**JavaScript 函数定义**

JavaScript 使用关键字 **function** 定义函数。

函数可以通过1.声明定义，也可以是一个2.表达式。

## 1.函数声明

在之前的教程中，你已经了解了函数声明的语法 :

function functionName(parameters) {  
  执行的代码  
}

函数声明后不会立即执行，会在我们需要的时候调用到。

## 实例

function myFunction(a, b) {  
    return a \* b;  
}

## 2. 函数表达式

JavaScript 函数可以通过一个表达式定义。

实际就是把一个**匿名函数对象**赋值给变量

函数表达式可以存储在变量中：

## 实例

var x = function (a, b) {return a \* b};

在函数表达式存储在变量后，变量也可作为一个函数使用：

## 实例

var x = function (a, b) {return a \* b};  
var z = x(4, 3);

以上函数实际上是一个 **匿名函数** (函数没有名称)。

函数存储在变量中，不需要函数名称，通常通过变量名来调用。

上述函数**以分号结尾**，因为它是一个执行语句。

## 3. Function() 构造函数

在以上实例中，我们了解到函数通过关键字 **function** 定义。

函数同样可以通过内置的 JavaScript 函数构造器（Function()）定义。

## 实例

var myFunction = new Function("a", "b", "return a \* b");//参数名称和语句  
  
var x = myFunction(4, 3);

实际上，你不必使用构造函数。上面实例可以写成2.函数表达式的形式

## 实例

var myFunction = function (a, b) {return a \* b}  
  
var x = myFunction(4, 3);

## 4. 函数提升（Hoisting）（使用函数声明定义函数时可使用）

在之前的教程中我们已经了解了 "hoisting(提升)"。

提升（Hoisting）是 JavaScript 默认**将当前作用域提升到前面**去的的行为。

提升（Hoisting）应用在变量的声明与函数的声明。

因此，函数可以在声明之前调用：

myFunction(5);  
  
function myFunction(y) {  
    return y \* y;  
}

**使用表达式定义函数时无法提升**。

函数提升mix函数内置对象Arguments（包含函数实参数组，是一个数组对象）

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<p>查找最大的数。</p>

<p id="demo"></p>

<script>

x = findMax(1, 123, 500, 115, 44, 88);

function findMax() {

var i, max = arguments[0];

if(arguments.length < 2) return max;

for (i = 0; i < arguments.length; i++) {

if (arguments[i] > max) {

max = arguments[i];

}

}

return max;

}

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

## Arguments 对象

JavaScript 函数有个**内置的对象** arguments 对象。

argument 对象包含了**函数调用的参数数组**。

通过这种方式你可以很方便的找到最后一个参数的值

## 5. 自调用函数

函数表达式可以 "自调用"。

自调用表达式会自动调用。

**如果表达式后面紧跟 () ，则会自动调用**。

不能自调用声明的函数。

**前面通过添加括号，来说明它是一个函数表达式**：

**后面通过添加空括号()，来自动调用自身**

## 实例

(function () {  
    var x = "Hello!!";      // 我将调用自己  
})();

5. 函数定义**作为对象的属性**，称之为**对象方法**。  
函数如果用于**创建新的对象**，称之为对象的构造函数。

6.函数的参数

## 函数显式参数(Parameters)与隐式参数(Arguments)

在先前的教程中，我们已经学习了函数的显式参数:

functionName(parameter1, parameter2, parameter3) { // 要执行的代码…… }

函数显式参数在函数定义时列出。形参

函数隐式参数在函数调用时传递给函数真正的值。也就是实参

## 参数规则

JavaScript 函数定义时**显示参数没有指定数据类型**。

JavaScript 函数**对隐式参数没有进行类型检测**。 不对实参进行类型安全检验

JavaScript 函数对隐式参数的**个数**没有进行检测。

## 默认参数

如果函数在调用时未提供隐式参数（实参），参数会默认设置为： **undefined**

有时这是可以接受的，但是建议最好为参数设置一个默认值：

## 实例

function myFunction(x, y) { if (y === undefined) { y = 0; } }

7.Javascript闭包

变量 **add** 指定了函数自我调用的返回字值。

自我调用函数只执行一次。设置计数器为 0。并返回函数表达式。

add变量可以作为一个函数使用。非常棒的部分是它可以访问函数上一层作用域的计数器。

这个叫作 JavaScript **闭包。**它使得函数拥有私有变量变成可能。

计数器**受匿名函数的作用域保护**，只能通过 add 方法修改。

|  |  |
| --- | --- |
| **Note** | **闭包是**可访问**上一层**函数**作用域里变量**的**函数**，**即便**上一层函数**已**经**关闭**。 |

