函数

函数是搭建JavaScript语言大厦的基本构件之一。一个函数本身就是一段JavaScript程序——包含用于执行某一任务或计算的一系列语句。要使用某一个函数，你必需在想要调用这个函数的执行域的某处定义它。

一个**函数的定义**（也称为**函数的声明**）由一系列的函数关键词组成，依次为：

- 函数的名称。
- 包围在括号()中，并由逗号区隔的一个函数引数（译注：实际参数）列表。
- 包围在花括号{}中，用于定义函数功能的一些JavaScript语句。

例如，以下的代码定义了一个名为square的简单函数（译注：其实并非看上去那么“简单”）：

```
function square(number) {
  return number * number;
}
```

函数square使用了一个引数，叫作number。这个函数只有一个语句，它说明该函数会将函数的引数（即number）自乘后返回。函数的return语句确定了函数的返回值。

```
return number * number;
```

原始参数（比如一个具体的数字）被作为值传递给函数；值被传递给函数，但是如果被调用函数改变了这个参数的值，这样的改变不会影响到全局或调用的函数。

如果你传递一个对象（即一个非实际值，例如矩阵或用户自定义的其它对象）作为参数，而函数改变了这个对象的属性，这样的改变对函数外部是可见的，如下面的例子所示：

```
function myFunc(theObject) {
  theObject.make = "Toyota";
}

var mycar = {make: "Honda", model: "Accord", year: 1998},
    x,
    y;

x = mycar.make;
myFunc(mycar);
y = mycar.make;
```

请注意，重新给参数分配一个对象，并不会对函数的外部有任何影响，因为这样只是改变了参数的值，

而不是改变了对象的一个属性值:

```
function myFunc(theObject) {
  theObject = {make: "Ford", model: "Focus", year: 2006};
}

var mycar = {make: "Honda", model: "Accord", year: 1998},
    x,
    y;

x = mycar.make;
myFunc(mycar);
y = mycar.make;
```

在第一段例子中，对象mycar被传递给了函数myFunc，进而函数改变了它。 第二段例子里，函数并没有改变传递来的对象；相反，它生成了一个新的恰好和传递的全局对象同名的局部变量，因此对传递来的全局对象没有任何影响。

在JavaScript语言中，一个函数可以在满足一定条件后才被定义。例如，下面的函数定义只有在num为0时，才定义函数myFunc:

```
if (num == 0){
  function myFunc(theObject) {
    theObject.make = "Toyota"
  }
}
```

如果num不为0，函数不会被定义，因而任何执行它的尝试将会失败。

此处要注意ECMAScript标准并不允许函数像上例那样出现在上下文里，仅仅允许直接在其他函数的内部或者在程序的顶级，因此上例在ECMAScript里是非法的。

警告：JavaScript语言的不同实现处理类似非标准构造的方式也是不同的，因而最好的方式是在写可迁移代码的时候就避免它。否则，你的代码可能会在一些浏览器下工作正常而对另一些出错（译者：为什么这里要举一个错误的例子再警告不要用？）。

除了以上讨论的定义函数的方法之外，你仍然可以用函数生成器从字符串实时生成函数，就像`eval()`。

方法是函数本身作为对象的属性。请参考有关对象和方法的用对象编程一文。

当然上述函数定义都用的是语法语句，函数也同样可以由**函数表达式**产生。这样的函数可以是匿名的;

它不必有名称。例如，上面提到的函数square也可这样来定义：

```
var square = function(number) {return number * number};
```

必要时，函数名称可与函数表达式同时存在，并且可以用于在函数内部代指其本身，或者在调试器堆栈跟踪中鉴别该函数：

```
var factorial = function fac(n) {return n<2 ? 1 : n*fac(n-1)};


print(factorial(3));
```

函数表达式在将函数作为一个引数传递给其它函数时十分方便。下面的例子演示了一个叫map的函数如何被定义，而后调用一个匿名函数作为其第一个参数：

```
function map(f,a) {
  var result = [],      i;
  for (i = 0; i != a.length; i++)
    result[i] = f(a[i]);
  return result;
}
```

下面的代码：

```
map(function(x) {return x * x * x}, [0, 1, 2, 5, 10]);
```

返回 [0, 1, 8, 125, 1000].

