

# JavaScript for PHP Developers

---

## JavaScript for PHP Developers

### 简介

- 1.1 本书范围
- 1.2 语言
- 1.3 学习环境
  - 1.3.1 浏览器
  - 1.3.2 JavaScriptCore
  - 1.3.3 Node.js和Rhino
- 1.4 较长的例子
- 1.5 开始学习

### JavaScript 语法

- 2.1 变量
  - \$在JavaScript中的作用
- 2.2 值
  - 2.2.1 typeof 自省
  - 2.2.2 null和undefined
- 2.3 数组
- 关联数组
- 2.4 条件
  - 2.4.1 三元操作符
  - 2.4.2 严格比较
  - 2.4.3 switch
  - 2.4.4 try-catch
- 2.5 while和for循环
- 2.6 for-in循环
- 2.7 其他操作符
  - 2.7.1 in
  - 2.7.2 字符串连接
  - 2.7.3 类型强制转换
  - 2.7.4 void
  - 2.7.5 逗号操作符

### 函数

- 3.1 默认参数
- 3.2 任意多个参数
- 3.3 arguments.length 技巧
- 3.4 返回值
- 3.5 函数是对象
- 3.6 一种不同的用法
- 3.7 作用域
- 3.8 提升
- 提升函数
- 3.9 闭包
- PHP中的闭包
- 3.10 作用域链
  - 3.10.1 WebKit控制台中的作用域链
  - 3.10.2 保持作用域
  - 3.10.3 保留引用而不是值
  - 3.10.4 循环中的闭包
  - 3.10.5 练习: onclick 循环
- 3.11 立即函数
  - 3.11.1 初始化
  - 3.11.2 私有性

- 3.12 传递和返回函数
  - 回调函数不是字符串
- 面向对象编程
  - 4.1 构造器和类
    - 4.1.1 返回对象
    - 4.1.2 关于this的更多内容
    - 4.1.3 增强构造器
  - 4.2 原型
  - 4.3 对象直接量
    - 4.3.1 访问属性
    - 4.3.2 令人混淆的点号
    - 4.3.3 对象直接量中的方法
  - 4.4 奇特的数组
  - 4.5 自身属性
    - 4.5.1 **proto**
    - 4.5.2 this或prototype
  - 4.6 继承
    - 4.6.1 通过原型继承
    - 4.6.2 通过复制属性来继承
    - 4.6.3 Beget对象
    - 4.6.4 “经典的”extend()
    - 4.6.5 借用方法
    - 4.6.6 结论

## 简介

---

多年来，**JavaScript** 一直名声狼藉。很多开发者认为编写 **JavaScript** 代码很痛苦，因为程序的行为不可预期。有时编写完成代码，打开一个浏览器进行测试，却得到一条毫无帮助的错误信息。因此，开发者通常懒得学习该语言。

事实上，历史上的大多数问题是由于DOM和BOM实现上的差异所导致，而很小程度上，是由于 **JavaScript** 语言本身所导致的。DOM表示Document Object Model（文档对象模型）。它是一个API（应用程序接口），用来操作XML，XHTML和HTML写成的那些结构化文档。DOM是一种独立于语言的API，它也存在于PHP中。在 **JavaScript** 中，这个API随处可见：以document开头的任何内容，都与DOM有关。一个有趣的历史事实是，DOM从 **JavaScript** 中诞生，并且后来由 **W3C**（World Wide Web Consortium）作为独立于 **JavaScript** 及任何其他语言的一个API而标准化。如今，我们仍然可以见到最初的DOM（宣布在称之为DOM0）的一些残留，例如 `document.images` 和 `document.links`，而在DOM1中，用更加通用的方法 `document.getElementById()` 和 `document.getElementsByTagName()` 代替了它们。

BOM表示Browser Object Model（浏览器对象模型）。对于那些没有正式定义的内容来说，这是一个不错的名字，它是与浏览器相关的属性和方法的一个集合，例如，可用屏幕的大小或状态栏。大多数这样的属性，都像 `innerwidth` 一样可以全局地使用，尽管你大多数时候只是看到他们用作window对象的属性（例如，`window.innerWidth`）。很长一段时间以来，BOM都没有标准化，因此你可以想象得到，它们在浏览器之间存在一些差异。HTML5开始浏览器之间的通用行为规范化，这包括通用的BOM对象。

需要认识的另一个术语是ECMAScript。这是取出了DOM和BOM之后的核心 **JavaScript** 语言。这是该语言涉及的语法，循环，函数，对象等部分。**JavaScript** 最初是Netscape的一项创新，随后又被其他浏览器厂商模仿，由此，为该语言建立一个标准以便让所有实现者（浏览器厂商和其他组织）都遵循就变得越来越有必要。这一标准是由欧洲计算机制造商协会（European Computer Manufacturers Association, ECMA；现在称为Ecma International）指定的，称为ECMAScript。如今，从技术上讲，**JavaScript** 是Mozilla对ECMAScript的实现（还有JScript，这是Microsoft的现实版本），但是，当人们提到 **JavaScript** 的时候，并不代表这个意思。

总而言之，当人们提到 `JavaScript` 的时候，他们表示的可能是一组话题，包括DOM，BOM和ECMAScript。因此

$$JavaScript = ECMAScript + DOM + BOM$$

## 1.1 本书范围

与Web诞生的早期相比，一个很好的现象是，现在有了各种各样可供使用的，高质量的JavaScript库，如jQuery和YUI（Yahoo! User Interface Library）。这些库提供一个API层，将DOM和BOM的浏览器实现差异（有的时候，是ECMAScript自身的差异）抽象化，从而使开发者从痛苦中解脱出来。

本书的范围是ECMAScript，也就是语言本身。你可能已经通过PHP熟悉了DOM，而BOM并不是很有趣的内容（它只是全局变量和函数的一个集合）。此外，你总是可以使用一个库来抽象化DOM和BOM。本书独立于任何特定的JavaScript库，允许你关心核心语言，并且在需要的时候自行选择一个库。

本书使用术语JavaScript来表示ECMAScript，因此使用“ECMAScript”的情况并不常见，而且看起来很别扭。本书关注的是语言的最流行版本ECMAScript 3(ES3)。这个标准跳过了版本4，并且本书出版的时候，最新的版本是ECMAScript 5.1，也写作ES5。尽管ES5已经变得广为可用，如果想要支持旧的浏览器的话（例如，Internet Explorer 9之前的版本），你还是不能指望它。本书第六章将会详细介绍ES5中有哪些新内容和变化，并且将帮助你决定是否使用它。在旧的浏览器中，它甚至可以通过“shim”或“polyfill”来工作，我们将在本书第六章中介绍这点。

## 1.2 语言

一旦取出了所有DOM/BOM内容，剩下的就是JavaScript了，这是一种美丽的，很精致的语言，只不过它具有与C语言类似的语法，还带有一些内建的对象。

这是一种广为流行的语言，在客户端，服务器，移动手机，桌面计算机上，甚至在shell上都很常见。即便在PHP中，我们也可以使用V8js PHP类来嵌入JavaScript。实际上，很难找到一个无法运行JavaScript的环境，这意味着，我们只需要学习一种语言，而随处都可以使用它。

从一名PHP程序员的视角来看，JavaScript又是一种奇怪的语言。在你阅读本书之前，有几点需要注意

- JavaScript中的函数是对象，数组和正则表达式也是对象。
- 函数提供作用域，通过将代码包含在一个函数中，从而实现局部变量作用域。
- 大量使用闭包（Closure）。尽管从PHP5.3开始就有了闭包，但闭包在PHP中并不常用。在JavaScript中，闭包随处可见。
- 原型（Prototype）是JavaScript中重要概念之一，在PHP中没有对应的概念。这是JavaScript实现代码重用和继承的一种方式。
- JavaScript没有类的概念。从PHP的角度来看，这真的很奇怪，我们将会详细讨论这一点。

## 1.3 学习环境

为了简单起见，我们不会让创建和维护用户交互的事务纠缠你。JavaScript甚至没有输入和输出概念。I/O是由运行JavaScript的环境所提供的。最常见的环境就是Web浏览器，其中，用户交互的任务落到了HTML肩上。要学习JavaScript，我们根本不需要这些，只要使用JavaScript控制台所提供的I/O就行。

控制台允许你快速输入一段代码并查看其运行情况，就像是在命令行中调用的交互式PHP Shell一样：

```
$ PHP -a
Interactive shell
php > echo "Hi";
Hi
```

有很多种JavaScript控制台，可供选择用来学习该语言。最好用的控制台，是浏览器所提供的控制台。

### 1.3.1 浏览器

在一个桌面的WebKit浏览器中（例如，Chrome或Safari），直接载入一个页面，在任何位置单击鼠标右键，并且从菜单中选择“inspect element”（审查元素）。这将会启动Web Inspector。单击Console标签页，这就准备好了。

在最新的Firefox版本中，可以在菜单栏中选择Tools-->Web Developer --> Web Console，从而打开控制台。此外，也可以安装Firebug扩展，该扩展在任何Firefox版本中都能工作。

Internet Explorer（从IE8开始）有一个F12键打开的Developer Tools，在其Script标签页下有一个控制台。

### 1.3.2 JavaScriptCore

如果你使用Mac系统，它已经带有一个JavaScript解释器，所以可以运行shell脚本。Windows系统也有一个内建的命令行JavaScript解释器，但没有控制台。

Mac系统中的JavaScript命令行解释器是一个叫做JavaScriptCore的程序，可以在/System/Library/Frameworks/JavaScriptCore.framework/Versions/Current/Resources/jsc下找到它。

也可以为它起一个别名，以便很容易地找到它。

1. 启动Terminal应用程序（例如，通过在Spotlight中输入“Terminal”）。
2. 输入如下内容：

```
alias  
jsc='/System/Library/Frameworks/JavaScriptCore.framework/Versions/Current/Res  
ources/jsc'
```

3. 现在，可以通过jsc启动控制台了

注意，与浏览器的控制台相比，这是一个更为简单的环境，因为它甚至没有BOM和DOM的概念，而只是核心JavaScript语言。

请自行给~/.profile行添加别名，以便当你需要的时候，它总是在哪里。

---

注意：JavaScriptCore引擎在基于WebKit的浏览器中，但是Chrome并没有使用它，Chrome拥有自己的JavaScript引擎，该引擎叫做V8。

---

### 1.3.3 Node.js和Rhino

如果你在自己的计算机上安装了Node.js或Rhino，那么，你可以使用它们的控制台，其优点是可以在任何操作系统上工作。

Node.js基于Google的V8 JavaScript引擎。当你安装它之后，可以在Terminal输入**node**来打开控制台。

Rhino是Mozilla使用Java编写的一款JavaScript解释器。当你下载了*rhino.jar*文件并将其保存到便于查找的位置之后（对我来说，就是home目录），可以通过输入如下内容打开控制台。

```
$ java -jar ~/rhino.jar
```

如果你选择使用Node.js，要注意其REPL（read-evaluate-print loop）类似于PHP -a（如果不比它更好的话），可以用来学习和体验该语言。

## 1.4 较长的例子

使用JavaScriptCore, Node.js或Rhino, 我们可以创建一个更长点的例子, 并且将其分别保存到文件中。实际上, 这展示了如何用JavaScript编写自己的Shell脚本。我们通过为外部文件传递参数来运行它们:

```
$ jsc test.js
$ node test.js
$ java -jar ~/rhino.jar test.js
```

## 1.5 开始学习

有了一个方便的JavaScript控制台, 我们就准备在JavaScript这边大地上开始一段旅程了。首先, 第二章将介绍JavaScript语言, 重点关注其与PHP的相似与不同。

# JavaScript 语法

和PHP一样, JavaScript拥有类似C的语法, 因此, 我们会很快熟悉它。本章将介绍语法基础, 重点强调在变量, 数组, 循环, 条件, 以及某些形形色色(并且有点奇怪的)操作符方便有哪些类似之处和区别。

## 2.1 变量

在PHP中, 要定义一个变量, 我们会这么写:

```
// PHP
$n = 1;
```

在JavaScript中, 对等的形式是:

```
// javascript
var n = 1;
```

没有美元符号, 只需要给出变量的名称。与在PHP中一样, 我们不需要定义变量类型, 因为变量的类型取决于其值。对于所有的类型, 都是用var。如果需要一个数值类型, 那么给该变量一个数字值。这种方法也同样适用于布尔类型和字符串:

```
var n = 1; // 数字
var b = true; // 布尔类型
var s = "hello"; // 字符串
```

也可以选择声明一个变量而并不使用一个值来初始化它。在这种情况下, 特殊值undefined会赋给该变量:

```
var a;
a; // `a` 有一个特殊值`undefined`
```

