**JavaScript**

# JavaScript简介

* 什么是JavaScript

JavaScript一种直译式脚本语言，是一种动态类型、弱类型、基于原型的语言，内置支持类型。它的解释器被称为JavaScript引擎，为浏览器的一部分，广泛用于客户端的脚本语言，最早是在HTML（标准通用标记语言下的一个应用）网页上使用，用来给HTML网页增加动态功能。

JavaScript是一种属于网络的脚本语言,已经被广泛用于Web应用开发,常用来为网页添加各式各样的动态功能,为用户提供更流畅美观的浏览效果。通常JavaScript脚本是通过嵌入在HTML中来实现自身的功能的。

1. 是一种解释性脚本语言（代码不进行预编译）。
2. 主要用来向HTML（标准通用标记语言下的一个应用）页面添加交互行为。
3. 可以直接嵌入HTML页面，但写成单独的js文件有利于结构和行为的分离。
4. 跨平台特性，在绝大多数浏览器的支持下，可以在多种平台下运行（如Windows、Linux、Mac、Android、iOS等）。

Javascript脚本语言同其他语言一样，有它自身的基本数据类型，表达式和算术运算符及程序的基本程序框架。Javascript提供了四种基本的数据类型和两种特殊数据类型用来处理数据和文字。而变量提供存放信息的地方，表达式则可以完成较复杂的信息处理。

* 日常用途：

1. 嵌入动态文本于HTML页面。
2. 对浏览器事件做出响应。
3. 读写HTML元素。
4. 在数据被提交到服务器之前验证数据。
5. 检测访客的浏览器信息。
6. 控制cookies，包括创建和修改等。
7. 基于Node.js技术进行服务器端编程。

* JavaScript脚本语言具有以下特点:

1. 脚本语言。JavaScript是一种解释型的脚本语言,C、C++等语言先编译后执行,而JavaScript是在程序的运行过程中逐行进行解释。
2. 基于对象。JavaScript是一种基于对象的脚本语言,它不仅可以创建对象,也能使用现有的对象。
3. 简单。JavaScript语言中采用的是弱类型的变量类型,对使用的数据类型未做出严格的要求,是基于Java基本语句和控制的脚本语言,其设计简单紧凑。
4. 动态性。JavaScript是一种采用事件驱动的脚本语言,它不需要经过Web服务器就可以对用户的输入做出响应。在访问一个网页时,鼠标在网页中进行鼠标点击或上下移、窗口移动等操作JavaScript都可直接对这些事件给出相应的响应。
5. 跨平台性。JavaScript脚本语言不依赖于操作系统,仅需要浏览器的支持。因此一个JavaScript脚本在编写后可以带到任意机器上使用,前提上机器上的浏览器支 持JavaScript脚本语言,目前JavaScript已被大多数的浏览器所支持。[3]

不同于服务器端脚本语言，例如PHP与ASP，JavaScript主要被作为客户端脚本语言在用户的浏览器上运行，不需要服务器的支持。所以在早期程序员比较青睐于JavaScript以减少对服务器的负担，而与此同时也带来另一个问题：安全性。

而随着服务器的强壮，虽然程序员更喜欢运行于服务端的脚本以保证安全，但JavaScript仍然以其跨平台、容易上手等优势大行其道。同时，有些特殊功能（如AJAX）必须依赖Javascript在客户端进行支持。随着引擎如V8和框架如Node.js的发展，及其事件驱动及异步IO等特性，JavaScript逐渐被用来编写服务器端程序。

* JavaScrip实现

虽然JavaScriptECMAScript通常被人们用来表达相同的含义，但是完整的javaScript实现应该由以下三个不同的部分组成：

1. 核心（ECMAScript）

ECMAScript定义了JavaScript语言的基础，在此基础上可以构建完善的脚本语言及语言的扩展。ECMAScript规定了语言的语法、类型、语句、关键字、保留字、操作符、对象等。

1. 文档对象模型（DOM）

文档对象模型是针对XML但经过扩展用于HTML的应该程序编程接口（API）。DOM把整个页面映射为一个多层节点接口。通过DOM创建的这个文档树形结构，开发人员可以获得控制页面内容和结构的主动权。借助DOM提供的API，可以实现对页面内容的删除、添加、替换或修改任何节点。

1. 浏览器对象模型（BOM）

使用BOM可以控制浏览器显示的页面以外的部分。如：弹出新窗口、移动缩放和关闭浏览器窗口、提供浏览器详细信息的navigator对象、提供浏览器所有加载页面的详细信息的location对象、提供用户显示器分辨率详细信息的screen对象、对cookies的支持等。

# HTML中使用JavaScript

## <script></script>标签

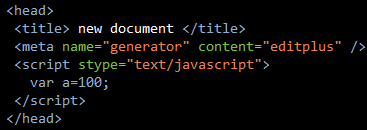
向HTML页面中插入javascript的主要方法就是使用<script>标签。

标签属性：

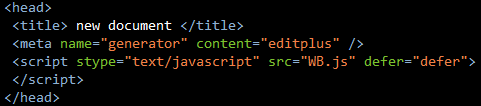
1. async：可选。表示应该立即下载脚本，但是不应妨碍页面中的其他操作，比如下载其他资源或等待加载其他脚本。只对外部脚本文件有效。
2. charset：可选。表示通过src属性指定的代码的字符集。
3. defer：可选。表示脚本可以延迟到文档完全被解析和显示之后再执行。只对外部脚本文件有效。
4. src：可选。表示包含要执行代码的外部文件。
5. type：必选。用于指定脚本语言的MIME类型（text/javascript）

## 在HTML中使用javascript的方法

1. 使用<script>标签直接将javascript代码锲入到HTML代码内部。



1. 使用<script>标签的src属性引入外部javascript代码文件（后缀名为.js）



说明：在一组<script></script>标签中不能同时存在外部javascript文件和javascript代码。如果同时存在解析器也只会执行其中的javascript，不会执行外部的javascript文件。

## 在HTML中<script>标签的位置

在HTML中锲入的javascript可以根据代码的实际功能来选择锲入的位置。比如需要随网页加载的时候一起执行的代码块，则可以锲入的网页的头部。如果是网页加载的时候不需要执行的javascript代码或外部javascript文件，则可以将其放在页面的尾部。使其不影响页面的加载效果。

当然外部引用的javascript文件，可以通过设置<script>标签的defer和anync属性来设置代码执行的时间。

## defer属性（defer=”defer”）

用于表明脚本在执行时不会影响页面的构造。脚本会被延迟到整个页面都解析完毕后再运行。立即下载，延迟执行。只适用于引入的外部javascript文件

## async属性（async=”async”）

用于改变处理脚本的行为，于defer属性类型，但是此属性不能保证脚本按照指定它们的先后顺序执行。只适用于外部javascript文件。

## <noscript></noscript>标签

用于在不支持或关闭javascript的浏览器中显示替代的文字。

# ECMAScript

## 语法

1. 区分大小写

在ECMAScript中，所有的一切都是区分大小写的，并使用 Unicode 字符集。

1. 标识符

标识符指的是变量、函数、属性的名字，或者函数的参数。

标识符的命名规则：

第一个字符必须是一个字母、下划线、美元符号或数字。

标识符不能是ECMAScript中的关键字和保留字

按照惯例，ECMAScript标识符采用驼峰大小写格式。例如：myCar、firstSecond。

1. 注释

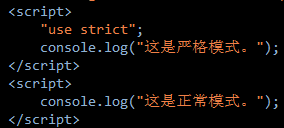
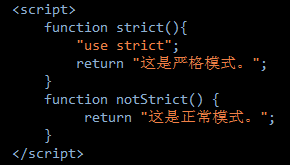
在ECMAScript中分为单行注释和多行注释，单行采用//注释，多行采用/\*….\*/注释

1. 严格模式

在ECMAScript5中引入了严格模式的概念。在严格模式中一些不确定的行为将得到处理，对于一些不安全的操作也会抛出错误。

启用严格模式，只需在代码顶部加入”use strict”

整个脚本文件： 定义函数严格模式：

1. 语句格式

ECMAScript中每条完整的语句都以分号结尾。如果一条语句独占一行的话，那么分号是可以省略的。（译者注：并不建议这么做。）但如果一行中有多条语句，那么这些语句必须以分号分开。 ECMAScript 规定了在语句的末尾自动插入分号（ASI）。（如果想要了解更多信息，请参阅 JavaScript 词法语法 。）虽然不是必需的，但是在一条语句的末尾加上分号是一个很好的习惯。这个习惯可以大大减少代码中产生 bug 的可能性。

在条件控制语句（如if语句）中，尽量使用代码块的方式编写语句。

if (test){

alert(test);

}

1. 关键字和保留字

关键字

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **break** | **do** | **instanceof** | **typeof** | **case** | **else** |
| **new** | **var** | **catch** | **finally** | **return** | **void** |
| **continue** | **for** | **switch** | **while** | **debugger (5版)** | **function** |
| **this** | **with** | **default** | **if** | **throw** | **delete** |
| **in** | **try** |  |  |  |  |

保留字

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **abstract** | **enum** | **int** | **short** | **boolean** | **export** |
| **interface** | **static** | **byte** | **extends** | **long** | **super** |
| **char** | **final** | **native** | **synchronized** | **class** | **float** |
| **package** | **throws** | **const** | **goto** | **private** | **transient** |
| **debugger** | **implements** | **protected** | **volatile** | **double** | **import** |
| **public** | **let (5版)** | **yield (5版)** |  |  |  |