调用函数

定义一个函数并不会自动的执行它。定义了函数仅仅是赋予函数以名称并明确函数被调用时该做些什么。调用函数才会以给定的参数真正执行这些动作。例如，一旦你定义了函数square，你可以如下这样调用它：

```
square(5);
```

上述语句以引数（译注：即实际参数）5来调用函数。函数执行完它的语句会返回值25。

函数一定要处于调用它们的域中，但是函数的声明可以在它们的调用语句之后，如下例：

```
print(square(5));
/* ... */
function square(n){return n*n}
```

函数的域是指函数被声明时的所在函数，或者函数在顶级被声明时指整个程序。注意只有使用如上的语

法形式（即如function funcName(){}）才可以。而形如下面的代码是无效的。

```
print(square(5));
square = function (n) {
  return n * n;
}
```

函数的引数并不局限于字符或数字。你也可以将整个对象传递给函数。函数show_props（其定义参见用对象编程有关章节）就是一个将对象作为引数的例子。

函数可以被递归；就是说函数可以调用其本身。例如，下面这个函数计算递归的阶乘值：

```
function factorial(n){
  if ((n == 0) || (n == 1))
    return 1;
  else
    return (n * factorial(n - 1));
}
```

你可以在其后计算下面5个阶乘值：

```
var a, b, c, d, e;
a = factorial(1);b = factorial(2);c = factorial(3);d = factorial(4);e =
factorial(5);
```

还有其它的方式来调用函数。常见的一些情形是需要函数被动态的调用，或者函数的引数数量是变化的，或者在调用函数的上下文中，函数的引数需要被实时设置为一个特定的对象。这将把函数本身转变为对象，且这些对象在转换中有不同的方法（参考函数对象一文）。作为此中情形之一，apply()方法可以被用于这种目的。（译者：此小节不明硬译！？）

函数的域

在函数内定义的变量不能从函数之外的任何地方取得，因为变量仅仅在该函数的域的内部有定义。相反对应的，一个函数可以取得在它的域中定义的任何变量和子函数。换言之，定义在全局域中的函数可以取得所有定义在全局域中的变量。而定义在一个函数内部的子函数可以取得定义在其父函数内的，或已经由其父函数取得的任何变量。

```
var num1 = 20,
    num2 = 3,
    name = "Chamahk";
function multiply() {
  return num1 * num2;
```

```
}

multiply();function getScore () {
  var num1 = 2,
      num2 = 3;

  function add() {
    return name + " scored " + (num1 + num2);
  }

  return add();
}

getScore();
```

闭包

（译注：请参考JavaScript指南的闭包一文）

闭包是JavaScript语言最有力的特性之一。JavaScript允许函数的嵌套，而且内嵌函数将能够完全取得外部函数内定义的变量和函数（以及如前所述，外部函数所能取得的其它变量和函数）。同样相反的，外部函数不能够取得内嵌函数定义的变量和函数。这提供了一种针对内嵌函数变量的安全机制。进而，因为内嵌函数可以取得外部函数的域，如果内部函数被设置成不依托外部函数的声明而存在的话，外部函数所定义的变量和函数可能会比外部函数本身生存的更长。当内嵌函数在某种情形下可以被外部函数的外部域获得时，一个闭包就产生了。

（译者：此处不甚明！？很多计算机术语翻译的太失败了，如此处的闭包-当然也有很多好的译文-字面和实质有如云泥，或者当代的汉语教育已死，我们快要不会说“好的中文”了；希望是iwo自己的局限所致）

```
var pet = function(name) {
        return name;

      return getName;
    myPet = pet("Vivie");

myPet();
```

```
    var getName = function() {
    }

},
```

