# 函数进阶

函数对任何语言来说都是一个核心的概念。通过函数可以封装任意多条语句，而且可以在任何地方、任何时候调用执行。

## 预定义函数

JavaScript 引擎中有一组可供随时调用的内建函数。

*  parseInt()；
*  parseFloat()；
*  isNaN()；
*  isFinite()；
*  encodeURI()；
*  decodeURI()；
*  encodeURIComponent()；
*  decodeURIComponent()；
*  eval()。

**黑盒函数**

一般来说，当我们调用一个函数时，程序是不需要知道该函数的内部工作细节的。我们可以将其看做一个黑盒子，您只需要给它一些值（作为输入参数），就能获取它输出的返回结果。这种思维适用于任何函数—既包括 JavaScript 中的内建函数， 也包括由任何个人或集体所创建的函数。

## 匿名函数

正如您所知，我们可以这样定义一个函数：

|  |
| --- |
| var f = function(a){  return a;  }; |

通过这种方式定义的函数常被称为**匿名函数**（即没有名字的函数），特别是当它不被赋

值给变量单独使用的时候。在这种情况下，此类函数有两种优雅的用法：

*  可以将匿名函数作为参数传递给其他函数，这样，接收方函数就能利用我们所传递的函数来完成某些事情。
*  可以定义某个匿名函数来执行某些一次性任务。

## 回调函数

既然函数与任何可以被赋值给变量的数据是相同的，那么它当然可以像其他数据那样

被定义、删除、拷贝，以及当成参数传递给其他函数。

一个函数就可以接收另一个函数作为参数,这种函数就称之为高阶函数。

在下面的示例中，我们定义了一个函数，这个函数有两个函数类型的参数，然后它会

分别执行这两个参数所指向的函数，并返回它们的返回值之和。

|  |
| --- |
| function invokeAdd(a, b){  return a() + b();  } |

下面让我们来简单定义一下这两个参与加法运算的函数（使用函数声明模式），它们只

是单纯地返回一个固定值：

|  |
| --- |
| function one() {  return 1;  }  function two() {  return 2;  } |

现在，我们只需将这两个函数传递给目标函数 invokeAdd()，就可以得到执行结果了：

|  |
| --- |
| > invokeAdd(one, two); |

事实上，我们也可以直接用匿名函数（即函数表达式）来代替 one()和 two()，以作

为目标函数的参数，例如：

|  |
| --- |
| > invokeAdd(function () {return 1; }, function () {return 2; }); |

或许，我们可以换一种可读性更高的写法：

|  |
| --- |
| > invokeAdd(  function () { return 1; },  function () { return 2; }  ); |

当我们将函数 A 传递给函数 B，并由 B 来执行 A 时， A 就成了一个**回调函数**（callback

functions）。如果这时 A 还是一个无名函数，我们就称它为**匿名回调函数**。

那么，应该什么时候使用回调函数呢？

### 回调示例

在编程过程中，我们通常需要将一个函数的返回值传递给另一个函数。

在下面的例子中，我们定义了两个函数：第一个是 multiplyByTwo()，该函数会通过一个循环将其所接受的三个参数分别乘以 2，并以数组的形式返回结果；第二个函数 addOne()只接受一个值，然后将它加 1 并返回。

|  |
| --- |
| function multiplyByTwo(a, b, c) {  var i, ar = [];  for(i = 0; i < 3; i++) {  ar[i] = arguments[i] \* 2;  }  return ar;  }  function addOne(a) {  return a + 1;  } |

现在，我们来测试一下这两个函数，结果如下：

|  |
| --- |
| > multiplyByTwo(1, 2, 3);  [2, 4, 6]  > addOne(100);  101 |

接下来，假设我们有三个元素，我们要实现这三个元素在两个函数之间的传递。这需要

定义另一个数组，用于存储来自第一步的结果。我们先从 multiplyByTwo()的调用开始：

|  |
| --- |
| > var myarr = [];  > myarr = multiplyByTwo(10, 20, 30);  [20, 40, 60] |