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>菜鸟教程(runoob.com)</title>

</head>

<body>

<p>局部变量计数。</p>

<button type="button" onclick="myFunction()">计数!</button>

<p id="demo">0</p>

<script>

var add = (function () {

var counter = 0;

return function () {return counter += 1;}

})(); //自调用，所以函数还要包裹一层括号，再在后面加上()

function myFunction(){

document.getElementById("demo").innerHTML = add();

}

</script>

</body>

</html>

显然，add是一个自调用的闭包

# 8. JavaScript 函数调用

JavaScript 函数有 **4** 种调用方式。

每种方式的不同方式在于 **this** 的初始化。

## *****this***** 关键字

一般而言，在Javascript中，this指向函数执行时的当前对象。

|  |  |
| --- | --- |
| **Note** | 注意 **this** 是保留关键字，你不能修改 **this** 的值。 |

## 调用 JavaScript 函数

在之前的章节中我们已经学会了如何创建函数。

函数中的代码在函数被调用后执行。

## 1.作为一个函数调用

## 实例

function myFunction(a, b) { return a \* b; } myFunction(10, 2); // myFunction(10, 2) 返回 20

[**尝试一下 »**](http://www.runoob.com/try/try.php?filename=tryjs_invoke_function)

以上函数不属于任何对象。但是在 JavaScript 中它始终是默认的全局对象。

在 HTML 中默认的全局对象是 HTML 页面本身，所以函数是属于 HTML 页面。

在浏览器中的页面对象是浏览器窗口(window 对象)。以上函数会自动变为 window 对象的函数。

myFunction() 和 window.myFunction() 是一样的：

## 实例

function myFunction(a, b) { return a \* b; } window.myFunction(10, 2); // window.myFunction(10, 2) 返回 20

[**尝试一下 »**](http://www.runoob.com/try/try.php?filename=tryjs_invoke_function_2)

|  |  |
| --- | --- |
| **Note** | 这是调用 JavaScript 函数常用的方法， 但不是良好的编程习惯  全局变量，方法或函数容易造成命名冲突的bug。 |

## 2.全局对象

当函数没有被自身的对象调用时， **this** 的值就会变成全局对象。

在 web 浏览器中全局对象是浏览器窗口（window 对象）。

该实例返回 **this** 的值是 window 对象:

## 实例

function myFunction() { return this; } myFunction(); // 返回 window 对象

[**尝试一下 »**](http://www.runoob.com/try/try.php?filename=tryjs_invoke_function_3)

|  |  |
| --- | --- |
| **Note** | 函数作为全局对象调用，会使 **this** 的值成为全局对象。 使用 window 对象作为一个变量容易造成程序崩溃。 |

## 3.函数作为方法调用

在 JavaScript 中你可以将函数定义为对象的方法。

以下实例创建了一个对象 (**myObject**), 对象有两个属性 (**firstName** 和 **lastName**), 及一个方法 (**fullName**):

## 实例

var myObject = {

firstName:"John",

lastName: "Doe",

fullName: function () {

return this.firstName + " " + this.lastName;

}

} ;

myObject.fullName(); // 返回 "John Doe"

[**尝试一下 »**](http://www.runoob.com/try/try.php?filename=tryjs_invoke_method)

**fullName** 方法是一个函数。函数属于对象。 **myObject** 是函数的所有者。

**this**对象，拥有 JavaScript 代码。实例中 **this** 的值为 **myObject** 对象。

测试以下！修改 **fullName** 方法并返回 **this** 值:

## 实例

var myObject = { firstName:"John", lastName: "Doe", fullName: function () { return this; } } myObject.fullName(); // 返回 [object Object] (所有者对象)

[**尝试一下 »**](http://www.runoob.com/try/try.php?filename=tryjs_invoke_method_2)

|  |  |
| --- | --- |
| **Note** | 函数作为对象方法调用，会使得 **this** 的值成为对象本身。 |

## 使用构造函数调用函数

如果函数调用前使用了 **new** 关键字, 则是调用了构造函数。

这看起来就像创建了新的函数，但实际上 JavaScript 函数是重新创建的对象：

## 实例

// 构造函数: function myFunction(arg1, arg2) { this.firstName = arg1; this.lastName = arg2; } // This creates a new object var x = new myFunction("John","Doe"); x.firstName; // 返回 "John"