实际上可能会比上面的代码复杂的多。在下面这种情形中，一个包含多种操作内嵌函数变量方法的对象会被返回。（译者：此处可能理解有误？！）

```
var createPet = function(name) {
  var sex;

  return {
    setName: function(newName) {
      name = newName;
    },

    getName: function() {
      return name;
    },

    getSex: function() {
      return sex;
    },

    setSex: function(newSex) {
      if(typeof newSex == "string" && (newSex.toLowerCase() == "male" ||
newSex.toLowerCase() == "female")) {
        sex = newSex;
      }
    }
  }
}

var pet = createPet("Vivie");
pet.getName();
pet.setName("Oliver");
pet.setSex("male");
pet.getSex();                    pet.getName();
```

在上面的代码中，外部函数的name变量对内嵌函数来说是可取得的，而除了通过内嵌函数本身，没有其它任何方法可以取得内嵌的变量。内嵌函数的内嵌变量就像内嵌函数的保险柜。它们会为内嵌函数保留“稳定”——而又安全——的数据参与运行。而这些内嵌函数甚至不会被分配给一个变量，或者不必一定要有名字。

```
var getCode = (function(){
  var secureCode = "0]Eal(eh&2";
  return function () {
```

```
    return secureCode;
  };
})();

getCode();
```

尽管有上述优点，使用闭包时仍然要小心避免一些陷阱。如果一个闭包的函数用外部函数的变量名定义了同样的变量，那在外部函数域将再也无法指向该变量。

```
var createPet = function(name) {   return {
    setName: function(name) {          name = name;                    }
  }
}
```

闭包中的神奇变量this是非常诡异的。使用它必须十分的小心，因为this指代什么完全取决于函数在何处被调用，而不是在何处被定义。一篇绝妙而详尽的关于闭包的文章可以在这里找到。

使用引数对象

函数的引数（译注：即实际参数）会被保存在一个类似数组的对象中。在函数内，你可以按如下方式找出传入的引数：

```
arguments[i]
```

其中i是引数的序数编号，以0开始。所以第一个传来的引数会是arguments[0]。引数的全部数量由arguments.length表示。

使用引数对象，你可以用比它正式声明会接受的更多引数来调用函数。这在你事先不知道会需要将多少引数传递给函数时十分有用。你可以用arguments.length来决定传递给函数的引数的数量，然后用arguments对象来取得每个引数。

例如，设想有一个用来连接字符串的函数。唯一事先确定的引数，是在连接后的字符串中用来分隔各个连接部分的字符（译注：比如例子里的分号“；”）。该函数定义如下：

```
function myConcat(separator) {
   var result = "",       i;
     for (i = 1; i < arguments.length; i++) {
      result += arguments[i] + separator;
   }
   return result;
}
```

你可以给这个函数传递任何数量的引数，它会将各个引数连接成一个字符串“表”：

```
myConcat(", ", "red", "orange", "blue");
myConcat("; ", "elephant", "giraffe", "lion", "cheetah");
myConcat(". ", "sage", "basil", "oregano", "pepper", "parsley");
```

请记住，arguments变量只是“矩阵式”的，但并不就是一个矩阵。称其为矩阵式是说它有一个数字指针和length属性。尽管如此，它并不拥有全部的矩阵操作方法。

更多信息请阅读JavaScript大参考里的函数对象一文。

预定义的函数

JavaScript语言有好些个顶级的预定义函数：

- eval
- isFinite
- isNaN
- parseInt and parseFloat
- Number and String
- encodeURI, decodeURI, encodeURIComponent, and decodeURIComponent (all available with Javascript 1.5 and later).

下面的几节介绍了这些函数。此类函数的更详细信息请阅读JavaScript大参考。

eval函数

eval函数对一串JavaScript代码字符求值，并且不限于特定的对象。eval的语法是这样的：

```
eval(expr);
```

这里的expr是一个被求值的字符串。

若该字符串是一个表达式，eval函数就对这个表达式求值。若这个引数表示了一个或多个JavaScript语句，eval函数就执行这些语句。eval所执行的代码的域，取决于调用代码所在的域。不要用eval对算数表达式求值；JavaScript语言会自动计算数学表达式。

