# 函数

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。

## 定义函数

ECMAScript 中的函数使用 function 关键字来声明，后跟一组参数以及函数体。  
函数的基本语法如下所示：

|  |
| --- |
| function functionName(arg0, arg1,...,argN) {  statements } |

一般来说，函数声明通常由以下几部分组成。

*  关键词 function。
*  函数名称，即这里的 sum。
*  函数所需的参数，即这里的 a、 b。一个函数通常都具有 0 个或多个参数。参数 之间用逗号分隔。
*  函数所要执行的代码块，我们称之为函数体。
*  return 子句。函数通常都会有返回值，如果某个函数没有显式的返回值，我们

就会默认它的返回值为 undefined。

以下是一个函数示例：

|  |
| --- |
| function sayHi(name, message) {  alert("Hello " + name + "," + message); } |

## 函数调用

如果我们需要使用一个函数，就必须要去调用它。调用的方式很简单，只需在函数名

后面加一对用以传递参数的括号即可。另外，对于“调用（to call）”这种操作，我们有时也可以将其称之为“请求（to invoke）”某个函数。

调用 sayHi()函数的代码如下所示：

|  |
| --- |
| sayHi("Nicholas", "how are you today?"); |

这个函数的输出结果是"Hello Nicholas,how are you today?"。函数中定义中的命名参数 name和 message 被用作了字符串拼接的两个操作数，而结果最终通过警告框显示了出来。

## 函数参数

在定义一个函数的同时，我们往往会设置该函数所需的调用参数。当然，您也可以不

给它设定参数，但如果您设定了，而又在调用时忘了传递相关的参数值， JavaScript 引擎就会自动将其设定为 undefined。

从技术角度来说，参数又可分为形参（形式参数）与实参（实际参数）两种，。形参是指定义函数时所用的那些参数，而实参则指的是在调用函数时所传递的那些参数。

对于那些已经传递进来的参数， JavaScript 是来者不拒的。但是，即便我们向函数传递再多的参数，多余的那部分也只会被默默地忽略掉。

实际上，我们还可以创建一些在参数数量方面更为灵活的函数。这得益于函数内部的

arguments 变量，该变量为内建变量，每个函数中都能调用。它能返回函数所接收的所有参数。 arguments.length 返回的是函数被调用时所接收的参数数量。arguments 实际上不是一个数组（虽然它有很多数组的特性），而是一个类似数组的对象。

## 函数返回值

ECMAScript 中的函数在定义时不必指定是否返回值。实际上，任何函数在任何时候都可以通过**return** 语句后跟要返回的值来实现返回值。请看下面的例子：

|  |
| --- |
| function sum(num1, num2) {  return num1 + num2; } |

这个 sum()函数的作用是把两个值加起来返回一个结果。我们注意到，除了 return 语句之外，没有任何声明表示该函数会返回一个值。调用这个函数的示例代码如下：

|  |
| --- |
| var result = sum(5, 10); |

这个函数会在执行完 return 语句之后停止并立即退出。因此，位于 return 语句之后的任何代码都永远不会执行。例如：

|  |
| --- |
| function sum(num1, num2) {  return num1 + num2;  alert("Hello world"); // 永远不会执行 } |

在这个例子中，由于调用 alert()函数的语句位于 return 语句之后，因此永远不会显示警告框。当然，一个函数中也可以包含多个 return 语句，如下面这个例子中所示：

|  |
| --- |
| function diff(num1, num2) {  if (num1 < num2) {  return num2 - num1;  } else {  return num1 - num2;  } } |

这个例子中定义的 diff()函数用于计算两个数值的差。如果第一个数比第二个小，则用第二个数减第一个数；否则，用第一个数减第二个数。代码中的两个分支都具有自己的 return 语句，分别用于执行正确的计算。  
另外，return 语句也可以不带有任何返回值。在这种情况下，函数在停止执行后将返回 undefined值。这种用法一般用在需要提前停止函数执行而又不需要返回值的情况下。比如在下面这个例子中，就不会显示警告框：

|  |
| --- |
| function sayHi(name, message) {  return;  alert("Hello " + name + "," + message); //永远不会调用 } |

