|  |
| --- |
| 原文地址 <http://msdn.microsoft.com/zh-cn/magazine/cc163419.aspx> |
| 原作者 Ray Djajadinata Ray Djajadinata在 Barclays Capital Singapore 开发 AJAX 应用程序，对此他很开心。您可以通过 [ray.djajadinata@gmail.com](mailto:ray.djajadinata@gmail.com) 与 Ray 联系。 |

最近，我面试了一个有五年 Web 应用程序开发经验的软件开发人员。四年半来她一直在从事 JavaScript 相关的工作，她自认为 JavaScript 技能非常好，但在不久之后我就发现实际上她对 JavaScript 知之甚少。话虽这样说，但我确实没有责备她的意思。JavaScript 真的是很有趣。很多人（包括我自己，直到最近！）都认为自己很擅长 JavaScript 语言，因为他们都知道 C/C++/C#，或者有一些以前的编程经验。  
在某种程度上，这种假设并不是完全没有根据的。用 JavaScript 很容易做些简单的事情。入门的门槛很低，该语言很宽松，它不需要您知道很多细节就可以开始用它进行编码。甚至非编程人员也可能用它在几个小时内为主页编写一些有用的脚本。  
的确，直到最近，仅仅凭借 MSDN® DHTML 参考资料和我的 C++/C# 经验，我也总能勉强利用这点 JavaScript 知识完成一些任务。只是当我开始编写真实的 AJAX 应用程序时，我才意识到实际上我的 JavaScript 知识还非常不够。这个新一代的 Web 应用程序的复杂性和交互性需要程序员以完全不同的方法来编写 JavaScript 代码。它们是真正的 JavaScript 应用程序！我们在编写一次性脚本时一直采用的方法已完全不再有效。  
面向对象编程 (OOP) 是一种流行的编程方法，很多 JavaScript 库中都使用这种方法，以便更好地管理和维护基本代码。JavaScript 支持 OOP，但与诸如 C++、C# 或 Visual Basic® 等流行的 Microsoft® .NET Framework 兼容语言相比，它支持 OOP 的方式非常不同，因此主要使用这些语言的开发人员开始可能会觉得在 JavaScript 中使用 OOP 很奇怪而且不直观。我写本文就是为了深入讨论 JavaScript 语言实际上如何支持面向对象编程，以及您如何使用这一支持在 JavaScript 中高效地进行面向对象开发。下面首先讨论对象（还能先讨论其他别的什么呢？）。  
  
JavaScript 对象是词典  
在 C++ 或 C# 中，在谈论对象时，是指类或结构的实例。对象有不同的属性和方法，具体取决于将它们实例化的模板（即类）。而 JavaScript 对象却不是这样。在 JavaScript 中，对象只是一组名称/值对，就是说，将 JavaScript 对象视为包含字符串关键字的词典。我们可以使用熟悉的“.”（点）运算符或“[]”运算符，来获得和设置对象的属性，这是在处理词典时通常采用的方法。以下代码段

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

var userObject = new Object();  
userObject.lastLoginTime = new Date();  
alert(userObject.lastLoginTime);

的功能与下面的代码段完全相同：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

var userObject = {}; // equivalent to new Object()  
userObject[“lastLoginTime”] = new Date();  
alert(userObject[“lastLoginTime”]);

我们还可以直接在 userObject 的定义中定义 lastLoginTime 属性，如下所示：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

var userObject = { “lastLoginTime”: new Date() };  
alert(userObject.lastLoginTime);

注意，它与 C# 3.0 对象初始值非常相似。而且，熟悉 Python 的人会发现在第二和第三个代码段中实例化 userObject 的方法与在 Python 中指定词典的方法完全相同。唯一的差异是 JavaScript 对象/词典只接受字符串关键字，而不是像 Python 词典那样接受可哈希化的对象。  
这些示例还显示 JavaScript 对象比 C++ 或 C# 对象具有更大的可延展性。您不必预先声明属性 lastLoginTime — 如果 userObject 没有该名称的属性，该属性将被直接添加到 userObject。如果记住 JavaScript 对象是词典，您就不会对此感到吃惊了，毕竟，我们一直在向词典添加新关键字（和其各自的值）。  
这样，我们就有了对象属性。对象方法呢？同样，JavaScript 与 C++/C# 不同。若要理解对象方法，首先需要仔细了解一下 JavaScript 函数。  
  
