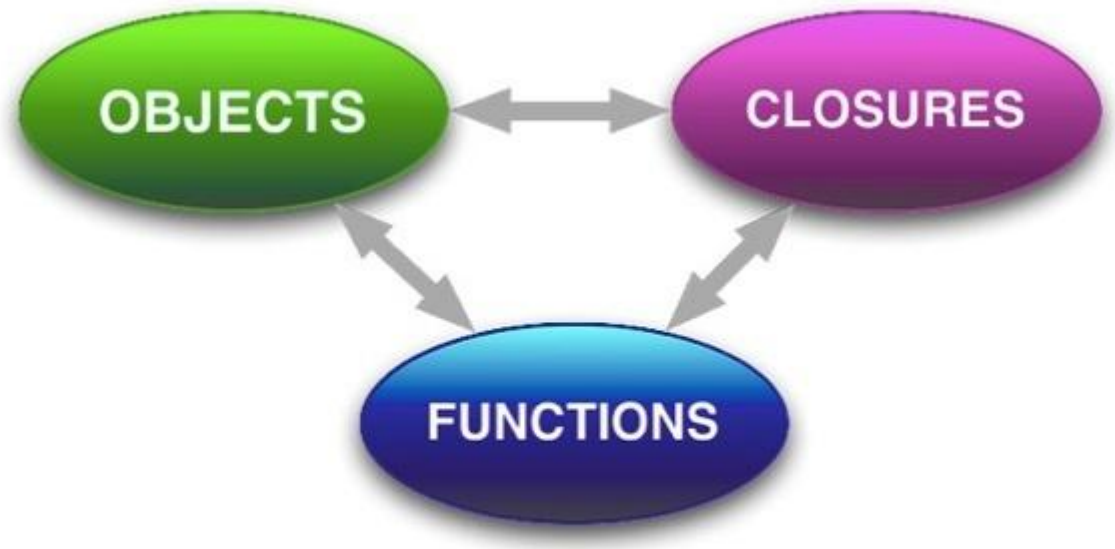


Figure 1.1 JavaScript consists of a close relationship between objects, functions and closures

深刻理解他们之间的关系，可以提高我们的 JavaScript 编程能力。这可以视为 JavaScript 编程的基础内功。

很多 JavaScript 开发者，尤其那些之前使用例如 Java，C++这种面向对象语言的开发者，

他们可能会过多的关注对象(objects)而忽视了函数(functions)和闭包(closures)所发挥的价值。

另外还有两个经常在 JavaScript 开发中遇到，但没有被充分利用的概念：定时器(timers)和正则表达式(regular expressions)。这两个概念会经常的被用到，但是由于经常被误解，他们的真正的能量没有完全的发挥出来。

对我们来说定时器在浏览器中的执行原理是一个很神秘的事，但是如果真正理解了它的本质，我们就可以编写出这样一些复杂的情景的代码：常驻的运算能力，平滑的动画效果。

另外如果能真正的理解正则表达式，可以让你将一些复杂又零碎的工作，变得简单和高效。

在 JavaScript 高级编程的旅程中，我们会遇到 with 和 eval()这两个概念，他们分别在第 10 章和第 9 章。

这两个概念被大部分 JavaScript 开发者忽视，误用，甚至是全盘否定。但看一下那些世界级的开发者的代码，我们会了解到，当这些特性被适当的使用时候，它们可以创造出不可思议的代码。

如果你想将 JavaScript 玩的发烧一些，我们可以利用他们来进行有趣的元编程，我们可以通过改造 JavaScript 本身，创造出你想要的语言。

通过学习这些内容，你可以不断的升级你的编程能力。

将我们所学的这些概念和亮点结合在一起，我们将拥有很强的的理解力和编写稳定并且跨浏览器的代码的能力。

1.3 介绍跨浏览器编程(Cross-browser considerations)

当我们具备了很好的 JavaScript 编程能力之后，并想利用他们开发一个基于浏览器的应用的时候，我们经常会陷入各种因为浏览器的差异导致的困境之中。

浏览器之间之间相距甚远，他们有很多的问题，bug，未实现的 API。我们需要一个综合的对策来应付这些浏览器差异性导致的问题。

我们在本书中使用的最终对策是基于雅虎公司的浏览器分级概念，叫做浏览器分级支持(Graded Browser Support)。

雅虎为他们希望支持的浏览器分了不同的等级，如下所示：

.A:最高支持力度，在这一级中的浏览器，会得到最全面的支持和最强大的功能和测试。

.C:很小的支持力度，这里的浏览器一般都是比较旧的，并且是很难用的。通常只会为这种浏览器提供 Html 和 CSS，没有 JavaScript

.X:不支持，一般都是未知浏览器

Table 1.1 This early 2011 Graded Browser Support matrix shows the level of browser support from Yahoo!

	Windows XP	Windows 7	Mac OS 10.6	iOS 3	iOS 4	Android 2.2
IE <6	C					
IE 6, 7, 8	A					
Firefox <3	C					
Firefox 3	A	A	A			
Firefox 4		A	A			
Safari < 5			C			
Safari 5+			A			
Safari for iOS				A	A	
Chrome	A					
WebKit for Android						A
Opera <9.5	C					
Netscape <8	C					

1.4 最佳实践

如果你想称为最佳的 JavaScript 开发者，良好的编程和跨浏览器能力是不够的，你还需要拥有测试能力，性能分析能力和 debug 能力。

让我们来看看一些实践：

1.4.1 最佳实践：测试(testing)

assert()函数，即：断言函数，这是一个用于测试的功能函数，他的目的是判断传入次函数的第一个参数是 true 还是 false。

一般代码结构如下：

assert(condition, message);

如果 condition 如果不为 true，message 就会被显示出来，例如：

assert(a == 1, “Disaster! A is not 1!”);

如果 a 不等于 1 那么后面的那句话就会显示出来。

由于 assert 函数并不是 JavaScript 的内置函数，所以我们要自己实现一个 assert 函数，我们会在第二章来讨论。

1.4.2 最佳实践：性能分析(performance analysis)

另一个很重要的实践是性能分析。我们会自己实现一个函数来分析我们自己写的代码的性能。

例如：

```
perf(“String Concatenation”, function(){
var name = “fred”;
for (var i = 0; i < 20; i++){
name += name;
}
})
```

在复杂的浏览器环境下开发我们的 JavaScript 应用，拥有一套完整的开发技巧是很必要的。

1.5 总结

跨浏览器编程是很困难的，比大多数人想象的还要困难。

为了搞定它，除了成为 JavaScript 这们语言的大师，我们还要了解浏览器的知识，这样才能写出健壮的最佳代码。

面对 JavaScript 编程的这些困难的挑战，已经有一些勇敢的灵魂为我们

作出了贡献：他们就是类库开发者。

在这些类库蕴含的光芒的指引下，我们会一步一步攀登，最终成为世界级的 JavaScript 开发者。

这段旅程将会是丰富，受益的。

让我们开始有趣的探索吧。

(转载本文章请注明作者和出处 Yann (yannhe.com)，请勿用于任何商业用途)

第二章.测试和debug(2.Testing and debugging)[完成]

翻译 Secrets of the JavaScript Ninja (JavaScript 忍者禁术)

第二章 测试和 debug(Testing and debugging)

本章重点： 1.测试工具

2.测试技术

3.构建一个测试框架

4.如何测试异步代码

目录链接：http://yannhe.com/secrets-of-the-javascript-ninja-javascript

本文链接：http://yannhe.com/2-testing-and-debugging

由于我们在之后的示例中用到的测试工具很少，几乎就是在用 assert 函数，所以本章不做详细的翻译了。

首先将 assert 函数给出，之后的章节用的的 assert 函数都是在调用此段代码,用的时候请引用下面的 assert.js:

```
(function()
{
    document.write("<style>#results li.pass { color: green;}#results
li.fail { color: red;}</style>");
```

```
    document.write("<ul id='results'></ul>");
    var results;
    this.assert = function assert(value, desc)
    {
        results = results || document.getElementById("results");
        var li = document.createElement('li');
        li.className = value ? "pass" : "fail";
        li.appendChild(document.createTextNode(desc));
        results.appendChild(li);
        if (!value)
        {
            li.parentNode.parentNode.className = "fail";
        }
        return li;
    };
    this.test = function test(name, fn)
    {
        results = document.getElementById("results");
        results = assert(true,
name).appendChild(document.createElement("ul"));
        fn();
    }
})()
```

2.1 debugging code

一般菜鸟在 debug 的时候都是用 alert()，现在我们看看在不同的浏览器中自带的高级点的工具：

.Firebug:请看 http://getfirebug.org

.IE Developer Tools: 在 IE8 和 9 和 10 中可用

.Opera Dragonfly: Opera 9.5 或更新可用

.WebKit Developer Tools: Safari 3 ,4, Chrome(推荐这个，我就是用这个)

2.1.1 Logging

一般用 console.log()

2.1.2 断点(Breakpoints)

2.3 测试框架

2.3.1 QUnit(http://docs.jquery.com/Qunit)

2.3.2 YUITest(http://developer.yahoo.com/yui/3/test/)

2.3.3 JSUnit(http://www.jsunit.net)

(转载本文章请注明作者和出处 Yann (yannhe.com)，请勿用于任何商业用途)

第三章.函数是根基(3.Functions are fundamental)[完成]

翻译 Secrets of the JavaScript Ninja (JavaScript 忍者禁术)

第三章 函数是根基(3.Functions are fundamental)

本章重点：

1.为什么能够理解函数是如此的重要

2.为什么函数是基本类型对象(first-class objects)

3.函数如何被浏览器调用

4.函数的声明(Delaring functions)

5.函数被调用的秘密

6.函数上下文(The context within a function)

目录链接：

http://yannhe.com/secrets-of-the-javascript-ninja-javascript 本文链接：http://yannhe.com/3-functions-are-fundamental 你可能感到很惊讶，为什么当我们讨论 JavaScript 的根基时是要讨论函数而非对象。

我们当然会关心对象(第六章会针对 object)，但是前提是讨论一些对象的实质问题。

当我们写 JavaScript 代码的时候，代码写成什么水平，是写成了平均水平，还是写成了 JavaScript 忍者水平。

就是取决于你是否真正理解了 JavaScript 是一门函数式语言(很重要的一语话)

你的水平就是取决于这个认知。

如果你在读本书，你一定不是一个初学者，我们会认为你是拥有很丰富的面向对象的开发经验(当然我们也会在第六争详细的讨论对象的高级概念)，

但是真正的理解函数在 JavaScript 中的意义，是唯一的一件我们可以挥舞的重要武器。

由于函数如此之重要，我们接下来的会花另外两个章节来深入讨论函数在 JavaScript 中的意义。

值得注意是，在 JavaScript 中，函数是自然类型的对象(first-class objects)。

这就意味着：

1.函数可以被视为普通对象，就像其他数据的类型一样。

2.函数可以被赋值给任何变量，也可以被声明，还有可以被视为另一个函数的参数。

3.1 函数的独特之处是什么？

3.1.1 为什么 JavaScrpit 的函数式如此重要？

3.1.2 sorting with a comparator

3.2 声明

3.2.1 函数的域

3.3 调用

我们都调用过 JavaScript 的函数，但是你是否停下来想一想，函数究竟是如何被调用的。

在本章节中我们会解释一下几种不同的函数调用方式。

不同的函数调用方式，会产生非常不同的影响。其中最重要的一个影响，就是 this 这个参数的定义。

这点不同比你想象要大的多，我们会在接下来的章节中讨论它，这个概念也会伴随着本书余下的内容，

深刻的理解这个概念会让我们写出忍者级别的代码来。

这里列出了四种函数调用方式，他们之间拥有细微的不同：

他们是：

1.作为函数，在这里函数被调用，以一种很直接，易懂的方式。

2.作为方法，方法是连接在对象上面的，被这个对象调用，这中形式就是面向对象编程。

3.作为构造器，在构造的过程中一个新的对象被创建出来。

4.经由函数的 apply()或 call()方法，这是一般比较复杂的概念，所以我们稍后在用到的时候再来讨论。

方法，这是一般比较复杂的概念，所以我们稍后在用到的时候再来讨论。

但是我们最后忘记了一点，函数表达式的最后都会有一个圆括号，它表示了这个函数被调用。

任何在圆括号中传入的变量，都会被翻译成一个列表，作为函数参数来使用。

在我们开始讨论上面那四种函数调用模式之前，我们先来详细的讨论下在函数被调用时候，参数是如何工作的。

3.3.1 变量称为参数(From arguments to function parameters)

调用函数时候传入多个变量，每个变量都会赋值到此函数定义的时候的参数。

例如第一个变量会成称为第一个参数，第二个变量会成为第二个参数，以此类推。

如果传入的变量和函数定义的变量的个数不同的时候并不会报错;