注意: 重新声明一个已有的变量, 则并不会将该变量的值设置回undefined:

```
var a = 1;
var a;
// `a` 仍然是1
```

可以用一条var语句来声明多个变量（并且可选择用一个值来初始化变量），只要用一个逗号将变量分开，并且用一个分号作为结束：

```
var pi = 3.1415,
    yeps = true,
    nopes,
    hi = 'Hello',
    world = 'world';
```

注意：从技术上讲，var是可选的。但是，除非变量在作用域链（第三章将详细介绍）中确定处于较高的位置，如果省略了var的话，最终会得到一个全局变量。你可能已经知道，并且可能有过亲身经历才理解，全局命名空间污染是很痛苦的事情。此外，声明一个全局变量的时候，使用和不使用var还有一些细微差异。简而言之，不要视图省略var，在声明变量的时候，一定要使用它。

可以使用字母，数字，下划线和美元符号来命名变量。然而，变量名不能以数字开头：

```
var _1v; // 有效
var v1; // 有效
var v_1; // 有效
var 1v; // 无效
```

## \$在JavaScript中的作用

在面试的时候，有时会问到这种刁钻的问题。简单地回答“是”，没什么用。

可以在变量名的任何位置使用\$。早期的ECMAScript标准建议，\$应该只用于自动生成的代码中，而这是开发者通常容易忽略的一个建议。在实际应用中，\$常常用作一个函数的名称，而该函数从HTML文档中选择一个DOM节点。

假如有了如下的HTML标记：

```
<div id='myid'></div>
```

要引用这个DOM，可以使用DOM方法 `getElementById()`：

```
var mydiv = document.getElementById("myid");
```

然而，这段代码在客户端编程中经常用到，因此，这需要录入很多代码。因此，很多库定义了一个类似的函数，如下所示：

```
function $(id){
    return document.getElementById(id);
}
```

然后，我们可以像下面这样做：

```
$("myid").innerHTML = "Hello world";
```

面试官通过刁钻的\$问题来检查应试者是否有超出一个特定的库之外的JavaScript经验。回答"\$回去一个DOM元素"，这对于很多库以及浏览器控制台来说，确实如此，但对于JavaScript自身来说就不对了。

注意：在大多数浏览器控制台中，诸如web inspector和Firebug中，\$()函数只是为了方便快速访问DOM节点，再有\$\$()，它对应document.querySelectorAll()，允许你使用一个CSS选择器来查找DOM元素，如下所示：

```
// <li> DOM节点的列表，这些<li>节点都是带有ID “menu”的一个元素的子元素
$$("#menu li")
```

## 2.2 值

JavaScript中有5种原始值类型，其他的一切都是对象。这5中原始类型是：

- 字符串
- 布尔类型
- 数字
- null
- undefined

放在单引号或双引号之间的任何内容，都是一个字符串值。与PHP不同，这里单引号和双引号是没有区别的。

不带引号的直接量true和false，用来创建布尔类型值。

对于JavaScript中的数字来说，没有float，int，doubles等，它们都是数字。可以使用一个"."来定义浮点值（例如。3.14）。

在值前面放一个0，将会得到一个八进制数值：

```
var a = 012;
a === 10; // true
```

注意：在ES5的“严格模式”中，八进制直接量现在已经废弃和禁止了（参见本书第六章第6.1节关于严格模式的详细介绍）。

在值的前面放一个0X，将会得到一个十六进制值：

```
var a = 0xff;
a === 255; // true
```

也可以使用科学表示法：

```
var a = 5e3; // 5 后面带3个0
a === 5000; // true
var b = 5E-3; // 将.移动到5左边的第三位
b === 0.005; // true
```

## 2.2.1 typeof 自省

可以使用操作符来搞清楚所操作值的值的类型：

```
var a = "test";
typeof a; // "string"
typeof 0xff; // "number"
typeof false; // "boolean"
```

有时候，typeof用作一个函数（例如，typeof(a)），但是，不建议这么做。记住，typeof不是一个函数，而是一个操作符。在这种用法中，typeof()之所以像是函数一样使用，是因为()也是一个操作符，它叫做分组操作符（grouping operator），通常用来覆盖操作符优先级。例如：

```
3 * (2 + 1); // 9
// 相反
3 * 2 + 1; // 7
```

---

注意：JavaScript中的typeof(a)和typeof a的关系，就像是PHP中的echo(\$a)和echo \$a；的关系（尽管hi使用二者的原因不同）。PHP中include(\$a)和include \$a也是基于相同的道理。即便这种写法有效，但仍然被看做一种糟糕的做法，因为它有点鱼目混珠。

---

## 2.2.2 null和undefined

到目前为止，我们介绍了5中原始类型中的3种，分别是字符串，布尔类型和数字。另外两种类型是null和undefined。

undefined类型只有唯一的一个值，就是值undefined，当我们声明一个变量而没有初始化它的时候，这个变量就会用undefined来初始化。此外，当我们有一个函数而它没有明确返回一个值，那么，它将返回undefined：

```
var a;
typeof a; // undefined
var b = 1;
b = undefined;
typeof b; // undefined
```

注意，typeof操作符总是返回一个字符串。在前面的例子中，b的值为undefined，但typeof b返回了字符串"undefined"。新手常常把字符串"undefined"和值undefined搞混淆。考虑一下如下代码的区别：

```
var b;
b === undefined; // true
b === 'undefined'; // false
typeof b === 'undefined'; // true
typeof b === undefined; // false
```

null类型只有一个值，就是值null。它很少用到，但是当你想要调用一个函数接受很多参数的时候，或者当你想要明确地区分未声明的变量和未初始化的变量的时候（因为undefined可能意味着未声明，也可能表示未初始化），可以将null传递到函数调用中。令人惊讶的是，当对一个null值使用typeof的时候，它返回"object"。



```
var a = null;
typeof a;// "object"
```

注意：我们可以期待，在ECMAScript下一个版本中typeof null 返回"null"。

## 2.3 数组

在PHP中，我们使用如下代码定义一个数组：

```
// PHP
$a = array(1,2,3);
```

在JavaScript中，我们去掉array并且使用方括号：

```
// javascript
var a = [1,2,3];
```

注意：PHP 5.4支持一种更为简单的语法，因此，声明数组变得和JavaScript中的类似了：

```
// php
$a = [1,2,3];
```

如下代码在JavaScript中也有效：

```
// javascript
var arr = Array(1,2,4);
var arrgh = new Array("a","B",'C');
arr; // [1,2,4]
arrgh;// ["a","B","C"]
```

关于这一语法，稍后还有更多讨论，但是记住，方括号表示法（数组直接量）是首选。

就像在PHP中一样，我们可以在一个数组中混合任何类型的值，包括其他的数组：

```
// javascript
var a = [a,'Yes',false,null, undefined,[1,2,3]];
```

在JavaScript中，数组是对象：

```
typeof a;// "object"
```

在本书中，稍后我们将讨论数组对象的属性和方法，这里只是简略地介绍：

```
var a = [1,2,3];
a.length; // 3，就像PHP中的count()
```

遗憾的是，我们不能像在PHP中那样，使用a[]来添加一个元素。我们需要传递所要添加的元素的索引：

```
a[3] = 4;
```

或者，更为常见的做法是，使用数组的长度为下一个索引：

```
a[a.length] = "five";
```

另外，还有一个push()方法，它类似于PHP的array\_push()函数：

```
a.push(6);
```

执行完所有这些语句后的最终结果是：

```
a; // [1,2,3,4,'five',6]
```

就像我们所预料的那样，可以使用元素的索引来访问它们：

```
var b = a[4]  
b; // 'five'
```

## 关联数组

JavaScript中没有关联数组，当我们需要关联数组的时候，使用一个对象。如果想要像下面这样：

```
// PHP  
$assoc = array('one'=>1,'two'=>2);
```

那么，在JavaScript中，我们可以像下面这样编写：

```
// javascript  
var assoc = {"one":1,"two":2};
```

因此，JavaScript与PHP的不同之处仅在于使用：而不是=>，并且将值包含在了花括号{}中。

将键包围了起来引号，我们把键称为属性（properties），因为它们真的是一个对象的属性，通常会被忽略。当键是有效的标识符的时候（意味着我们可以用作变量名），引号是可选的。

```
var assoc = {one:1,two:2};
```

给这个关联数组/对象/字典添加更多元素：

```
assoc.three = 3;
```

要删除属性，使用delete操作符：

```
delete assoc.two;
```

在ES3中，要统计一个对象中的属性的数目，没有比在一个for-in循环中遍历所有属性更简单的方法了。稍后，我们将会更为详细地介绍如何做到这一点。

要访问一个对象的属性，使用点表示法（dot notation）：

```
assoc.one; // 1
```

此外，也可以使用括号表示法（square bracket notation），这看上去与PHP中的数组相似：

```
assoc['one']; // 1
```

## 2.4 条件

JavaScript中的if语法与PHP中的if语法是相同的：

```
var a,b;
if(a === 'hello'){
    b = 'world';
}else if(a === 'goodbye'){
    b = 'bye';
}else{
    b = 'what?';
}
```

尽管PHP针对if-then-else结构有几种替代的算法，但是在JavaScript中，只有唯一的一种if结构（如果三元操作符不算的话）。因此，我们不能使用elseif来替代else if。

### 2.4.1 三元操作符

条件语法的缩写形式叫做三元操作符（ternary operator），因为它是接受3个操作数的一个操作符，看上去很熟悉，如下所示：

```
var num = 11;
var whatlist = (num%2)?"odd":"even";
num + ' is ' + whatlist; // 11 is odd;
```

你可以已经了解到，要避免将多个三元操作符叠加使用而不使用括号，因为这很难读懂。另一个原因是，它们在JavaScript的工作方式存在差异。例如：

```
// PHP
echo true?"a":true?"b":"b";// "b"
// 等同于
echo (true?"a":true)?"b":"c";// "b"
echo ("a")?"b":"c"; // "b"
```

在JavaScript中，语法是相同的，但是得到的结果是不同的，因为计算机顺序所不同：

```
// javascript
true?"a":true?"b":"c";// "a"
// 等同于
true?'a':'i\'m ignored';// 'a'
```

## 2.4.2 严格比较

你是否注意到目前为止所有的代码中使用的都是三元相等符号===。在JavaScript中，比较的方式和PHP类似：

- 三个相等符号表示严格比较（strict comparison），对表达式的值和类型都会进行比较。
- 两个相等符号表示宽松比较（loose comparison），只是比较值，如果需要的话，其中一个表达式的值会强制转型为另一个表达式的类型，以执行类型比较。

为了使调试工作更为容易，最好是知道你要比较的表达式的类型，因此，使用===是一种比较好的做法。这更容易一些，因为你在开发过程中捕获错误，从而避免程序“随意”地运行而其中的值成为不可预期的。你还需要记住强制转型的规则和习惯，这些和PHP中不完全相同。

通常，有很多假（false）的值（在非严格的比较中，这些值会强制转型为false），包括：

- 字符串""
- 数字0
- 值false
- Null
- undefined
- 特殊的数值NaN，它表示“not a number”（不是一个数字）。就像PHP的NaN常量。

在非严格的比较中，可能还有陷阱：

```
null == undefined; // true
"" == 0; // true
```

但是，

```
null === undefined; // false
"" === 0; // false
```

因为：

```
typeof null === typeof undefined; // false, "object" !== "undefined"
typeof 0 === typeof ""; // false, 'number' !== 'string'
```

所有的对象都是真的（true）。数组是对象，因此，在宽松比较中，它们总是强制转型为true。这和PHP所有不同，在PHP中，空数组是假的：

```
// php
if(array()){
    echo 'hello';
    // 在PHP中，不会执行这段代码
}
```

```
if([]){
    console.info("hello");
    // 在JavaScript中，会执行这行代码
}
```

并不是所有的假的值都非严格地与另一个假的值相等：

```
undefined == null;// true
undefined == 0;// false
```

再回顾一下，你可以总是使用===进行严格比较，从而使得身边的这些比较问题变得简单且容易。你不需要记住强制转型规则和PHP中的区别。

### 2.4.3 switch

除了仅有的一点区别外，switch控制语句与PHP中的工作方式相同。这一点区别就是，case是使用严格比较来计算的：

```
// javascript
var a = "";
var result = '';
switch(a){
  case false:
    result = 'a is false';
    break;
  case a+'hello':// 这里允许用表达式
    result = 'what?';
    break;
  case "":
    result = 'a is an empty string';
    break;
  default:
    result = '@#$';
}
```

这里得到的结果是“a is an empty string”，而在PHP中，类似的代码会得到“a is false”。

### 2.4.4 try-catch

try-catch语句块也几乎与PHP中相同。如下是PHP中的例子：

```
// PHP
try{
  throw new Exception('ouch');
}catch(Exception $e){
  $msg = $e->getMessage();
}
```

下面使用JavaScript来编写它：

```
// JavaScript
var msg = "";
try{
  throw new Error('ouch');
}catch(e){
  msg = e.message;
}
msg;// 'ouch'
```