JavaScript 函数是最棒的  
在很多编程语言中，函数和对象通常被视为两样不同的东西。在 JavaScript 中，其差别很模糊 — JavaScript 函数实际上是具有与它关联的可执行代码的对象。请如此看待普通函数：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

function func(x) {  
    alert(x);  
}  
func(“blah”);

这就是通常在 JavaScript 中定义函数的方法。但是，还可以按以下方法定义该函数，您在此创建匿名函数对象，并将它赋给变量 func

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

var func = function(x) {  
    alert(x);  
};  
func(“blah2”);

甚至也可以像下面这样，使用 Function 构造函数：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

var func = new Function(“x”, “alert(x);”);  
func(“blah3”);

此示例表明函数实际上只是支持函数调用操作的对象。最后一个使用 Function 构造函数来定义函数的方法并不常用，但它展示的可能性非常有趣，因为您可能注意到，该函数的主体正是 Function 构造函数的 String 参数。这意味着，您可以在运行时构造任意函数。  
为了进一步演示函数是对象，您可以像对其他任何 JavaScript 对象一样，在函数中设置或添加属性：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

function sayHi(x) {  
    alert(“Hi, “ + x + “!”);  
}  
sayHi.text = “Hello World!”;  
sayHi[“text2”] = “Hello World... again.”;  
  
alert(sayHi[“text”]); // displays “Hello World!”  
alert(sayHi.text2); // displays “Hello World... again.”

作为对象，函数还可以赋给变量、作为参数传递给其他函数、作为其他函数的值返回，并可以作为对象的属性或数组的元素进行存储等等。**图 1** 提供了这样一个示例。  
http://i.msdn.microsoft.com/Platform/Controls/CollapsibleArea/resources/plus.gifFigure 1 JavaScript 中的函数是最棒的

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

var myDog = {  
    “name” : “Spot”,  
    “bark” : function() { alert(“Woof!”); },  
    “displayFullName” : function() {  
        alert(this.name + “ The Alpha Dog”);  
    },  
    “chaseMrPostman” : function() {   
        // implementation beyond the scope of this article   
    }      
};  
myDog.displayFullName();   
myDog.bark(); // Woof!

记住这一点后，向对象添加方法将是很容易的事情：只需选择名称，然后将函数赋给该名称。因此，我通过将匿名函数分别赋给相应的方法名称，在对象中定义了三个方法：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

alert(“NaN is NaN: “ + isNaN(NaN));  
  
function x() {  
    this.isNaN = function() {   
        return “not anymore!”;  
    };  
}  
// alert!!! trampling the Global object!!!  
x();  
  
alert(“NaN is NaN: “ + isNaN(NaN));

C++/C# 开发人员应当很熟悉 displayFullName 函数中使用的“this”关键字 — 它引用一个对象，通过对象调用方法（使用 Visual Basic 的开发人员也应当很熟悉它，它在 Visual Basic 中叫做“Me”）。因此在上面的示例中，displayFullName 中的“this”的值是 myDog 对象。但是，“this”的值不是静态的。通过不同对象调用“this”时，它的值也会更改以便指向相应的对象，如**图 2** 所示。  
http://i.msdn.microsoft.com/Platform/Controls/CollapsibleArea/resources/plus.gifFigure 2 “this”随对象更改而更改

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

alert(“NaN is NaN: “ + isNaN(NaN));  
  
function x() {  
    this.isNaN = function() {   
        return “not anymore!”;  
    };  
}  
// alert!!! trampling the Global object!!!  
x();  
  
alert(“NaN is NaN: “ + isNaN(NaN));

**图 2** 中的最后一行表示的是将函数作为对象的方法进行调用的另一种方式。请记住，JavaScript 中的函数是对象。每个函数对象都有一个名为 call 的方法，它将函数作为第一个参数的方法进行调用。就是说，作为函数第一个参数传递给 call 的任何对象都将在函数调用中成为“this”的值。这一技术对于调用基类构造函数来说非常有用，稍后将对此进行介绍。  
有一点需要记住，绝不要调用包含“this”（却没有所属对象）的函数。否则，将违反全局命名空间，因为在该调用中，“this”将引用全局对象，而这必然会给您的应用程序带来灾难。例如，下面的脚本将更改 JavaScript 的全局函数 isNaN 的行为。一定不要这样做！