[**尝试一下 »**](http://www.runoob.com/try/try.php?filename=tryjs_invoke_constructor)

构造函数的调用会创建一个新的对象。新对象会继承构造函数的属性和方法。

|  |  |
| --- | --- |
| **Note** | 构造函数中 **this** 关键字没有任何的值。 **this** 的值在函数调用时实例化对象(new object)时创建。 |

## 作为函数方法调用函数

在 JavaScript 中, 函数是对象。JavaScript 函数有它的属性和方法。

**call()** 和 **apply()** 是预定义的函数方法。 两个方法可用于调用函数，两个方法的第一个参数必须是对象本身。

## 实例

function myFunction(a, b) { return a \* b; } myObject = myFunction.call(myObject, 10, 2); // 返回 20

[**尝试一下 »**](http://www.runoob.com/try/try.php?filename=tryjs_invoke_call)

## 实例

function myFunction(a, b) { return a \* b; } myArray = [10, 2]; myObject = myFunction.apply(myObject, myArray); // 返回 20

[**尝试一下 »**](http://www.runoob.com/try/try.php?filename=tryjs_invoke_apply)

两个方法都使用了对象本身作为第一个参数。 两者的区别在于第二个参数： apply传入的是一个参数数组，也就是将多个参数组合成为一个数组传入，而call则作为call的参数传入（从第二个参数开始）。

在 JavaScript 严格模式(strict mode)下, 在调用函数时第一个参数会成为 **this** 的值， 即使该参数不是一个对象。

在 JavaScript 非严格模式(non-strict mode)下, 如果第一个参数的值是 null 或 undefined, 它将使用全局对象替代。

|  |  |
| --- | --- |
| **Note** | 通过 call() 或 apply() 方法你可以设置 **this** 的值, 且作为已存在对象的新方法调用。 |

# 4. javascript函数能重载吗?

javascript函数的重载和java的**重载方式不一样。**  
定义JavaScript函数时，函数名是[函数对象](https://www.baidu.com/s?wd=%E5%87%BD%E6%95%B0%E5%AF%B9%E8%B1%A1&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuj6zPj0LmWmvnWmYmW6d0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHb4PjmdnWRz)的标识，参**数数量只是这个函数的属性（即Arguments）**。靠定义参数数量不同的函数实现重载是不行的。  
调用函数时，js通过函数名找到对应的[函数对象](https://www.baidu.com/s?wd=%E5%87%BD%E6%95%B0%E5%AF%B9%E8%B1%A1&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLuj6zPj0LmWmvnWmYmW6d0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHb4PjmdnWRz)，然后根据函数按照定义时的参数，和表达式参数列表**按顺序匹配**，**多余的参数舍去**，**不够的参数按undefined处理**，然后执行函数代码。所以，js重载函数需要通过函数代码判断参数值和类型实现。  
因此，**通常定义函数时，把必选参数放在参数列表最前面**，可选参数放在参数放在参数列表必须参数后面，方便函数重载。

<script type="text/javascript">

        //js是不支持重载的，但是**通过arguments以及prototype可以实现重载和继承的效果**：

        //1，通过arguments对象实现**重载的效果**，去判断函数的参数个数：

        function doAdd(){

            if(arguments.length==1){

               alert(arguments[0]+10);

            }else if(arguments.length==2){

               alert(arguments[0]+ arguments[1]);

            }

        }

        doAdd(10); //outputs 20

        doAdd(20,30); //outputs 50

    /\*备注：开发人员定义的js**函数都可以接受任意个数的参数**（根据NetScript 文档，**最多能接受25个**） ，而不会引发错误，**任何遗漏的参数都会用undefined 代替，多余的参数被将忽略**。\*/

    //2，通过prototype对象实现**继承的效果**,添加新的属性和方法

     function persona(){

        this.a = 'A';

        this.sayA = function(){

            alert('My name is a')

        }

     }

      function personb(){

            this.b = 'B';

            this.sayB = function(){

                alert('My name is b')

            }

         }

    var a = new persona();

    personb.prototype = a;

        var b = new personb(); //此时b函数拥有了a的属性和方法

    alert(b.a);//outputs ：A

    b.sayA(); //outputs ：my name is a

    alert(b.b);//outputs ：B

        b.sayB(); //outputs ：my name is a

    </script>

5.