与PHP相比，在JavaScript中有如下几个显著的区别：

- 抛出的是Error对象，而不是Exception对象。

- 捕获时，并未声明类型。
- 访问message属性，而不是调用getMessage()方法。

在JavaScript中，也有finally（在PHP 5.5以后的版本中，也有finally），但是，实际很少使用它，特别是考虑到它可能在IE中导致一些bug：

```
var msg = '';  
try{  
    throw new Error('ouch');  
}catch(e){  
    msg += e.message;  
}finally{  
    msg += ' - finally';  
}  
msg; // ouch - finally
```

正如你所看到的，不管try语句是否抛出一个错误，finally语句块中的语句总是会执行。

注意：稍后，我们将更多介绍变量作用域，但是注意，catch语句块对作用域规则是一个例外，该规则就是：没有块作用域，只有函数作用域。在前面的示例中，e只是在catch语句块中可见。然而，只有对Error对象e才使用这一例外原则，如果我们在catch语句块中定义了其他的变量，那么，这些变量的作用域会跨出语句块：

```
try{  
    throw new Error();  
}catch(e){  
    var oops = 1;  
}  
typeof e; // "undefined"  
typeof oops; // "number"
```

当执行catch语句块的时候，try-catch语句块还会有一些性能上的影响，应该避免将其放入到对性能有影响的代码段中（例如，应该将try-catch语句块放到循环之外）。

## 2.5 while和for循环

while，do-while和for循环在JavaScript中和PHP中的用法完全相同，都是照搬了C的语法。让我们轻松片刻，看看计算机从1到100的总和的一个例子。

本小结中代码在两种语言中都能运行，但是在JavaScript中，应该像下面这样开始：

```
var $i,$sum;
```

这不是必须的，但是，这会帮助你养成总是声明变量的好习惯。如果你漏掉了这一行，代码也能工作，但是，\$i和\$sum编程了全局变量。变量名称前面的\$，对名称来说，是一个不常用但绝对有效的部分。

如下是两种语言都适用的while循环的示例：

```

$i = 1;
$sum = 0;
while($i<=100){
    $sum += $i++;
}
$sum;// 5050
$i;// 101

```

\$sum现在是5050.

类似地，如下是两种语言都适用的do-while循环：

```

$i = 1;
$sum = 0;
do{
    $sum += $i++;
}while($i<= 100);
$sum;// 5050

```

在PHP中for循环也是一样的，如下的示例同样在两种语言中都有效：

```

$i = 1;
$sum = 0;
for($i=1,$sum=0;$i<=100;$i++){
    $sum += $i;
}
$sum;// 5050

```

在所有这些示例中，\$i最终结果都是101，而不是100.你能说出原因吗？

## 2.6 for-in循环

在JavaScript中，当我们需要一个关联数组的时候，我们使用对象。并且，我们用一个for-in循环而是foreach循环来遍历对象。

假设你在PHP中定义了如下的数组

```

//PHP
$clothes = array(
    'shirt'=>'black',
    'pants'=>'jeans',
    'shoes'=>'none', // 结尾的逗号没有问题，我们甚至推荐这样做
);

```

在JavaScript中，相同的数据如下所示：

```

// javascript
var clothes = {
    shirt:'black',
    pants:'jeans',
    shoes:'none' // 由于旧的IE的存在，结尾放逗号会有问题
};

```

在PHP中，我们像下面这样遍历数据：

```
// php
$out = '';
foreach($clothes as $key=> $value){
    $out .= $key.': '.$value.'\n';
}
```

在JavaScript中，我们使用for-in循环：

```
// javascript
var out = '';
for(var key in clothes){
    out += key + ": " + clothes[key] + '\n';
}
```

在该循环中，我们不能直接访问下一个属性的值，因此，我们使用方括号表示法clothes[key]来获取值。

这要注意，枚举属性的顺序不是固定的并且依赖于具体实现。换句话说，shoes可能出现在shirt之前，尽管其在定义中位于后面。如果想要保持这个顺序，我们需要使用一个常规的数组来存储数据。

这里我只是为了便于说明，如果你暂时忘记了PHP中的foreach必须使用=>来访问值，你也可以在PHP中像下面这样做，以模拟JavaScript的for-in循环：

```
$out = '';
foreach(array_keys($clothes) as $key){
    $out .= $key.': '.$clothes[$key].'\n';
}
```

## 2.7 其他操作符

在深入函数这个重要话题之前，我们先来看看JavaScript中几个奇怪的操作符。

### 2.7.1 in

在上一个小结中，in操作符用来遍历一个对象的属性。它也可以用来检查一个对象中是否存在某个属性：

```
if('shirt' in clothes){ // true
    // TODO
}
if('hat' in clothes){ // false
}
```

也可以使用如下代码来进行检查：

```
if(clothes['shirt']){// true
    // TODO
}
```

或者，更常用的方式是使用点表示法：

```
if(clothes.shirt){ // true
    // TODO
}
```



不同之处在于，`in`只是检查该属性是否存在，它根本不查看其值。使用`clothes.shirt`检查，只有该属性存在并且其值为非假的时候，才会返回`true`。

例如，如果我们添加一个新的属性：

```
clothes.jecket = undefined;
```

那么：

```
"jecket" in clothes; // true
```

但是：

```
!!clothes.jecket; // false
```

因为`clothes.jecket`的值是`undefined`，只是一个假值，并且会强制转型为`false`。

这类似PHP中的`isset()`和`empty()`之间的区别。

此外，更麻烦一点，我们可以使用`typeof`操作符来只检查属性是否存在而不检查其值：

```
typeof clothes.jecket !== 'undefined'; // true
// 但是请注意，当一个属性是'undefined'值的时候，相应地就会得到false
clothes.i_am_undefined = undefined;
typeof clothes.i_am_undefined === typeof clothes.i_dont_exist; // true
```

这种写法的更简洁的方式是，与`undefined`值比较：

```
clothes.jecket !== undefined; // true
```

## 2.7.2 字符串连接

在查看`for-in`循环的示例的时候，你可能已经注意到了，`+`操作符用来连接字符串：

```
var alphabet = 'a'+'b'+'c';
alphabet; // 'abc'
```

这可能有些奇怪，但是`+`操作符既可以用来将数字相加，也可以用来连接字符串：

```
var numbers = '1'+'2'+'3';
var sum = 1 + 2 + 3;
numbers; // '123'
sum; // 6
```

那么，我们如何知道该执行哪一种操作呢？这取决于操作数的类型。

不必说，`+`操作符经常会导致错误，因此，记住变量类型就很重要。

一种替代性的方法是，将要添加的字符串连续存储到一个数组中，完成后，再将数组的元素链接起来。这种连接方法甚至在较早的IE 7和更早版本中也能很好地工作：

```
['R',2,'-', 'D',2].join(''); // 'R2-D2'
```

最后，还可以使用字符串的`concat()`方法，该方法不需要我们为了连接而创建临时的数组对象：

```
''.concat('R',2,'-', 'D',2);// 'R2-D2'  
String.prototype.concat('R',2,'-', 'D',2);// 'R2-D2'
```

### 2.7.3 类型强制转换

由于+既可以用于加法也可以用于字符串连接，我们也可以使用+来快速地转换变量的类型：

```
+'1';// 将字符串“1”转化为数字1  
'+1;// 将数字1转换为字符串1
```

说到类型强制转换，我们可以使用!!将变量转化为布尔类型：

```
!!1;//true  
!!"";// false
```

第一个!取反，并且如果该值是假的话，会返回一个布尔true；第二个!再次取反，得到了转换为布尔类型后我们想要的结果。此外，内建的构造函数（我们稍后将会详细介绍）也可以用来执行类型强制转换：

```
Number("1") === 1;// true  
String(100) === "100";// true  
Boolean(0) === false;// true  
Boolean(1) === "true";//true
```

### 2.7.4 void

我们可能会遇到的另一个操作符是void。它接受任意的表达式作为操作数，并且返回undefined：

```
var a = void 1;  
typeof a === 'undefined';// true
```

void操作符很少用到，但是有时候，它会用作链接中的href。和typeof操作符一样，人们常常错误地将其与分组操作符()一起使用，这使其看上去像是一个函数调用：

```
<!-- 这不是一种好的做法 -->  
<a href="javascript:void(0)">click and nonthing happens</a>
```

这种糟糕的做法看上去挺无辜的，但是，你思考一下会发现，这么做确实很傻。在这个过程中，发生了如下的事情：

- 用户单击，计算一个JavaScript表达式。
- 直接量0作为一个操作数发送给分组操作符()。
- 分组操作符返回组中最终的值0。
- 现在发送给void操作符。
- void在这里是一个黑洞，它接受操作符并且返回undefined。

即便不考虑这种让链接什么也不做的糟糕语意，人们还是会将href设置为javascript:undefined。

### 2.7.5 逗号操作符

逗号操作符是另一个有点奇怪的操作符。它是JavaScript中优先级最低的操作符，它直接返回传递给它的最后一个操作数的值：

```
var a = ('hello','there');  
a; // "there"  
var a = ("hello","there","young","one")  
a; // "one"
```

它类似于&&和||，然而，当输出结果一目了然的时候，这些表达式会停止计算，但是逗号操作符则会继续计算：

```
0 && 7 && 1; // 0  
0 || 7 || 1; // 7  
0,7,1; // 1
```

当JavaScript只期望一个表达式，而你想要偷偷插入另一个表达式的时候，逗号操作符很有用。

注意：你是否注意到，前面的例子中，&&和||并不总是像它们在PHP中那样返回一个布尔值。相反，它们返回相应的表达式的值。这可能有点令人混淆，特别是在如下的常用模式中，也按照默认情况编写JavaScript的时候。

```
// javascript  
var a = 100;  
var b = a || 200;  
b; // 100
```

```
// php  
$a = 100;  
$b = $a || 100;  
var_dump($b); // bool(true)
```

## 函数

函数是JavaScript中的重要主题，因为JavaScript的函数有很多用途。其基本形式看上去就像一个PHP函数一样：

```
// 在两种语言中都有效  
function sum($a, $b){  
    return $a + $b;  
}  
sum(3,5); // 8
```

### 3.1 默认参数

就像在PHP中一样，还没有一种语法运行一个函数带有默认值。ECMAScript未来的一个版本有计划要这么做，但是现在，我们必须在函数体内部自己做到这一点。假设我们想要让第二个参数有一个默认的值2：

```
// php  
function sum($a, $b = 2){  
    return $a + $b;  
}
```

```
// js
function sum(a, b){
    b = b || 2;
    // 有时候也写作
    // b || (b = 2)
    return a + b;
}
```

这是一种简短的语法，在很多情况下都能很好地工作，但是在这个特定的示例中，它有点太简单了，因为**b || 2**是对**b**的宽松比较。如果将0作为一个参数传递，这种比较将会得到**false**，并且**b**变成2：

```
sum(3); // 5
sum(3,0); // 5，并非我们期待的结果
```

一个较长一些的版本是使用**typeof**：

```
function sum(a, b){
    b = typeof b === "undefined"?2:b;
    return a + b;
}
sum(3,0); // 3
sum(3); // 5
```

## 3.2 任意多个参数

我们不一定必须传递函数所期望的所有参数，并且JavaScript也不会为此抱怨。换句话说，没有必须的参数。如果要让一个参数成为必需的，我们需要在函数体中强调这一点。

我们也可以传递比函数所期望的参数更多的参数，多余的参数会被愉快地忽略掉，而不会导致任何错误：

```
sum(); // NaN
sum(1,2); // 3
sum(1,2,100,100); // 3
```

我们可以准备接受多个参数，并且优雅地处理这种情况。在PHP中，我们可以使用**func\_get\_args()**，它返回传递给函数的参数的一个数组。在JavaScript中，我们使用类似数组的对象**arguments**。这是可供函数体使用的一个对象。之所以称之为“类似数组的”，因为它像一个数组一样，其属性是可以从索引0开始枚举的，并且它还有一个**length**属性：

```
function sum(/*nothing here*/){
    for(var i = 0,result = 0;i < arguments.length;i++){
        result += arguments[i];
    }
    return result;
}

sum(); // 0
sum(11); // 11
sum(1,2); // 3
sum(1,2,3); // 6
sum(1,10,2,9,3,8); // 33
```

记住，尽管arguments看上去像是一个数组，但它实际上是一个对象，并且它没有诸如push(), pop(), 这样的数组方法。然而，我们可以很容易地将这个类似数组的对象转化为一个数组。请准备好了解prototype相关的一些新内容，我们稍后将会进行详细介绍：

```
function sum(){
    var args = Array.prototype.slice.call(arguments);
    typeof arguments.push;// "undefined"
    typeof args.push;// "function"
}
```

注意：当对函数使用typeof操作符的时候，将会返回字符串“function”

