**函数表达式**

JavaScript 函数可以通过一个表达式定义。

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。

|  |  |
| --- | --- |
| **Note** | 上述函数以分号结尾，因为它是一个执行语句。 |

**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");  
  
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：

**实例**

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

|  |  |
| --- | --- |
| **Note** | 在 JavaScript 中，很多时候，你需要避免使用 **new** 关键字。 |

**函数提升（Hoisting）**

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。

**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。

通过添加括号，来说明它是一个函数表达式：

**实例**

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

以上函数实际上是一个 **匿名自我调用的函数** (没有函数名)。

**函数是对象**

在 JavaScript 中使用 **typeof** 操作符判断函数类型将返回 "function" 。

但是JavaScript 函数描述为一个对象更加准确。

JavaScript 函数有 **属性** 和 **方法**。

arguments.length 属性返回函数调用过程接收到的参数个数：

**实例**

function myFunction(a, b) {  
    return arguments.length;  
}

toString() 方法将函数作为一个字符串返回:

**实例**

function myFunction(a, b) {  
    return a \* b;  
}  
  
var txt = myFunction.toString();

JavaScript 函数对参数的值没有进行任何的检查。

## 函数显式参数(Parameters)与隐式参数(Arguments)

函数显式参数在函数定义时列出。

函数隐式参数在函数调用时传递给函数真正的值。

## 参数规则

JavaScript 函数定义时显式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。

JavaScript 函数对隐式参数的个数没有进行检测。

## 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"

## 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"

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

## 实例

function myFunction(a, b) {

return a \* b;

}

myArray = [10, 2]; myObject = myFunction.apply(myObject, myArray); // 返回 20

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。 |