## 没有重载

ECMAScript 函数不能像传统意义上那样实现重载。而在其他语言（如 Java）中，可以为一个函数编写两个定义，只要这两个定义的签名（接受的参数的类型和数量）不同即可。如前所述， ECMAScirpt函数没有签名，因为其参数是由包含零或多个值的数组来表示的。而没有函数签名，真正的重载是不可能做到的。  
如果在 ECMAScript 中定义了两个名字相同的函数，则该名字只属于后定义的函数。请看下面的例子：

|  |
| --- |
| function addSomeNumber(num){  return num + 100; } function addSomeNumber(num) {  return num + 200; } var result = addSomeNumber(100); //300 |

在此，函数 addSomeNumber()被定义了两次。第一个版本给参数加 100，而第二个版本给参数加200。由于后定义的函数覆盖了先定义的函数，因此当在最后一行代码中调用这个函数时，返回的结果就是 300。

## 变量的作用域

这是一个至关重要的问题。特别是当我们从别的语言转向 JavaScript 时，必须要明白一点，即**在 JavaScript 中，变量的定义并不是以代码块作为作用域的，而是以函数作为作用域。**也就是说，**如果变量是在某个函数中定义的，那么它在函数以外的地方是不可见的。**而如果**该变量是定义在 if 或者 for 这样的代码块中的，它在代码块之外是可见的。**另外，在 JavaScript 中，术语**“全局变量”指的是定义在所有函数之外的变量**（也就是定义在全局代码中的变量），与之相对的是**“局部变量”，所指的则是在某个函数中定义的变量**。其中，**函数内的代码可以像访问自己的局部变量那样访问全局变量**，反之则不行。

下面来看一个具体示例，请注意两点：

 函数 f()可以访问变量 global。

 在函数 f()以外，变量 local 是不存在的。

|  |
| --- |
| var global = 1;  function f() {  var local = 2;  global++;  return global;  } |

这里还有一点很重要，如果我们声明一个变量时没有使用 var 语句，该变量就会被默

认为全局变量。

**最佳实践**

 尽量将全局变量的数量降到最低，以避免命名冲突。因为如果有两个人在同一段脚本的不同函数中使用了相同的全局变量名，就很容易导致不可预测的结果和难以察觉的 bug。

 最好总是使用 var 语句来声明变量。

## 变量提升

下面， 我们再来看一个例子，它显示了关于局部和全局作用域的另一个重要问题。

|  |
| --- |
| var a = 123;  function f() {  alert(a);  var a = 1;  alert(a);  }  f(); |

第一个 alert()实际上显示的是undefined，这是因为**函数域始终优先于全局域**，所以局部变量 a 会覆盖掉所有与它同名的全局变量，尽管在 alert()第一次被调用时， a 还没有被正式定义（即该值为undefined），但该变量本身已经存在于本地空间了。这种特殊的现象叫做提升（hoisting）。

也就是说，当 JavaScript 执行过程进入新的函数时，这个函数内被声明的所有变量都

会被移动（或者说提升）到函数最开始的地方。这个概念很重要，必须牢记。另外需要注

意的是，被提升的只有变量的声明，这意味着，只有函数体内声明的这些变量在该函数执

行开始时就存在，而与之相关的赋值操作并不会被提升，它还在其原来的位置上。

## 函数也是数据

在 JavaScript 中，函数实际上也是一种数据。这概念对于我们日后的学习至关重要。

也就是说，我们可以把一个函数赋值给一个变量。

|  |
| --- |
| var f = function() {  return 1;  }; |

上面这种定义方式通常被叫做函数标识记法（function literal notation）。

function(){ return 1;}是一个函数表达式。函数表达式可以被命名，称为命名

函数表达式（named function expression， NFE）。所以以下这种情况也是合法的，虽然我们不常常用到（在这里， myFunc 是函数的名字，而不是变量：

|  |
| --- |
| var f = function myFunc() {  return 1;  }; |

这样看起来，似乎命名函数表达式与函数声明没有什么区别。但它们其实是不同的。

两者的差别表现于它们所在的上下文。

我们还可以像变量那样使用函数—也就是说，我们可以将它拷贝给不同的变量。

|  |
| --- |
| > var sum = function(a, b) {  return a + b;  };  > var add = sum;  > typeof add;  "function"  > add(1, 2);  3 |

由于函数也是赋值给变量的一种数据，所以函数的命名规则与一般变量相同—即函

数名不能以数字开头，并且可以由任意的字母、数字、下划线和美元符号组合而成。