JavaScript 在处理这个问题上很优雅，处理的具体细节如下：

·如果调用时候传入的参数数目多于函数定义的变量个数，多出来的变量不会赋值到函数定义时候的有名字的参数上。但我们仍然有方法获取到这些多出来的变量。

·如果函数定义的参数数目多于传入的变量，多出来的变量则会是 undefined。

非常有趣的一个地方是每个函数在被调用的时候，隐性的传入两个参数：arguments 和 this

隐性的意思是，这两个参数不会在函数定义的时候出现，但是他们会在函数被调用的时候悄悄的传入函数，并且作用域函数的作用域中。

不过他们可以被直接引用，就像其他显性定义的函数参数一样。

下面让我们来讨论一下这两个隐性的参数。

1.THE ARGUMENTS PARAMETER

argments 参数，即函数调用的时候传入函数的变量集合。

这个集合具有 length 属性，表示集合内变量的个数。

并且集合中的变量可以根据下标索引被引用。

argments 看起来很像 JavaScript 中的 Array，但是它并不是 Array。

尽管他们有共同的特征，例如 length 属性，可以通过下标索引到集合中的变量，可以通过 for 语句来进行遍历。

但是你要是用其他 Array 中的方法来调用 argments，就会报错了。

所以 argments 只是"array-like"

相比起 argments，this 这个参数更有意思。

2.THE THIS PARAMETER

当一个函数被调用的时候，我们可以看到变量赋到了显性的参数上，但同时，一个叫做 this 的隐性参数也会传递到函数中。

参数 this 关联到一个对象，这个对象隐性的与函数的调用发生关系，我们叫做函数上下文(function context)

函数上下文的概念来自于面向对象的语言，例如 java 中的 this 指向的是一个类的实例，在类上会定义方法。

但是要注意，这里的方法调用只是我们之前提到的四种调用模式的其中一种。

话说回来，JavaScript 的 this 指向什么，和 java 中的模式是不同的，并不根据函数的定义，而是根据函数是如何被调用的，也就是根据函数的调用模式的不同，this 的指向也不同。

根据这个事实，我们可以换种更清晰的称呼，我们就将 this 叫做调用的上下文(invocation context)，当然我们没有经过官方的协商来给 this 换个称呼。

我们下面将要来看看之前说的四种函数调用模式的不同之处，我们会发现其中最终要的一个不同之处就是在调用时候 this 指向的不同。我们会利用很长的篇幅来继续讨论这个话题。所以不要担心你现在没有理解这

个问题。

现在让我们来看看函数是如何被调用的。

3.3.2 函数调用模式(Invocation as a function)

“作为一个函数来调用"，从字面来看好像没啥意义，函数当然是作为函数来调用，除了函数还能有什么其他的吗？

其实，我们说一个函数的调用模式是作为一个函数来调用，是要与其它三种调用模式做区分。

其他的三种调用模式为：方法调用模式，构造器调用模式，apply/call 调用模式。

当一个函数的调用方式不是这三种之一的时候，我们就将这个调用模式叫做函数调用模式。

这种调用模式的触发是由于函数后面加上()操作符，并且这个函数并不属于任何一个对象的一个属性(如果是属于对象的一个属性，那么这个模式就成为了方法调用模式，我们下面会讨论)。

简单的实例如下：

```
function ninja(){};
ninja();

var samurai = function(){};
samurai();
```

当我们用这种模式来调用函数的时候，函数的上下文(function context)是全局的上下文(global context)，
即 window 对象。

我们先不用代码来说明这个问题，在下面讨论方法调用模式的时候，我们再来写一些有比较的代码。

3.3.3 方法调用模式(Invocation as a method)

一个函数是一个对象的属性，当这个函数被调用时候，这个函数就视为这个对象的一个方法，例如：

```
var o = {};
o.whatever = function(){};
o.whatever();
```

OK,so what?这个函数就叫做“方法”(method),但是，这有啥意义吗？

如果你是来着与面向对象编程的开发人员，你应该记得，方法中的 this 就是这个方法所属的对象。
在这个例子中，也是这么回事。
当这个函数作为一个对象的方法被调用的时候，这个对象就成为了这个函数的上下文(function context),
并且通过 this 函数作为载体存在于函数之中。

这就是最重要的意义之一。

JavaScript 允许面向对象编程的存在(构造器是另外一个，我们稍后讨论)

让我们看几个例子，来对比一下函数调用模式和方法调用模式的不同：

Listing 3.3 Illustrating the difference between function and method invocations

```
function creep(){ return this; }

assert(creep() == window, "Creeping in the window");

var sneak = creep;

assert(sneak() == window, "Sneaking in the window");

var ninja1 = {
    skulk: creep
};

assert(ninja1.skulk() === ninjal, "The 1st ninja is skulking");

var ninja2 = {
    skulk: creep
};

assert(ninja2.skulk() === ninja2, "The 2nd ninja is skulking");
```

运行后，我们得出结论：

函数式的调用模式的 this 为 window

方法式的调用模式为方法所属的对象。

这里值得注意的一个地方，ninja1 和 ninja2 的 skulk 属性引用的是同一个函数 creep

但是当 creep 执行的时候，产生的 this 却是各自不同的。

所以说函数的上下文的产生，并不由函数的定义来决定，而是由函数的调用来决定。

3.3.4 构造器调用模式(Invocation as a constructor)

作为构造器被调用，函数的定义并没有什么不同，不同的地方在于调用的方式。

如果想让一个函数作为构造器被调用，需要用到 new 关键字。

例如，我们的 creep 函数

```
function creep(){ return this;}
```

如果我们想让 creep 函数作为构造器来被调用，我们需要这么写：

```
new creep();
```

但是，就算我们用构造器的模式来调用 creep()，这个函数也不适合作为构造器。

让我来讨论下构造器的特性：

THE SUPER-POWERS OF CONSTRUCTORS

将一个函数作为构造器来调用是一个很有用处的亮点，以下是这些亮点：

1.一个新的空对象被创造出来

2.这个对象被传递给这个构造器作为 this 参数，也就是说这个对象是这个构造器函数的上下文

3.如果没有显性的 return 语句，这个新的对象则会被隐式的 return，并成为这个构造器的值。

这些正是说明了 creep()函数不适合作为构造器。

构造器的目的是为了创建一个新的对象，并且 set it up，返回它，让它作为构造器的值。

让我构建一个更加有意义的函数：一个可以 set up the skulking ninjas

例子 3.4

```
Listing 3.4 Using a constructor to set up common objects
function Ninja(){
    this.skulk = function() { return this;}
}
var ninja1 = new Ninja();
var ninja2 = new Ninja();
assert(ninja1.skulk() === ninja1, "The 1st ninja is skulking");
assert(ninja2.skulk() === ninja2, "The 2nd ninja is skulking");
```

在这个例子中，我们创建了一个构造函数 Ninja()，当我们利用 new 关键字来调用它的时候，

一个空的对象实例会被创造出来并传递道函数中作为 this 存在。

这个构造器同时创建了 skulk 属性，并将此属性作为一个方法赋予给它创造出的新的对象实例。

我们调用了两次构造函数，并产生了 ninja1 和 ninja2 两个不同的实例

对象。

当我们想写一个构造函数的时候，函数的写法会和普通的函数很不一样，让我们接下来详细讨论下。

函数的编写：

构造函数的目的是用来初始化一个新的对象。

当然构造函数也可以和普通函数一样被调用或者象方法一样先赋给一个对象的属性然后调用。

例如，我们可以这样调用 Ninja()函数：

```
var whatever = Ninja();
```

但是结果就变成，skulk 属性被赋予到了 window 上，并且 window 会被 return 并存储在 whatever 中。

由于构造函数的代码和调用模式和其他函数不同，所以一般不会用 new 之外的方式来调用一个构造函数。

如何来标示一个函数是否为构造函数呢？

一般小写字母开头的函数名字就是普通函数例如 skulk()，

creep(),sneak(),doSomethinWonderful()等等，

大写字母开头的既是构造函数，

Ninja(),Samurai(),Ronin(),KungFuPanda()等等。

3.3.5 Invocation with the apply and call methods

到目前为止，我们看到不同的函数调用模式之间最大的差异之处在于，函数上下文的不同，即隐性的 this 参数。

函数调用模式中，this 为 window

方法调用模式中，this 为方法所属的对象

构造器调用模式中，this 为创建的新对象

但是，如果我想要让 this 成为任意我希望的对象，如何做到呢？

如何能够显性操作？

利用 APPLY 和 CALL 方法(USING THE APPLY AND CALL METHODS)

JavaScript 提供给我一种途径，我们可以用任何对象作为函数上下文，并将它显性的操作。

这个途径就是通过每个函数都具备的方法：apply()和 call()

是的，所谓每个函数自身具备的方法，即：函数作为自然类型的对象(firs-class objects)，函数可以拥有属性，包括方法(methods)

当我们通过 apply()来调用一个函数的时候，我们需要传递两个参数给 apply()：

第一个参数：用于作为函数上下文的对象；

第二个参数：一个参数数组

另外 call()是另一个更加简单的形式，我们只需要传递 argment list 来代替参数数组

例 3.5 展示了这两个方法：

```
List 3.5 Using the apply and call methods to supply the function context
function juggle(){
    var result = 0;
    for (var n = 0; n < arguments.length; n++){
        result += arguments[n];
    }
    this.result = result;
}
```

```
var ninja1 = {};
var ninja2 = {};
juggle.apply(ninja1, [1,2,3,4]);
juggle.call(ninja2, 5, 6, 7, 8);
assert(ninja1.result === 10, "juggled via apply");
assert(ninja2.result === 26, "juggled via call");
```

我们看到通过 apply()或者 call()，我们将 ninja1 或 ninja2 作为参数传入到了函数 juggle()中，

并且让 ninja1 称为了函数上下文。

另外要注意的是 apply 和 call 的区别在于，后面的参数，一个是数组，另一个是 argmuments list

这种用法在回调函数中会更加有用

函数上下文在回调函数中的应用(FORCING THE FUNCTION CONTEXT IN CALLBACKS)：

```
Listing 3.6 Building a for-each function to demonstrate setting an
arbitrary function context
function forEach(list, callback){
       for (var n = 0; n < list.length; n++){
          callback.call(list[n], n);
       }
}
var list = ['shuriken', 'katana', 'nunchucks'];
forEach(
    list,
    function(index){
       console.log(index);
       console.log(this);
       assert(this == list[index], "Got the expected value of "+
list[index]);
    }
)
```

3.4 总结(Summary)

在本章，我们学到：

1.基于对 JavaScript 是一门函数式语言的理解，你可以写出更对位的代码。

2.函数是自然类型的对象(fist class objects)，就象其他对象一样，它可以：

```
1) 通过字面量创建
2) 赋予给一个变量
3) 作为参数传递
4) 作为另一个函数的返回值
5) 拥有变量和方法
```

3.函数不同于其他对象的一个超级能力就是它可以被调用

4.函数通过字面量来创建，可以没有函数名

5.函数的域的定义再 JavaScript 中相比起其他语言有些特别：

```
1)函数内部的变量的有效域是定义它的地方到函数结束处
2)内部函数再包裹它的函数内是处处有效的，甚至是定义这个内部函数的前面也有效
```

6.一个函数的参数列表和它实际的 argument 列表的长度可以不同

```
1)如果没有赋与的参数，那么这个参数就是 undefined
2)超出预设长度的 arguments 不会绑定到任何参数上
```

7.所有函数被调用的时候，都会隐性的传入两个 arguments

```
1)arguments: 真正传入的 arguments 集合
2)this: 函数上下文的载体
```

8.函数可以有多种调用方式，调用的方式决定了函数上下文的不同

```
1)最普通的函数调用模式：函数上下文是 window
2)方法调用模式：函数上下文是拥有这个方法的对象
3)构造器调用模式：函数上下文是新生成的对象
4)apply() or call()：函数上下文可以是我们指定的任何对象
```

到此为止，我们介绍了函数的工作原理，下一个章节我们会讨论函数化的知识还有如何应用它

(转载本文章请注明作者和出处 Yann (yannhe.com)，请勿用于任何商业用途)

第四章.挥舞函数(4.Wielding functions)[完成]

翻译 Secrets of the JavaScript Ninja (JavaScript 忍者禁术)

第四章.挥舞函数(4.Wielding functions)

本章重点：

1.为什么匿名函数如此重要

2.函数中的递归

3.函数可以被引用后再调用

4.如何为函数缓存索引

5.利用函数的能力来实现记忆

6.利用函数上下文

7.处理参数长度

8.判断一个对象是否为函数

在上一章我们了解到函数作为自然类型的对象(first-order objects),并且了解到什么是函数式编程。在本章，我们会利用函数来解决一些问题，也许以后做 web 开发时候可以用到。