## 3.3 arguments.length 技巧

Andrea Giammerchi建议的处理可选参数的一种聪明方法是使用arguments对象：

```
// 这里我们有一个函数，它带有所有4个默认的
// 参数，模仿PHP的声明
// function sum($a = 1, $b = 2, $c = 3, $d = 4) ....
function sum(a,b,c,d){
    //注意：不需要break
    switch(arguments.length){
        case 0:a = 1;
        case 1:b = 2;
        case 2:c = 3;
        case 3:d = 4;
    }
    return a+b+c+d;
}

// 测试
sum();// 10
sum(1);// 10
sum(11); // 20
sum(1,2,3,24); // 30
sum(11,22); // 40
```

## 3.4 返回值

函数总是会返回一个值。如果一个函数不能使用return语句，那么，它会隐式地返回值undefined:

```
// 明确的“return”
function iReturn(){
    return 101;
}
iReturn() === 101; // true
typeof iReturn() === "number"; // true

// 隐式的“return”
function noReturn(){
    // 没有内容
}
```

```
noReturn() === undefined; // true
typeof noReturn === 'undefined'; // true
```

## 3.5 函数是对象

JavaScript中函数是对象，理解这一点很重要。它们带有自己的一些属性和方法。

例如，看看下面这个简单的函数：

```
function sum(a,b){
    return a+b;
}
```

函数对象的length属性告诉我们，在函数定义的()部分中列出了多少个形式参数，换句话说，该函数期望有多少个参数：

```
sum.length; // 2
```

函数对象的call()和apply()方法，提供了调用该函数的一种替代的方法：

```
sum.call(null,3,4); // 7
sum.apply(null,[3,4]); // 7
```

两者之间的区别在于，apply()所接受的参数是以数组的形式传递给该函数的，而不是一个一个地传递的。这两种方法，类似我们在PHP中的如下用法：

```
// PHP
call_user_func('sum',2,3); // 5
call_user_func_array('sum',array(3,4)); // 7
```

函数是对象，这一事实意味着，我们可以给函数对象添加属性和方法。这可能很有用，例如，我们可以缓存函数调用的结果以避免重复性的工作。

---

注意：如果你不明白为何传递给call()和apply()的第一个参数是null,暂时可以忽略这个问题。稍后，我们会对此了解更多。提示：函数是一个对象，在需要的时候，sum()函数可以在其函数体中用this来引用。

---

## 3.6 一种不同的用法

函数是对象，一次可以像所有其他的数据类型一样，将它们赋值给变量。如下的代码段，展示了一种完全有效并且很常用的语法：

```
var sum = function(a,b){

    return a+b;

}

sum(2,2); // 4
```

这种语法展示了所谓的函数表达式（function expression），这和我们所熟悉的另一种叫做函数声明（function declaration）的语法不同。函数声明是一种更为常用的语法。并且在PHP 5.3之前的版本中，这是唯一可用的语法。

---

**注意：**根据上下文的不同，function关键字具有不同的语义含义，在函数表达式中，它是一个操作符。在函数声明中，它是一条语句。

---

注意函数表达式末尾的分号。和函数声明不同，函数表达式必须有一个分号。

---

**注意：**JavaScript有一种分号插入机制，如果你忘了分号的话，解释器会为你添加分号。然而，最好是你自行添加分号，以避免歧义。当你精简自己的代码以减少其在产品代码中大小的时候，这种做法是很有帮助的，由于使用了精简，所有的代码几乎在一行。

---

你还可能会碰到另一种语法，叫做命名的函数表达式（named function expression，NFE）。它如下所示：

```
var sum = function plum(a,b){
    return a+b;
};

sum(21,21); // 42

plum(21,21); // Error:plum() 没有定义
```

在进行调试中，当调试器读取函数对象的非标准但通常可用的named属性的时候，这种语法很有用。

```
sum.name; // "plum"
```

这种语法之所有叫做命名的函数表达式，是因为我们给函数指定了一个名称。如果没有指定名称（这种情况更为常见），最终得到一个未命名的函数表达式（unnamed function expression），或者直接称之为函数表达式（function expression）或匿名函数（anonymous function）。

不推荐这样的做法：为表达式设置的名称（标识符）和你分配给他的变量的名称不同（就像前面的实例中的plum和sum）。这会造成阅读代码时容易混淆，并且，在很多IE版本中，这会在封闭作用域中错误地创建两个符号（变量）。换句话说，在现代浏览器中，typeof plum将会是“undefined”，而在较早的IE版本中，它将会是“function”。因此，最好将名字设置为与你分配给的变量的名字一致，例如：

```
var sum = function sum(a,b){
    return a+b;
}
```

这对调试很有用，并且如果监测到变量的名字没有出现在函数体中，那么去掉函数的名称是一种很好的精简。

## 3.7 作用域

JavaScript中没有块作用域，只有函数作用域。

在一个函数中定义的任何变量，对于函数来说都是局部的，而且无法在函数之外看到它。全局变量是那些在任何函数之外定义的变量：

```

if(true){
    //即便在一个花括号块中，也是全局的
    var true_global = 1;
}
if(false){
    var false_global = 1;
}
var sum = function(){
    var local = 1;
    is_it_local = 1;
    return true_global + local + is_it_local;
};

true_global; // 1
false_global; // undefined
local; //ReferenceError: local 未定义
is_it_local; // ReferenceError: is_it_local 未定义

sum(); // 3
true_global; // 1
false_global; // undefined
local; // ReferenceError : local 未定义
is_it_local; // 1

```

前面的代码段中，需要注意以下几点：

- true\_global总是可用的。
- 即便是在一段不会执行的代码块中，false\_global也总是声明了的，尽管它没有初始化。使用它将会返回undefined（这是所有变量的默认值），并且，这不是一个错误。
- 在函数sum()之外，local是不用的，它是该函数的局部变量。视图在其局部作用域之外使用它，将会产生错误。
- 在使用sum()函数之前，is\_is\_local还没有声明，这将会导致错误。调用sum()会给is\_it\_local分配一个值，并且由于漏掉了var语句，会将这个值放到全局作用域中。这是应该避免的做法，因为我们要防止全局命名控件被污染。

注意：“没有块作用域”这种说法，也有下面这些例外：

- ECMAScript计划在下一个版本中纳入块作用域，但是要使用let而不是var来实现。
- catch语句块中的错误对象，其作用域就是其语句块：

```

try{
    throw new Error('Yuck');
}catch(err){
    err.message; // Yuck
}
typeof err; // "undefined"

```

## 3.8 提升

当程序执行进入一个新的作用域（例如，在一个新函数，全局作用域或eval()中），在函数中任何地方所定义的所有变量都移动或提升（hoisted）到作用域的顶层。这一点需要注意，因为它容易令人混淆：



```
var a = 1;
function hoistingTest(){
    console.log(a);
    var a = 2;
    console.log(a);
}
hoistingTest();// 控制台显示“undefined”，然后2
```

你可能期望第一个`console.log()`会给出全局变量`a`的值1。但是相反，我们得到了`undefined`，因为局部变量`a`的声明提升到了顶层。只有声明得到了提升，赋值为2的操作并没有提升到作用域的顶层。就像下面函数这样编写的：

```
function hoistingTest(){
    var a;
    console.log(a);
    a = 2;
    console.log(a);
}
```

为了防止混淆，很多开发者采用了这样的惯例：总是在顶部声明所有的变量，而不管在哪里使用它们。这可以看做是一种令人烦恼的手动提升（`manual hoisting`），因为毕竟这只是其表面的工作方式，并且提升是JavaScript解释器的工作。但是，这使得代码更加明白并且易于阅读。另外一种做法，也就是在需要变量的时候再定义它们，则会导致你每次阅读一个（可能很长的）函数时都需要在内心完成提升过程（`mental hoisting`）。

```
function mine(){
    // 在顶部一次声明所有变量
    var a,b = 2, c= 3;
    // 随后，在任意地方使用变量
    // ...
    a = b;
    // ...
    return c + a;
}
```

你可以把“单个的`var`放在函数的顶部”这一规则，看做类似于定义一个PHP类并且将所有的属性放在类的顶部，而不是让其散落在方法中。

## 提升函数

函数只是分配一个变量的对象，因此，也可以期望它们得到提升。然而，根据它们定义的方式，这会有所区别。

考虑如下的例子：

```
// 全局作用域
function declare(){}
var express = function(){};
(function(){
    // 布局作用域
    console.log(typeof declare);// "function"
    console.log(typeof express); // "undefined"
    function declare(){}
    var express = function(){};
    console.log(typeof declare); // "function"
    console.log(typeof express); // "function"
})();
```

局部函数declare()和express()都“覆盖”了具有相同名字的全局变量，因为它们都提升到了函数的顶层，尽管它们都是后定义的。但是，在函数表达式express()的例子中，只有var提升了，而函数声明（declare()）则是与其值（实现）一起提升的。记住了提升，函数就好像是像下面这样编写的一样：

```
// 全局作用域
function declare(){}
var express = function(){};
(function(){
    // 局部作用域
    function declare(){} // 声明并给定一个值
    var express = undefined; // 在这里声明

    console.log(typeof declare); // "function"
    console.log(typeof express); // "undefined"
    express = function(){}; // 在这里实现
    console.log(typeof declare); // "function"
    console.log(typeof express); // "function"
})();
```

## 3.9 闭包

闭包是一个函数及非局部变量的环境。在JavaScript中，每个函数都有一个闭包，因此，闭包没有什么特别的，它们也只是函数。但是，你应该理解函数的行为，因为做不到这一点将会导致很多调试难题。

## PHP中的闭包

从PHP 5.3开始，PHP中也有闭包了（<http://php.net/manual/en/functions.anonymous.php>）。

我们已经了解了，在JavaScript中，定义一个函数的常用语法是使用一个函数表达式：

```
// javascript
var sum = function(a,b){
    return a+b;
};
```

在PHP（PHP 5.3及其以后的版本）中，也有一种类似的语法：

```
// php
$sum = function($a,$b){
    return $a + $b;
};
$sum(9,11); // 20
```

和JavaScript不同，PHP函数不会自动访问全局环境或父级环境。如果想让来至父级环境的变量在闭包中可用，我们需要使用use来声明意图：

```
// php
$global_ten = 10;
function papa(){
    global $global_ten; //将全局变量局部化
    $hundred_more = 100;
    $sum = function($a,$b) use ($global_ten,$hundred_more){
        return $a + $b + $global_ten + $hundred_more;
    };
    return $sum(9,11);
}

echo papa(); // 130
```

在JavaScript中，没有use，且在内部函数中访问其环境变量的行为是自动的：

```
// javascript
var global_ten = 10;
function papa(){
    var hundred_more = 100;
    var sum = function(a, b){
        return a + b+ global_ten + hundred_more;
    };
    return sum(9,11);
}
papa(); // 130
```

papa()可以访问了定义该函数的环境中的变量。在这个例子中，这是全局环境并且是global\_ten变量。

sum()可以访问其父级函数papa()的环境，因此，它可以访问hundred\_more。此外，更进一步，sum()也能够看到其父级函数所能看到的所有变量，这意味着sum()也能够访问global\_ten。

这种父级函数和子级函数的关系形成了一条链，我们称之为作用域链（scope chain）。

---

**注意：**在PHP中，当你使用use捕获感兴趣的变量的时候，可以选择通过传引用或传值（就像示例中一样）的方式来传递它们。在JavaScript中，我们则总是在闭包作用域中引用变量。

---

## 3.10 作用域链

每次JavaScript解释器进入一个新的函数，它都会看一下附近那些局部变量是可用的。它收集这些变量，并且将他们放到一个特殊的 *variables* 对象中。

如果在第一个函数内部还定义了另一个函数，并且解释器进入到第二个函数体内，会为此内部函数创建另一个`variables`对象。这个内部函数还会得到一个特殊的`scope`属性，它指向了外部函数的`variables`对象。外部函数就是定义内部函数的地方，这就像是字典中的情况一样。这就是为什么有“词法”这个术语（“词法定义”和“字典定义”是同义）。它强调了函数的定义与其执行之间的区别。

在函数定义的过程中设置`scope`属性，而不是在执行期间设置。从程序员的视角来看，`variables`对象和`scope`属性都是可见的，但是，后续的代码段使用`_variables`和`_scope`以便说明（在ECMAScript规范中，用`[[Scope]]`表示`_scope`）。

如果你的函数没有定义于另一个函数之中，它就是在全局空间（全局作用域）中。全局空间中的这个位置，也有一个`variables`对象，包括了全局作用域中的所有变量。因此，全局函数的`scope`属性将保存对全局`variables`对象的一个引用。

来看一个全局函数：

```
var global_ten = 10;
// __variables 对象包含所有全局变量
// {
//   global_ten: 10,
//   papa:function(){...}
//   ... 所有其他的全局变量
// }

function papa(){
  var hundred_more = 100;
  // __variables 对象拥有{hundred_more:100}
  // __scope属性指向外部的词法环境（换句话说，全局的__variables对象）的variables对象
}
```