## 声明

在JavaScript中可以使用三种关键字来对变量进行声明：var、let、const。这三种声明方式的都是相同的，只是表达的语义不同。

### var修饰符

声明一个变量，并可选地将其初始化为一个值。

**语法：**



varnameN

变量名。变量名可以定义为任何合法标识符。

valueN

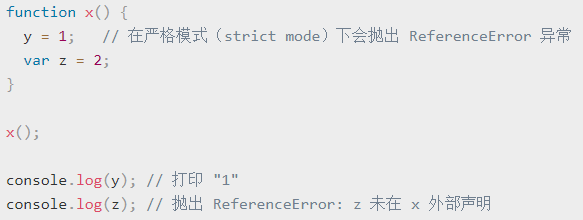
变量的初始化值。该值可以是任何合法的表达式。默认值为 undefined。

**描述：**

变量声明，无论发生在何处，都在执行任何代码之前进行处理。用 var 声明的变量的作用域是它当前的执行上下文，它可以是嵌套的函数，也可以是声明在任何函数外的变量。如果你重新声明一个 JavaScript 变量，它将不会丢失其值。

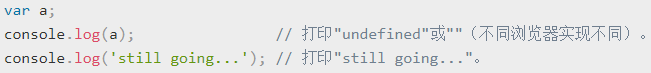
将赋值给未声明变量的值在执行赋值时将其隐式地创建为全局变量（它将成为全局对象的属性）。声明和未声明变量之间的差异是：

1. 声明变量的作用域限制在其声明位置的上下文中，而非声明变量总是全局的。

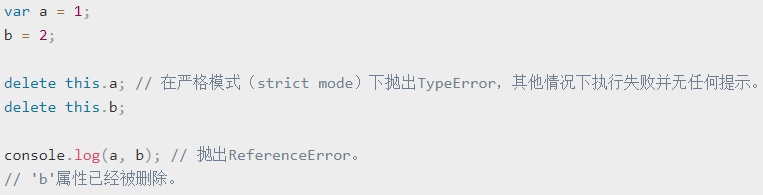


1. 声明变量在任何代码执行前创建，而非声明变量只有在执行赋值操作的时候才会被创建。





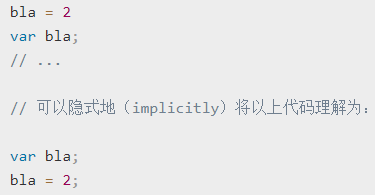
1. 声明变量是它所在上下文环境的不可配置属性，非声明变量是可配置的（如非声明变量可以被删除）。



由于这三个差异，未能声明变量将很可能导致意想不到的结果。因此，建议始终声明变量，无论它们是在函数还是全局作用域内。 而且，在 ECMAScript 5 严格模式下，分配给未声明的变量会引发错误。

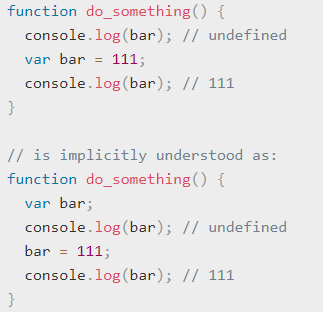
**变量提升**

由于变量声明（以及其他声明）总是在任意代码执行之前处理的，所以在代码中的任意位置声明变量总是等效于在代码开头声明。这意味着变量可以在声明之前使用，这个行为叫做“hoisting”。“hoisting”就像是把所有的变量声明移动到函数或者全局代码的开头位置。



因此，建议始终在作用域顶部声明变量（全局代码的顶部和函数代码的顶部），这可以清楚知道哪些变量是函数作用域（本地），哪些变量在作用域链上解决。

重要的是，提升将影响变量声明，而不会影响其值的初始化。当到达赋值语句时，该值将确实被分配：

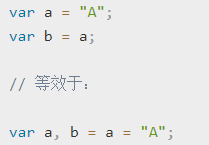


**例子**

声明并初始化两个变量：



给两个变量赋值成字符串值：

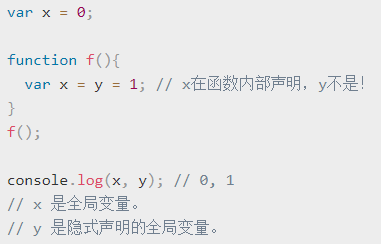


留意其中的顺序：



在这里，x 和 y 在代码执行前就已经创建了，而赋值操作发生在创建之后。当"x = y"执行时，y 已经存在，所以不抛出ReferenceError，并且它的值是'undefined'。所以 x 被赋予 undefined 值。然后，y 被赋予'A'。于是，在执行完第一行之后，x === undefined && y === 'A' 才出现了这样的结果。

多个变量的初始化



隐式全局变量和外部函数作用域

看起来像是隐式全局作用域的变量也有可能是其外部函数变量的引用。



### let修饰符

let 语句声明一个块级作用域的本地变量，并且可选的将其初始化为一个值。

**语法**



var1, var2, …, varN

变量名。必须是合法的标识符。

value1, value2, …, valueN

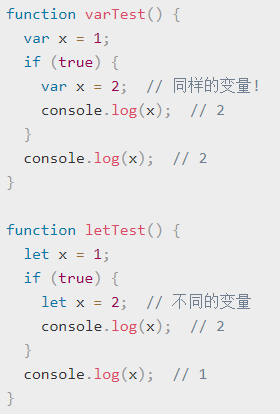
变量的初始值。可以是任意合法的表达式。

**描述**

let允许你声明一个作用域被限制在块级中的变量、语句或者表达式。与var关键字不同的是，var声明的变量只能是全局或者整个函数块的。

1. 作用域规则

let声明的变量只在其声明的块或子块中可用，这一点，与var相似。二者之间最主要的区别在于var声明的变量的作用域是整个封闭函数。



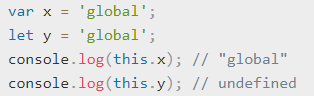
1. 简化内部函数代码

当用到内部函数的时候，let会让你的代码更加简单。



上面这段代码的意图是创建5个li,点击不同的li能够打印出当前li的序号。如果不用let，而改用var的话，将总是打印出 Item 5 is Clicked，因为 j 是函数级变量，5个内部函数都指向了同一个 j ,而 j 最后一次赋值是5。用了let后，j 变成块级域（也就是花括号中的块，每进入一次花括号就生成了一个块级域）,所以 5 个内部函数指向了不同的 j 。

在程序或者函数的顶层，let并不会像var一样在全局对象上创造一个属性，比如



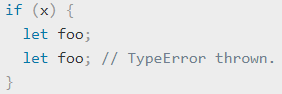
1. 模仿私有接口

在处理构造函数的时候，可以通过let声明而不是闭包来创建私有接口。

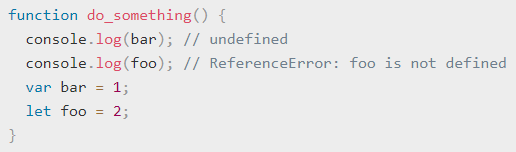


1. let 的暂存死区与错误

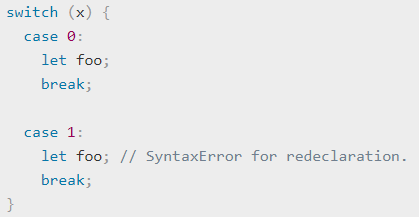
在同一个作用域中用let重复定义一个变量将引起 TypeError.



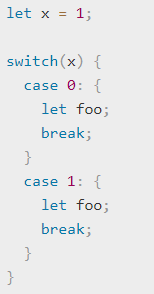
在 ECMAScript 2015 中，let 绑定不受变量提升的约束，这意味着 let 声明不会被提升到当前执行上下文的顶部。在块中的变量初始化之前，引用它将会导致 ReferenceError（而使用 var 声明变量则恰恰相反，该变量的值是 undefined ）。这个变量处于从块开始到 let 初始化处理的”暂存死区“之中。



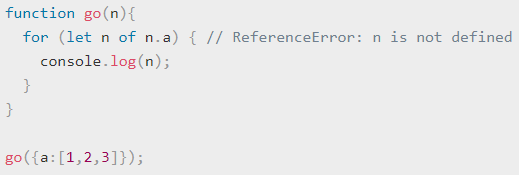
在 switch 声明中你可能会遇到这样的错误，因为一个switch只有一个作用块。



但是，需要指出，在case语句后面可以创建新的作用域块，形成新的词法环境，这样就不会产生上述重复声明（redeclaration）的语法错误。

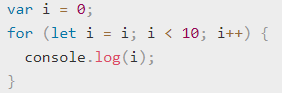


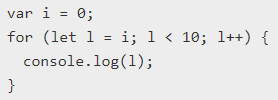
let后跟一个函数传递的参数时将导致循环内部报错。



1. 循环定义中的let作用域

循环体中是可以引用在for声明时用let定义的变量，尽管let不是出现在大括号之间.(注：该方法在 火狐 45.4.0 ,Centos7 下，报错 ReferenceError: can't access lexical declaration `i' before initialization)





域作用规则

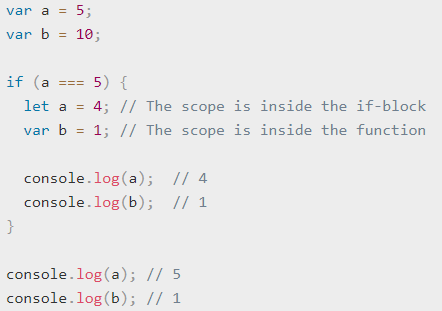


在这个例子中，expr2, expr3, 和 statement 都是包含在一个隐含域块中，其中也包含了 expr1.

**例子**

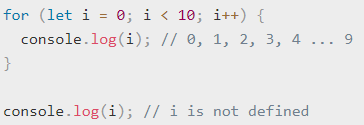
let 对比 var

let的作用域是块，而var的作用域是函数



let 在循环中

可以用 let 来代替 var ，在 for 定义块中使用块级变量.