我们展示的例子并不会直接解决你开发中的问题，那岂不成了另一个什么指南之类的书了。

我们知道，本书的本质目的是让你能够真正的了解这门语言的精髓。

4.1 匿名函数(Anonymous functions)

不知道你是否已经熟悉了匿名函数，匿名函数的确是一个十分重要的概念需要我们来理解，如果你还在为 JavaScript 的忍者头巾而奋斗。他们是是否重要的特点，并且是一个函数化语言的灵魂，例如 Scheme

匿名函数通常会被用于后续使用，例如存储一个变量，作为一个对象的方法，用于回调函数(例如 timeout 或者事件处理)

```
Listing 4.1: Common examples of using anonymous functions
window.onload= function(){ assert(true, 'power!');};
var ninja = {
    shout: function(){
        assert(true, "Ninja");
    }
}
ninja.shout();

setTimeout(function(){ assert(true, 'Forever!')}, 500)
```

我们将会在本书的后面看到大量的匿名函数，因为是否能将 JavaScript 使用的很有力量，取决于你是否将它作为函数式语言在使用。所以我们将会在后面加入很重的函数式的代码。

函数式编程专注于：小，每段小代码只做一个事。

4.2 递归(Recursion)

递归是是否有用的技术。你可能以为递归只是用于数学计算。很多情况下是这样的。

但是他同样适合于其他事情，例如遍历一个树，包括 DOM 本身，我们在 web 开发中经常遇到。

所以递归是个很有用的概念，通过它我们可以更加深刻的理解函数如何在 JavaScript 中作用。

让我们开始通过最简单的方式来看看递归

4.2.1 递归在普通函数中(Recursion in named functions)

我们来写一个例子，这个例子是判断一个字符串是否为对称

我们的实现如下：

```
function isPalindrome(text){
    if (text.length <= 1) return true;
    if (text.charAt(0) != text.charAt(text.length - 1)) return false;
    return isPalindrome(text.substr(1, text.length - 2));
}


Listing 4.2: Chirping using a named function
funcction chirp(n){
    return n > 1? chirp(n -1) + "-chirp": "chirp";
}

assert(chirp(3) == "chirp-chirp-chirp", "Calling the named function comes
naturally.")
```

上面的例子都是在用实名函数在实现递归，让我们下面看看如何利用匿名函数实现递归。

4.2.2 递归在对象的方法中(Recursion with object methods)

我们对上面的例子做个改造，让一个匿名函数赋予在一个对象的属性上。

Listing 4.3: Method recursion within an object

```
var ninja={
chirp: function(n){
return n > 1 ? ninja.chirp(n - 1) + “-chirp” : “chirp”;
}
}
assert(ninja.chirp(3) == “chirp-chirp-chirp”, “An object property isn’t too confusing, either.”)
```

4.2.3 引用的丢失问题(The pilfered reference problem)

我们对上个例子继续改造一下，我们增加一个新的对象，samurai，让它继续引用 ninja 对象的匿名函数。

```
Listing 4.4 Recursion using a missing function reference
var ninja = {
    chirp: function(n){
        return n > 1? ninja.chirp(n - 1) + "-chirp" : "chirp";
    }
}

var samurai = { chirp : ninja.chirp};
ninja = {};
try{
    assert(samurai.chirp(3) == "chirp-chirp-chirp", "Is this going to work?");
}
catch(e){
    assert(false, "Uh, this isn't good! Where'd ninja.chirp go?");
}
```

这里的引用关系是，samurai 和 ninja 同事引用了一个函数,如图 4.2：

这并不是问题所在，问题在于这个共同的函数引用了 ninja 自身，他并不在乎是谁调用的他。

我们可以修复这个问题，相比于通过在匿名函数中引用 ninja 对象，不如通过函数上下文(this)来搞定这个问题：

```
var ninja = {
    chirp: funciton(n){
        return n > 1 ? this.chirp(n - 1) + "-chirp" : "chirp";
    }
}
```

还记得函数作为方法被调用的时候函数上下文就是对象本身吗？

所以当 samurai.chirp()调用的时候，this 就指向了 samurai。

4.2.4 内部实名函数(Inline named functions)

```
Listing 4.5: Using a inline function in a recursive fashion
var ninja = {
    chirp: function signal(n){
        return n > 1 ? signal(n - 1) + "-chirp": "chirp";
    }
}
assert(ninja.chirp(3) == "chirp-chirp-chirp", "Works as we would expect
it to!");

var samurai = {chirp: ninja.chirp};
ninja = {};

assert(samurai.chirp(3) == "chirp-chirp-chirp", "The method correctly
calls itself.");

Listing 4.6: Verifying the identity of an inline function
var ninja=function myNinja(){
    assert(ninja == myNinja,"This function is named two things at once!");
}

ninja();

assert(typeof myNinja == "undefined", "But myNinja isn't defined outside
of the function.")
```

4.2.5 callee 属性(The callee property)

我们来看另外的一个函数的概念：arguments 参数的属性 callee

```
Listing 4.7: Using arguments.callee to reference the calling function

var ninja={
    chirp: function(n){
        return n>1? arguments.callee(n-1) + "-chirp" : "chirp";
    }
}

assert(ninja.chirp(3) == "chirp-chirp-chirp", "arguments.callee is the
function itself.")
```

arguments 有一个属性叫做 callee，这个属性指向的是当前执行的函数。这个属性可以一直用于在函数内部获取到函数自身。

4.3 函数作为对象(Fun with function as objects)

在 JavaScript 中的函数并不像其他语言中的函数，JavaScript 给予了函数很多的能力，不只是被当作自然对象(first-class objects)

我们已经看到，函数可以拥有属性，可以拥有方法，可以被赋到变量和属性上，但是最牛的一个能力是他们可以被调用(callable)

首先让我们看一下将函数缓存在一个集合中，然后要研究“memoizing“这个技术。

4.3.1 存储函数(Storing functions)

很多时候，我们会想要存储一些具有唯一标识的函数。

当我们想添加一个函数到一个集合中的时候，我们需要判断这个函数之前是否已经被添加到这个集合中了。

之前我们用的普通的方法是将函数添加到一个数组里，每次添加新的函数的时候都要遍历一次数组进行比较，我们下面要做得更好一些。

我们可以利用函数自身的属性来达到这个目的。

```
Listing 4.8: Storing a collection of unique functions
var store = {
    nextId: 1,
    cache: {},
    add: function(fn){
        if (!fn.id){
            fn.id = store.nextId++;
            return !!(store.cache[fn.id] = fn);
        }
    }
}

function ninja(){};

assert(store.add(ninja), "Function was safely added.");
assert(!store.add(ninja), "But it was only added once.");
```

这里要注意：!!是一个很简单的方式，让任意 JavaScript 表达式变成 Boolean 的方式，

例如：!!“hello” === true and !!0 === false

4.3.2 自身缓存函数(Self-memoizing functions)

Memoization 是让函数具有记忆能力的技术，可以让函数记住上一次计算的结果。他可以提升效率。

MEMOIZING EXPENSIVE COMPUTATIONS

如果是负责计算，这里的情况很适合。我们会构建一个"answer cache" ，它会存储上一次运行的结果。

```
Listing 4.9: Meemoizing previously-compued values
function isPrime(value){
    if (!isPrime.anwers) isPrime.answers={};
    if (isPrime.answers[value] != null){
        return isPrime.answers[value];
    }
    var prime = value != 1; // 1 can never be prime
    for (var i = 2; i < value; i++){
        if (value % i == 0){
            prime = false;
            break;
        }
    }
    return isPrime.answers[value] = prime;
}

assert(isPrime(5), "5 is prime!");
assert(isPrime.answers[5], "The answer was cached!");
```

MEMOIZING DOM ELEMENTS

```
function getElements(name){
    if (!getElements.cache) getElements.cache = {};
    return getElements.cache[name] =
        getElements.cache[name] || document.getElementsByTagName(name);
}
```

4.3.3 Faking array methods

我们来构造自定义的一个类似 Array 的对象

```
Listing 4.10: Simulating array-like methods
<input id="first"/>
<input id="second"/>
var elems = {
    length: 0,
    add: function(elem){
        Array.prototype.push.call(this, elem);
    },
    find: function(id){
        this.add(document.getElementById(id));
    }
}

elems.find("first");
assert(elems.length == 1 && elems[0].nodeType, "Verify that we have an
element in our stash");

elems.find("second");
assert(elems.length == 2 && elems[1].nodeTpe, "Verify the other
insertion");
```

4.4 Variable-length argument lists

JavaScript 是一门很灵活的语言.其中具体的一个特点是他可以让函数接受任意长度的 arguments。

我们下面要看一些列子来如何应用这个特性来增强他们的能力,通过这些例子我们可以学到：

1.如何支持任意数量的 arguments

2.如何利用 variable-length argument 列表来实现函数的重载

3.如何认识和使用 argument 列表的 length 属性

让我们来利用 apply()来搞定这些

4.4.1 Using apply() to supply variable arguments

如果我们求一个集合的最大值，我们会想到 Math

例如:

var biggest = Math.max(1,2);
var biggest = Math.max(1,2,3);
var biggest = Math.max(1,2,3,4);

但是，我们不能这么做：

var biggest = Math.max(listp[0], list[1], list[2]);

如果搞定这个问题?通过 apply()或者 call()

```
Listing 4.11: Generic min and max funcctions for arrays
function smallest(array){
    return Math.min.apply(Math, array);
}

function largest(array){
    return Math.max.apply(Math, array);
}

assert(smallest([0,1,2,3]) == 0, "Located the smallest value.");
assert(largest([0,1,2,3]) == 3, "Located the largest value.");
```

这里通过 apply 将数组的参数转换成了正常参数：

Math.min(0,1,2,3);

4.4.2 Function overloading

在 3.2 章节，我们知道 argments 参数是隐性的传递给被调用的函数的，现在让我们详细的看一下这它。

所有函数都接受到这个隐性的参数，它给了我们力量来处理未知数目的参数。

让我们来看看如何通过它来实现函数的重载(function overloading)

DETECTING AND TRAVERSING ARGUMENTS

在大部分面向对象的语言中，重载一般是通过定义同样的名字的函数，通过定义不同的参数来实现区别。但是在 JavaScript 不是这样，在 JavaScript 中我们可以只用一个函数，在函数内部通过判断参数的个数来实现逻辑划分。

在下面的例子中，我们要 merge 两个对象的属性到一个 root 对象中。

```
Listing 4.12: Traversing variable-length argument lists
function merge(root){
    for (var i = 1; i < arguments.length; i++){
        for (var key in arguments[i]){
            root[key] = arguments[i][key];
        }
    }
    return root;
}

var merged = merge(
    {name: "Batou"},
    {city: "Niihama"});

assert(merged.name == "Batou", "The original name is intact.");
```

```
assert(mmerged.city == "Niihama", "And the city has been copied over.");
```

SLICING AND DICING AN ARGUMENTS LIST

我们要利用 arrays 的 slice()方法来忽略 arguments 的第一个参数

让我们来看看例 4.13

```
Listing 4.13: Slicing the arguments list
function multiMax(multi){
    return multi*Math.max.apply(Math, arguments.slice(1));
}

assert(multiMax(3,1,2,3) == 9, "3*3 =9(First arg, by largest.");
```

运行之后报错，因为 argument 不是 Array，所以他没有 slice 方法。

让每重写一下这段代码

```
Listing 4.14: Slicing the arguments list - successfully this time
function multiMax(multi){
    return multi * Math.max.apply(Math,
       Array.prototype.slice.call(arguments, 1));
}

assert(multiMax(3,1,2,3) == 9, "3*3=9 (First arg, by largest.");
```

我们利用 Array 的 slice()方法，让 arguments 看起来也是一个 array，尽管它自身不是。

FUNCTION OVEERLOADING APPROACHES

普通的做法是根据参数的不同在函数内部写 if-then-else-if 代码块，但是这样又太不简洁。

我们可以利用一个不常用的函数的属性来实现代码的简洁化。

THE FUNCTION'S LENGTH PROPERTY

函数中有一个我们很少知道的属性，但是它能告诉我们这个函数是如何被定义的，它就是 length 属性。

请不要把这个属性和 arguments 的 length 属性混淆，它是专指，被显性定义的函数的参数的个数。

因此，如果我们只是定义了一个入参的函数，那么这个属性值就是 1。

代码示例如下：

```
function makeNinja(name){}
function makeSamurai(name, rank){}
assert(makeNinja.length == 1, “Only expecting a single argument”);
assert(makeSamurai.length == 2, “Two arguments expected”);
```