然后，用循环遍历每个元素，并将它们分别传递给 addOne()。

|  |
| --- |
| > for (var i = 0; i < 3; i++) {  myarr[i] = addOne(myarr[i]);  }  > myarr;  [21, 41, 61] |

如您所见，这段代码可以工作，但是显然还有一定的改善空间。特别是这里使用了两个循环，如果数据量很大或循环操作很复杂的话，开销一定不小。因此，我们需要将它们合二为一。这就需要对 multiplyByTwo()函数做一些改动，使其接受一个回调函数，并在每次迭代操作中调用它。具体如下：

|  |
| --- |
| function multiplyByTwo(a, b, c, callback) {  var i, ar = [];  for(i = 0; i < 3; i++) {  ar[i] = callback(arguments[i] \* 2);  }  return ar;  } |

函数修改完成之后，之前的工作只需要一次函数调用就够了，我们只需像下面这样同

时将初始值和回调函数传递给它：

|  |
| --- |
| > myarr = multiplyByTwo(1, 2, 3, addOne);  [3, 5, 7] |

同样， 我们还可以用匿名函数来代替 addOne()， 这样做可以节省一个额外的全局变量。

|  |
| --- |
| > multiplyByTwo(1, 2, 3, function (a){  return a + 1;  });  [3, 5, 7] |

而且，使用匿名函数也更易于随时根据需求调整代码。例如：

|  |
| --- |
| > multiplyByTwo(1, 2, 3, function(a){  return a + 2;  });  [4, 6, 8] |

## 即时函数

目前我们已经讨论了匿名函数在回调方面的应用。接下来，我们来看匿名函数的另一

个应用示例—这种函数可以在定义后立即调用。比如：

|  |
| --- |
| (  function(){  alert('boo');  }  )(); |

我们只需将匿名函数的定义放进一对括号中，然后外面再紧跟一对括号即可。其中，第二对括号起到的是“立即调用”的作用，同时它也是我们向匿名函数传递参数的地方。

|  |
| --- |
| (  function(name){  alert('Hello ' + name + '!');  }  )('dude'); |

另外，您也可以将第一对括号闭合于第二对括号之后。这两种做法都有效。

|  |
| --- |
| (function () {  // …  } () ); |

**使用即时（自调）匿名函数的好处是不会产生任何全局变量**。**当然，缺点在于这样的**

**函数是无法重复执行的**（除非您将它放在某个循环或其他函数中）。这也使得即时函数非常

适合于执行一些一次性的或初始化的任务。

## 内部（私有）函数

函数与其他类型的值本质上是一样的，因此，没有什么理由可以阻止我们在一个函数内部定义另一个函数。

|  |
| --- |
| function outer(param) {  function inner(theinput) {  return theinput \* 2;  }  return 'The result is ' + inner(param);  } |

我们也可以改用函数标识记法来写这段代码：

|  |
| --- |
| var outer = function (param) {  var inner = function (theinput) {  return theinput \* 2;  };  return 'The result is ' + inner(param);  }; |

当我们调用全局函数 outer()时，本地函数 inner()也会在其内部被调用。由于

inner()是本地函数，它在 outer()以外的地方是不可见的，所以我们也能将它称为私

有函数。

|  |
| --- |
| > outer(2);  "The result is 4"  > outer(8);  "The result is 16"  > inner(2);  ReferenceError: inner is not defined |

使用私有函数的好处主要有以下几点：

*  有助于我们确保全局名字空间的纯净性（这意味着命名冲突的机会很小）。
*  确保私有性—这使我们可以选择只将一些必要的函数暴露给“外部世界”，而保

留属于自己的函数，使它们不为该应用程序的其他部分所用。

## 函数内部属性

在函数内部，有两个特殊的对象： arguments 和 this。其中， arguments 是一个类数组对象，包含着传入函数中的所有参数( **且和参数是一一映射关系，改变则另一方也改变**）。虽然 arguments 的主要用途是保存函数参数，