### const修饰符

常量是块级作用域，很像使用 let 语句定义的变量。常量的值不能通过重新赋值来改变，并且不能重新声明。

**语法**



nameN

常量名称，可以是任意合法的标识符。

valueN

常量值，可以是任意合法的表达式。

**描述**

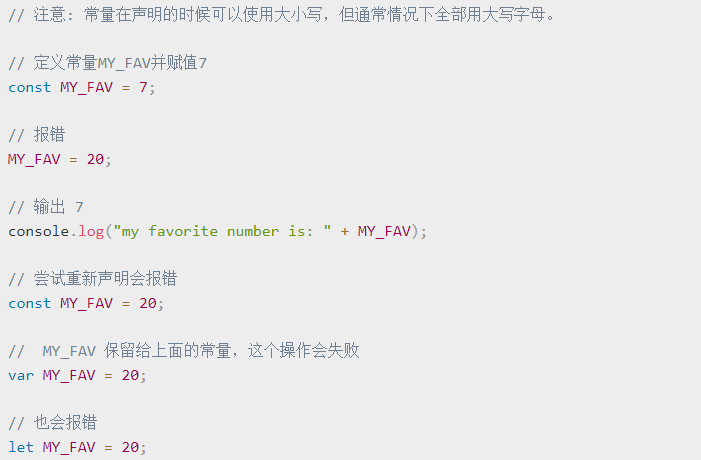
此声明创建一个常量，其作用域可以是全局或本地声明的块。 与var变量不同，全局常量不会变为窗口对象的属性。需要一个常数的初始化器；也就是说，您必须在声明的同一语句中指定它的值（这是有道理的，因为以后不能更改）。

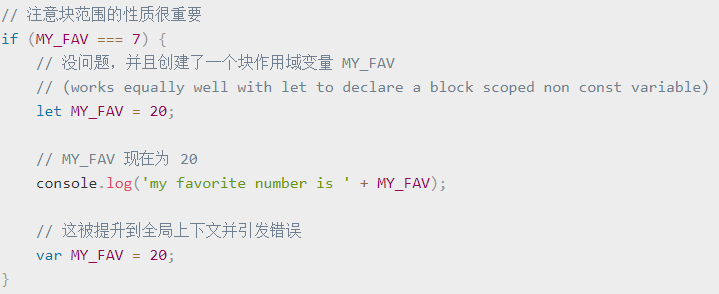
const声明创建一个值的只读引用。但这并不意味着它所持有的值是不可变的，只是变量标识符不能重新分配。例如，在引用内容是对象的情况下，这意味着可以改变对象的内容（例如，其参数）。

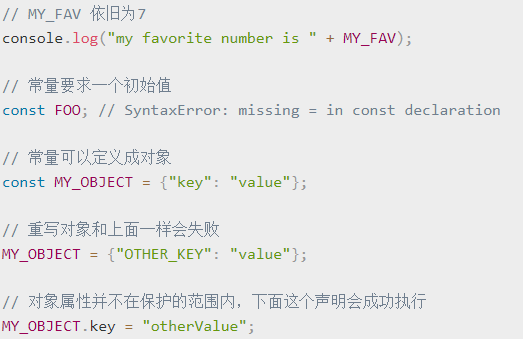
一个常量不能和它所在作用域内的其他变量或函数拥有相同的名称。

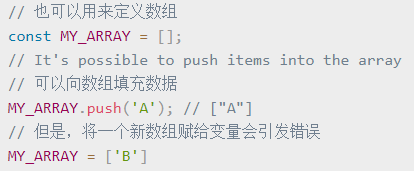
**示例**

下面的例子演示了常量的特性。在浏览器的控制台试一下这个例子。









## 变量

### 变量命名规则

在应用程序中，使用变量来作为值的符号名。变量的名字又叫做标识符，其需要遵守一定的规则。

一个 JavaScript 标识符必须以字母、下划线（\_）或者美元符号（$）开头；后续的字符也可以是数字（0-9）。因为 JavaScript 语言是区分大小写的，所以字母可以是从“A”到“Z”的大写字母和从“a”到“z”的小写字母。

你可以使用大部分 ISO 8859-1 或 Unicode 编码的字符作标识符，例如 å 和 ü。你也可以使用 Unicode 转义字符 作标识符。

合法的标识符示例：Number\_hits，temp99，$credit 和 \_name。

### 声明变量

你可以用以下三种方式声明变量：

使用关键词 var 。例如 var x = 42。这个语法可以用来声明局部变量和全局变量。

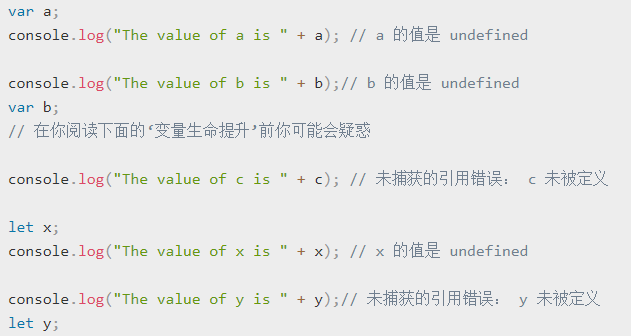
直接赋值。例如x = 42。在函数外使用这种形式赋值，会产生一个全局变量。在严格模式下会产生错误。因此你不应该使用这种方式来声明变量。

使用关键词 let 。例如 let y = 13。这个语法可以用来声明块作用域的局部变量。

### 变量求值

用 var 或 let 语句声明的变量，如果没有赋初始值，则其值为 undefined 。

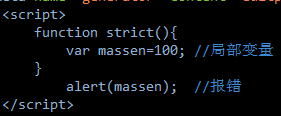
如果访问一个未声明的变量会导致抛出一个引用错误（ReferenceError）异常：



ECMAScript的变量是松散类型的，变量只是一个占位符，可以用来保存任何类型的数据。通过使用var操作符来定义变量

var message=100;

1. 使用var操作符定义的变量将成为定义该变量的作用于中的局部变量。如果在一个函数体中用var定义的变量，该变量只在函数体中有效，函数退出后，该变量就会被销毁。如果不使用var定义变量，则该变量为全局变量。但是不推荐使用此方法。（在局部作用域中定义全局变量将会很难维护，并且在严格模式下会抛出ReferenceError错误）



1. 变量可以同时定义多个，变量名之间用逗号隔开



## 数据类型

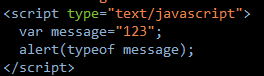
ECMAScript中定义了5中基本数据类型：undefined、null、boolean、number、string和1种复杂数据类型：object。

ECMAScriot不支持任何创建自定义类的机制，所有值最终都将是上述6种数据类型之一。

### typeof操作符

由于javascript是松散类型的语言，所以在ECMAScript中定义了一个typeof操作符用于检测变量的数据类。

语法格式：alert(typeof 变量); //说明：alert()函数，用于输出结果。



可能返回值：undefined-----未定义：表示变量没有定义

boolean-------布尔值

string----------字符串

number-------数值

object---------对象或null

function-------函数：表示值为一个函数

### undefined类型

undefined类型只有一个值，即特殊的undefined。在使用var声明变量但为对其初始化赋值，这个变量的值就是undefined

功能：

使用字面量undefined的主要目的是用于比较。还有一个是用于正式区分空对象指针和未初始化的变量。

说明：

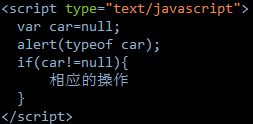
变量未初始化赋值和变量未被定义是不同的，变量未被定义alert()返回错误提示。

### null类型

null类型也只有一个值，即特殊的值null。表示为一个空对象指针。

功能：

如果定义了一个变量用于将来保存对象，那么最好将该变量初始化为null而不是其他值。这样只要直接检查null值就可以知道相应的变量是否已经保存了一个对象的引用。

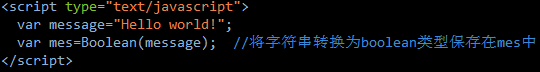


说明：只要变量是用于保存对象的引用但还未保存对象时，必须将变量的值保存为null

### boolean类型

boolean类型是ECMAScript中使用得最多的一种类型，该类型只有两个字面值：true和false。这两个值于数字值不是一回事，因此true不一定等于1，而false也不一定等于0。

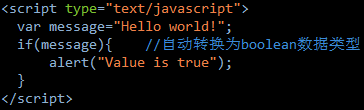
在ECMAScript中所有类型的值都有与这两个boolean值等价的值。要将一个值转换为对应的boolean值，可以调用转型函数Boolean()



|  |  |  |
| --- | --- | --- |
| 数据类型 | 转化为true的值 | 转换为false的值 |
| bollean | true | false |
| string | 任何非空字符串 | “”（空字符串） |
| number | 任何非零数字值（包括无穷大） | 0和NaN |
| object | 任何对象 | null |
| undefined | n/a（不适用） | undefined |

功能：

只要用于程序中的流程判断和逻辑的判断，经常与判断语句结合使用。



### number类型

ECMAScript中规定了number类型用于保存所有的数值（整数和浮点数），保存方式为八进制、十进制、十六进制。但是在作算数计算的时候所有的都会转换成十进制进行数值计算

在javascript中保存数值的方式，可以保存正零和负零（正零和负零被认为相等）

浮点数值：

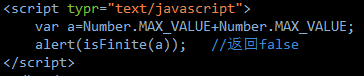
浮点数值即小数，在内存中会占用整数值两倍的内存空间。如果浮点数值本身表示的是一个整数（1.0），那么该值也会被转换为整数。浮点数值的最高精度是17位小数，但是在进行算术计算的时候精度远远不如整数，所以不能将浮点数用于比较判断，这样会报错

数值范围：

ECMAScript并不能保存世界上所有的数值，ECMAScript能够表示的最小数值保存在Number.MIN\_VALUE中，这个值是5e-324。最大值保存在Number.MAX\_VALUE中，这个值是1.7976931348623157e+308。