可以用链表将其关系表示如下：

```
papa.__scope->global__variables
```

我们再来看看前面的代码示例，这次带上行号：

```
1. var global_ten = 10;
2.
3. function papa(){
4.   var hundred_more = 100;
5.   var sum = function(a,b){
6.     return a + b + global_ten + hundred_more;
7.   };
8.   return sum(9,11);
9. }
10.
11. papa(); // 130
```

从第1行开始，解释器将其隐藏的`_variables`对象包含所有的变量（例如，`global_ten`和`papa`）。

由于其中之一是函数，它需要一个`_scope`属性。`papa.__scope`指向了全局的`_variables`。

当解释器需要查找一个变量的值的时候，查找过程真的很简单，因为只需要看一个位置：

```
[global __variables] global_ten,papa
```

接下来，解释器到了第11行。它进入papa()函数，并且在第4行找到了自己。它看看附近，并且创建了另一个\_\_variables对象，该对象包含了hundred\_more和sum。现在，你看到为什么要提升了：解释器必须找到散落在函数体中的所有变量，并且将它们保存在一个方便的位置。继续前进到第5行，发现sum也是一个函数，因此，它需要一个\_\_scope属性。现在，sum.\_\_scope应该指向papa() \_\_variables，因为这是sum()词法定义之处。作用域的链表（作用域链）变成如下所示：

```
sum.__scope->papa's __variables -> global __variables
```

现在，如果解释器需要查找一个变量，他会在papa()局部在\_\_variables中查找。如果没有找到该变量，解释器将会遵从这样的顺序：“好了，我在函数papa()中，并且这里没有这个名称的局部变量。让我来看看papa.\_\_scope指向哪里。哦，看，是另一个\_\_variables对象，其中包含更多的变量。我最好也到那里检查一下。”

```
[papa() local __variables] hundred_more, sum
[global __variables] global_ten, papa
```

接下来，在第8行，代码调用sum()，因此，解释器进入这个函数并且在第6行找到它。在这里，还是创建了另外一个\_\_variables对象，它包含a和b。这些变量没有使用var声明，而是以用作函数的形式参数的方式来声明的。

通常作为一个局部作用域，现在变成了一个闭包作用域，因为现在手边还有另一个直接的局部作用域：

```
[sum() local __variables] a, b
[papa() closure __variables] hundred_more, sum
[global __variables] global_ten, papa
```

当解释器必须执行第6行的算术运算的时候，它开始查找替代变量的值。你可以假设解释器是按照下面这样来思考的：

```
a // 在[local]作用域中查找到它了，我们干的不错。
b // 也在[local]作用域，真的太好了
global_ten // 哦，没在[local]中，我们深入下去找找看。
    // sum()的__scope是什么？哦，我明白了，是一个[closure]
    // 也没在[closure]中，继续进行。
    // [closure]是papa()的__variables对象，那么，papa.__scope是什么？
    // 它是[global __variables]
    // 我在[global]作用域中找到global_ten，太好了！
    // 这是最后一次机会了，否则我恐怕要抛出一个错误了。
hundred_more // 没有在[local]中，但是在[closure]中找到了，万事大吉！
```

我们现在看到了一个链（作用域链）是如何以链表的形式创建的，其中。每个函数都知道自己词法的\_\_scope。当解释器需要一个变量，并且在本地无法找到它，它就开始按照作用域的链来查找该变量。

### 3.10.1 WebKit控制台中的作用域链

当在WebKit控制台（Chrome或Safari）中调试的时候，可以看到作用域对象以及它们所包含的内容。我已经创建了一个可供联系的HTML文件。步骤如下：

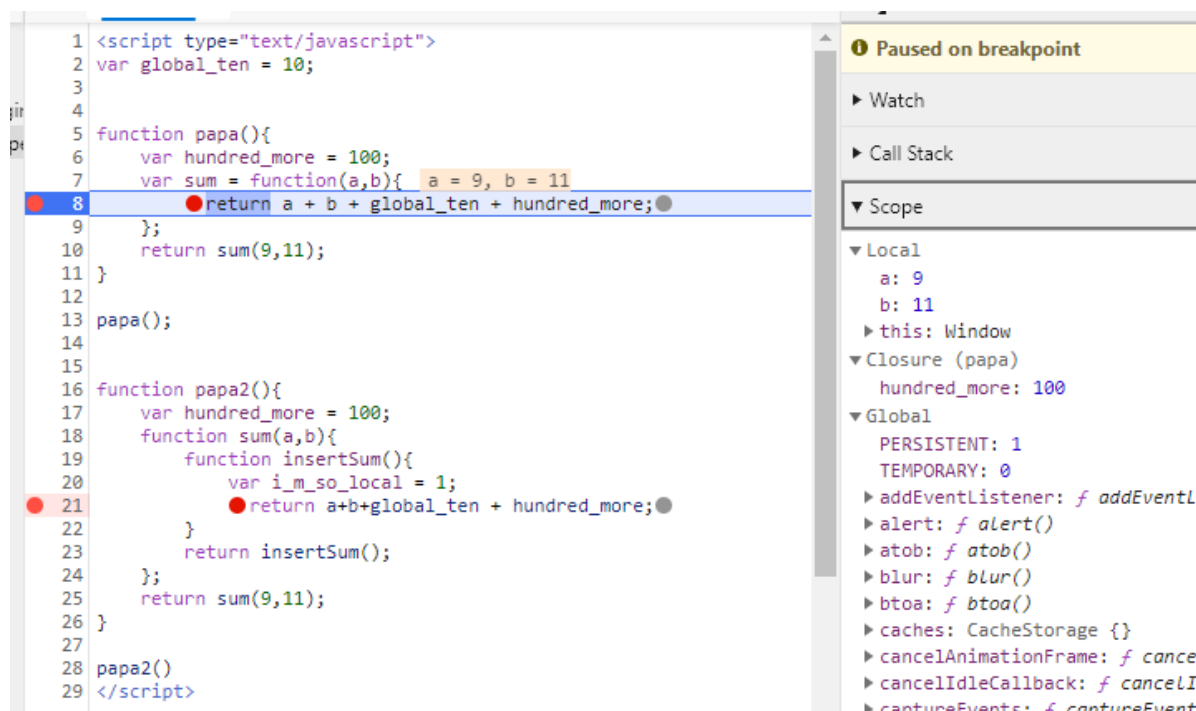
1. 在一个WebKit浏览器中载入该文件。
2. 在任何位置单击鼠标右键，并且选择“Inspect element”以打开Web Inspector。
3. 单击Script标签。
4. 通过在第8行和第21行上单击，插入断点。

现在，应该能够看到如图3-1所示的内容。

现在，刷新该页面，将会看到代码执行停在了第8行。看一下右边的菜单，在Scope Variables部分，将会清晰地看到包含Global，Closure和Local部分的链，以及其中可用的变量（见图3-2）。



```
1 <script type="text/javascript">
2 var global_ten = 10;
3
4
5 function papa(){
6     var hundred_more = 100;
7     var sum = function(a,b){
8         return a + b + global_ten + hundred_more;
9     };
10    return sum(9,11);
11 }
12
13 papa();
14
15
16 function papa2(){
17     var hundred_more = 100;
18     function sum(a,b){
19         function insertSum(){
20             var i_m_so_local = 1;
21             return a+b+global_ten + hundred_more;
22         }
23         return insertSum();
24     };
25     return sum(9,11);
26 }
27
28 papa2()
29 </script>
```



Paused on breakpoint

Watch

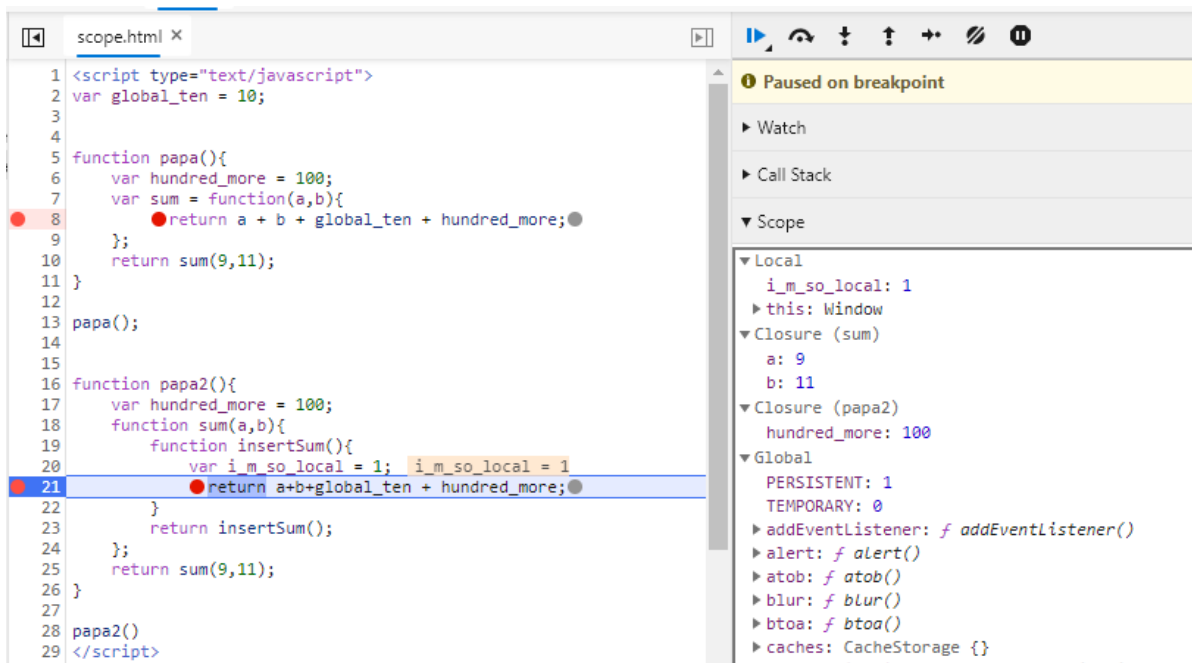
Call Stack

Scope

- Local
  - a: 9
  - b: 11
  - this: Window
- Closure (papa)
  - hundred\_more: 100
- Global
  - PERSISTENT: 1
  - TEMPORARY: 0
  - addEventListener: f addEventL
  - alert: f alert()
  - atob: f atob()
  - blur: f blur()
  - btoa: f btoa()
  - caches: CacheStorage {}
  - cancelAnimationFrame: f cance
  - cancelIdleCallback: f cancelI
  - captureEvents: f captureEvent

Global部分真的很大，但是，如果向下滚动的话，会看到global\_ten。你可能注意到，图3-2中的Closure中漏掉了sum。它应该在那里的，只不过在Chrome这个版本的Web Inspector屏幕截图中，它没有显示出来。如果在Safari中做同样的事情，将会看到不同的情况，其中包含了sum以及arguments对象。

现在，如果在调试器中单击Play按钮继续执行，将会到达第21行的第二个断点。在作用域列表中，有两个Closure部分，因为函数InnerSum()定义于sum()的内部，并且它在作用域链中创建了另一个链接（见图3-3）。



### 3.10.2 保持作用域

现在我们知道，局部函数可以访问它们的定义所在的环境的作用域。当执行移动出一个局部函数（例如，`innerSum()`或`sum()`）之外的時候，这个函数的局部作用域（其`__variables`对象）不再可用，并且可以销毁并销毁并进行垃圾回收，除非有人仍然需要它。这很有趣，很强大，但是同样也容易导致错误。

假设`sum()`不是一个局部函数，而是一个全局函数。函数仍然定义于一个局部环境中，但是，变量`sum`是在全局作用域中声明的：

```
var global_ten = 10;
var sum;

function papa(){
    var hundred_more = 100;
    sum = function(a,b){
        return a + b + global_ten + hundred_more;
    }
    return sum(9,11);
}

papa(); // 130
```

现在，在调用`papa()`之后，我们有了一个全新的`sum()`函数，可以从全局作用域来调用它。当调用`sum()`的时候，它可以访问诸如`global_ten`（不必使用PHP中`$GLOBAL`这样的特殊语法）这样的所有全局变量。但是，可能有些令人惊讶的是，它还可以访问`hundred_more`并且可以看到其值：

```
sum(9,11); // 130
```

这里发生了什么？在`papa()`返回之后，其局部作用域发生了什么？前面的例子中，当`sum`是局部的，那么该局部作用域会释放掉，因为没有人再需要它。但是，当`sum()`是全局并且定义于`papa()`之中，那么`sum.__scope`引用了`papa()`的局部作用域。因此，该作用域需要保留以备将来使用，不能销毁。`sum.__scope`保留了对该作用域的引用。