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

Dog spot = new Dog();

到这里，我们已经介绍了如何创建对象，包括它的属性和方法。但如果注意上面的所有代码段，您会发现属性和方法是在对象定义本身中进行硬编码的。但如果需要更好地控制对象的创建，该怎么做呢？例如，您可能需要根据某些参数来计算对象的属性值。或者，可能需要将对象的属性初始化为仅在运行时才能获得的值。也可能需要创建对象的多个实例（此要求非常常见）。  
在 C# 中，我们使用类来实例化对象实例。但 JavaScript 与此不同，因为它没有类。您将在下一节中看到，您可以充分利用这一情况：函数在与“new”运算符一起使用时，函数将充当构造函数。  
  
构造函数而不是类  
前面提到过，有关 JavaScript OOP 的最奇怪的事情是，JavaScript 不像 C# 或 C++ 那样，它没有类。在 C# 中，在执行类似下面的操作时：  
  
Dog spot = new Dog();  
将返回一个对象，该对象是 Dog 类的实例。但在 JavaScript 中，本来就没有类。与访问类最近似的方法是定义构造函数，如下所示：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

function DogConstructor(name) {  
    this.name = name;  
    this.respondTo = function(name) {  
        if(this.name == name) {  
            alert(“Woof”);          
        }  
    };  
}  
  
var spot = new DogConstructor(“Spot”);  
spot.respondTo(“Rover”); // nope  
spot.respondTo(“Spot”); // yeah!

那么，结果会怎样呢？暂时忽略 DogConstructor 函数定义，看一看这一行：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

var spot = new DogConstructor(“Spot”);

“new”运算符执行的操作很简单。首先，它创建一个新的空对象。然后执行紧随其后的函数调用，将新的空对象设置为该函数中“this”的值。换句话说，可以认为上面这行包含“new”运算符的代码与下面两行代码的功能相当：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

// create an empty object  
var spot = {};   
// call the function as a method of the empty object  
DogConstructor.call(spot, “Spot”);

正如在 DogConstructor 主体中看到的那样，调用此函数将初始化对象，在调用期间关键字“this”将引用此对象。这样，就可以为对象创建模板！只要需要创建类似的对象，就可以与构造函数一起调用“new”，返回的结果将是一个完全初始化的对象。这与类非常相似，不是吗？实际上，在 JavaScript 中构造函数的名称通常就是所模拟的类的名称，因此在上面的示例中，可以直接命名构造函数 Dog：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

// Think of this as class Dog  
function Dog(name) {  
    // instance variable   
    this.name = name;  
    // instance method? Hmmm...  
    this.respondTo = function(name) {  
        if(this.name == name) {  
            alert(“Woof”);          
        }  
    };  
}  
  
var spot = new Dog(“Spot”);

在上面的 Dog 定义中，我定义了名为 name 的实例变量。使用 Dog 作为其构造函数所创建的每个对象都有它自己的实例变量名称副本（前面提到过，它就是对象词典的条目）。这就是希望的结果。毕竟，每个对象都需要它自己的实例变量副本来表示其状态。但如果看看下一行，就会发现每个 Dog 实例也都有它自己的 respondTo 方法副本，这是个浪费；您只需要一个可供各个 Dog 实例共享的 respondTo 实例！通过在 Dog 以外定义 respondTo，可以避免此问题，如下所示：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

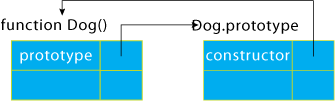
function respondTo() {  
    // respondTo definition  
}  
  
function Dog(name) {  
    this.name = name;  
    // attached this function as a method of the object  
    this.respondTo = respondTo;  
}