如果计算得到的值超出javascript数值范围，那么这个数值将被自动转换成特殊的Infinity值(正无穷)。负值为-Infinity（负无穷）。而且该值将无法继续参与下次的计算（Infinity无法参与计算）。Number.NEGATIVB\_INFINITY和Number.POSITIVE\_INFINITY也可以得到负和正Infinity的值

isFinite()函数：用于判断数值是不是有穷的，函数的参数为有穷则会返回true



NaN：非数值。是一个特殊的数值，这个数值用于表示一个本来要返回数值的操作数未返回数值的情况（这样就不会抛出错误）。例

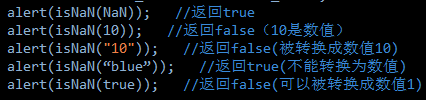
如，在其他编程语言中，任何数值除以0都会导致错误，从而停止代码执行。但在ECMAScript中，任何数值除以0会返回NaN，因此不会影响其他代码的执行。

NaN本身有两个非同寻常的特点。1.任何涉及NaN的操作（例如：NaN/10）都会返回NaN，这个特点在多步计算中可能导致问题。2.NaN与任何值都不相等，包括NaN本身。



isNaN()函数：用于检测所传的参数是否“不是数值”。参数可以是任何类型。isNaN()接收到一个值后，首先将其转换为数值。任何不

能被转换为数值的值都会导致这个函数返回true



isNaN()可以适用于对象。在基于对象调用isNaN()函数时，会首先调用对象的valueOf()方法，然后确定该方法返回的值是否可

以转换为数值。如果不能，则基于这个返回值再调用toString()方法，再测试返回值

数值转换：

有3个函数可以将非数值转换为数值：Number()、parseInt()、parseFloat()。

Number()：转型函数，可以用于任何数据类型

如果是boolean值，true和false将分别转换为1和0

如果是数值，只是简单的传入和返回

如果是null值，返回0

如果是undefined，返回NaN

如果是字符串，则遵循以下规则：

如果字符串中包含数字，将其转换为十进制数值。

如果字符串中包含有效的浮点格式，将其转换为对应的浮点数值。

如果字符串中包含有效的十六进制格式，将其转换为相同大小的十进制整数值。

如果字符串是空的，则将其转换为0

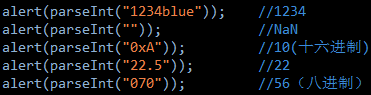
其余的将会转换为NaN

parseInt()：此函数在转换字符串时，更多的是看其是否符合数值模式。它会忽略字符串前面的空格，直接找到第一个非空格字

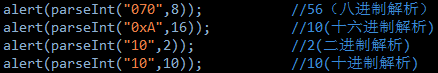
符。如果第一个字符不是数值字符，则直接返回NaN。parseInt()在转换空字符串会返回NaN（Number()则会返回0）。

如果第一个字符是数值字符，则会继续解析第二个以此类推，直到遇到非数值字符或解析完剩余的字符。

如果字符串中的第一个字符是数字字符，parseInt()也能够识别出各种整数格式（八进制、十进制、十六进制），但是parseInt()无法有效的转换浮点数值，因为小数点并不是有效的数字字符，所以parseInt()只适用于转换整数型数值字符串。



在ECMAScript5中已经不具有解析八进制值的能力，前导的零会被认为无效，从而将这个值当成“0”，得到十进制的0。严格模式中也是如此。但是可以为这个函数提供第二个参数：转换时使用的基数（即多少进制）。这个在解析的时候很有必要。

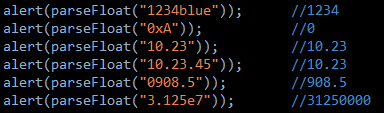


parseFloat()：与parseInt()类似，parseFloat()函数也是从第一个字符（位置0）开始解析每个字符，直到解析到字符串末尾，

或者解析到遇见一个无效的浮点数字字符为止。字符串中的第一个小数点是有效的，而第二个小数点的无效的，因此它后面的字符串将被忽略。



parseFloat()函数始终会忽略前导的零。所以parseFloat()只能解析十进制值的字符串。



### string类型

string类型用于表示字符串，字符串可以用双引号或单引号表示



特点：

ECMAScript中的字符串是不可变的，也就是字符串一旦创建，它们的值就不能改变。要改变某个变量保存的字符串，首先要销毁原来的字符串，然后再用另一个包含新值的字符串填充该变量



转意字符串：

使用转义字符来表示字符集中定义的字符，比如ASCll里面的控制字符及回车换行等字符，这些字符都没有现成的文字代号。所

以只能用转义字符来表示 。二、某一些特定的字符在编辑语言中被定义为特殊用途的字符。这些字符由于被定义为特殊用途，

它们失去了原有的意义。

|  |  |
| --- | --- |
| 字面量 | 含义 |
| \n | 换行 |
| \t | 制表符 |
| \b | 空格 |
| \r | 回车 |
| \f | 进纸 |
| \\ | 斜杠 |
| \’ | 单引号，在用单引号表示的字符串中使用：’He said,\’hey.\’ ’ |
| \” | 双引号，在用双引号表示的字符串中使用：”He said,\”hey.\’’ ” |
| \xnn | 以十六进制代码nn表示的一个字符（其中n为0~f）：\x41表示“A” |
| \unnnn | 以十六进制代码nnnn表示的一个Unicode字符（其中n为0~f）：\u03a3表示希腊字符Σ |

length属性：返回字符串的长度，字符串有几个字符组成（包括空字符）。

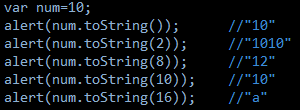


转换为字符串：

要把一个值转换为字符串有两种方法：1.使用toString()方法。2.使用加号操作符与一个字符串拼接。

toString()方法：

其中数值、布尔值、对象和字符串值都有toString()方法，但是null和undefined值没有此方法。（字符串返回的是一个字符串副本）多数情况下调用toString()方法不必传递参数。但是调用数值的toString()方法是，可以传递一个参数：输出的值的参数。（默认情况下数值toString()方法返回的是以十进制格式的字符串，可以通过传递基数参数来指定返回的格式）



转型函数

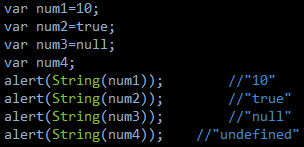
如果不知道要转换的数值是不是null或undefined的情况下，还可以使用转型函数String()，这个函数能都将任何类型的值转换为字符串。

遵循规则：

如果值有toString()方法，则调用该方法（没有参数）并返回相应的结果

如果值是null，则返回”null”

如果只是undefined，则返回”undefined”



### Object类型

ECMAScript中的对象其实就是一组数据和功能的集合。对象可以通过new操作符后跟要创建的对象类型的名称来创建。而创建Object类型的实例并为其添加属性和（或）方法，就可以创建自定义对象。

调用Object构造函数创建对象

在ECMAScript中Object类型是所有创建Object实例的基础，Object类型所具有的任何属性和方法也同样存在于更具体的对象中

Object的每个实例都具有下列属性和方法：

Constructor：保存着用于创建当前对象的函数。Object()构造函数

hasOwnProperty()：用于检查给定的属性在当前对象实例中（而不是在实例的原型中）是否存在。其中参数的属性名必须以字

符串形式指定。



isPrototypeOf()：用于检查传入的对象是否是另一个对象的原型。参数为对象名。



propertyIsEnumerable()：用于检查给定的属性是否能够使用for….in语句来枚举。参数属性名必须以字符串形式指定。

toLocaleString()：返回对象的字符串表示，该字符串与执行环境的地区对应。

toString()：返回对象的字符串表示。

valueOf()：返回对象的字符串、数值或布尔值表示。通常与toString()方法的返回值相同。

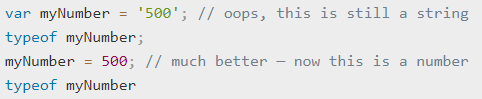
### 动态类型

JavaScript是一种“动态类型语言”，这意味着不同于其他一些语言(译者注：如C、JAVA)，您不需要指定变量将包含什么数据类型（例如number或string）

例如，如果你声明一个变量并给它一个带引号的值，浏览器就会知道它是一个字符串：



即使它包含数字，但它仍然是一个字符串，所以要小心：



尝试依次将上述代码输入您的控制台，看看结果是什么（无须输入//之后的注释）。 我们使用了一个名为typeof()的特殊函数 ——它会返回所传递给它的变量的数据类型。 第一次在上面的代码中调用它，它应该返回string，因为此时myNumber变量包含一个字符串'500'。 看看它第二次将返回什么。

## 操作符

ECMAScript中定义了一组用于操作数值的操作符，包括算术操作符、位操作符、关系操作符、相等操作符。在ECMAScript中操作符能够适用于很多值，例如字符串、数值、布尔值，对象。但是在应用于对象时，相应的操作符通常都会调用对象的valueOf()和（或）toString()方法，以便取得可以操作的值。

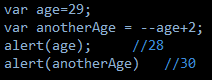
### 一元操作符

只能操作一个值的操作符叫做一元操作符。

递增和递减操作符：++、--

前置型：将操作符放于变量前面，效果为：自身加1后再赋值给自身



后置型：将操作符放于变量后面，效果为：先进行操作，再自身加1并赋值给自身



递增和递减操作符适用于任何值、字符串、布尔值、浮点数值、对象。但是遵循以下规则：

在应用于一个包含有效数字字符的字符串时，先将其转换为数字值，再执行递增递减操作。字符串变量变成数值变量。

在应用于一个不包含有效数字字符的字符串时，将变量的值设置为NaN。字符串变量变成数值变量。

在应用于布尔值false时，先将其转换为0再执行递增递减操作。布尔值变量编程数值变量。

在应用于布尔值true时，先将其转换为1再执行递增递减操作。布尔值变量编程数值变量。

在应用于浮点数值是，直接进行递增递减操作。

在应用于对象时，先调用valueOf()方法，以取得一个可供操作的值，然后对该值进行递增递减操作。如果结果是NaN，则

再调用toString()方法让后在执行操作。对象变量变成数值变量

一元加和减的操作符：

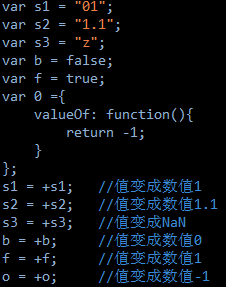
ECMAScript中定义的一元加和减于数学书中完全一样，一元加操作符以一个加号（+）表示，一元减操作符以一个减号（-）