isFinite函数

isFinite函数对引数求值，判断其是否为有限的数。isFinite的语法是：

```
isFinite(number);
```

其中number是需要求值的数。

若引数是非数字（译注：术语为NaN）、正无穷或负无穷，本方法会返回false，否则为true。（译

者：此处隐含的意思即函数是一种方法。此为行文精采之处，但奈何不明言之？）

下面的代码检查用户的的输入，判断其是否为有限的数字。

```
if(isFinite(ClientInput)){


}
```

isNaN函数

isNaN函数对引数求值，判断其是否为“NaN”（即“不是一个数字”的缩写）。isNaN的语法是：

```
isNaN(testValue);
```

这儿的testValue是你需要对其求值的引数。

当求值后知道结果不是数字时，parseFloat和parseInt函数返回"NaN"。而isNaN函数在传递来的引数为"NaN"时返回真，否则为非真。（译者：有点儿绕，不过当然还是对的）

下面的代码对引数floatValue求值，判断其是否为数字，然后执行相应的程序：

```
var floatValue = parseFloat(toFloat);

if (isNaN(floatValue)) {
   notFloat();
} else {
   isFloat();
}
```

parseInt和parseFloat函数

parseInt和parseFloat这两个“解析”函数，当引数为一个给定字符串时，将返回一个数字值。

parseFloat的语法是：

```
parseFloat(str);
```

此处函数parseFloat将解析它的引数，即字符串str，并尝试返回一个浮点数。若它遇到了不是正负号（+或-）、数目字（0-9）、小数点或者一个指数的字符，它将返回当前位置的值，并忽略该字符和所有其后的字符。若第一个字符就无法被转换为数字，它将返回“NaN”（即“不是一个数字”的缩写）。

parseInt的语法是：

```
parseInt(str [, radix]);
```

parseInt将解析它的第一个引数，字符串str，并尝试返回一个指定基数radix（即进位制）表示的整数，此处的基数radix（即进位制）由第二个可选的引数指定。例如，基数为十时会转换为十进制数，八时为八进制，十六则为十六进制，以此类推。若基数大于十，字母将用来表示大于十的数目位。例如，对十六进制数（基数为16），字母A到F会被用作数目字。

若parseInt遇到了一个不能以给定基数表示的字符，将忽略它和其后的所有字符，并返回一个解析到当前位置为止的整数值。若第一个字符就无法被转换为以给定基数表示的数字，它将返回“NaN”。即parseInt函数会将字符串截取为整数值。

Number和String函数

Number和String函数让你能够将一个对象转化为数字或字符串。此类函数的语法如下：（译者：此处隐含的意思即函数可对“对象”进行操作。此亦为行文精采之处，不明言之而潜移默化地使用“对象可以用作函数的引数”的概念，而汉译的“对象”一词似略不达意，且将此处深意破坏殆尽。）

```
var objRef;
objRef = Number(objRef);
objRef = String(objRef);
```

上面的objRef是一个对象引用。Number函数使用对象的alueOf()方法；而String函数使用对象的toString()方法。（译者：此处不明！？）

下例把对象Date转换为可读的的字符串。

```
var D = new Date(430054663215),
    x;
x = String(D);
```

下例把String对象转换为Number对象。

```
var str = "12",
    num;
num = Number(str);
```

你可以自己试一下。使用DOM的write()方法和JavaScript语言的typeof运算符。

```
var str = "12",
    num;
document.write(typeof str);
document.write("<br/>");
num = Number(str);
document.write(typeof num);
```

escape和unescape函数（译注：JavaScript 1.5以上已废止）

escape和unescape函数在非ASCII编码字符下工作不正常，已经被废弃。在JavaScript 1.5和之后的版本中，请使用encodeURI、decodeURI、encodeURIComponent和decodeURIComponent。

escape和unescape函数让你能编码和解码字符串。escape函数返回引数的ISO拉丁字符集的十六进制编码。而unescape函数返回该十六进制编码值相应的ASCII编码字符串值。

这些函数语法分别是：

```
escape(string);
unescape(string);
```

以上函数主要在服务器端的JavaScript脚本中，用来编码和解码URLs中的名字/值对。