这一点对于理解和记住JavaScript闭包（函数）很重要：它们会保留对环境引用，这是它们定义于其中的一个环境，也叫做词法环境。

函数papa()也保留了它的引用环境，但它恰好是全局作用域，而这个作用域总是存在的。

### 3.10.3 保留引用而不是值

保留的只是对环境的引用，而不是具体的值，理解这一点很重要。我们来看一个例子：

```
var global_ten = 10;
var sum;
function papa(){
    var hundred_more = 100;
    sum = function(a,b){
        return a + b + global_ten + hundred_more;
    };
    hundred_more = 100000; // 这前面的例子不同
}

papa(); // 这次没有明确的返回值
sum(9,11); // 100030
```

在定义sum()的时候，hundred\_more的值为100，但稍后，它变成了100000。当papa()返回的时候，其局部作用域已经用新的值100000更新了，这是sum()能够访问的作用域。sum()将看到引用环境的最新状态。

注意，当你在PHP中使用use来“局部化”一个变量的时候，与此有所区别：

```
$hundred_more = 100;
$sum = function($a,$b) use($hundred_more){
    return $a+$b+$hundred_more;
};
$hundred_more = 100000;
echo $sum(0,0); // 100
```

在JavaScript中做同样的时候，将会返回100000，因为只是保留了对hundred\_more的引用，而不是保留其值。

```
var hundred_more = 100;
var sum = function(a,b){
    return a + b + hundred_more;
};
hundred_more = 100000;
sum(0,0); // 100000
```

### 3.10.4 循环中的闭包

我们来看另外一个例子，其中这一区别更为明显。我们重新定义一个函数数组：



```

var fns = [];
function definer(){
    for(var i = 0; i < 5; i++){
        fns.push(function(){
            return i;
        });
    }
}

definer();

```

这里，我们从0到4循环，定义了5个函数。可以像下面这样调用这些函数：

```

fns[0]() ;
fns[1]() ;
fns[2]() ;
fns[3]() ;

```

如果你没有阅读前面的章节，可能会狭隘地认为每次函数调用都会返回每个函数定义时的i值。但是，所以这个5个函数都是在同样的环境中定义的，并且能够访问同样的局部作用域。

当definer()返回的时候，i的值为5而不是4，因为在最后一次迭代的时候，有一个最后的i++。这是在保留的环境中存在的值，且所有的5个函数都可以看到它。因此，所有5个函数都返回5：

```

fns[0]() ; // 5
fns[1]() ; // 5
fns[2]() ; // 5
fns[3]() ; // 5

```

那么，如果要让所有的函数都返回一个递增的值，该怎么办呢？可以使用另一个闭包，将i的值“局部化”到一个local\_i中：

```

var fns = [];
function definer(){
    for(var i=0;i<5;i++){
        fns.push(function(local_i){
            return function(){
                return local_i;
            };
        })(i);
    }
}

definer()

```

这里引入了另一个作用域对象。在这个新的作用域中，有一个局部变量local\_i。它是使用i的当前值来初始化并保留的。实际上，现在5个函数中的每一个函数都有自己环境，而且与其他4个不相同。

现在，得到的结果更加符合你的期望：

```
fns[0]() // 0
fns[1]() // 1
fns[2]() // 2
fns[3]() // 3
fns[3]() // 4
```

再进行一个小小的改进：简化一下`definer()`中的循环。你可以引入一个工具函数`binder()`，它返回一个新的函数以及局部变量`i`的值：

```
var fns = [];
function binder(i){
    return function(){
        return i;
    }
}

function definer(){
    for(var i=0;i<5;i++){
        fns.push(binder(i));
    }
}

definer();

//测试
fns[0]() // 0
fns[1]() // 1
fns[2]() // 2
fns[3]() // 3
fns[3]() // 4
```

### 3.10.5 练习：onclick 循环

我们来看一个更加实际的示例，它展示了同样的思路，即引用保留的是环境变量的引用而不是值。

假设在带有递增的ID属性的一个HTML页面中，有3个按钮：

```
<button id='button-1'>One</button>
<button id='button-2'>Two</button>
<button id='button-3'>Three</button>
```

然后，在一个循环中分配click事件处理程序：

```
<script type='text/javascript'>
for(var i =1;i<=3;i++){
    document.getElementById('button-'+i).onclick = function(){
        alert('This is butotn:' +i);
    }
}
</script>
```

这段代码看上去似乎可以很好地工作，但是，你单击任何按钮总是会给出：

```
"This is button:4"
```

## 3.11 立即函数

在前面的`definer()`函数中，有一处较新且之前没有提及的语法，即最后的`()`。它告诉解释器，获取刚刚定义的函数并且立即执行它：

```
var immediate_one = function(){
    return 1;
}();

immediate_one; // 1
```

这里可以看到，调用了一个一次性函数，它运行并且返回一个值，然后，将该变量赋值给`immediate_one`。

如果这个示例是：

```
var immediate_one = function(){
    return 1;
};
```

那么，`immediate_one`应该是一个函数，而不是一次函数调用的返回值，并且它不会运行，只不过定义了该函数。

那么，如下的语法怎么样呢？

```
// 语法错误
function myfun(){
    return 1;
}();
```

这是不合法的语法，因为`myfun()`是使用函数声明语法定义，而不是函数表达式语法。二者之间的区别有时候不是很显著，除非你考虑到代码在程序中位于何处。

可以将该函数作为一个操作数传递给分组运算符，从而修正这一语法错误：

```
// 有效语法
(function myfun(){
    return 1;
})();
```

我们所需要的只是将函数包含到圆括号中，这有何区别？在这个例子中，函数不再位于主程序代码中，而是作为一个操作数传递给一个分组运算符`()`，它直接返回该函数。这将函数声明转换为函数表达式，并且后面的`()`执行该函数。函数声明是不能作为操作数的，因此，解释器消除了改语法的歧义，并且将这个函数当做一个函数表达式对待。

像下面这样放置分组运算圆括号，也是有效的：

```
// 也是有效的语法
(function myfun(){
    return 1;
})();
```

对于处理JavaScript中变量作用域的工作方式来说，立即函数是一种有用且常见的构造。立即函数有时候也叫自执行（selfexecuting）或自调用（self-invoking）函数，这容易让人混淆，因为它可能是指一个递归函数。它们也叫做立即调用函数表达式（immediately-invoked function expressions, IIFE），但是，这个术语有点太拗口了。即便在不必须的时候，为了可读性，我们也常常使用分组运算。在下面的示例中，hardtoread和easytoread的值都是1，但开始的(符号的出现，告诉读者这里还有些内容要继续（否则的话，很容易误认为hardtoread是一个函数而不是一个函数的返回值）：

```
var hardtoread = function(){
    return 1;
}();

var easytoread = (function(){
    return 1;
})();

hardtoread === easytoread; // true
```

### 3.11.1 初始化

立即函数的一种用法是，进行一次性初始化。通常，你想要进行一些初始化，而又不会遗留下全局变量。如果这一初始化足够复杂，以至于需要临时变量，你可以将所有的初始化代码包含在一个立即函数中，让该函数为你执行清理工作。

如下的代码将单击事件处理程序分配给按钮，这是一种很好的选择：

```
(function(){
    for(var i=1;i<=3;i++){
        document.getElementById('button'+i).onclick = function(){
            // ...
        }
    }
})();
```

如果没有立即函数，变量i将会保留在全局作用域中。正如你所看到的，该函数结束（close over）了在代码段中用到的变量，闭包（closure）的名字就是由此而来的。

### 3.11.2 私有性

立即函数也用来实现私有性。你可以让全局函数定义于其他函数的闭包之中，并且共享这个闭包的本地作用域。

如下是一个简单的迭代器实现：

```
var next, previous, rewind; // 全局变量
(function(){
    // 私有性数据
    var index = -1;
    var data = ['eeny', 'meeny', 'miny', 'moe'];
    var count = data.length;
    next = function(){
        if(index < count){
            index++;
        }
        return data[index];
    };

    previous = function(){
```

```

        if(index>=0){
            index--;
        }
    };

    rewind = function(){
        index = -1;
    };
}());

```

在这个示例中，`index`，`data`和`count`是私有的，因为这个迭代器的用户只需要向前走或向后走的方法，并且可能不会访问数据和指针，它们不会决心去把数据和指针搞乱。

先来测试下向前和向后的方法

```

next(); // "eeny"
next(); // "meeny"
previous(); // "eeny"

```

再来一些测试，看看回转和重新开始

```

var a;
rewind();
while(a = next()){
    // 用'a'做些事情
}

```

## 3.12 传递和返回函数

你可能看到过传递函数的例子，但是值得再次提及的是，你可以把函数作为参数传递给其他函数，并且也可以让函数返回其他的函数作为其他返回值。

此前，有一个函数返回了5个函数的一个完整数组。那有什么理由不能只返回一个呢？

函数是对象，并且和其他任何对象一样，它们也可以作为参数传递。如下的示例展示了这些特点：

```

function goForIt(a,b,what){
    return function(){
        return what(a,b);
    };
}

```

该函数接受名为`what`的参数，该参数期待是一个函数。我们也可以把分配给`what`的函数称为回调函数（callback function）。`goForIt()`函数返回一个不同的（匿名）函数，该函数返回执行回调的结果：

```

var sumOneAndTwo = goForIt(1,2,function(one,two){
    return one + two;
});

```

`sumOneAndTwo`的值是多少？它不是3，但它是一个函数，该函数返回值3：

```

sumOneAndTwo(); // 3

```

注意执行回调的那一行代码：

```
return what(a,b);
```

它也可以写成下面这样子：

```
what.call(null,a,b);
```

或者是：

```
what.apply(null,[a,b]);
```

稍后，我们将会看到关于null参数的更多的介绍。

你可以在使用时再来定义回调函数，如下所示：

```
var sumOneAndTwo = goForIt(1,2,function(one,two){  
    return one + two;  
});
```

这就像是在PHP中将一个闭包作为一个回调函数传递。

但是，你也可以重用不同的函数，而这个函数是你已经定义过的。

```
function sum(a,b){  
    return a + b;  
}  
  
var mysum = goForIt(1,99,sum);  
mysum();// 100  
  
// Also  
goForIt(1,99,sum)(); // 100
```

## 回调函数不是字符串

注意，不要将回调函数放在引号中，这意味着，不要将其当做一个字符串传递：

```
// 不要这么做  
goForIt(11,22,"sum")(); // TypeError:not a function
```

在前面的代码中，sum是一个函数对象而不是字符串，一i那次，将其放到引号中，意味着视图将一个字符串当做函数执行，而这无法工作。

---

**注意：**在PHP中，定义回调函数的传统式是，将函数名称用作传递给call\_user\_func()的字符串，但是，自从该语言采用了闭包，这种做法变得越来越少见了。

---

然而，有一些浏览器函数（例如，setTimeout()）接受回调函数，并且能够很好地处理人们传递一个字符串的情况。

假设有一个显示时间的函数：

```
function showTime(){
    alert(new Date());
}
```

可以规划让它1秒钟执行一次（1000毫秒）：

```
setTimeout(showTime,1000);
```

你也可能会看到，有人是像下面这样做到这一点的：

```
setTimeout("showTime()",1000);
```

这也有效，但是，感觉好像有错，因为它期望你传递一个函数值而不是一个字符串。它之所以有效，是因为setTimeout()会检测你传递的是什么，如果它是一个字符串，会计算它。一个实现示例大致如下所示：

```
function setTimeout(callback, when){
    // ... 当时间正确 ...
    if(typeof callback === "string"){
        (function(){
            eval(callback);
        })();
    }else{
        callback();
    }
}
```

总而言之，我们将函数值作为一个回调函数传递的时候，不用引号也不用()，因为我们并不想要在传递的时候就执行它；我们想要稍后再合适的时候再调用它。

最后一点要注意，你是否注意到，可以通过添加圆点括号，从而立即执行由另一个函数所返回的一个函数。

```
goForIt(11,22,sum)(); // 33
```

## 面向对象编程

本章介绍JavaScript面向对象特性，包括对象，构造器函数和原型。本章还将讨论代码重用和继承。

### 4.1 构造器和类

在PHP种，如果有一个Dog类，可以使用如下代码创建这个类的一个\$fid0实例：

```
// PHP
$fid0 = new Dog();
```

JavaScript有一个类似的语法：

```
// JavaScript
var fido = new Dog();
```

一个重要的区别是，Dog在JavaScript中不是一个类，因为该语言中没有类。Dog只是一个函数。但是，用来创建对象的函数称为构造函数（constructor functions）。

从语法上看，一般的函数和一个构造器函数之间并没有区别。区别在于其用途。因此，为了实现可读性，构造器函数名称的首字母通常大写。

当你使用new运算符调用一个函数的时候，它总是返回一个对象。在函数体内部，这个对象称为this。即便在函数中不做任何特殊的事情，也会有this对象。记住，如果不使用new调用的话，没有一条显式的return语句的每个函数都会返回undefined：

```
function Dog(){
    this.name = "Fido";

    this.sayName = function(){
        return "Woof! " + this.name;
    }
}

var fido = new Dog();
fido.sayName(); // "Woof! Fido"
```