表示。

一元加操作符放于变量前面没有任何影响，但是在对非数值应用的时候，操作符会像Number()转型函数一样对该值进行转

型。布尔值false和true将被转换为0和1，字符串或根据规则进行转换，对象会调用valueOf()和（或）toString()方法，

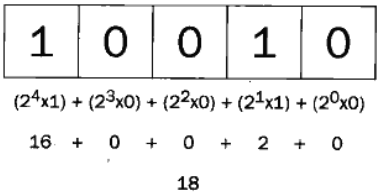
进行转换。



一元减操作符主要用于表示负数，在操作非数值的值和一元加操作符一样，先转换再执行。

### 位操作符

位操作符用于在最基本的层次上，即按内存中表示数值的位来操作数值。ECMAScript中的所有数值都以IEEE-754 64位格式存储，但是位操作符并不直接操作64位的值，而是先将64位的值转换成32位的整数，然后执行操作，最后再将结果转换回64位。（计算机中所有数据都是以二进制的形式存储在内存中，即用0和1表示，排序位从右至左）对于有符号的整数，32位中的前31位用于表示整数的值。第32位用于表示数值的符号：0表示正数，1表示负数。这个表示符号的位叫做符号位，符号位的值决定了其他位数值的格式。其中，正数以纯二进制格式存储，31位中的每一位都表示2的幂。第一位表示2的0次方，第二位表示2的1次方，一次类推。没有用到的位以0填充。例如：18的二进制表示是00000000000000000000000000010010，或者更简洁的10010。



负数同样以二进制码存储，但使用的格式是二进制补码。计算一个数值的二进制补码，需要经过以下3个步骤：

求这个数值绝对值的二进制码（例如，要求-18的二进制补码，先求18的二进制码）

求二进制反码，即将0替换为1，将1替换为0

得到的二进制反码加1

### 布尔操作符

布尔操作符主要用于测试两个值的关系，一般与判断语句或循环语句一起使用。布尔操作符分为3个：非（NOT）、与（AND）、或（OR）

逻辑非

逻辑非操作符由一个叹号（！）表示，可以应用于ECMAScript中的任何值。这个操作符会返回一个布尔值。

逻辑非操作符首先会将操作数转换为一个布尔值，然后再对其求反。

如果操作数是一个对象，返回false；

如果操作数是一个空字符串，返回true；

如果操作数是一个非空字符串，返回false；

如果操作数是一个数值0，返回true；

如果操作数是任意非0数值（包括Infinity），返回false；

如果操作数是null，返回true；

如果操作数是NaN，返回true；

如果操作数是undefined，返回true；

逻辑非操作符也可以用于将一个值转换为与其对应的布尔值。而同时使用两个逻辑非操作符，实际上就会模拟Boolean()转型函

数的行为。



逻辑与

逻辑与操作符是由两个和号（&&）表示，有两个操作数。



|  |  |  |  |
| --- | --- | --- | --- |
| 第一个操作数的值 | 第二个操作数的值 | 结果 | 说明 |
| true | true | true | 第一个操作数的值为true时会继续对第二个操作数求值 |
| true | false | false | 第一个操作数的值为true时会继续对第二个操作数求值 |
| false | true | false | 第一个操作数的值为false时不会对第二个操作数求值 |
| false | false | false | 第一个操作数的值为false时不会对第二个操作数求值 |

逻辑与操作可以应用于任何类型的操作数，而不仅仅是布尔值。在有一个操作数不是布尔值的情况下，逻辑与操作就不一定返

回布尔值；此时遵循以下规则：

如果第一个操作数是对象，则返回第二个操作数；

如果第二个操作数是对象，则只有在第一个操作数的求职结果为true的情况下才会返回该对象；

如果如果两个操作数都是对象，则返回第二个操作数；

如果有一个操作数是null，则返回null；

如果有一个操作数是NaN，则返货NaN；

如果有一个操作数是undefined，则返回undefined

逻辑与操作属于短路操作，即如果第一个操作数能够决定结果，那么就不会再对第二个操作数求值。如果第一个值是false，则就不会对第二个操作数求值。如果第一个操作数是true，则会继续对第二个操作数求值并判断结果。在逻辑与中操作中不要使用未定义的值。





逻辑或

逻辑或操作符由两个竖线符号（||）表示，有两个操作数



|  |  |  |  |
| --- | --- | --- | --- |
| 第一个操作数的值 | 第二个操作数的值 | 结果 | 说明 |
| true | true | true | 第一个操作数的值为true，直接返回结果，不会对第二个操作数进行求值 |
| true | false | true | 第一个操作数的值为true，直接返回结果，不会对第二个操作数进行求值 |
| false | true | true | 第一个操作数的值为false，继续对第二个操作数进行求值后判断结果 |
| false | false | false | 第一个操作数的值为false，继续对第二个操作数进行求值后判断结果 |

与逻辑与操作符相似，如果有一个操作数不是布尔值，逻辑或也不一定返货布尔值；此时遵循以下规则：

如果第一个操作数是对象，则返回第一个操作数；

如果第一个操作数的求值结果为false，则返回第二个操作数；

如果两个操作数都是对象，则返回第一个操作数；

如果两个操作数都是null，则返回null；

如果两个操作数都是NaN，则返会NaN；

如果两个个操作数是undefined，则返回undefined

逻辑或操作符也是属于短路操作符。也就是如果第一个操作数的求值结果为true，就不会对第二个操作数求值。逻辑或操作是

不要使用为定义的操作数





### 乘性操作符

ECMAScript中定义了3个乘性操作符：乘法、除法、求模。操作数未非数值的情况下会执行自动的类型转换。如果参与乘法计算的某个操作数不是数值，后台会先使用Number()转型函数将其转换为数值。

乘法

乘法操作符由一个星号（\*）表示，用于计算两个数值的乘积。



在处理特殊值的情况下，乘法操作符遵循以下规则：

如果操作数都是数值，执行常规的乘法计算。如果乘积超出了ECMAScript数值的表示范围，则返回Infinity或-Infinity

如果有一个操作数是NaN，则结果是NaN；

如果是Infinity与0相乘，则结果是NaN；

如果是Infinity与非0数值相乘，则结果是Infinity或-Infinity；

如果是Infinity与Infinity相乘，则结果是Infinity；

如果有一个操作数不是数值，则在后台调用Number()将其转换为数值，再进行计算。

除法

除法操作符是由一个斜线符号（/）表示，执行第二个操作数除第一个操作数的计算。



除法操作符在对特殊的值也有特殊的处理规则：

如果操作数都是数值，执行常规的除法计算。如果除积超出了ECMAScript数值的表示范围，则返回Infinity或-Infinity

如果有一个操作数是NaN，则结果是NaN；

如果是Infinity被Infinity除，则结果是NaN；

如果是零被零除，则结果是NaN；

如果是非零的有限数被零除，则记过是Infinity或-Infinity；

如果是Infinity被任何非零数值除，则结果是Infinity或-Infinity

如果有一个操作数不是数值，则后台调用Number()将其转换为数值，再执行计算。

求模

求模（余数）操作符由一个百分号（%）表示



求模操作符在处理特殊值的时候，会遵循以下规则：

如果操作数都是数值，则执行常规的除法计算，返回除得的余数；

如果被除数是无穷大值而除数是有限大的数值，则结果为NaN；

如果被除数是有限大的数值而除数是零，则结果为NaN；

如果是Infinity被Infinity除，则结果为NaN；

如果被除数是有限大的数值而除数是无穷大的数值，则结果是被除数；

如果被除数是零，则结果是零；

如果有一个操作数不是数值，则在后台调用Number()将其转换为诶数值，在进行计算。

### 加性操作符

加性操作符和乘性操作符类似，也会在后台转换不同的数据类型。

加法

加法操作符使用加号（+）表示



如果两个操作符都是数值，执行常规的加法计算，然后根据下列规则返回结果：

如果有一个操作数是NaN，则结果是NaN；

如果是Infinity加Infinity，则结果是Infinity；

如果是-Infinity加-Infinity，则结果是-Infinity；

如果是Infinity加-Infinity，则结果是NaN；

如果是+0加+0，则结果是+0；

如果是-0加-0，则结果是-0；

如果是+0加-0，则结果是+0

如果有一个操作数是字符串，那么就要应用如下规则：

如果两个操作数都是字符串，则将第二个操作数与第一个操作数拼接起来；

如果只有一个是操作数是字符串，则加另一个操作数转换为字符串，然后再将两个字符串拼接起来；

如果有一个操作数是对象、数值、布尔值，则调用toString()方法取得相应的字符串值，然后再应用前面关于字符串的规则，对

undefined和null，则分别调用String()函数并取得字符串”undefined”和”null”。

减法

减法操作符使用减号（-）来表示。

遵循的规则如下：

如果两个操作数都是数值，则执行城轨的算术减法操作并返回结果；

如果有一个操作数是NaN，则结果是NaN；

如果是Infinity减Infinity，则结果是NaN；

如果是-Infinity减-Infinity，则结果是NaN；

如果是Infinity减-Infinity，则结果是Infinity；

如果是-Infinity减Infinity，则结果是-Infinity；

如果是+0减+0，则结果是+0；

如果是+0减-0，则结果是-0；

如果是-0减-0，则结果是+0；

如果有一个操作数是字符串、布尔值、null、undefined，则先后台调用Number()函数将其转换为数值，然后再根据前面的

规则执行减法计算。如果转换的结果是NaN，则减法的结果就是NaN;