所以针对一个函数，我们可以断定 2 个事情：

1.通过函数 length 属性，我们可以知道他的显性参数的数目

2.通过 arguments 的 length 属性，我们可以知道调用的时候真正传递进去的参数个数。

让我们看看如同运用这个属性来实现函数的重载

OVERLOADING FUNCTIONS BY ARGUMENT COUNT

这里有很多方式来根据参数实现函数的重载。

一个普遍的做法是根据参数的类型，另一种是根据参数的个数。

让我们看看如何通过参数的个数来实现：

```
var ninja = {
    whatever: function(){
        switch(arguments.length){
            case 0:
                /* do something */
                break;
            case 1:
                /* do something else */
            case 2:
                /* do yet something else */
                break;
            // and so on...
        }
    }
}
```

在这段代码中，每个 case 对应的是 argument 实际传入的数量。

但是不够简洁，不够忍者。

让我们再写一段代码，如果我们想让重载的逻辑在调用的时候：

var ninja = {}
addMethod(ninja, 'whatever', function(){/* do something */});
addMethod(ninja, 'whatever', function(a){/ do something else */});*
addMethod(ninja, 'whatever', function(a,b){/ yet something else */});*

通过这种方式，我们在调用的时候才定义重载的逻辑，漂亮并且简洁吧。

但是我们还没有定义 addMethod 这个函数，下面我们来实现它。

让我们来看例 4.15

```
Listing 4.15 A method overloading function
function addMethod(object, name, fn){
    var old = object[name];
    object[name] = function(){
        if (fn.length == arguments.length)
            return fn.apply(this, arguments)
        else if (typeof old == 'function')
            return old.apply(this, arguments);
    }
}
```

我们的 addMethod()函数接受了三个参数：

1.一个对象，作为载体

2.一个要被绑定的方法名字

3.要被绑定的方法的声明

让每看例 4.16 来测试一下我们的新函数：

```
Listing 4.16 Testing the addMethod function

var ninjas = {
    values: ["Dean Edwards", "Sam Stephenson", "Alex Russell"]
}

addMethod(ninjas, "find", function(){
    return this.values;
})

addMethod(ninjas, "find", function(name){
    var ret = [];
    for (var i = 0; i < this.values.length; i++)
        if (this.valuespi].indexOf(name) == 0)
            ret.push(this.values[i]);
    return ret;
})

addMethod(ninjas, "find", function(first, last){
    var ret = [];
    for(var i = 0; i < this.values.length; i++)
        if (this.values[i] == (first + " " + last))
            ret.push(this.values[i]);
    return ret;
})

assert(ninjas.find().length == 3, "Found all ninjas");
assert(ninjas.find("Sam").length == 1, "Found ninja by first name");
```

```
assert(ninjas.find("Dean", "Edwards").length == 1, "Found ninja by first
and last name");
assert (ninjas.find("Alex", "X", "Russell") == null, "Found nothing");
```

这段代码很整洁，因为我们没有将函数真正存储在一个明显的数据结构中。我们是通过闭包(closures)来实现的，我们会在下一个章节来讨论闭包

本章到此为止，我们学习了一个函数如何被当作一个自然对象来使用，我们还要知道如何判断一个对象是否是一个函数。

4.5 Checking for functions

大多数浏览器都可以通过 typeof 来判断一个对象的类型，

例如：

function ninja(){}
assert(typeof ninja == “function”, “Functions have a type of function”);

当然有些浏览器是不支持这么写。这就要设计到浏览器兼容性问题

4.6 Summary

1.匿名函数可以让代码更简洁

2.递归函数让我们知道函数如何被引用：

```
1)通过名字
2)通过方法
3)通过变量
4)通过 arguments 的 callee 属性
```

3.函数拥有一些属性，我们可以利用这些属性来存储信息，包括：

```
1)存储函数，可以后续来调用和引用
2)利用函数的属性来实现缓存(memoization)
```

4.通过控制函数上下文，我们可以让一个对象的方法不再为这个对象服务，可以利用 Array 和 Math 拥有的方法来为我们所用，来计算我们自己的数据。

5.函数可以根据参数的不同而执行不同的逻辑(function overloading).

6.利用 typeof 关键字我们可以检查一个对象实例是否为函数.这里要考虑浏览器兼容性问题。

(转载本文章请注明作者和出处 Yann (yannhe.com)，请勿用于任何商业用途)

第五章.闭包（5.Closing in on closures)[完成]

这章写得极为精彩，让我明白了好多以前根本不知道的概念。

闭包是一个神奇的东西，新果说过，闭包就是可以当作鞭子来用，可以打出鞭子头一样的力道。

所谓的函数化这个概念中，闭包是一个很重要并且很精髓的概念。

想要理解函数化编程，就要深刻的理解闭包。

本章重点：

1.闭包的定义，闭包是什么，闭包如何工作

2.利用闭包来实现一些简单的开发

3.利用闭包实现性能上的增强

4.利用闭包实现私有域

5.1 闭包如何工作（How closures work)

问：闭包是什么？

答：简单来说，closure 是一块域，这块域是由创建一个 function 而来，这个 function 可以访问和操作它的外部的变量

(Simply put, a closure is the scope created when a function is declared that allows the function to access and manipulate variables that are external to that function.)

这个概念最好还是用代码来解释，所以让我们看看 5.1 这个例子：

Listing 5.1: A simple closuer

```
var outerValue = 'ninja';
function outerFunction(){
    assert(outerValue == 'ninja', "I can see the ninja")
}
outerFunction();
```

你可能写过 n 多次这样的代码，可你竟然没有意识到你在创建闭包(closure)！

你不相信？估计是因为你没有感觉到有任何的惊喜。

因为 out value 和 outer function 的作用域是全局域(global scope)，而全局域是永远不会消失(自从这个页面被加载就存在了)，

另外就算是这个 function 可以访问 outer value，这个特点也没有实用的价值。

所以就算 closure 已经存在了，它的用处也不明显。

让我们给他加上一点料，来看看 5.2 这个列子吧：

Listing 5.2: A not-so-simple closue

```
var outerValue = 'ninja';
var later;

function outerFunction(){
    var innerValue = 'samurai';

    function innerFunction(){
        assert(outerValue, "I can see the ninja");
        assert(innerValue, "I can see the samurai");
    }

    later = innerFunction;
}

outerFunction();
```

```
later();
```

首先，我们定义了 later 这个变量，我们会在后面用到它。

然后，我们在 outer function 中定义了 inside 这个变量，这样 inside 就被限制在 outerFunction 域内了。

然后，我们在 outerFunction 中定义了一个 innerFunction，innerFunction 可以访问我们已经定义过的所有外部变量。

是的！我们可以这样做！记住上一章我们讨论过的吗？

function 是自然类型的对象(first－class objects)，它和变量一样在任何地方都可以被创建。

我们当然也可以将 inner function 赋给 later 变量，以便稍后调用。

一切就绪，我们开始运行。

我们调用 outer function，创建了 inner function，然后 inner function 赋给了 later 变量（全局变量），然后通过调用全局变量 later，等价于调用了 inner function。

发生了什么事情？

1.inner function 可以访问 outerValue，因为 outerValue 属于全局域，

2.由于 js 的特性，在正常情况下，全局域中我们是无法访问到 innerValue 的，但是通过将 innerFunction 赋给 later 变量，later 就称为了域之间的桥梁，我们自然在全局域可以访问到 innerValue。

这是什么原理呢？答案就是，闭包（closures）！

当我们创建 innerFunction 的时候，不仅仅是定义了一个 function，我们还创建了一个域。

在外部掉用 innerFUnction 的时候，虽然调用发生在全局域，但是 innerFunction 还是会访问当初被定义的时刻的那个原始域。这个域就是它的闭包（closure）

让我们来看看下一个例子：5.3

Listing 5.3: What else closures can see

```
var outerValue = 'ninja';
var later;

function outerFunction()
{
    var innerValue = 'samuai';

    function innerFunction(paramValue)
    {
        assert(outerValue, "Inner can see the ninja");
        assert(innerValue, "Inner can see the samurai");
        assert(paramValue, "Inner can see the wakizashi");
        assert(tooLate, "Inner can see the ronin");
    }

    later = innerFunction;
}

assert(!tooLate, "Outer can't see the ronin");

var tooLate = 'ronin';

outerFunction();
later('wakizashi');
```

这个例子中表达了 3 个关于 closure 的概念：

1.function 的参数也被包含在闭包之中

2.所有的外部变量都属于闭包可访问的范围（外部域），包括那些在闭包后面创建出来的变量。

3.在同一个域中，不可以访问后面才创建的变量

第 2 和第 3 点解释了为什么只有 inner closure 可以看见 tooLate，而 outer closure 则不可以看见 tooLate。

虽然闭包并不引人注意（你并不能通过工具检测到“闭包”这个对象），但是我门还是要意识到闭包其实是占用了内存的。

请记住，闭包威力无穷，但是也有危险。

如果你有强烈的意愿想用好闭包，闭包毫无疑问是威力无穷的。

但是它理所当然需要系统开销。

所有闭包中的信息数据都已经缓存在内存之中，

如何释放掉这块内存呢？

这里有两种办法：

1.通过 JavaScript 引擎的 GC 垃圾回收机制

2.刷新或者关闭当前页面。

5.2 让闭包用起来(Putting closures to work)

现在，我们已经明白了闭包是什么，并且知道它如何使用（虽然是很 high level 的），

现在我们要真正让闭包用起来。

5.2.1 私有变量(Private variables)

在 JavaScript 中如何实现让一个域拥有私有的变量呢？

答案就是使用闭包。

我们看到大部分 JavaScript 菜鸟都是在用面向对象(Object-oriented)的方式来编写 JavaScript,

这样做带来的缺点就是无法实现私有变量。

在例子 5.4 中我们会看到闭包是如何搞定私有变量的

Listing 5.4：Using closures to approxiate private variables

```
function Ninja()
{
    var slices = 0;

    this.getSlices = function()
    {
        return slices;
    }

    this.slice = function()
    {
        slices++;
    }
}

var ninja = new Ninja();
```

```
ninja.slice();

assert(ninja.getSlices() == 1,
    "We're able to access the internal slice data.");
assert(ninja.slices == undefined,
    "And the private data is inaccessible to us.");
```

这个例子表明了，Ninja 这个函数中的数据状态，是由 Ninja 这个函数来维护，

外部是无法干预到 Ninja 这个函数内部的变量的。

函数内部的变量，只允许 Ninja 的内部方法来访问。

现在，让我们回过神来，我们要继续探索闭包的另一个牛逼的用法。

5.2.2 Callbacks and timers

另一个闭包的牛逼用法，就是实现了回调(callbacks)和定时器(timer)

让我们来看一个利用 jQuery 来实现 Ajax 请求的例子 5.5

Listing 5.5: Using closures from a callback for an Ajax request

```
var jQuery = function(){
    return {
        click:function(){}
    }
};

jQuery('#testButton').click(function(){
    var elem$ = jQuery("div");
    elem$.heml("Loading...");

    jQuery.ajax({
        url: test.html,
        success: function(html){
            assert(elem$,
                "We can see elem$, via the closure for this callback.");
```

```
            // elem$.html(html);
        }
    })
})
```

jQuery 大家制定用的很熟悉了，这里就不过多解释了，如果你看不懂这个例子，先去 http://www.w3cschool.cn/index-30.html 看看 jQuery 的概念吧，我只能帮你到这里了 Brother。

下面让我们看看 Timer 的例子：

Listing 5.6: Using a closure in a timer interval callback

```
var elem = document.getElementById("box");
var tick = 0;

var timer = setInterval(function(){
    if (tick < 100){
        elem.style.left = elem.style.top = tick + "px";
        tick++;
    }else{
        clearInterval(timer);
        assert(tick == 100,
            "Tick accessed via a closure.");
        assert(elem,
            "Element also acessed via a closure.");
        assert(timer,
            "Timer reference also obtained via a closure.")
    }
}, 10);
```

下面，让我们来看看如何让 function 的上下文(contexts)为我们工作。

5.3 Binding function contexts

我们在上一章已经讨论果函数上下文这个概念，你还记得如何更改一个函数的上下文吗？

对了！用.call()和.apply()。

ps:你还记得 call 和 apply 的区别吗?不记得的话就回去看看吧。

在下面这个例子中，我们想让一个对象的方法绑定到一个 dom 元素上

Listing 5.7:Bingding a specific context to a function

```
var button = {
    clicked : false,
    click: function(){
        assert(this == elem, "This is the elem, not the object button")
        this.clicked = true;
        assert(button.clicked, "The button has been clicked");
    }
}

var elem = document.getElementById("test 5.7");
elem.addEventListener("click", button.click, false);
```

运行上面这个例子之后，我们会发现，结果并不是我们预期的。

原因是 click 方法并没有向我们预期的那样绑定在了 button 对象上。

根据我们之前学习的第三章的知识，如果我们这么调用：
button.click()
那么函数的的上下文就会是 button 对象了。

但是在这个例子中，函数的上下文则是 dom 中的元素，不是 js 中的 button 对象。

如何解决这个问题呢？

答案还是利用闭包！

我们来改进一下上面的例子，请看例子 5.8

Listing 5.8: Bingding a specific context to an event handler

```
function bind(context, name){
    return function(){
        return context[name].apply(context, arguments);
    };
}

var button = {
    clicked : false,
    click: function(){
        assert(this == button, "This is the object button")
        this.clicked = true;
        assert(button.clicked, "The button has been clicked");
    }
}

var elem = document.getElementById("test 5.8");
elem.addEventListener("click", bind(button,"click"), false);
```

bind()函数被广泛的应用在了 JavaScript 的原型库中(Prototype JavaScript Library)，

这样就可以优雅的写出典型的面向对象的代码(Object-oriented)

即便如此，在需要编写事件机制的代码时候，还是很容易出现函数上下文错误的问题。

让我们来看看面向对象的代码风格如何实现 bind 的，请看例子:5.9.