这样，所有 Dog 实例（即用构造函数 Dog 创建的所有实例）都可以共享 respondTo 方法的一个实例。但随着方法数的增加，维护工作将越来越难。最后，基本代码中将有很多全局函数，而且随着“类”的增加，事情只会变得更加糟糕（如果它们的方法具有相似的名称，则尤甚）。但使用原型对象可以更好地解决这个问题，这是下一节的主题。  
  
原型  
在使用 JavaScript 的面向对象编程中，原型对象是个核心概念。在 JavaScript 中对象是作为现有示例（即原型）对象的副本而创建的，该名称就来自于这一概念。此原型对象的任何属性和方法都将显示为从原型的构造函数创建的对象的属性和方法。可以说，这些对象从其原型继承了属性和方法。当您创建如下所示的新 Dog 对象时：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

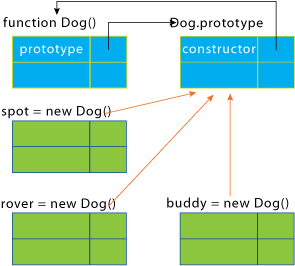
var buddy = new Dog(“Buddy“);

buddy 所引用的对象将从它的原型继承属性和方法，尽管仅从这一行可能无法明确判断原型来自哪里。对象 buddy 的原型来自构造函数（在这里是函数 Dog）的属性。   
在 JavaScript 中，每个函数都有名为“prototype”的属性，用于引用原型对象。此原型对象又有名为“constructor”的属性，它反过来引用函数本身。这是一种循环引用，**图 3** 更好地说明了这种循环关系。  
图 3 **每个函数的原型都有一个 Constructor 属性**   
  
现在，通过“new”运算符用函数（上面示例中为 Dog）创建对象时，所获得的对象将继承 Dog.prototype 的属性。在**图 3** 中，可以看到 Dog.prototype 对象有一个回指 Dog 函数的构造函数属性。这样，每个 Dog 对象（从 Dog.prototype 继承而来）都有一个回指 Dog 函数的构造函数属性。**图 4** 中的代码证实了这一点。**图 5** 显示了构造函数、原型对象以及用它们创建的对象之间的这一关系。  
http://i.msdn.microsoft.com/Platform/Controls/CollapsibleArea/resources/plus.gifFigure 4 对象具有其原型的属性

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

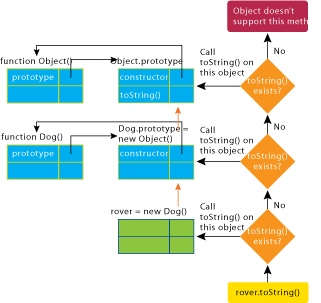
var spot = new Dog(“Spot”);  
  
// Dog.prototype is the prototype of spot  
alert(Dog.prototype.isPrototypeOf(spot));  
  
// spot inherits the constructor property  
// from Dog.prototype  
alert(spot.constructor == Dog.prototype.constructor);  
alert(spot.constructor == Dog);  
  
// But constructor property doesn’t belong  
// to spot. The line below displays “false”  
alert(spot.hasOwnProperty(“constructor”));  
  
// The constructor property belongs to Dog.prototype  
// The line below displays “true”  
alert(Dog.prototype.hasOwnProperty(“constructor”));

图 5 **实例继承其原型**   
  
某些读者可能已经注意到**图 4** 中对 hasOwnProperty 和 isPrototypeOf 方法的调用。这些方法是从哪里来的呢？它们不是来自 Dog.prototype。实际上，在 Dog.prototype 和 Dog 实例中还可以调用其他方法，比如 toString、toLocaleString 和 valueOf，但它们都不来自 Dog.prototype。您会发现，就像 .NET Framework 中的 System.Object 充当所有类的最终基类一样，JavaScript 中的 Object.prototype 是所有原型的最终基础原型。（Object.prototype 的原型是 null。）  
在此示例中，请记住 Dog.prototype 是对象。它是通过调用 Object 构造函数创建的（尽管它不可见）：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

Dog.prototype = new Object();