如果有一个操作数是对象，则调用对象的valueOf()方法以取得表示该对象的数值，如果得到的值是NaN，则减法的结果是

NaN。如果对象没有valueOf()方法，则调用toString()方法并将得到的字符串转换为数值

### 关系操作符

关系操作符包括：小于（<）、大于（>）、小于等于（<=）、大于等于（>=），用于对两个值进行比较。这几个操作符返回一个布尔值。

当关系操作符的操作数使用了非数值时，也要进行数据转换或完成某些奇怪的操作，并遵循以下规则：

如果两个操作数都是数值，则执行数值比较；

如果两个操作数都是字符串，则比较两个字符串对应的字符编码值。

如果一个操作数是数值，则将另一个操作数转换为一个数值，然后执行数值比较。

如果一个操作数是对象，则调用这个对象的valueOf()方法，用得到的结果按照前面的规则执行比较。如果对象没有valueOf()

方法，则调用toString()方法，并用得到的结果根据前面的规则执行比较

如果一个操作数是布尔值，则先将其转换为数值，然后再执行比较。

### 相等操作符

相等和不相等

ECMAScript中的相等操作符由两个等号（==）表示，如果两个操作符相等在，则返回true。而不相等操作符由叹号后面跟等于

号（!=）表示，如果两个操作数不相等，则返回true。这两个操作符都会先转换操作数（强制转换），然后再比较它们的相等性。

在转换不同的数据类型时，相等和不相等操作符遵循下列基本规则：

如果有一个操作数是布尔值，则在比较相等性之前先将其转换为数值—false转换为0，而true转换为1；

如果一个操作数是字符串，另一个操作数是数值，在比较相等性之前先将字符串转换为数值；

如果一个操作数是对象，另一个操作数不是，则调用对象的valueOf()方法，用得到的基本类型值按照签名档额规则进行比

较；

这两个操作符在进行比较时遵循下列规则：

null和undefined是相等的；

要比较相等性之前，不能将null和undefined转换成其他任何值。

如果有一个操作数是NaN，则相等操作符返回false，则不相等操作符则返回true。NaN不等于NaN

如果两个操作数都是对象，则比较它们是不是相同一个对象。如果两个操作数都指向同一个对象，则相等操作符返回true，

否则返回false。

特殊情况及比较结果：

|  |  |
| --- | --- |
| 表达式 | 值 |
| null == undefined | true |
| “NaN” == NaN | false |
| 5 == NaN | false |
| NaN == NaN | false |
| NaN != NaN | true |
| false == 0 | true |
| true == 1 | true |
| true == 2 | false |
| undefined == 0 | false |
| null == 0 | false |
| “5” == 5 | true |

### 全等和不全等

全等和不全等与相等和不相等基本类型，只是全等和不全等还要比较操作数之间的数据类型是否相等。全等操作符由3个等号（===）表示，它只在两个操作数未经转换就相等的情况下返回true



不全相等操作符由一个叹号后跟两个等号（!==）表示，它在两个操作数未经转换就不相等的情况下返回true。



### 条件操作符

条件操作符是ECMAScript中最灵活的一种操作符，通过表达式得到布尔值来判定个变量赋那个值。

语法格式：



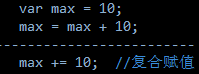
说明：如果（num1>num2）返回true，则将num1赋值给max。如果（num1>num2）返回false，则将num2赋值给max。

### 赋值操作符

简单的赋值操作符由等于号（=）表示，其作用就是把右侧的值赋给左侧的变量。



如果在等于号前面再添加乘性操作符、加性操作符、位操作符，就可以完成复合赋值操作



每个主要算术操作符都有对应的复合赋值操作符

乘/赋值（\*=）；

除/赋值（/=）；

模/赋值（%=）；

加/赋值（+=）；

减/赋值（-=）；

左移/赋值（<<=）；

有符号右移/赋值（>>=）；

无符号右移/赋值（>>>=）。

### 逗号操作符

使用逗号操作符可以在一条语句中执行多个操作

同时定义多个变量并赋值



逗号操作符还可以用于赋值，逗号操作符总会返回表达式中的最后一项



## 语句

### if语句

用于根据表达式返回的布尔值，来判断是否执行语句。

语法格式：

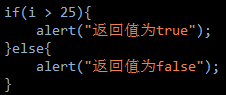


圆括号中的表达式返回为boolean值，返回true，则执行花括号中的代码。

### if…..else….语句

用于与if语句类型，只是多了一个执行代码块

语法格式：



说明：如果(i>25)表达式的值为true，则只执行第一个花括号中的代码。如果(i>25)表达式的值为false，则只执行第二个花括号中的代码。

### for循环语句

for循环语句，用于判定条件表达式中的结果来重复执行语句。

语法格式：



说明：其中(var i=0;i<10;i++)条件表达式中，定义变量i并初始化为0后判断i是否小于10，返回值为true，则执行花括号中

的代码后i自加1。然后再判断自加1后得到的值在判断语句中否返回true，继续执行花括号中的代码，直到判断语句返回false。则跳出for循环语句。

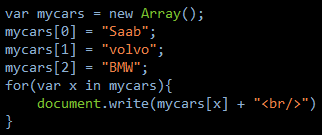
### for…in…语句

for…in语句是一种精准的迭代语句，可用于遍历数组或者对象的属性（对数组或者对象的属性进行循环操作）。

语法格式：



使用for…in循环遍历数组，其中变量X用于存储数组的下标值



使用for…in循环遍历对象属性，其中变量x用于存储对象的属性



说明：如果循环的对象的变量值为null或undefined，for…in语句会抛出错误，所以建议在使用for…in循环之前，先检测确认该对象的值不是null或undefined。

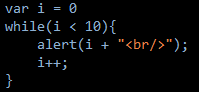
### while循环语句

while语句属于前测试循环语句，在循环体内的代码执行之前，就会对出口条件求值。因此，循环体内的代码有可能永远不会被执行。

语法格式



实例：



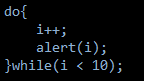
说明：在使用while循环语句的时候一定要注意条件表达式中的所用变量的值，如果条件表达式返回的值永远的true，则循环永远不

会停止，最终导致浏览器崩溃。

### do…while循环语句

do…while循环语句是while语句的一种变体，是一种后测试循环语句。在检测条件是否成立之前会先执行一遍循环体内的代码。

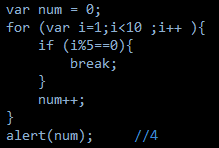
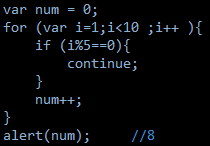
语法格式



### break和continue语句

break和continue语句用于在循环中精确地控制代码的执行。其中break语句会立即退出循环，强制继续执行循环后面的语句。而continue语句虽然也是立即退出循环，但退出循环后会从循环的顶部继续执行。

break用于跳出整个循环语句，继续执行循环语句后面的代码。而continue用于跳出本次循环，继续执行下次循环。

### label语句

使用label语句，可以在代码中添加标签，以便将来使用。

### with语句

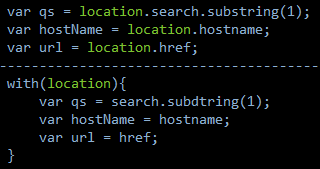
with语句的作用是将代码的作用域设置到一个特定的对象中。

定义with语句的目的主要是为了简化多次编写同一个对象的工作。

语法格式：



实例：



使用with语句关联了location对象。这一位置在with语句的代码块内部，每个变量首先被认为是一个局部变量，而如果在局部环境中找不到该变量的定义，就会查询location对象中是否有同名的属性。如果发现了同名的属性，则以location对象属性的值作为变量的值。

严格模式下不允许使用with语句，否则将视为语法错误。

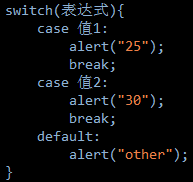
由于大量使用with语句会导致性能下降，同时也会给调试代码造成困难，因此在开发大型应用程序时，不建议使用with语句。

### switch语句

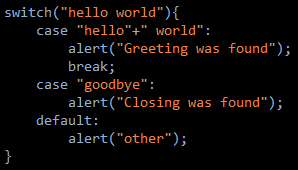
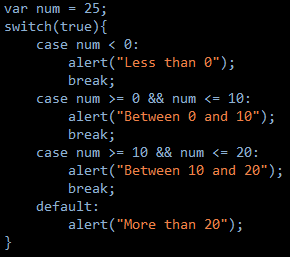
switch语句与if语句的关系最为密切，也是一种普遍使用的流程控制语句。

表达式中的值与case提供的值相等，则执行case后面的语句，其余的语句都不执行。

语法格式：



在ECMAScript中switch语句中可以使用任何数据类型，无论是字符串，还是对象。每个case的值不一定是常量，可以是变量，甚至是表达式。

switch语句在比较值时使用的是全等操作符，因此不会发生类型转换。

## 引用类型

引用类的值（对象）是引用类型的一个实例。在ECMAScript中，引用类型是一种数据结构，用于将数据和功能组织在一起。新对象是使用new操作符后跟一个构造函数来创建的。构造函数本身就是一个函数，只不过该函数是出于创建新对象的目的而定义的。

### Object类型

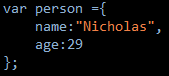
到目前为止，我们看到的大多数引用类值都是Object类型的实例；而Object也是ECMAScript中使用最多的一个类型。Object的实例在应用程序中存储和传输数据是非常理想的选择。

创造Object实例有两种方式：1.使用new操作符和Object构造函数。2.使用字面量

使用构造函数



使用字面量



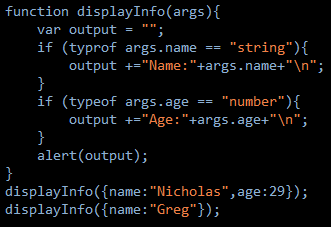
使用字面量的方式创建的对象，避免了使用构造函数创建包含大量属性的对象的过程。

对象的属性名称都是字符串类型，即使是定义了数值类型，后台也会自动将其转换为字符串类型。

使用字面量的方式创建字符串，如果留空其花括号，则可以定义只包含默认属性和方法的对象，与使用构造函数创建的对象相

同。

在通过对象字面量定义对象时，实际上不会调用Object构造函数。对象字面量也是向函数传递大量可选参数的首选方式



访问对象属性

在ECMAScript中访问对象的属性有两种方法：1.使用一般的点表示法。2.使用方括号表示法

使用方括号表示法，属性名必须是以字符串的形式放在方括号中。  
 使用方括号表示法的有点事可以通过变量来访问属性



如果变量名中包含会导致语法错误的字符或者属性名使用了关键字或保留字，则可以使用方括号表示法来避免报错。



不过一般情况下我们还是使用点表示法。

### Array类型

Array类型也是在ECMAScript中最常用的类型，与其他语言不同。ECMAScript中的数组里的每一项可以保存任何类型的数据，数组的大小是可以随着数据的增加动态调整的其数组的长度以容纳新增的数据。