注意：在JavaScript中，与在PHP中一样，当你没有给构造器函数传递参数的时候，圆括号是可选的，因此，var fido = new Dog;也是有效的。

如果在控制台输入fido，将会看到它有两个属性：name和sayName。有些控制台，还会给出一个名为\_\_proto\_\_的特殊属性，但你现在可以忽略它。

看一下sayName属性，它指向一个函数。在JavaScript中，函数是对象，因此，可以将它们分配给属性，在这种情况下，我们也可以称其为方法（method）。在JavaScript中，属性和方法之间真的没有区别。方法只不过是可调用的属性。

### 4.1.1 返回对象

当使用new调用任何函数的时候，会发生如下的事情：

1. 在后台自动创建一个“空”的对象，通过this引用该对象：

```
var this = {}; // 伪代码，如果你使用它，将产生一个语法错误
```

2. 程序员可以任意地给this添加属性：

```
this.name = "Fido";
```

3. 在函数的末尾，隐式地返回this：

```
return this; // 不是一个错误，到那时你不需要使用它
```

程序员可以通过返回一个不同的对象，从而改变步骤3：



```
function Dog(){
    var nothis = {
        noname: "Anonymous"
    };
    this.name = "Fido";
    return nothis;
}

var fido = new Dog();
fido.name; // undefined
fido.noname; // "Anonymous"
```

在这个示例中，我们添加给this的任何内容，在函数返回的时候都直接销毁了。你也可以删除它，在这种情况下，你真的不需要new所提供的魔力，可以像对待一般函数一样调用它并实现同样的效果：

```
function Dog(){
    return {
        noname: "Anonymous"
    };
}

var fido = Dog(); // 没有new，但是这一次需要()
fido.name; // undefined
fido.noname; // "Anonymous"
```

然而，请注意，返回任何this以外的内容，都将会导致 `instanceof` 运算符和 `constructor` 属性无法像预期的那样工作：

```
function Dog(){
    return {
        noname: "Anonymous"
    };
}

var fido = new Dog();
fido instanceof Dog; // false
fido.constructor === Object; // true
```

当使用new的时候，你可以返回一个定制的对象（而不是this），但是，它必须是一个对象。尝试返回一个非对象（一个标量），将会导致返回值被忽略，并且最终任然会得到this：

```
function Dog(){
    this.name = "Fido";
    return 1;
}

var fido = new Dog();
typeof fido; // "object"
fido.name; // "Fido"
```

### 4.1.2 关于this的更多内容

正如你所知道的，构造函数和常规函数之间没有区别，只不过二者的用途不同。因此，如果在一个非构造器函数中给this添加属性，会发生什么事情呢？换句话说，当你调用一个构造器函数（它添加给了this）并且忘记使用new调用它的时候，会发生什么事情？

```
function Dog(){
    this.thisIsTheName = "Fido";
    return 1;
}

var fido = new Dog();
var one = Dog();

// fido是一个常规对象
typeof fido; // "object"
fido.thisIsTheName; // "Fido"

// one是1，作为一个非构造器的返回值
typeof one; // "number"
one.thisIsTheName; // undefined

// 什么？
thisIsTheName; // "Fido"
```

这里令人惊讶之处在于，通过调用Dog()而不使用new，创建了全局变量thisIsTheName。这是因为，省略了new意味着这是一次常规函数调用，并且现在this指向了全局对象。添加给该全局对象的属性，可以用作全局变量（稍后会更多地介绍全局对象）。

这是危险的行为，因为你不要污染全局命名空间。这就是为什么这点很重要：在应该使用new的时候一定要使用new。命名构造器的时候，遵从首字母大写的惯例很重要，因为这样你可以给代码阅读者一个提示，以便他们知道函数的目标用途：

```
function Request(){} // 哈！这是一个构造器函数
function request(){} // 只是一个普通函数
```

---

注意：在ECMAScript 5的严格模式中，这一危险的行为得到了修正

---

### 4.1.3 增强构造器

如果你有点偏执狂，可以通过编程来确保即便调用者忘记了new，函数也会像构造器函数一样工作。可以使用instanceof运算符，它接受一个对象和一个构造器函数引用，并且返回true或false：

```
fido instanceof Dog; // true
```

如下是实现自行强化的构造器的一种方式：

```
function Dog(){
    // 检查调用者是否忘记了'new'
    if(!(this instanceof Dog)){
        return new Dog();
    }
    // 真正的工作开始了...
```

```

    this.thisIsTheName = "Fido";
    // 真正的工作结束

    // 暗含的工作在此结束...
    // 返回this
}

var newFido = new Dog();
var fido = Dog();

newfido.thisIsTheName; // "Fido"
fido.thisIsTheName; // "Fido"

```

this instanceof Dog这一行检查新创建的this对象是否是由Dog创建的。var fido = Dog();一行没有使用new，因此，this指向了全局对象。全局对象肯定不是由Dog创建的。毕竟，它甚至在Dog之前就存在了，因此，检查失败并且执行到return new Dog();这一行。

注意：你其实并不知道使用哪个构造函数创建了全局对象，因为，这是依赖于环境的一个内部类实现。

查询和咨询“谁创建了对象”的另一种方法是，使用多有对象都有的 `constructor` 属性。它也是一个可写的属性，因此，它并真的可靠，参见下面的示例：

```

function Dog(){}
var fido = new Dog();

fido.constructor === Dog; // true, 像预料的一样
fido.constructor = "I like potatoes";
fido.constructor === Dog; // false, ...等等，什么？
fido.constructor; // "I like potatoes"

```

## 4.2 原型

PHP中并没有原型（prototypes）的概念，但是，在 JavaScript 中，这却是一个重要的概念。让我们来看一个示例：

```

function Dog(name){ // 构造器函数
    this.name = name;
}
// 给'prototype'属性添加一个成员
Dog.prototype.sayName = function(){
    return this.name;
}

var fido = new Dog("Fluff");
fido.sayName(); // "Fluff"

```

这里发生了什么情况？

1. 由一个常用的函数Dog，显然它创建用来作为一个构造器函数，因为它一首字母D开头，并且在函数体中引用了this。
2. 在幕后，Dog()函数和任何其他函数一样，自动得到一个叫做prototype的属性（我们知道，函数是对象，因此，它们可以由属性）。总是会为每个函数，构造器，以及其他对象创建prototype属性。

3. 给 `prototype` 属性添加一个新的属性，该属性名为 `sayName`。这个属性恰好是一个函数，因此，我们可以说它是一个方法。
4. `sayName()` 访问了 `this`。
5. 使用 `new Dog()` 创建一个 `fido` 对象，这使得 `fido` 可以访问添加给 `prototype` 属性的所有属性。否则的话，如果我们没有使用 `new` 调用 `Dog()`，其中的 `prototype` 和所有内容都将忽略。
6. 即便 `sayName` 不是 `fido` 对象的属性，`fido.sayName()` 也工作得很好。

一个对象可以访问的属性和方法不属于它自己，而是属于创建该对象的构造器函数的 `prototype` 所引用的对象。

## 4.3 对象直接量

在本书中，我们已经见过几次对象直接的使用（例如，讨论在 `JavaScript` 表示 `PHP` 的关联数组的时候）。

对象直接量其实就是键—值对，它们用逗号隔开，并且用花括号包围起来。

```
var obj{  
  name : "Fluffy",  
  legs : 4,  
  tail : 1  
}
```

注意：在最后一个属性的后面保留结尾的逗号，这是不好的做法，因为某些环境（早期 Internet Explorer）无处理它，并且会导致严格错误。

可以一开始带有一些属性（或者根本就没有属性），并且随后在添加一些属性：

```
var obj = {};  
obj.name = 4;  
obj.tail = a;
```

### 4.3.1 访问属性

使用对象量表示法创建一个对象之后，可以使用点号表示法（dot notation）来访问这些属性。

```
var desc = typeof +"Has +" obj.legs + "legs and"+obj.tail,  
doc; // "Fluffy is 4 legs and 1 (tail);"
```

此外不太常见，因为将属性名当做字符串传递有点长且笨拙。然而，当实现穿不知道，它很有用。例如，当遍历素有的属性的时候：

```
var obj = {};  
for(var property in obj){  
  all.push(property+": "+obj[property]);  
}  
var desc = all.join(', ');  
desc; // "Name:Fluffy,legs:4, tail:1"
```

或者，另一个例子是，当在运行时计算属性名的时候：

```
var obj = {
  foo: "Foo",
  bar: "Bar",
  foobar: "Foo + Bar = BFF"
};
var fprop = "foo", bprop = "bar";
obj[fprop]; // "Foo"
obj[bprop]; // "Bar"
obj[fprop + bprop]; // "Foo + Bar = BFF"
```

当属性不是一个有效的标识符的时候，方括号表示法时必需的（对象直接量中包含属性名的引号也是必需的）：

```
var fido = {};
fido.number-of-paws = 4; // ReferenceError
fido['number-of-paws'] = 4; // This is OK
```

### 4.3.2 令人混淆的点号

在 JavaScript 中，点号用来访问属性，但是在 PHP 中，它们用来连接字符串。当你内心还处在 PHP 模式中，但是却要用 JavaScript 来编写代码的时候，你往往会按照习惯把点号的用法搞混淆了：

```
// javascript
var world = "world";
var result = "Hello ". world;
```

这很有趣，这在 JavaScript 中不是语法错误。result 包含了值 undefined。这是因为点号前后的空格时可选的，因此，如下这样也是有效：

```
"hello".world; // undefined
```

换句话说，你想要访问字符串对象 "hello" 的属性 world。字符串直接量在幕后转换为对象，就好像你执行了 new String("hello")（后面很快将更详细地介绍它）。由于这个对象没有这样一个属性，结果是 undefined。

### 4.3.3 对象直接量中的方法

我们可以使用对象直接量表示法给一个对象添加方法吗？绝对可以。方法只不过是属性，而该属性恰好指向函数对象：

```
var obj = {
  name: "Fluffy",
  legs: 4,
  tail: 1,
  getDescription: function() {
    return obj.name + " has " + obj.legs + " legs and " + obj.tail + "
tail(s)";
  }
};

obj.getDescription(); // "Fluffy has 4 legs and 1 tial(s)"
```

---

注意：在 getDescription() 中，我们可以使用 this 来替换 obj。

稍后，可以给一个已有的对象添加方法：

```
obj.getAllProps = function(){
    var all = [];
    for(var property in obj){
        if(typeof obj[property] !== "function"){
            all.push(property + ": " + obj[property]);
        }
    }
    return all.join(', ');
};

obj.getAllProps(); // "name: Fluffy, legs: 4, tail: 1"
```

是否注意到，使用如下的代码过滤掉了一些属性？

```
typeof obj[property] !== "function"
```

情况很可能是这样，你不想让函数出现在属性的列表中，但是，由于函数就像是所有其他属性一样，如果你没有过滤它们的话，它们将会出现。可以尝试删除掉这个过滤器，看看会发生什么情况。

**警告：** 在字符串连接环境中访问一个函数对象，将会把函数对象转换为一个字符串。这通过调用 `toString()` 方法而实现，继承至 `Object` 的所有对象都会响应该方法调用。函数对象通过返回函数的源码，从而实现 `toString()`，尽管这不是标准的实现方法，并且不同引擎的实现会在换行和空白上有所不同。

## 4.4 奇特的数组

总的来说，`obj`现在看上去就像是一个有趣的数组，或者说像是一个 `PHP` 的关联数组，其中，它的一些属性像函数一样起作用。实际上，早期的 `PHP` 版本根本没有对象的概念。当对象随后添加进来之后，人们称其为“奇特的数组”（fancy array）。如今，在 `PHP` 中，很容易在一个对象和一个关联数组之间来回切换：

```
// PHP
$num = array(
    'name'=>"Fluffy",
    'legs'=>4,
    'tail'=>1,
);
echo $num['name']; // "Fluffy"
var_dump($num->name); // NULL，这不是一个对象
```

这里 `$num` 是一个数组，因此，将 `name` 当做一个属性访问是行不通的。然而，我们可以把 `$num` 转换为一个对象：

```
// PHP
$num = (object)$num; // $num 现在是一个对象
echo $num['name']; // 重大的错误，这不再是一个数组
echo $num->name; // "Fluffy"
```

正如你所看到的，关联数组和对象是如此的接近，因此，JavaScript 决定合二为一，仅仅使用对象来表示两种概念。

要继续对 PHP 和 JavaScript 进行类比，你可以把 JavaScript 的对象直接想象成是幕后转换为对象的关联数组。就好像你在 PHP 中做了如下的事情一样：

```
// PHP
$mutt = (object)array(
    'name'=>"Fluffy"
);

echo $mutt->name; // "Fluffy"
```