Listing 5.9: An example of the function binding code used in the Protoype library

```
Function.prototype.bind = function(){
```

```
    var fn = this,
    args = Array.prototype.slice.call(arguments),
    object = args.shift();

    return function(){
        return fn.apply(object,
            args.concat(Array.prototype.slice.call(arguments)))
    };
};

var myObject = {};
function myFunction(){
    return this == myObject;
}

assert( !myFunction(), "Context is not set yet");

var myFunction = myFunction.bind(myObject);
assert(myFunction(), "Context is set properly");
```

注意，这里.bind()的目的并不是要取代.apply()或者.call()。

这个例子想要说明的是，我们可以通过闭包和匿名函数(anonymous function)来操控上下文(context),

进而实现，当延迟调用此函数的时候，上下文也不会更改。

这种情形在回调(callbacks)和事件机制(event handlers and timers)中会很有帮助。

5.4 滚雪球式的函数(Partially applying functions)

同学们，注意啦，精彩的章节来了！！！想知道如何利用闭包聚集力量，发出致命一击吗？

忍着师傅要传授绝密武功了，请仔细的阅读本章节。

在学习此章节前，我们先来思考几个题外话。

你们有没有感觉到，想要做好任何一类事情，都可以找到这个事情的一个巧劲。

例如，打羽毛球，不应该用蛮力击球，而是应该找到，放松的通过一段位移，挥舞牌子产生惯性的力来击球。说白了就是一种放松情况下产生的力道。

例如，李小龙悟到的打拳的发力，不应该用蛮力，而是让肌肉放松，将手臂想象成鞭子，将拳头甩出去打人。

例如，弹钢琴，不应该用蛮力，应该用手指产生的惯性力来敲打键盘，有一句话，钢琴是弹的不应该是按的。这里关于钢琴的正确发力的学问，叫做触键。

例如，制造一个大雪球，如果用蛮力来做，这是在地上堆砌雪球，而正确的做法是先在山顶制造一个小雪球，然后通过向下滚动，逐渐让雪球逐渐变大，最后成长为一个大雪球。

总结看来，这些巧劲的共性，就是利用距离和时间转化为力量（类似与势能转换为动能）。而不是蛮力本身的传递。

这样的例子太多了，我们做任何事情，都应该意识到，核心的问题是如何找到那种巧劲，编程也是如此。

那么 JavaScript 的巧劲是什么呢，请看下文分解？

“滚雪球式”(“Partially applying”)是一个非常有趣的技术，

利用这个技术，我们可以预先将参数安装到函数中，而不用立即执行此

函数。

等到我们想执行此函数的时候，我们可以在任意时刻触发。

这里就是将时间(提前按照参数)和距离(中间增加函数)看作势能，

而实际调用此函数的那一瞬间则是是将势能转换为动能。

我们来看看本书作者如何定义的。

本书作者引用了传统的一个词汇，“currying”

定义如下：

首先将一批函数转入一个函数(然后这个函数返回一个新的函数)，这中形式就叫“做科里化”（currying）

我想到新果曾经在解释 currying 的时候写的一个例子，如下：

用 currying 化实现 2 个数相加。

```
var add = function(a)
{
    return function(b){

        return a + b;
    }
}

assert(add(1)(2) == 3, "Curring: add two integer");
```

下面我们来看本书中的如何通过例子来表达 currying：

例如我们现在有个需求，需求是我们想将一个字符串转换为 CSV 格式的一个字符串数组。

如果是菜鸟，我们一般这样写：

```
var elements = "val1, val2, val3".split(/, \s*/);
```

有没有觉得上面这种写法很笨拙，每次想将一个字符串转换为 CSV 格式的时候，还必须得记住这个特定得正则表达式。

同学们，我们得目标是成为牛逼闪闪得 JavaScript Ninja，我们要用国际范的方式来搞定这个问题。

so 让我们创建一个 csv()方法来做这个事吧。

让我们想象一下，如果用 currying 化来编写这个 csv()呢？

请看例子 5.10

Listing 5.10:Partially applying arguments to a native function

```
Function.prototype.curry = function(){
    var fn = this,
    // 这里就是在预装参数，将参数抓住，缓存在变量 args 中
    args = Array.prototype.slice.call(arguments);
    return function(){
        return fn.apply(this, args.concat(
            Array.prototype.slice.call(arguments)))
    }
}

String.prototype.csv = String.prototype.split.curry(/,\s*/);
var results = ("Mugan, Jin, Fuu").csv();

assert(results[0] == "Mugan" &&
       results[1] == "Jin" &&
       results[2] == "Fuu",
       "The text values were split properly");
```

上面这个例子看懂了吗？

curry 这个函数做的事情，是将函数中的 this 和 arguments 缓存在了闭包之中。

当 split 函数调用 curry 的时候，curry 中的 this 就是 split 函数本身，

正则表达式这个参数则是预存在 curyry 中的 arguments。

虽然这段代码实现已经很不错了，但是我们还可以进行一些改进。

上面这个 curry 函数，实现了将所有参数缓存在闭包之中。

当调用闭包返回的新的函数的时候才去处理之前预装的所有参数。

但是如果我不想让所有的参数全部预装，而是只预装其中一部分，

另外一部分参数要在调用闭包返回的那个新的函数中，将这部分参数传入。

这类问题，已经在其他语言中实现了，Oliver Steele 就是其中的一个作者。

他开发的类库叫

Functional.js(http://osteele.com/sources/javascript/functional/)

让我们看看我们自己的实现，例子：5.12

Listing 5.12: A more-complex “partial” function

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

        }
        return fn.apply(this, args);
```

```
    }
}

var delay = setTimeout.partial(undefined, 10);
delay(function(){
    assert(true,"A call to this function will be delayed 10 ms");
})


var bindClick = document.body.addEventListener.partial("click",
undefined, false);

bindClick(function(){
    assert(true, "Click event bound via curried function.");
})
```

partial()的实现和 curry()有些相像。

在第一批的参数中，我们用 undefined 代表了那部分缺省参数。

在后面 delay 调用的时候，才将真正想要被处理的参数传递进去。

5.5 函数的重写(Overriding function behavior)

JavaScript 是一个很有趣的语言，在 JavaScript 中，

你可以在用户毫无感知的情况下，完全的控制一个函数的内部行为。

具体来说有两种方法可以做到这件事：

1.更改已经存在的函数（不需要通过闭包）

2.基于已经存在的函数，来创建一个新的函数，在这个新的函数中加入自身更改的逻辑

记得第四章的一个例子吗？函数结果的记忆能力(memoization)

让我们来看看另一种实现的版本

5.5.1 函数的记忆能力(Memoization)

所谓 memoization，即让一个函数具备一种可以记忆它历史被调用时候产生的运算结果的能力。

在第四章中，我们实现 memoization 的方法是采用了直接去更改一个已经存在的函数。

这样做真是太粗暴了。

我们需要优化这个实现。

让我们创建一个叫做 memoized()的方法，在例子 5.13 中，

我们实现了用 memoized()来记住一个已经存在的函数的返回值。

在这个例子中我们并没有用到闭包

Listing: A memorization method for functions.

```
Function.prototype.memoized = function(key){
    this._values = this._values || {};
    return this._values[key] !== undefined ?
        this._values[key] :
        this._values[key] = this.apply(this, arguments);
}


function isPrime( num ){
    var prime = num != 1;
    for ( var i = 2; i < num; i++)
    {
        if (num % i == 0){
            prime = false;
            break;
        }
    }
    return prime;
}
```

```
assert(isPrime.memoized(5), "The function works; 5 is prime.");
assert(isPrime._values[5], "The answer has been cached.");
```

在上面这个例子中，结果都被缓存在了_values 中。

有趣的一点是，计算和存储是在一个 single step 中。

结果是通过.apply()来调用父函数来完成，然后直接将结果存储在了父函数的属性_values 中。

所以整个事件链是这样：

计算出结果，保存结果，返回结果。所有这些都是在一个逻辑代码块中完成(a single logical unit of code)

这样做的缺点在于：调用 isPrime()的时候还必须记得要调用.memoized()方法，来实现 memoization。

如果解决这个缺点呢？

用闭包！

Listing 5.14: A technique for memorizing functions using closures

```
Function.prototype.memoized = function(key){
    this._values = this._values || {};
    xyz = this._values;
    return this._values[key] !== undefined ?
        this._values[key] :
        this._values[key] = this.apply(this, arguments);
}

// 这里得 memoize 就是利用闭包的特性，来隐性的更改了 ipPrime 的行为
Function.prototype.memoize = function(){
    var fn = this;                          //#1
    return function(){                       //#2
        return fn.memoized.apply(fn, arguments);
    }
}
```

```
var isPrime = (function ( num ){
    var prime = num != 1;
    for ( var i = 2; i < num; i++)
    {
        if (num % i == 0){
            prime = false;
            break;
        }
    }
    return prime;
}).memoize();
```

在 5.13 这个例子的基础上，我们通过一点改进，完美的解决了之前的那个缺陷。

在本例中，我们创建了一个新的方法，memoize()。

在 memozie()中，利用 memoized()重新包裹了原始函数(original function)。

这样一来，memoize 返回的函数就永远会是具备了 momoized 能力的函数(#2)

请注意，在 momozie()这个方法中，我们构建了一个闭包，

在这个闭包中通过 fn 这个变量拷贝了原始函数：isPrime 函数(通过获取其上下文 ＃1)，

这是一个很通用的技巧：每个函数都拥有自己的上下文，所以上下文从来都不是闭包的一部分。

但是我们可以利用一个变量来关联原始函数的上下文的值，

这样就等同于函数上下文(isPrime context)成为了闭包的一部分。

由于原始函数已经成为了闭包的一部分，那么闭包自然可以在返回的新

函数中，

随意的操作原始函数(isPrime)，让其具备记忆能力.

在例 5.14 中我们还使用了一个特别技巧：立即执行一个函数（immediately)

在后面 5.5.3 这个章节中，我们会详细的讨论这个技巧。

例 5.14 是一个很好的展现了闭包的威力的例子，但是我们也要知道，这种技巧同样也会有缺陷。

如果过多的利用闭包来隐性的修改一个函数的逻辑。那么将会让这个函数成为一个很难被继承的函数。

过于极端的使用这个技巧也是不可取的。

不过无所谓，因为后面我们将会学习一种技巧来搞定这个缺陷，

那就是将这部分逻辑封装在一个可以延迟调用的函数中。

5.5.2 函数包装(Function wrapping)

函数包装是一个的用来封装函数功能的技巧。

如果想要继承或者创建一个新的函数的时候，通过函数包装可直接实现。

最有价值的一个场景是：在我们想要重写(override)一些已经存在的函数的情况下，

并且可以保持在原始函数中那些有用的部分可以在被包装后仍然有效。

另外一个普遍的场景是：兼容不同的浏览器。

例如在 Opera 浏览器中，有一个获取 title 属性的 bug。

Prototype 类库采用了函数包装这个技术来对付这个 bug。

为了防止在 readAttribute()函数中，过多的出现 if/else 这样的代码快(这是一个比较丑陋，并且不是一个特别好的分割逻辑的方式)。

Prototype 选择利用函数包装(function wrapping)来重写(override)那个老的方法，

但是原始函数中的那些正常的逻辑还应该有效

(这一段比较难翻译，之前读的时候，并没有完全理解作者真正想要表达的那些有价值的细节)

让我看例 5.15

Listing 5.15: Wrapping an old function with a new piece of functionality

```
function wrap(object, method, wrapper){
    var fn = object[method];                //#1
    return object[method] = function(){     //#2
        return wrapper.apply(this, [fn.bind(this)].concat(
            Array.prototype.slice.call(arguments)));
    }
}