创建数组的方式有两种：1.使用Array构造函数。2.使用字面量表示

构造函数



如果Array()函数中的参数为数值，则表示需要设置长度为几的数组。



如果直接将数组中每项的值通过参数传递进去，则每项值必须要以字符串的形式传递。



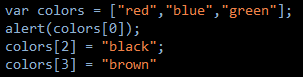
数组字面量表示法



使用数字字面量表示法时，也不会调用Array构造函数

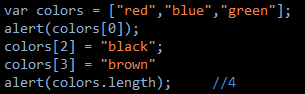
读取数组中的值

在读取和设置数组值时，使用方括号并提供相应值的下标值。第一项的下标值为0



设置数组的值也可以使用方括号表示法来设置，如果之前对的索引有值，则直接替换之前的值。

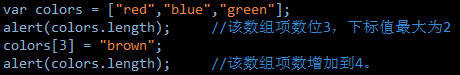
数组的项数保存在length属性中，这个属性始终会返回0或更大的值。



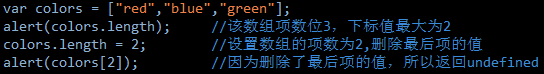
增加和减少数组的项

使用方括号表示法可以实现增加和减少数组的项数和对应的值

增加数组的项数和值



减少数组的项数



增加和删除数组的项数和值，都是在数组的最后面（右面）操作的，

检测数组

我们知道检测值是否为基本数据类型，可以使用typeof操作符来实现。但是我们在检测引用数据类型的使用就无法使用typeof

操作符了。为此我们可以使用instanceof操作符来检测值是否为引用数据类型。使用instanceof始终返回一个boolean值。

语法格式：



instanceof有一个问题在于，如果在同一个全局执行环境中使用instanceof是没有问题。但是如果处于多个全局执行环境中是不

能使用instanceof操作符（数组支持框架之间传递值，但是每个框架都有独自的全局执行环境，每个框架中Array构造函数是

不一样的。所以在处于多个框架中是不能使用instanceof操作符实现跨框架检测引用数据类型）

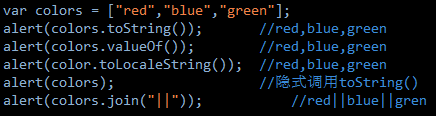
在ECMAScript中新增了Array.isArray()方法，用于检测某个值是不是数据，而不管它是在哪个全局执行环境中创建的。Array.isArray()始终返回一个boolean值。



数组转换成字符串的方法

所有对象都具有toLocaleString()、toString()、valueOf()方法。此3个方法默认的是以逗号分隔符将数据的每项拼接成字符串表

示。如果需要使用不同的分隔符来拼接，则可以使用join()方法，此方法只接收一个参数，即分隔符。



如果数组中的某一项的值是null或者undefined，则该值在join(0、toLocaleString()、toString、valueOf()返回的结果中以空字

符串表示。

栈方法（后进先出）

ECMAScript中也为数组提供了像数组行为类似于其他数据结构的方法（栈方法）。栈是一种数据结构，最新添加的项最早被移

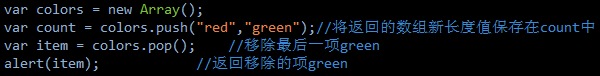
除。而栈中项的插入和移除，值发生在一个位置-----栈的顶部。为此ECMAScript为数组专门提供了push()、pop()方法，以便

实现类似栈的行为。

push()：该方法可以接收任意数量的参数，把他们逐个添加到数组的末尾，并返回修改后数组的长度。



pop()：该方法从数组末尾移除最后一项，减少数组的length值，然后返回被移除的项

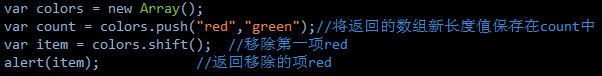


队列方法（先进先出）

栈数据结构的访问规则是后进先出，则对列数据结构的访问规则是先进先出。队列在列表的末端添加项，从列表的前端移除

项。使用push()方法实现在队列末端添加项并返回新数组的长度。使用shift()方式实现移除数组中第一个项并返回该项，同时

将数组长度减1。



unshift()：该方法实现了在数组前端添加任意多个项，并返回新数组的长度值。可以与pop()方法一起实现先进后出的数据结构



数组的重排序方法

数组中提供了2个方法用来重新排序：reverse()、sort()。

reverse()：用于将数组原有的排序反转。



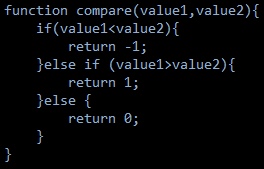
sort()：默认情况下sort()方法实现数据的升序排列。sort()方法会调用每个数组项的toString()转型方法，然后比较得到的字符

串，以确定如何排序。如果数组中的每一项都是数值，sort()方法比较的也是字符串。



因此sort（）方法可以接收一个比较函数作为参数，以便我们指定那个值位于哪个值的前面。其中比较函数接收两个参数，如果第一个参数应该位于第二个之前，则返回一个负数。如果两个参数相等，则返回0.如果第一个参数应该位于第二个参数之后，则返回一个正数。

简单的比较函数：



这个比较函数可以适用于大多数数据类型，只要将其作为参数传递给sort()方法即可。



如果想实现降序，只需要交换比较函数返回的值即可。

reverse()和sort()方法的返回值是已经排序之后的数组。

对于数值类型或者其valueOf()方法返回数值类的对象类型，可以使用一个更简单的比较函数。这个函数只要用第二个值

减第一个值即可。

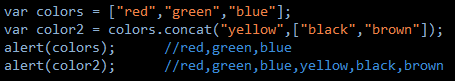


数组的操作方法

ECMAScript为操作已经包含在数组中的项提供了很多方法。concat()、slice()、splice()

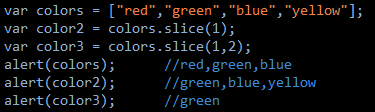
concat()：基于当前数组中的所有项创建一个新数组。此数组为当前数组的副本，可以通过传递的参数来增加数组中的项。参数

可以是一个或多个数组，也可以是值



slice()：基于当前数组中的一个或多个项创建一个新数组。此方法可以接收一个或两个参数，即要返回项的开始和结束位置（不

包含结束项）。不会影响原始数组。传递的参数为原始数组的下标值



如果slice()方法的参数有一个负数，则用数组长度加上该数来确定项的位置。例如：数组长度为5，slice(-2,-1)与slice(3,4)

得到的结果一样。如果结束位置小于起始位置，则返回空数组。

splice()：该函数主要用途是向数组的中部插入项，但是使用这种方法的方式有3中情况：

删除：可以删除任意数量的项，只需要指定2个参数（要删除的第一项的位置和要删除的项数）。例如：splice(0,2)删

除数组的前两项。

插入：可以向指定位置插入任意数量的项，只需要提供3个参数（起始位置、要删除的项数（此项为0）、要插入的项）。

例如：splice(2,0,”red”,”green”)

替换：可以向指定位置插入任意数量的项，且同时删除任意数量的项，只需要指定3个参数（起始位置、要删除的项

数、要插入任意数的项）。例如： splice(2,1,”red”,”green”)splice()方法始终返回一个数组，这个数组中的项就是

删除原有数组中的项。



数组项位置查找方法

ECMAScript5为数组实例添加了两个位置方法：indexOf()、lastIndexOf()

indexOf()：接收两个参数（要查找的项和查找的起点位置（可选）），此方法从数组的开头（位置0）开始向后查找。在计较查

找的时候使用的是全等操作符，要求查找的项必须严格相等，如果没找到，则返回-1。

lastIndexOf()：接收两个参数（要查找的项和查找的起点位置（可选）），此方法从数组的末尾开始向前查找。在计较查找的时

候使用的是全等操作符，要求查找的项必须严格相等，如果没找到，则返回-1。

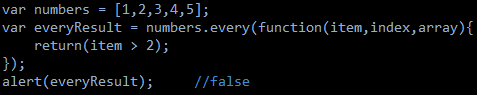


迭代方法

ECMAScript5为数组定义了5个迭代方法。每个方法都接收两个参数：要在每一项上运行的函数和（可选）运行该函数的作用域对象---影响this的值。传入这些方法中的函数会接收三个参数：数组项的值、该项在数组中的位置、数组对象本身

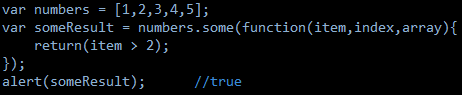
every()：对数组中的每一项运行与给定的函数，如果该数组中的每一项在给定函数中返回的结果都返回true，则返回true，只