因此，正如 Dog 实例继承 Dog.prototype 一样，Dog.prototype 继承 Object.prototype。这使得所有 Dog 实例也继承了 Object.prototype 的方法和属性。  
每个 JavaScript 对象都继承一个原型链，而所有原型都终止于 Object.prototype。注意，迄今为止您看到的这种继承是活动对象之间的继承。它不同于继承的常见概念，后者是指在声明类时类之间的发生的继承。因此，JavaScript 继承动态性更强。它使用简单算法实现这一点，如下所示：当您尝试访问对象的属性/方法时，JavaScript 将检查该属性/方法是否是在该对象中定义的。如果不是，则检查对象的原型。如果还不是，则检查该对象的原型的原型，如此继续，一直检查到 Object.prototype。**图 6** 说明了此解析过程。  
图 6 **在原型链中解析 toString() 方法** (单击该图像获得较大视图)  
  
JavaScript 动态地解析属性访问和方法调用的方式产生了一些特殊效果：

* 继承原型对象的对象上可以立即呈现对原型所做的更改，即使是在创建这些对象之后。
* 如果在对象中定义了属性/方法 X，则该对象的原型中将隐藏同名的属性/方法。例如，通过在 Dog.prototype 中定义 toString 方法，可以改写 Object.prototype 的 toString 方法。
* 更改只沿一个方向传递，即从原型到它的派生对象，但不能沿相反方向传递。

**图 7** 说明了这些效果。**图 7** 还显示了如何解决前面遇到的不需要的方法实例的问题。通过将方法放在原型内部，可以使对象共享方法，而不必使每个对象都有单独的函数对象实例。在此示例中，rover 和 spot 共享 getBreed 方法，直至在 spot 中以任何方式改写 toString 方法。此后，spot 有了它自己版本的 getBreed 方法，但 rover 对象和用新 GreatDane 创建的后续对象仍将共享在 GreatDane.prototype 对象中定义的那个 getBreed 方法实例。  
http://i.msdn.microsoft.com/Platform/Controls/CollapsibleArea/resources/plus.gifFigure 7  继承原型

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

function GreatDane() { }  
  
var rover = new GreatDane();  
var spot = new GreatDane();  
  
GreatDane.prototype.getBreed = function() {  
    return “Great Dane”;  
};  
  
// Works, even though at this point  
// rover and spot are already created.  
alert(rover.getBreed());  
  
// this hides getBreed() in GreatDane.prototype  
spot.getBreed = function() {  
    return “Little Great Dane”;  
};  
alert(spot.getBreed());   
  
// but of course, the change to getBreed   
// doesn’t propagate back to GreatDane.prototype  
// and other objects inheriting from it,  
// it only happens in the spot object  
alert(rover.getBreed());

静态属性和方法  
有时，您需要绑定到类而不是实例的属性或方法，也就是，静态属性和方法。在 JavaScript 中很容易做到这一点，因为函数是可以按需要设置其属性和方法的对象。由于在 JavaScript 中构造函数表示类，因此可以通过在构造函数中设置静态方法和属性，直接将它们添加到类中，如下所示：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

function DateTime() { }  
  
    // set static method now()  
    DateTime.now = function() {  
        return new Date();  
    };  
  
    alert(DateTime.now());

在 JavaScript 中调用静态方法的语法与在 C# 中几乎完全相同。这不应当让人感到吃惊，因为构造函数的名称实际上是类的名称。这样，就有了类、公用属性/方法，以及静态属性/方法。还需要其他什么吗？当然，私有成员。但 JavaScript 本身并不支持私有成员（同样，也不支持受保护成员）。任何人都可以访问对象的所有属性和方法。但我们有办法让类中包含私有成员，但在此之前，您首先需要理解闭包。  
  
闭包  
我没有自觉地学习过 JavaScript。我必须快点了解它，因为我发现如果没有它，在实际工作中编写 AJAX 应用程序的准备就会不充分。开始，我感到我的编程水平好像降了几个级别。（JavaScript！我的 C++ 朋友会怎么说？）但一旦我克服最初的障碍，我就发现 JavaScript 实际上是功能强大、表现力强而且非常简练的语言。它甚至具有其他更流行的语言才刚刚开始支持的功能。  
JavaScript 的更高级功能之一是它支持闭包，这是 C# 2.0 通过它的匿名方法支持的功能。闭包是当内部函数（或 C# 中的内部匿名方法）绑定到它的外部函数的本地变量时所发生的运行时现象。很明显，除非此内部函数以某种方式可被外部函数访问，否则它没有多少意义。示例可以更好说明这一点。  
假设需要根据一个简单条件筛选一个数字序列，这个条件是：只有大于 100 的数字才能通过筛选，并忽略其余数字。为此，可以编写类似**图 8** 中的函数。  
http://i.msdn.microsoft.com/Platform/Controls/CollapsibleArea/resources/plus.gifFigure 8 根据谓词筛选元素