// Example adapted from Prototype
if (Prototype.Browser.Opera){               //#3
    wrap(Element.Methods, "readAttribute",
        function(original, elem, attr){
            return attr == 'title' ? elem.title : original(elem, attr);
        })
```

```
}

//#1 缓存原始函数
//#2 返回新的包装函数
//#3 实现函数包装
```

我们来分析一下 wrap()函数是如何工作的。

在传入 wrapp()的参数中包括：

1.一个基础的对象(object)

2.这个基础对象想要被包装的方法(method)

3.新的包装函数(wrapper)

首先，我们将原始方法保存在变量 fn 中(#1)；

我们在后面会通过匿名函数的闭包(anonymous function’s closure)来调用它。

然后我们用一个新的函数来重写这个方法(#2)

这个新的函数执行了之前传进来的 wrapper 函数(通过一个闭包)，并且传给它一个重新构造的参数列表.

在构建这个参数列表时，我们想让第一个参数是我们正在重写的原始函数(original function)，

这样一来，我们创建一个数组来连接原始函数(原始函数的上下文已经被绑定，通过 bind()函数，我们之前在例 5.3 中创建的，同样适用于包装函数)，然后将原始参数追加到这个数组中。

在 apply()函数中，如我们所见，用这个数组作为了他的参数列表。

Prototype 类库利用 wrap()函数，实现了对一个已经存在的函数的重写(在这个例子中是重写了 readAttribute 函数)
将这个函数替换成了一个包装后的新函数(#3).
然而，这个新的函数仍然可以拥有原始函数的原始功能(通过原始参数)。
这就意味着，一个函数被重写之后，仍然可以保留原始的功能，这是一个很安全的技术。

结论就是，wrap()函数是一个可以重用的函数，我们可以用一种比较隐蔽的方式，
来重写任何对象的方法中的已有的功能。
这又是一个彰显闭包威力的案例。

现在，让我来看一个经常会被用到，也很古怪的一个表达式。
它同样也是函数式编程的一个重要部分。

5.6 即时函数(Immediate functions)

让我们来看一个简单的代码结构，这是一个函数化编程中很重要的部分，同样也是闭包中我们还未展示的一个亮点。
代码如下：

```
(function(){})()
```

这中模式的代码，毫无疑问的可以用在很多地方，
它给 JavaScript 带来了很多出乎意料的能力。

这段代码中的花括号和原括号看起来比较陌生，

让我们来一点一点的分析一下：

首先，让我们忽略第一个原括号里的内容，让代码变成这样：
(…)()
我们知道，我们可以通过函数名加原括号(functionName())这样的代码表达式，来调用任意的一个函数

但是我们可以用任意的一个关联函数实例的表达式，来替换函数名(function name).
这正是为什么我们可以调用一个指向函数的变量(variableName())。

所以在(…)()这个代码中，第一组圆括号只是用来划定逻辑范围，第二组圆括号则是操作符。

不同位置的圆括号表示的意义不同，这里可能会有一点小混乱。

如果函数的调用的操作符换成另一个样子，用～～代替()，代码变成这样(…)~~这样会不会清晰一些。

最后，第一组括号里要是一个函数或者引用函数的变量。所以代码还是会变成这样：
(functionName)()
虽然如果没有第一组括号，这个代码仍然有效。

现在，如果我们用一个匿名函数来代替第一组括号里的内容，代码是这呀：
(function(){})();
如果函数体的内部加上一些逻辑语句，代码变成这样：

```
(function(){
    statement-1;
    statement-2;
    ...
    stateent-3;
})
```

注意，在我们处理一个函数的时候，我们会的得到一个闭包，

我们可以在这个函数中访问外部变量。

总的来说，这个简单的代码结构，叫做即时函数，

稍后我们会看到它的价值。

让我们来看看即时函数(immediate function)和域(scope)的协作。

5.6.1 临时域和私有变量(Temporary scope and private variables)

利用即时函数，我们可以开始构建一个有趣的封闭空间来做些事情。

正是因为即时函数是直接执行的，所以在这个函数域中的变量是受保护的，

我们可以利用这个来创建一个临时域，来维护数据状态。

注：变量在 JavaScript 中的作用域是依赖于定义这个变量的函数。

利用创建一个临时函数，我们可以用它来来装载我们的变量。

来看这段小代码：

```
(function(){
    var numClicks = 0;
    document.addEventListener("click", function(){
        alert( ++numClicks );
```

```
    }, false);
})()
```

就像及时函数的名字一样，这个函数会立即执行，click handler 会立即被绑定。

一个闭包被创建出来，允许 numClicks 变量来存储数据。

这是即时函数非常通用的一个方式，就像一个简单的功能包。

如论怎样，我们要记得他们依旧是函数，他们可以被用的更加有趣，例如：

```
document.addEventListener("click",(function(){
    var numClicks = 0;
    return function(){
        alert( ++numClicks);
    };
})(),false)
```

这个例子相比其之前那个例子，会让人有点迷惑。

在这个例子中，我们同样创建了一个即时函数，

但是这一次我们加了一个返回值：一个类似事件处理者的函数(event handler)。

正象其他表达式一样，这个返回值传递给了 addEventListener 函数。

这个内部函数理所当然的可以访问 numClicks 变量，利用闭包的特性。

这个技巧是用一个很特别的视角来观察作用域。

在大多数的变成语言中，一个作用域是依赖它所在的代码快的。

但是在 JavaScript 中变量的作用域是依赖它所在的闭包

(In JavaScript, variables are scoped based upon the closure that they are in.)

此外，利用这个简单的结构，我们可以将作用域延伸到当前代码块和上一级代码块。

通过调用一个函数，让一些代码的作用域可以和传入这个函数的参数相连接。

这种能力无疑是很有用的，并且彰显了 JavaScript 的灵活性。

List 5.16: Using an immediate function as a variables shortcut

```
(function(v){
   Object.extend(v,
   {
      href:   v._getAttr,
      src:    v._getAttr,
      type:   v._getAttr,
      action: v._getAttrNode,
      disabled:   v._flag,
      checked:    v._flag,
      readonly:   v._flag,
      multiple:   v._flag,
      onload:     v._getEv,
      onunload:   v._getEv,
      onclick:    v._getEv
   });
})(Element._attributeTranslations.read.values);
```

在这个例子中，Prorotype 继承了一个对象的属性和方法。

在这个代码中，一个临时的变量被创建出来，目的是操作 Element._attributeTranslations.read.values,
这个变量通过第一个参数，传入了这个即时执行的匿名函数。

这就意味着，第一个参数现在已经连接了当前的数据结构和作用域。

这个技巧用在循环(looping)上更有用，我们会在下面讨论。

5.6.2 循环(Loops)

即时函数的另一个有用的地方是，它利用循环和闭包可以解决一些丑陋的问题。

例 5.17 就是一个拥有普遍问题的代码

Listing 5.17: A problematic piece of code in which the iterator is not maintained in the closure.

```
<div>DIV 0</div>
<div>DIV 1</div>
var div = document.getElementsByTagName("div");
for (var i = 0; i < div.length; i++){
    div[i].addEventListener("click", function(){
        alert("div #" + i + " was clicked.");
    }, false);
}
```

我们的目的想看到点击每个

的时候，会显示它的原始的值。

但是当我们点击“DIV 0”的时候，我们竟然看到它显示的是 “div #2 was clicked.“
哪里出错了呢？为什么 DIV 0 显示的是 2

在上个例子中，我们遇到了一个适用闭包和循环时候常见的问题，

也就是说，在函数被绑定后，闭包抓住的变量(在这个例子对应的是 i)已经被更改

这就意味着，每个被绑定的处理函数(function handler)都会显示 i 的最后的值，在这个例子中是 2。

这就证明了我们已经在 5.2.2 章节讨论过的一个问题：

闭包记住的是对变量的引用，而不是闭包创建时刻变量的值。

这一点区别，让很多人都走了弯路。

不过不要害怕，我们可以用另一个闭包和即时函数来修正当前这个闭包的毛病，俗话说的好：以毒攻毒(fighting fire with fire, so to speak)。

请看例 5.18

Listing 5.18: Using an immediate funtion to handle the iterator properly

```
<div>DIV 0</div>
<div>DIV 1</div>
var div = document.getElementsByTagName("div");
for (var i = 0; i < div.length; i++){
    (function(i){
       div[i].addEventListener("click", function(){
       alert("div #" + i + " was clicked.");
       }, false);
    })(i)
}
```

通过在 for 循环内加入即时函数，我们将正确的值传递给了及时函数的，进而让工作者(handler)得到了正确的值。

这个例子很清楚的展示了，如何利用即时函数和闭包来控制作用域里的变量和值。

5.6.3 类库包装(Library wrappping)

另一个闭包和即时函数很重要的用途，就是用于 JavaScript 类库的开发。当我们开发一个类库时，很重要的一点是，我们不希望变量去污染全局命名空间，尤其是一些临时变量

为了解决这个问题，闭包和即时函数十分有用，它可以帮助我们让类库尽可能的私密，并且可以有选择的让将一些变量链接到全局命名空间。jQuery 就专注于这个原则，完全的封装它的功能，只是有选择的将一些变量关联到全局空间。

例如 jQuery 自身这个变量，如下面：

```
(function(){
    var jQuery = window.jQuery = functon(){
        // Initialize
    }
    // ...
})()
```

注意：这里有两次赋值，

首先 jQuery 构造器(作为一个匿名函数)赋值给了 window.jQuery，这样就将其暴露到了全局空间。

尽管如此，并不能保证我们内部域中没有别的变量叫做 jQuery，所以我们将它赋予了一个本地变量 jQuery。

在我们自己的创建的世界中，jQuery 这个本地变量的含义是确定的。

由于类库将变量和函数封装的十分优雅，也就让最终使用类库的用户拥有了很好的灵活性。

当然，我还可以用另外一种形势的代码，同样实现这个效果，如下所示：

```
var jQuery = (function(){
    function jQuery(){
        // Initialize
    }
    //...
    return jQuery;
})();
```

这段代码和之前的那个例子实现了同样的效果，只是代码组织形势不同而已。

我们在匿名函数中定义了一个 jQuery 函数，它可以自由的被用在这个域中，

然后我们将它 return，这样可以将它赋值给同样叫做 jQuery 的全局变量。

通常情况下，是否使用这个技巧，取决于你是否想暴露唯一的一个变量到外部。

这个技巧看起来会清晰一些。

总值，这么多种不同的形势，到底选择哪个，完全取决于开发者的意愿。

5.7 总结(Summary)

在这个章节中，我们讨论了闭包，在函数化编程(functional programming)中，闭包是十分关键的一个概念。

JavaScript 这们语言设计的是按照函数化编程(functional programming)的思想设计的。
理所当然，在 JavaScript 中我们可以用闭包。

在一开始的基础部分，我们了解了闭包如何实现，并且看到了在一个应用中如何使用闭包。
我们讨论了很多闭包的特别的用处，包括用闭包实现私有变量，实现回调和 timers。

接下来我们展示了很多闭包高级的用法，闭包可以帮助重构 JavaScript 这个语言。
包括强制控制函数上下文，离散化函数，重写函数的行为。

然后我们又深入的探讨了即时函数，正如我们所学，即时函数重新定义了如何应用这们语言。

最终，对闭包的深入理解，会带来很多好处，尤其在我们开发 JavaScript 复杂应用的时候。
并且利用闭包可以解决很多不可避免的公共问题。

在本章的示例代码中，我们介绍了一点 prototypes 的概念，接下来的一章让我们来认真的研究一下 prototypes。

*(转载本文章请注明作者和出处 **Yann (yannhe.com)**，请勿用于任何商业用途)*

第六章.原型与面向对象(6.Object-orientation with prototypes)[完成]

翻译 Secrets of the JavaScript Ninja (JavaScript 忍者禁术)

第六章.原型与面向对象(6.Object-orientation with prototypes)

本章重点：

1.利用函数实现构造器

2.解释 prototyes

3.利用 prototypes 实现对象的扩展

4.avoiding common gotchas

5.利用 inheritace 构建 classes

看到 prototypes 你可能会觉得他与 object 是紧密联系的，但是，再次提醒各位，我们的重点还是 function，Prototypes 是一种很方便的定义 classes 的途径，但是它的本质是属于 function 的特性

总的来说，如果你很明显的在使用 prototypes 就是在使用一种 classical-style 形式的面向对象编程和继承的技术。

让我们开始看看如何使用 prototypes

6.1 实例化和原型(Instantiation and prototypes)

所有 functions 都拥有一个属性：prototype，

初始状态中 prototype 指向一个空的 object。

这个属性只是在这个函数被当作构造器来调用的时候，

才有作用，其余的情况下，没啥用。

在第三章中，我们利用关键字 new 来调用一个函数，这个函数就成为了构造器，

它会产出一个新的对象实例作为他的函数上下文(function context)

鉴于对象实例化这个部分很重要，下面我们详细的讨论一下，以便我们能真正理解这个知识点。

6.1.1 对象实例化(Object instantiation)

通常最简单的一种创建对象的方式如下：