这等同于JavaScript中的：

```
// javascript
var mutt = {
    name:"Fluffy"
};

mutt.name; // "Fluffy"
```

## 4.5 自身属性

在 JavaScript 中，对象所拥有的属性和继承自 prototype 对象的属性之间，有一个明确的区别。访问这二者的语法是相同的，但是，有时候你需要知道，一个属性是属于你的对象还是来自于其他某个地方。

自身属性是指使用对象直接量表示法或通过一次赋值添加给对象的那些属性：

```
var literal = {
    mine:"I pwn you"
};
literal.mine; // "i pwn you"
var assigned = {};
assigned.min = "I pwn you";
```

自身属性还包括添加给this以及由构造器函数返回的属性：

```
function Builder(what){
    this.mine = what;
}

var constructed = new Builder("pwned");
constructed.mine; // "pwned"
```

然而，请注意，这两个对象都访问了方法toString()，而它们都没有定义该方法：

```
literal.toString(); // "[object Object]"
constructed.toString(); // "[object Object]"
```

toString() 方法对于这两个对象来说，都不是自身方法。它是来自于一个 prototype 的方法。如果你想要分辨出自身属性和 prototype 属性之间的区别，你可以使用另外一个名为的 hasOwnProperty() 方法，它接受一个属性/方法的名称的字符串的形式：

```
literal.hasOwnProperty("mine"); // true
constructed.hasOwnProperty('mine'); // true
literal.hasOwnProperty('toString'); // false
constructed.hasOwnProperty('toString'); // false

literal.hasOwnProperty('hasOwnProperty'); // false
```

让我们做更多一些思考。`toString()`方法来自何处？我们如何才能够搞清楚谁是原型？

## 4.5.1 proto

对象拥有原型，但是，它们没有`prototype`属性，只有函数才会有这个属性。然而，很多环境为每个对象提供了一个特殊的`__proto__`属性。`__proto__`并不是随处都可用的，因为它只是对于调试和学习有用。`__proto__`是一个属性，它表明了对象以及创建该对象的构造器函数的`prototype`属性之间的秘密关系：

```
constructed.prototype; // 为定义，对象没有这个属性
constructed.constructor === Builder; // true, "who's your constructors?"

// 秘密联系以暴露了
constructed.constructor.prototype === constructed.__proto__; // true
```

链式的`__proto__`调用允许你刨根问底。这个根底就是内建的`Object()`构造器函数。`Object.prototype`是所有对象的父对象。所有对象都继承它。这就是定义`toString`之处：

```
Object.prototype.hasOwnProperty("toString"); // true
```

可以从`constructed`对象开始追踪`toString`：

```
constructed.__proto__.__proto__.hasOwnProperty("toString"); // true
```

那么`literal`对象呢？它的链条稍微短一些：

```
literal.__proto__.hasOwnProperty("toString"); // true
```

这是因为`literal`并不是一个定制的构造器创建的，这意味着，它是由`Object()`幕后创建的，而不是由继承自`Object()`的别的方法（例如，`Builder()`）创建的。

当你临时需要一个简单的对象的时候，可以使用`{}`，如下的语法也有效：

```
({}).__proto__.hasOwnProperty('toString'); // true
```

---

注意：前面提到了，像`var o = {}`这样的对象直接量，创建了“空”对象。“空”这个单词之所以放在引号中，是因为对象实际上并不是空的或空白的。每个对象，即便它没有任何自身属性，它也已经有一些属性和方法是可用的，这就是从它的原型链继承而来的那些属性和方法。

---



## 4.5.2 this或prototype

当使用构造器函数的收，你可以给this或构造器函数的 `prototype` 添加属性。你可能会问应该使用哪一个。

添加给 `prototype` 要更为高效一些，并且占用内存较少一些，因为这些属性和函数仅创建一次，并且可以供使用相同构造器所创建的所有对象来重用。添加给this的任何内容，都会在你每次实例化一个新对象的时候创建一次。

因此，你打算重用或者在示例之间共享的任何成员，都应该添加给 `prototype`；而在每个实例中都拥有不同的值的任何属性，都应该是添加给this的自身属性。更为常见的做法时，方法归于原型，而属性归于this，除非这些属性在实例中都是保持一致的。

谈到代码重用，这又引发下一个问题：可以重用那些使用了继承的代码吗？

## 4.6 继承

到目前为止，我们已经学习了：

- 如何用直接量表示法或构造器函数来创建对象。
- 原型是什么（每个函数的一个属性）。
- 自身属性和原型属性。
- 对象从其原型以及其原型的原型那里继承属性等。

现在，我们来讨论一下继承，因为你可能会感到奇怪，在没有类的语言中，继承如何工作呢？事实上，根据我们的目标和喜好，有另外的方法来实现继承。

### 4.6.1 通过原型继承

实现继承的默认方法是使用原型，我们使用一个父构造器创建一个对象，并且将其设置为子构造器的一个原型：

如下是要作为父构造器的一个构造器函数：

```
function NormalObject(){
  this.name = 'normal';
  this.getName = function(){
    return this.name;
  };
}
```

还有一个构造器：

```
function PreciousObject(){
  this.shiny = true;
  this.round = true;
}
```

继承的部分如下：

```
PreciousObject.prototype = new NormalObject();
```

好了，现在你可以创建Precious对象了，它具备Normal对象的所有功能：

```
var crystal_ball = new PreciousObject();
crystal_ball.name = "Ball, Crystal Ball.";
crystal_ball.round; // true
crystal_ball.getName(); // "Ball, Crystal Ball."
```

---

注意：本章中的示例受到了Jim Bumgardner博客帖子“Theory of the Precious Object”（<http://krazydad.com/blog/2008/07/31/theory-of-the-precious-object/>）的启发，而不是常见的“将汽车扩展为交通工具”。这是一篇很有趣的帖子，它阐明了对象的宝贵之处，就像是J. R.R Tolkien的科幻小说<<魔界>>或者像iPhone一样，具有某种公共的品质。

---

注意，你需要使用`new NormalObject()`来创建一个对象，并且将其赋值给`PreciousObject`函数的`prototype`属性，因为原型只是一个对象。如果想到了类，你知道，一个类总是继承自另一个来。并且，如果考虑到JavaScript的情况，你可能会期望一个构造器函数继承自另一个构造器函数。但是，情况并非如此。在JavaScript中，你继承了一个对象。

如果有几个继承了`NormalObject`对象的构造器函数，你可能每次会创建`new NormalObject()`，但这并非必须的。你可以创建一个`Normal`对象，并且将其作为子对象的原型重用。甚至开始的时候，并不需要整个`NormalObject`构造器函数。既然你继承了一个对象，所需要的只是一个对象，而不管它是如何创建的。

做同样的事情的另一种方式是，用对象直接量表示法来创建一个（单体）的`Normal`对象，并且将其用作其他对象的基类对象：

```
var normal = {
  name: 'normal',
  getName: function(){
    return this.name;
  }
};
```

然后，由`PreciousObject()`创建的对象可以像下面这样继承`normal`：

```
PreciousObject.prototype = normal;
```

### 4.6.2 通过复制属性来继承

由于继承都是关于代码重用的，实现它的另一种方式是，直接将一个对象的属性复制到另一个对象。假设有下面这些对象：

```
var shiny = {
  shiny: true,
  round: true
};

var normal = {
  name: 'name me',
  getName: function(){
    return this.name;
  }
};
```

`shiny`如何得到`normal`的属性？这里有一个简单的`extend()`函数，它遍历并复制属性：

```
function extend(parent, child){
    for(var i in parent){
        if(parent.hasOwnProperty(i)){
            child[i] = parent[i];
        }
    }
}

extend(normal, shiny); // 继承
shiny.getName(); // "name me"
```

复制属性看上去好像代价太高，并且可能会影响到性能，但是，相对于众多人物来说，它还是不错的选择。你还可以看到，这是实现混合继承和多继承的一种很容易的方式。

---

注意：使用这种模式，`instanceof`和`isPrototypeOf()`不会像预期的那样工作。

---

### 4.6.3 Beget对象

JavaScript的杰出人物，JSON的发明者Douglas Crockford，提供了另一种方式，使用一个仅仅能够设置其prototype的临时构造器函数，来实现了继承。

```
function begetObject(o){
    function F(){}
    f.prototype = o;
    return new F();
}
```

你创建了一个新的对象，但不是从头开始创建的，它是从另一个已有的对象继承了一些功能。例如，假设你有如下父对象：

```
var normal = {
    name: 'name me',
    getName: function(){
        return this.name;
    }
};
```

然后，可以有一个新的对象，它继承自该父对象：

```
var shiny = begetObject(normal);
```

可以给新的对象添加额外的功能：

```
shiny.round = true;
shiny.preciousness = true;
```

正如你所看到的，这里没有属性赋值，也没有看到任何构造器函数。一个新的对象继承自一个已有的对象。这实际上是社区支持的一种好思路，现在，它以`Object.create()`的形式称为ECMAScript 5的一部分，我们将会在第6章中看到这点。

作为闭包和优化的一个练习，你能否修改`begetObject()`以使得不会每次都要创建`F()`。

#### 4.6.4 “经典的”`extend()`

让我们用另外一种方式来实现继承，以此作为本章的结束。这种方法可能是最接近 **PHP** 的，因为这看上去就像是继承自另外一个构造器函数的一个构造器函数。因此，它看上去有点像是继承自另一个类的一个类。

主要内容如下：

```
function extend(child, parent){
    var F = function(){};
    F.prototype = parent.prototype;
    child.prototype = new F();
}
```

通过这种方法，我们传递了两个构造器函数给`extend()`。在`extend()`执行之后，使用第一个构造器（子构造器函数）创建的任何新的对象，都通过 `prototype` 属性，得到了第二个构造器（父构造器函数）的所有属性和方法。

---

注意： 这种方法通常称为“经典的”，因为它看上去和类的思想最为接近。

---

只需要给`extend`添加两个小内容：

- 让子构造器函数保留对父构造器函数的一个引用，以备不时之需。
- 重置子构造器函数的`constructor`属性，使其指向子构造器函数，以防在自省的时候需要（你将会在第5章看到关于这一属性的更多介绍）。

```
function extend(child, parent){
    var F = function(){};
    F.prototype = parent.prototype;
    child.prototype = new F();
    child.prototype.parent = parent;
    child.prototype.constructor = child;
}
```

在这个方法中，没有涉及`new Parent()`的实例。这意味着，在父构造器函数中添加给`this`的自身属性将不会被继承。只有添加给父对象的原型属性，才会被继承。并且这在很多情况下都是成立的。通常，我们会将想要重用的属性添加给原型。

考虑如下的设置：

```
function Parent(){
    this.name = "Papa";
}
Parent.prototype.family = "Bear";
function Child(){}
extend(Child, Parent);
```

`Name`属性是会被继承的，但是`family`属性会被继承。

```
new Child().name; // undefined
new Child().family; // "Bear"
```

并且，子对象可以访问父对象：

```
Child.prototype.parent === Parent; // true
```

### 4.6.5 借用方法

使用 `call()` 和 `apply()` 使得我们有机会重用代码，而根本不必处理继承。毕竟，继承本来就是用来帮助我们重用代码的。

如果你看到了想要一个方法，可以临时借用它，传递自己的对象在需要`this`的位置取代它：

```
var object = {
  name: 'normal',
  sayName: function(){
    return "My name is " + this.name;
  }
};

var precious = {
  shiny: true,
  name: "iPhone"
};
```

如果`precious`想要获得`object.sayName()`的益处并且不想扩展任何内容，它可以直接像下面这样使用：

```
object.sayName.call(precious); // "My name is iPhone"
```

如果将方法借用和经典的继承结合起来，那么，可以你既获得自身属性，也可以胡德原型属性：

```
function Parent(name){
  this.name = name;
}

Parent.prototype.family = "Bear";

function Child(){
  Child.prototype.parent.apply(this, arguments);
}

extend(Child, Parent);
```

所有`this`属性变成了子对象的自身属性。并且，通过 `arguments` 和 `apply()` 的魔力，如果愿意的话，我们也可以让参数传递给构造器函数：

```
var bear = new Child("Cub");
bear.name; // "Cub"
bear.family; // "Bear"
bear.hasOwnProperty("name"); // true
bear.hasOwnProperty("family"); // false
```

### 4.6.6 结论

正如你所看到的，很多种选项可以用来实现继承。你可以根据手边的任务，个人的偏好或团队的偏好来选择一种方法。你甚至可以构建自己的解决方法，或者使用你所选择的库自带的方案。然而，注意，深继承链在JavaScript项目种并不常见，因为这种语言允许你直接复制其他对象的属性和方法，或者“借用”这些属性和方法实现自己的任务。或者说，就像四人组在<<Design Pattern>>一书所说的：“对象

组合优先于类继承”。