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

function filter(pred, arr) {  
    var len = arr.length;  
    var filtered = []; // shorter version of new Array();  
    // iterate through every element in the array...  
    for(var i = 0; i < len; i++) {  
        var val = arr[i];  
        // if the element satisfies the predicate let it through  
        if(pred(val)) {  
            filtered.push(val);  
        }  
    }  
    return filtered;  
}  
  
var someRandomNumbers = [12, 32, 1, 3, 2, 2, 234, 236, 632,7, 8];  
var numbersGreaterThan100 = filter(  
    function(x) { return (x > 100) ? true : false; },   
    someRandomNumbers);  
  
// displays 234, 236, 632  
alert(numbersGreaterThan100);

但是，现在要创建不同的筛选条件，假设这次只有大于 300 的数字才能通过筛选，则可以编写下面这样的函数：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

var greaterThan300 = filter(  
    function(x) { return (x > 300) ? true : false; },   
    someRandomNumbers);

然后，也许需要筛选大于 50、25、10、600 如此等等的数字，但作为一个聪明人，您会发现它们全部都有相同的谓词“greater than”，只有数字不同。因此，可以用类似下面的函数分开各个数字：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

function makeGreaterThanPredicate(lowerBound) {  
    return function(numberToCheck) {  
        return (numberToCheck > lowerBound) ? true : false;  
    };  
}

这样，您就可以编写以下代码：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

var greaterThan10 = makeGreaterThanPredicate(10);  
var greaterThan100 = makeGreaterThanPredicate(100);  
alert(filter(greaterThan10, someRandomNumbers));  
alert(filter(greaterThan100, someRandomNumbers));

通过观察函数 makeGreaterThanPredicate 返回的内部匿名函数，可以发现，该匿名内部函数使用 lowerBound，后者是传递给 makeGreaterThanPredicate 的参数。按照作用域的一般规则，当 makeGreaterThanPredicate 退出时，lowerBound 超出了作用域！但在这里，内部匿名函数仍然携带 lowerBound，甚至在 makeGreaterThanPredicate 退出之后的很长时间内仍然如此。这就是我们所说的闭包：因为内部函数关闭了定义它的环境（即外部函数的参数和本地变量）。  
开始可能感觉不到闭包的功能很强大。但如果应用恰当，它们就可以非常有创造性地帮您将想法转换成代码，这个过程非常有趣。在 JavaScript 中，闭包最有趣的用途之一是模拟类的私有变量。  
  
模拟私有属性  
现在介绍闭包如何帮助模拟私有成员。正常情况下，无法从函数以外访问函数内的本地变量。函数退出之后，由于各种实际原因，该本地变量将永远消失。但是，如果该本地变量被内部函数的闭包捕获，它就会生存下来。这一事实是模拟 JavaScript 私有属性的关键。假设有一个 Person 类：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

function Person(name, age) {  
    this.getName = function() { return name; };  
    this.setName = function(newName) { name = newName; };  
    this.getAge = function() { return age; };  
    this.setAge = function(newAge) { age = newAge; };  
}

参数 name 和 age 是构造函数 Person 的本地变量。Person 返回时，name 和 age 应当永远消失。但是，它们被作为 Person 实例的方法而分配的四个内部函数捕获，实际上这会使 name 和 age 继续存在，但只能严格地通过这四个方法访问它们。因此，您可以：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

var ray = new Person(“Ray”, 31);  
alert(ray.getName());  
alert(ray.getAge());  
ray.setName(“Younger Ray”);  
// Instant rejuvenation!  
ray.setAge(22);  
alert(ray.getName() + “ is now “ + ray.getAge() +   
      “ years old.”);