```
var o = {};
```

但是这种形式有些缺陷，如果是一个面向对象背景的视角来看，这样做是不便利，并且缺少封装。

JavaScript 提供了一种途径，虽然和大多数的语言都不用一一样，例如面向对象阵营中的 Java，C++。

JavaScript 利用 new 这个关键字，通过构造器函数来实例化一个对象，

Prototypes 作为对象的蓝图(PROTOTYPES AS OBJECT BLUEPRINTS)

让我们来看看利用函数，用和不用 new 关键字会造成什么结果，注意观察 prototype 属性是否添加到了新的对象实例中

例 6.1

```
Listing 6.1: Creating an new instance with prototyped method

function Ninja(){}

Ninja.prototype.swingSord = function(){
    return true;
}
var ninja1 = Ninja();
assert(ninja1 === undefined , "No instance of Ninja created.");

var ninja2 = new Ninja();
assert(ninja2 && ninja2.swingSword(), "Instance exists and method is
callable.");
```

这个例子说明，函数的 prototype 作为 new 出来的对象的蓝图，但是前题是这个函数是被当作构造函数来调用才行。

实例的属性(INSTANCE PROPERTIES)

当使用构造函数通过 new 关键字来构建一个对象实例的时候，这就意味着我们可以在构造函数中通过 prototype 来初始化一些值。

让我们来看例 6.2

```
Listing  6.2:Observing the precedence of initialization activities
function Ninja(){
    thisswung = false;
    this.swingSword = function(){
        return ! this.swung;
    }
```

```
}

Ninja.prototype.swingSword = function(){
    return this.swung;
}

var ninja = new Ninja();
assert(ninja.swingSword(), "Called the instance method, not the prototype
method.");
```

1.对象的属性从 prototype 来

2.对喜爱那个的属性从构造函数里面来

当这两者冲突的时候，构造函数优先于 prototype。

因为构造函数中的 this 就是对象实例本身，我们可以在构造函数中做任何初始化的事情。

关联问题(RECONCILING REFERENCES)

这里有个很重要的问题要理解，就是 JavaScript 如何处理对象实例的属性来关联 prototype 的时序问题。

在上个例子中，时序问题我们可以能会如此解读：首先一个新的实例对象被创建出来，并传递给了构造函数，然后构造函数的 prototype 属性会被复制到对象实例中。

然后事实是构造函数内部的逻辑覆盖了 prototype 的值。

我们会知道，这并不是像我们想的那样，简单的将 protype 拷贝

例如，如果 prototype 的值仅仅是简单的拷贝，在这个对象之后对 prototype 的任何改变都不应该再来影响这个对象才对。但事实是这样

吗？

让我们来看例 6.3

```
Listing 6.3: Observing the behavior of changes to the prototype
function Ninja(){
    this.swung = true;
}
var ninja = new Ninja();

Ninja.prototype.swingSword = function(){
    return this.swung;
}
```

assert(ninja.swingSword(), “Method exists, even out of order.”);

很明显，prototype 不是简单的拷贝

简单的时序如下：

1.当你查看一个对象的属性，首先，这个对象检查真身的属性中是否存在，如果存在则使用这个值，如果不存在…

2.他会检查与他有关的那个 prototype，如果 prototype 存在这个属性，则使用此值，如果不存在…

3.这个值就是 undefined

我们后面会看到，真正的时序会比现在这个复杂一些，不过我们先这么理解。

这一切是如何工作的？

JavaScript 中每个对象丢有一个名字叫做 constructor 的属性，这个属性关联的是创建此对象的构造函数。由于 prototype 是构造函数的一个属性，所以每个对象都能有方法找到与他相关的 prototype。

在本例中，如果我们查看执行此例子时候的 console(in Chrome),我们会看到对象的结构如下：

```
>ninja.constructor
 function Nunja(){
    this.swung = true;
 }
>ninja.constructor.prototype.swingSword
 function(){
    return this.swung
 }
```

这里的更新了 prototype 就同时更新 object 中的属性值，我们叫他“同步更新”，

这个特性会给我带来很多的用处，这个特点在其他语言中是很少见到的。利用这个特性，我们可以构建一个函数化的框架，使用这个框架的用户可以使用到未来的函数实现，哪怕对象已经被创建出来了。

```
Listing 6.4: Fruther observing the behavior of changes to the prtotype
function Ninja(){
    this.swung = true;
    this.swingSword = function(){
        return !this.swung;
    }
}

var ninja = new Ninja();

Ninja.prototype.swingSword = function(){
    return this.swung;
}

asswert(ninja.swingSwrod(), "Called the instance method, not the
prototype method.");
```

本例中我们看到，在 ninja 被创建之后，通过更改 prototype 的值可以影响到 ninja 中的对应属性。

6.1.2 通过构造器归类对象(Object typing via constuctors)

例 6.5

```
Listing 6.5: Examining the type of an instance and its constructor
function Ninja(){}
var ninja = new Ninja();

assert (typeof ninja == "object", "The type of the instance is object.");
assert (ninja instanceof Ninja, "instance of identifies the
constructor.");
assert (ninja.constructor == Ninja, "The ninja object was created by the
Ninja function.")
```

在本例中，我们定义了一个构造函数，然后利用它创建了一个对象实例.然后我们利用 type of 来检验这个实例的类型。这里并不很明显，因为一个实例指定是对象，结果指定是“object”.相比起 typeof，更有意思的是 instanceof，通过 instanceof 我们可以清晰的看出一个对象实例的构造函数。

另外，我们知道对象都有一个 constructor 属性，这个属性关联的就是当初创建这个对象实例的构造函数。

注意，constructor 属性除了可以查询到原始的构造函数，我们还可以利用这个属性来创建一个新的 Ninja 对象实例.

例：6.6

```
Listing 6.6: Instantiating a new object using a reference to a construcor
```

```
function Ninja(){}
var ninja = new Ninja();
var ninja2 = new ninja.constructor();

assert(ninja2 instanceof Ninja, "It's a Ninja!");
```

注意：一个对象实例的 constructor 属性是可以被更改的，

到这里我们只是接触了一下 prototypes 的皮毛，下面我们来看看 prototypes 真正的有趣的地方。

6.1.3 继承和原型链(Inheritance and the prototype chain)

我们之前看到 instanceof 这个关键字，

如果想要应用它，我们需要理解 JavaScript 的继承机制，还有 prototype chain 扮演着什么样的角色。

让我们看一下例 6.7，在这个例子中我们会试图将一个继承加入到对象实例中。

```
Listing 6.7: Trying to achieve inheritance with prototypes
function Person(){}
Person.prototype.dance = function(){};
function Ninja(){};

Ninja.prototype = { dance: Person.prototype.dance };

var ninja = new Ninja();
assert( ninja instanceof Ninja, "ninja receives functionality from the
Ninja prototype");
assert( ninja instanceof Person, "... and the Person prototype");
assert( ninja instanceof Object, "... and Object prototype");
```

本例中，目的是让 Ninja 继承 Person 的 dance 属性，但是运行的结果发现，ninja instanceof Person 这句是 false。

说明 ninja 并不是一个 Person

虽然 Ninja 复用了 Person 的 dance 属性，但是他也不意味着就是一个 Person
如果我们想让 Ninja 复用 Person 的所有属性，那么就需要 copy 多次，这绝不是继承。

注意：即便什么也不做，每个对象都会继承 Object，你可以通过这句话来检验一下：
console.log({}.constructor)

我们的真正目的是 prototype chain。

例如一个 Ninja 可以是一个 person,

一个 person 可以是 Mammal，一个 Mammal 可以是一个 Animal，最终都成为一个 Object。

创建 prototyp chain 的一个方式是通过其他对象的 prototype，例如：
SubClass.prototype = new SuperClass();
这样通过 SubClass 创建出来的对像，会拥有 SuperClass 的所有属性。

另外它的 prototype 还会指向 SuperClass 的 prototype

我们来更改一下上面的例子，看例 6.8：

```
Listing 6.8 Achieving inheritance with prototypes
function Person(){}
Person.prototype.dance = function(){};

function Ninja(){}

Ninja.prototype = new Person();
```

```
var ninja = new Ninja();
assert(ninja instanceof Ninja, "ninja receives functionality from the
Ninja prototype");
assert(ninja instanceof Person, "... and the Person prototype");
assert(ninja instanceof Object, "... and the Object prototype");
assert(typeof ninja.dance == "function", "... and can dance!");
```

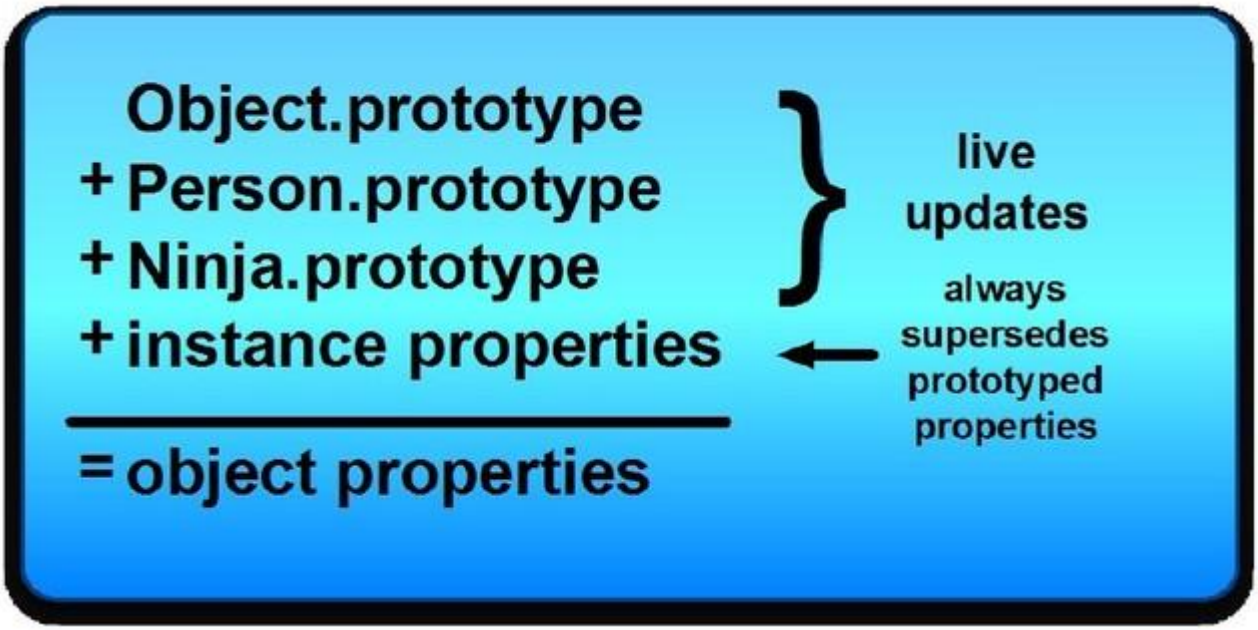
注意，不要用 Ninja.prototype = Person.prototype;

如果这样做，那么更改 Ninja prototype 的时候同样会更改 Person prototype(因为他们是同一个对象)

这里要注意，prototype 继承模式中，所有继承链中的函数都是实时更新的(live update)。

我们用一个图来解释 prototype chain

图 6.6



在图 6.6 中，我们注意到，对象的所有属性都是继承自 Object 的 prototype。

所有自然对象(native objects)的 constructors 属性(例如 Object,Array,String,Number,RegExp, and Function)都拥有 prototype 属性，并且它是可以被更改和继承的。

这样一来，每个上面提到的对象的 constructors 都是 functions 本身.

在语言层面，这是一个很有用的特点，利用它，我们可以扩展这门语言本身。

例如，JavaScript1.6 版本会加入一些有用的方法，例如 Arrays 的 forEach().

如果我们想在 1.6 版本之前就是用 forEach()，并且当 JavaScript 升级到 1.6 之后能同步使用新的特点，那么请看下面这个例子,我们针对旧的浏览器，实现一个 forEach()

```
Listing 6.9: Implementing the JavaScrript 1.6 array forEach method in a
future-compatible manner

if (!Array.prototype.forEach){
    Array.prototype.forEach = function(fn, callback){
        for (var i = 0; i < this.length; i++){
            fn.call(callback || null, this[i], i, this);
        }
    }
}