但这个对象还有一个名叫 callee 的属性，该属性是一个指针，指向拥有这个 arguments 对象的函数.

|  |
| --- |
| function factorial(num){  if (num <=1) {  return 1;  } else {  return num \* factorial(num-1)  } } |

定义阶乘函数也可以用递归算法；如上面的代码所示，在函数有名字，而且名字以后也不会变的情况下，这样定义没有问题。但问题是这个函数的执行与函数名 factorial 紧紧耦合在了一起。为了消除这种紧密耦合的现象，可以像下面这样使用 arguments.callee。

|  |
| --- |
| function factorial(num){  if (num <=1) {  return 1;  } else {  return num \* arguments.callee(num-1)  } } |

## 函数属性和方法

前面曾经提到过， ECMAScript 中的函数是对象，因此函数也有属性和方法。每个函数都包含两个属性： length 和 prototype。其中， length 属性表示函数希望接收的命名参数的个数，如下面的例子所示。

|  |
| --- |
| function sayName(name){  alert(name);  }  function sum(num1, num2){  return num1 + num2;  }  function sayHi(){  alert("hi");  }  alert(sayName.length); //1  alert(sum.length); //2  alert(sayHi.length); //0 |

每个函数都包含两个特殊的方法： apply()和 call()。这两个方法的用途都是**在特定的作  
用域中调用函数**，实际上等于设置函数体内 this 对象的值。首先， apply()方法接收两个参数：一个是在其中运行函数的作用域，另一个是参数数组。其中，第二个参数可以是 Array 的实例，也可以是arguments 对象。例如：

|  |
| --- |
| function sum(num1, num2){  return num1 + num2; } function callSum1(num1, num2){  return sum.apply(this, arguments); // 传入 arguments 对象 } function callSum2(num1, num2){  return sum.apply(this, [num1, num2]); // 传入数组 } alert(callSum1(10,10)); //20 alert(callSum2(10,10)); //20 |

在上面这个例子中， callSum1()在执行 sum()函数时传入了 this 作为 this 值（因为是在全局作用域中调用的，所以传入的就是 window 对象）和 arguments 对象。

call()方法与 apply()方法的作用相同，它们的区别仅在于接收参数的方式不同。对于 call()方法而言，第一个参数是 this 值没有变化，变化的是其余参数都直接传递给函数。换句话说，在使用call()方法时，传递给函数的参数必须逐个列举出来，如下面的例子所示。

|  |
| --- |
| function sum(num1, num2){  return num1 + num2;  }  function callSum(num1, num2){  return sum.call(this, num1, num2);  }  alert(callSum(10,10)); //20 |

结果与使用 apply()没有什么不同。至于是使用 apply()还是 call()，完全取决于你采取哪种给函数传递参数的方式最方便。

## 返回函数的函数

正如之前所提到的，函数始终都会有一个返回值，即便不是显式返回，它也会隐式返

回一个 undefined。既然函数能返回一个唯一值，那么这个值就也有可能是另一个函数。

例如：

|  |
| --- |
| function a() {  alert('A!');  return function(){  alert('B!');  };  } |

在这个例子中，函数 a()会在执行它的工作（弹出'A!'）之后返回另一个函数。而所返

回的函数又会去执行另外一些事情（弹出'B!'）。我们只需将该返回值赋值给某个变量，然后就可以像使用一般函数那样调用它了。

|  |
| --- |
| > var newFunc = a();  > newFunc(); |

上面第一行执行的是 alert('A!')，第二行才是 alert ('B!')。

如果您想让返回的函数立即执行，也可以不用将它赋值给变量，直接在该调用后面再

加一对括号即可，效果是一样的：

|  |
| --- |
| > a()(); |