未在构造函数中初始化的私有成员可以成为构造函数的本地变量，如下所示：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

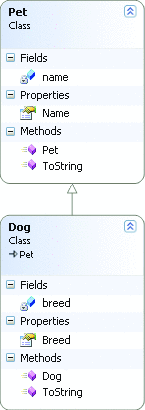
function Person(name, age) {  
    var occupation;  
    this.getOccupation = function() { return occupation; };  
    this.setOccupation = function(newOcc) { occupation =   
                         newOcc; };  
    
    // accessors for name and age      
}

注意，这些私有成员与我们期望从 C# 中产生的私有成员略有不同。在 C# 中，类的公用方法可以访问它的私有成员。但在 JavaScript 中，只能通过在其闭包内拥有这些私有成员的方法来访问私有成员（由于这些方法不同于普通的公用方法，它们通常被称为特权方法）。因此，在 Person 的公用方法中，仍然必须通过私有成员的特权访问器方法才能访问私有成员：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

Person.prototype.somePublicMethod = function() {  
    // doesn’t work!  
    // alert(this.name);  
    // this one below works  
    alert(this.getName());  
};

从类继承  
到这里，我们已经了解了构造函数和原型对象如何使您在 JavaScript 中模拟类。您已经看到，原型链可以确保所有对象都有 Object.prototype 的公用方法，以及如何使用闭包来模拟类的私有成员。但这里还缺少点什么。您尚未看到如何从类派生，这在 C# 中是每天必做的工作。遗憾的是，在 JavaScript 中从类继承并非像在 C# 中键入冒号即可继承那样简单，它需要进行更多操作。另一方面，JavaScript 非常灵活，可以有很多从类继承的方式。  
例如，有一个基类 Pet，它有一个派生类 Dog，如**图 9** 所示。这个在 JavaScript 中如何实现呢？Pet 类很容易。您已经看见如何实现它了：  
图 9 **类**

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

// class Pet  
function Pet(name) {  
    this.getName = function() { return name; };  
    this.setName = function(newName) { name = newName; };  
}  
  
Pet.prototype.toString = function() {  
    return “This pet’s name is: “ + this.getName();  
};  
// end of class Pet  
  
var parrotty = new Pet(“Parrotty the Parrot”);  
alert(parrotty);

现在，如何创建从 Pet 派生的类 Dog 呢？在**图 9** 中可以看到，Dog 有另一个属性 breed，它改写了 Pet 的 toString 方法（注意，JavaScript 的约定是方法和属性名称使用 camel 大小写，而不是在 C# 中建议的 Pascal 大小写）。**图 10** 显示如何这样做。  
http://i.msdn.microsoft.com/Platform/Controls/CollapsibleArea/resources/plus.gifFigure 10 从 Pet 类派生

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

// class Dog : Pet   
// public Dog(string name, string breed)  
function Dog(name, breed) {  
    // think Dog : base(name)   
    Pet.call(this, name);  
    this.getBreed = function() { return breed; };  
    // Breed doesn’t change, obviously! It’s read only.  
    // this.setBreed = function(newBreed) { name = newName; };  
}  
  
// this makes Dog.prototype inherits  
// from Pet.prototype  
Dog.prototype = new Pet();  
  
// remember that Pet.prototype.constructor  
// points to Pet. We want our Dog instances’  
// constructor to point to Dog.  
Dog.prototype.constructor = Dog;  
  
// Now we override Pet.prototype.toString  
Dog.prototype.toString = function() {  
    return “This dog’s name is: “ + this.getName() +   
        “, and its breed is: “ + this.getBreed();  
};  
// end of class Dog  
  
var dog = new Dog(“Buddy”, “Great Dane”);  
// test the new toString()  
alert(dog);  
  
// Testing instanceof (similar to the is operator)  
// (dog is Dog)? yes  
alert(dog instanceof Dog);  
// (dog is Pet)? yes  
alert(dog instanceof Pet);  
// (dog is Object)? yes  
alert(dog instanceof Object);