["a", "b", "c"].forEach(function(value, index, array){
    assert(value, "Is in position" + index + "out of " + (array.length
- 1))
})
```

我们已经看到，我们可以同那个 prototypes 来增强 native JavaScript objects;

现在，让我们来关注一下 Dom

6.1.4 HTML DOM prototypes

浏览器的一个有趣的地方是所有的 DOM 元素都是继承于 HTMLElement 构造器.

通过操作 HTMLElement 的 prototype，浏览器就会提供我们扩展任何 HTML 节点的能力。

让我们来看例 6.10

```
Listing 6.10: Adding a new method to all HTML elemeents via the HTMLElement prototype
<div id="a">I'm going to be removed.</div>
<div id="b">Me too!</div>
<script>
    HTMLElement.prototype.remove = function(){
        if (this.parentNode)
            this.parentNode.removeChild(this);
    };

    var a = document.getEleementById("a");
    a.parentNonde.removeChild(a);

    document.getElementById("b").remove();

    assert(!document.getElementById("a"), "a is gone.");
    assert(!document.getElementById("b"), "b is gone too.");
```

我们添加了一个新的 remove()方法，通过更改 HTMLElement 构造器的 prototype。

6.2 The Gotchas!

6.2.1 对象的扩展(Extending Object)

我们可能会犯的及其严重的错误就是去扩展 native Object.prototype.

难点在于一旦我们扩展了这个 prototype，所有的对象将会受到影响。

让我们来看一个例子

这里我们想在 Object 上增加一个 keys()方法，它会返回一个包含所有属性名的列表

看例子 6.11

```
Listing 6.11: Unexpected behavior of adding extra properties to the Object
prototype
Object.prototype.keys = function(){
    var keys = [];
    for (var p in this)
        keys.push(p);
    return keys;
}

var obj = {a:1, b:2, c:3};
assert(obj.keys().length == 3, "There are three properties in this
object.");
```

结果是报错的，我们需要 hasOwnProperty()，它能区别哪些是真正属于对象实例的属性，并不是引用的 prototype.让我们来看例 6.12

```
Listing 6.12: Using the hasOwnProperty() method to tame Object prototype
extensions
Object.prototype.keys = function(){
    var keys = [];
    for (var i in this)
        if (this.hasOwnProperty(i))
            keys.push(i);
    return keys;
};
```

```
var obj = {a:1, b:2, c:3};

assert(obj.keys().length == 3, "There are three properties in this
object.");
```

6.2.2 Number 的扩展(Exending Number)

```
Listing 6.13: Adding a method to the Number prototype.
Number.prototype.add = function(num){
    return this + num;
}

var n = 5;
assert(n.add(3) == 8, "It works when the number is in a variable");
assert((5).add(3) == 8, "Also worrks if a number is wrapped in
parentheses.");
assert(5.add(3) == 8, "What about a simple literal?");
```

本例运行后，浏览器会报错。

It turns out that the syntax parser can’t handle the literal case.

6.2.3 Subclassing native objects

```
Listing 6.14: Subclassing the Array object
function MyArray(){}
MyArray.prototype = new Array();
var mine = new MyArray();
mine.push(1,2,3);
assert(mine.length == 3, "All the items are on our sub-classed array.");
assert(mine instanceof Array, "Verify that we implement Array
functionality.");
```

这种模式就是制造子类，但是在 IE 中，浏览器不允许 Array 被子类化，

所以这样做是危险的。

让我用另一种方式来实现，例 6.15

```
Listing 6.15: Simulating Array functionality but without the true
sub-classing.
```

```
function MyArray(){}
MyArray.prototype.length = 0;

(function(){
    var methods = ['push', 'pop', 'shift', 'unshift', 'slice', 'splice',
'join'];
    for (var i = 0; i< methods.length; i++)(function(name){
        MyArray.prototype[name] = function(){
            return Array.prototype[name].apply(this, arguments);
        }
    })(methods[i]);

})();

var mine = new MyArray();
mine.push(1,2,3);
assert(mine.length == 3, "All the items are on our sub-classed array.");
assert(!(mine instanceof Array), "We aren't subclassing Array, though.");
```

6.2.4 实例化的问题(Instantiation issues)

我们已经看到函数可以被当作普通函数调用，也可以作为构造器被调用，也许你还是很模糊，不知道哪个代码是哪个，让我们用些例子来详细解释一下。

```
Listing 6.16: The result of leavving off thee new operator from a function
call.
function User(first, last){
    this.name = first + " " + last;
}
var user = User("Ichigo", "Kurosaki");
assert(user , "User instantiated");
assert(user.name == "Ichigo Kurosaki", "user name correctly assigned");
```

运行会报错

```
Listing 6.17: An example of accidentally introducing a variablee into the
global namespace
function User(first, last){
```

```
    this.name = first + " " + last;
}
var name = "Rukia";
var user = User("Ichigo", "Kurosaki");
assert(name == "Rukia", "name was set to Rukia.");
Listing 6.18: Determining if we're called as constructor
function Test(){
    return this instanceof arguments.callee;
}
assert(!Test(), "We didn't instantiate, so it returns false.");
assert(new Test(), "We did instantiate, returning true.");
```

复习一下以前学过的概念：

1.我们可以获得当期被调用的函数的的引用，通过 arguments.calle(我们在第四章学过)

2.普通函数的函数上下文是全局域的

3.instanceof 关键字可以判断一个对象的构造器

在本例中我们看到这样的表达：
this instanceof arguments.callee
如果为 true 表示是作为构造函数被调用的，如果为 false 则表示是作为普通函数被调用的。

这就是所，在函数中，我们可以判断出，它是否作为构造函数被调用。如果我们不是忍者，如果这个函数没有作为构造函数被调用，我们会抛出一个异常来提醒用户，下次正确使用。但是我们可以做的更好，让我们看看如何修复这个问题

```
Listing 6.19: Fixing things on the caller's behalf
function User(first, last){
    if (! (this instanceof arguments.callee)){
        return new User(first, last);
    }
```

```
    this.name = first + " " + last;
}
var name = "Rukia";
var user = User("Ichiggo", "Kurosaki");

assert(name == "Rukia", "Name was set to Rukia.");
assert(user instanceof User, "user instantiated");
assert(user.name == "Ichigo Kurosaki", "User name correctly assigned");
```

6.3 编写像类一样的代码(Writing class-like code)

JavaScript 可以允许我们通过 prototype 来实现集成，针对于大多数面向对象背景的开发者来说，JavaScript 的继承系统是比较熟悉的。

一般来说，这些开发者会渴求几点：

1.一套可以比较轻量的构建新的构造函数和属性的系统

2.一个简单的方式可以执行 prototype 的继承

3.一个路径可以利用函数的 prototype 来覆盖 methods

有 2 个比较突出的 JavaScript 类库实现了类的继承：base2 和 Prototype。我们会提取其中的精华部分来展示。

```
Listing 6.20: An example of somewhat classical-style inheritance syntax
var Person = Object.subClass({
    init: function(isDancing){
        this.dancing = isDancing;
    },
    dance: function(){
        return this.dancing;
    }
});

var Ninja = Person.subClass({
    init: function(){
```

```
        this._super(false);
    },
    dance: function(){
        // Ninja-specific stuff here
        return this._super();
    },
    swingSword: function(){
        return true;
    }
});

var person = new Person(true);
assert(person.dance(), "The person is dancing.");

var ninja = new Ninja();
assert(ninja.swingSword(), "The sword is swinging.");
assert(!ninja.ance(), "The ninja is not dancing.");
assert(person instanceof Person, "Person is a Person.");
assert(ninja instanceof Ninja && ninja instanceof Person, "Ninja is a Ninja
and a Person.");
```

这里我们通过 subclass()方法来构建了一个子类出来，现在我们需要实现这个方法。

```
Listing 6.21: A sub-classing method
(function(){
    //#1 Determines if functions can be serialized
    var initializing = false,
    fnTest = /xyz/.test(function(){xyz;}) ? /\b_super\b/ : /.*/;

    //#2 Creates a subclass
    Object.subClass = function(prop){
        var _super = this.prototype;

        //#3 Instantiates the superclass
        initializing = true;
        var proto = new this();
        initializing = false;

        //#4 Copies properties into prototype
        for (var name in prop){
            proto[name] = typeof prop[name] == "function" &&
```

```
            typpeof _super[name] == "function" &&
fnTest.test(prop[name]) ?

            //#5 Defines overriding function
            (function(name, fn){
                return function(){
                    var tmp = this._super;

                    this._super = _super[name];

                    var ret = fn.apply(this, arguments);
                    this._super = tmp;

                    return ret;
                }
            })(name, prop[name]) :
            prop[name]
        }

        //#6 Creates a dummy class constructor
        function Class(){
            if (!initializing && this.init)
                this.init.apply(this, arguments);
        }

        //#7 Populates class prototype
        Class.prototype = proto;

        //#8 Overrides constructor reference
        Class.constructor = Class;

        //#9 Makes class extendable
        Class.subClass = arguments.callee;

        return Class;

    };
})();
```

这里最重要的两处实现为 initialization 和 super-method protions,

我们下面来一步步的分析这些代码。

让我们开始看看可能以前你们见过的东西。

6.3.1 检查函数是否可序列化(Checking for function serializability)

很不幸，我们代码的一开始就是一段秘传的代码，它可能会让人迷惑。

这段代码的目的是检查浏览器是否支持函数的序列化。

函数序列化是获取函数的代码文本资源。

在大多数浏览器中，函数的 toString()方法就可以搞定。

在我们的代码中，我们这样写的：
/xyz/.test(function(){xyz;})
如何函数可以被序列化，那么结果就是 true(关于正则表达式我们后面会讨论)

我们写出这样的代码，为了后面用到：
superPattern= /xyz/.test(function(){ xyz; }) ? /\b_super\b/ : /.*/;
这里的 superPattern 变量，我们后面会用于检验一个函数是否包含"_super".

现在让我们看看 sub-classing 方法的代码。

6.3.2 subClasses 的实例化(Initialization of subclasses)

在这里，我们会声明一个 subclass 的方法，我们这样写的：

```
Object.subClass = function(properties){
    var _super = this.prototype;
```

这里添加到 Object 上的 subClass 方法，接受唯一的一个参数，这个参数就是一个属性组，我们需要遍历这个属性组并将它们加到 subclass 中。

普通的代码一般会写成这样：

```
function Person(){}
function Ninja(){}
NInja.prototype = new Person();
assert((new Ninja()) instanceof Person, "Ninjas are peeople too!");
```

6.3.3 保留父级的方法(Preserving super methods)

大多数语言都支持继承，一个方法被重写后我们可以访问重写后的方法。这是很有用的，有时候我们会重写一个方法，但大部分时候我们只是想要加强一个方法。

在我们的代码中我们创建了一个新的方法叫做_super,它是关联着父类的方法。

例如例 6.20，当我们想调用父类方法的构造函数的时候，我们这样写的：

```
var Person = Object.subclass({
    init: function(isDancing){
        this.dancing = isDancing;
    }
})

var Ninja = Person.subclass({
    init: function(){
        this._super(false);
    }
})
```

利用_super 我们可以省去重新写父类代码的麻烦。

实现这个方法需要多个步骤。

简单来说我们需要 merge 父类和传递进来的属性。

在开始，我们创建了一个实例，我们将此实例作为 prototype，

代码如下：

```
initializing = true;
var proto = new this();
initializing = false;
```

记得之前我们讨论过的如何保护初始化吗？对，是通过 initializing 这个变量标识。

如果我们不考虑父类的函数，我们可以这么写。
for(var name in properties) proto[name] = properties[name];
但是我们要考虑父类的函数，我们会通过_super 来引用父类的函数。

我们首先要查明我们是否需要 wrap 子类函数。我们可以通过下面的表达式：

```
typeof properties[name] == "function" &&
typeof _super[name] == "function" &&
superPattern.test(properties[name])
```

这个表示包括了如下的检查项：

1.子类的这个属性是否是个函数?

2.父类的这个属性是否是个函数?

3.子类函数中是否包含_super()?

只有所有条件都满足的时候我们才开始 wrap 这个函数,具体代码如下：

```
(function(name, fn){
    return function(){
        var tmp = this._super;
        this._super = _super[name];
        var ret = fn.apply(this, arguments);
        this._super = tmp;
        return ret;
    }
})(name, properties[name])
```

6.4 总结(Summary)

在这一章中，我们看到通过 prototype 我们将面向对象带进了 JavaScript。

我首先介绍了 prototype 的概念，他所扮演的角色。我们也介绍了是否用 new 关键字来调用一个函数的区别。

接下来，我们学习了如何辨别一个对象的类型。

我们还学习了面向对象中的继承概念，以及学习了如何运用 prototype 链来影响继承。

我们实现了 supclass 方法来构建一个子类。

在最后我们还一睹了正则表达式，在下一个章节我们会深入的学习它。

(转载本文章请注明作者和出处 Yann (yannhe.com)，请勿用于任何商业用途)