所使用的原型 — 替换技巧正确设置了原型链，因此假如使用 C#，测试的实例将按预期运行。而且，特权方法仍然会按预期运行。  
  
模拟命名空间  
在 C++ 和 C# 中，命名空间用于尽可能地减少名称冲突。例如，在 .NET Framework 中，命名空间有助于将 Microsoft.Build.Task.Message 类与 System.Messaging.Message 区分开来。JavaScript 没有任何特定语言功能来支持命名空间，但很容易使用对象来模拟命名空间。如果要创建一个 JavaScript 库，则可以将它们包装在命名空间内，而不需要定义全局函数和类，如下所示：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

var MSDNMagNS = {};  
  
MSDNMagNS.Pet = function(name) { // code here };  
MSDNMagNS.Pet.prototype.toString = function() { // code };  
  
var pet = new MSDNMagNS.Pet(“Yammer”);

命名空间的一个级别可能不是唯一的，因此可以创建嵌套的命名空间：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

var MSDNMagNS = {};  
// nested namespace “Examples”  
MSDNMagNS.Examples = {};   
  
MSDNMagNS.Examples.Pet = function(name) { // code };  
MSDNMagNS.Examples.Pet.prototype.toString = function() { // code };  
  
var pet = new MSDNMagNS.Examples.Pet(“Yammer”);

可以想象，键入这些冗长的嵌套命名空间会让人很累。 幸运的是，库用户可以很容易地为命名空间指定更短的别名：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

// MSDNMagNS.Examples and Pet definition...  
  
// think “using Eg = MSDNMagNS.Examples;”   
var Eg = MSDNMagNS.Examples;  
var pet = new Eg.Pet(“Yammer”);  
alert(pet);

如果看一下 Microsoft AJAX 库的源代码，就会发现库的作者使用了类似的技术来实现命名空间（请参阅静态方法 Type.registerNamespace 的实现）。有关详细信息，请参与侧栏“OOP 和 ASP.NET AJAX”。  
  
应当这样编写 JavaScript 代码吗？  
您已经看见 JavaScript 可以很好地支持面向对象的编程。尽管它是一种基于原型的语言，但它的灵活性和强大功能可以满足在其他流行语言中常见的基于类的编程风格。但问题是：是否应当这样编写 JavaScript 代码？在 JavaScript 中的编程方式是否应与 C# 或 C++ 中的编码方式相同？是否有更聪明的方式来模拟 JavaScript 中没有的功能？每种编程语言都各不相同，一种语言的最佳做法，对另一种语言而言则可能并非最佳。  
在 JavaScript 中，您已看到对象继承对象（与类继承类不同）。因此，使用静态继承层次结构建立很多类的方式可能并不适合 JavaScript。也许，就像 Douglas Crockford 在他的文章 [Prototypal Inheritance in JavaScript](http://javascript.crockford.com/prototypal.html) 中说的那样，JavaScript 编程方式是建立原型对象，并使用下面的简单对象函数建立新的对象，而后者则继承原始对象：

[**[复制]**](http://bbs.51js.com/viewthread.php?tid=80172&highlight=###)

**Code:**

function object(o) {  
        function F() {}  
        F.prototype = o;  
        return new F();  
    }

然后，由于 JavaScript 中的对象是可延展的，因此可以方便地在创建对象之后，根据需要用新字段和新方法增大对象。   
这的确很好，但它不可否认的是，全世界大多数开发人员更熟悉基于类的编程。实际上，基于类的编程也会在这里出现。按照即将颁发的 ECMA-262 规范第 4 版（ECMA-262 是 JavaScript 的官方规范），JavaScript 2.0 将拥有真正的类。因此，JavaScript 正在发展成为基于类的语言。但是，数年之后 JavaScript 2.0 才可能会被广泛使用。同时，必须清楚当前的 JavaScript 完全可以用基于原型的风格和基于类的风格读取和写入 JavaScript 代码。  
  
展望  
随着交互式胖客户端 AJAX 应用程序的广泛使用，JavaScript 迅速成为 .NET 开发人员最重要的工具之一。但是，它的原型性质可能一开始会让更习惯诸如 C++、C# 或 Visual Basic 等语言的开发人员感到吃惊。我已发现我的 JavaScript 学习经历给予了我丰富的体验，虽然其中也有一些挫折。如果本文能使您的体验更加顺利，我会非常高兴，因为这正是我的目标。