要有一项返回false，则整个every()方法返回false。

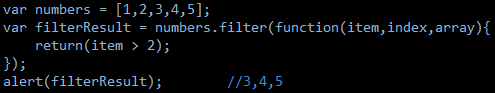


some()：这个方法与every()类似，都是用于查询数组中的项是否符合函数指定的函数表达式。不同的在用some()方法只要函

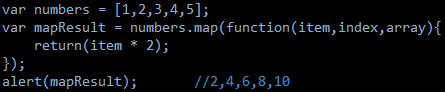
数中有一项满足函数表达式，则整个方法返回true



filter()：该方法用于将数组中的每一项运行与指定函数中，并返回一个数组（该数组的每项是原数组中符合函数表达式中的项）。

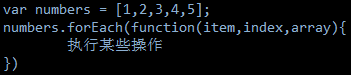


map()：该方法也是返回一个数组。但是返回的数组中的项是原数组中的每项通过指定的函数表达式运算后的结果。



forEach()：这个方法只是对数组中的每一项按照指定的函数表达式运行，但是不会返回任何值。本质上与使用for循环迭代数

组一样。



缩小方法

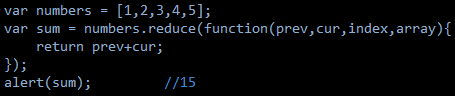
ECMAScript5中新增了两个缩小数组的方法：reduce()、reduceRight()。这两个方法都会迭代数组的所有项，然后构建一个最终

返回的值。

reduce()：此方法从数组的第一项开始，逐个便利到最后。该方法接收两个参数（一个在每一项上调用的函数和（可选）作为缩

小基础的初始值）。作为参数的函数接收4个参数（前一个值、当前值、项的索引、数组对象）这个函数返回的任何值

都会作为第一个参数自动传给下一项。第一次迭代发生在数组的第二项上，因此第一个参数是数组的第一项，第二个参数就是数组的第二项使用reduce()方法可以执行求数组中所有值之和的操作：



说明：第一次执行回调函数，prev是1，cur是2。第二次，prev是3（1加2的结果），cur是3（数组的第三项），

直到每项都访问一遍，最后返回结果

reduceRight()：该方法与reduce()方法类似，只是访问的方向相反。

### Date类型

ECMAScript中的Date类型使用自UTC 1970年1月1日零时开始经过的毫秒数来保存日期。使用这种数据存储格式的条件下，Date类型保存的日期能够精确到1970年1月1日之前或之后的285616年

创建日期对象

使用new操作符调用Date构造函数



在调用Date构造函数而不传递参数的情况下，新创建的对象会自动获取当前日期和时间。如果想根据特定的日期和时间创建日

期对象，必须传入表示该日期的毫秒数（从1970年1月1日到该日期止进过的毫秒数）为此ECMAScript提供了两个方法：

Date.parse()、Date.UTC()

Date.parse()：此方法接受一个表示日期的字符串参数，然后尝试根据这个字符串返回相应日期的毫秒数。ECMA-262没有定义

Date.parse()应该支持那种日期格式，所以因具体情况而定。

日期格式：

“月/日/年”，如：6/13/2004；

“英文月名 日,年”，如：January 12,2004；

“英文星期几 英文月名 日 年 时:分:秒 时区”，如：Tue May 25 2004 00:00:00 GMT-0700；

ISO 8601扩展格式YYYY-MM-DDTHH:mm:ss.sssZ（例如：2004-05-25T00:00:00）。只有兼容ECAMScript5的实

现支持这种格式。



如果传入Date.parse()方法的字符串不能表示日期，则返回NaN。如果直接将表示日期的字符串传递给Date构造函

数，也会在后台调用Date.parse()。

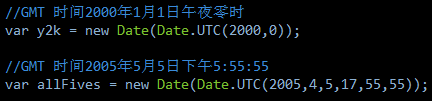


Date.UTC()：该方法也会返回表示日期的毫秒数，但是与Date.parse()在构建值时使用不同的信息。该方法的参数分别是：年份、

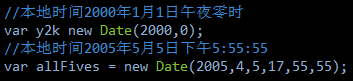
基于0的月份（一月是0，二月是1）、月中的那一天（1到31）、小时（0到23）、分钟、秒、毫秒数。在这些参

数中只有前两个参数（年和月）是必须的。如果没有提供月中的那一天，则假设天数为1；如果省略其他参数，则

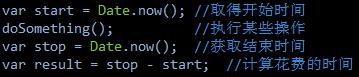
统统假设为0。



同样如果Date.UTC()方法的参数直接传递给Date构造函数，则Date()构造函数也会在后台调用Date.UTC()。但是不同的是Date构造函数接收的Date.UTC()方法的参数时日期和时间是基于本地时区而非GMT来创建。



Date.now()：该方法用于返回调用该方式时的日期和时间的毫秒数。



Date实例中继承的方法

与其他引用类型一样，Date类型也重写了toLocaleString()、toString()、valueOf()。

toLocaleString()方法会按照与浏览器设置的地区相适应的格式返回日期和时间，格式中会包含AM或PM，但是不包含时区信息。

toString()方法通过返回带有时区信息的日期和时间，起哄时间一般以军用时间（小时范围0到23）表示。

valueOf()方法不会返回字符串，而是返回日期的毫秒表示。因此可以用于比较日期值。

日期的格式化

Date类型定义了专门用于将日期格式化为字符串的方法

toDateString()：以特定于实现的格式显示星期几、月、日和年；

toTimeString()：以特定于实现的格式显示时、分、秒和时区；

toLocaleDateString()：以特定于地区的格式显示星期几、月、日和年；

toLocaleTimeString()：以特定于实现的格式显示时、分、秒；

toUTCString()：以特定于实现的格式完整的UTC日期。

日期/时间组件方法

|  |  |
| --- | --- |
| 方法 | 说明 |
| getTime() | 返回表示日期的毫秒数；与valueOf()方法返回的值相同 |
| setTime(毫秒) | 以毫秒数设置日期，会改变整个日期 |
| getFullYear() | 取得4位数的年份 |
| getUTCFullYear() | 返回UTC日期的4位数年份 |
| setFullYear(年) | 设置日期的年份。传入的年份值必须是4位数字 |
| setUTCFullYear(年) | 设置UTC日期的年份。传入的年份值必须是4为数字 |
| getMonth() | 返回日期中的月份，其中0表示一月，11表示十二月 |
| getUTCMonth() | 返回UTC日期中的月份，其中0表示一月，11表示十二月 |
| setMonth(月) | 设置日期的月份。传入的月份值必须大于0，超过11则增加年份 |
| setUTCMonth(月) | 设置UTC日期的月份。传入的月份值必须大于0，超过11则增加年份 |
| getDate() | 返回日期月份中的天数（1到31） |
| getUTCDate() | 返回UTC日期月份中的天数（1到31） |
| setDate(日) | 设置日期月份中的天数。如果传入的值超过了该月中应用的天数，则增加月份 |
| setUTCDate(日) | 设置UTC日期月份中的天数。如果传入的值超过了该月中应用的天数，则增加月份 |
| getDay() | 返回日期中星期的星期几（其中0表示星期日，6表示星期六） |
| getUTCDay() | 返回UTC日期中星期的星期几（其中0表示星期日，6表示星期六） |
| getHours() | 返回日期中的小时数（0到23） |
| getUTCHours() | 返回UTC日期中的小时数（0到23） |
| setHours(时) | 设置日期中的小时数。传入的值超过了23则增加月份中的天数 |
| setUTCHours(时) | 设置UTC日期中的小时数。传入的值超过了23则增加月份中的天数 |
| getMinutes() | 返回日期中的分钟数（0到59） |
| getUTCMinutes() | 返回UTC日期中的分钟数（0到59） |
| setMinutes(分) | 设置日期中的分钟数。传入的值超过59则增加小时数 |
| setUTCMinutes(分) | 设置UTC日期中的分钟数。传入的值超过59则增加小时数 |
| getSeconds() | 返回日期中的秒数（0到59） |
| getUTCSeconds() | 返回UTC日期中的秒数（0到59） |
| setSeconds(秒) | 设置日期中的秒数。传入的值超过了59会增加分钟数 |
| setUTCSeconds(秒) | 设置UTC日期中的秒数。传入的值超过了59会增加分钟数 |
| getMilliseconds() | 设置日期中的毫秒数 |
| getUTCMilliseconds() | 设置UTC日期中的毫秒数 |
| setMilliseconds(毫秒) | 设置其中的毫秒数 |
| setUTCMilliseconds(毫秒) | 设置UTC其中的毫秒数 |
| getTimezoneOffset() | 返回本地时间与UTC时间相差的分钟数。例如，美国东部标准时间返回300，在某地进入夏令时的情况下，这个值会有所变化 |

说明：UTC日期指的是在没有时区偏差的情况下（将日期转换为GMT时间）的日期值

### RegExp类型

ECMAScript通过RegExp类型来支持正则表达式。当您检索某个文本时，可以使用一种模式来描述要检索的内容。RegExp 就是这种模式。RegExp 对象用于存储检索模式。

创建RegExp对象

使用字面量法：



其中‘正则表达式’部分可以是任何简单或复杂的正则表达式，可以包含字符类、限定符、分组、向前查找、反向引用。

每个正则表达式都可以带有一个或多个标志，用以标明正则表达式的行为

其中正则表达式的匹配模式支持下列3个标志：

g：表示全局模式，即模式将被应用于所有字符串，而非在发现第一个匹配项时立即停止。

i：表示不区分大小写模式，即在确定匹配项时忽略模式与字符串的大小写，遇到第一个匹配的就立即停止。

m：表示多行模式，即在到达一行文本末尾时还会继续查询下一行中是否存在与模式匹配的项。



以上3个标志可以同时一起使用

在正则表达式中使用的所有元字符都必须转义。元字符包括

( [ { \ ^ $ | } ? \* + . ] )

使用RegExp构造函数

使用RegExp构造函数创建正则对象，构造函数需要接收两个参数：一个是要匹配的字符串模式，一个是可选择的标志字

符串。



RegExp实例属性

global：布尔值，表示是否设置了g标志。

ignoreCase：布尔值，表示是否设置了i标志。

lastIndex：整数，表示开始搜索下一个匹配项的字符位置，从0算起。

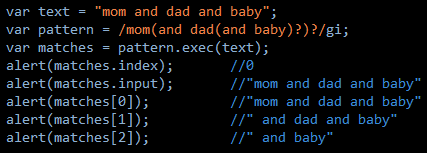
multiline：布尔值，表示是否设置了m标志。

source：正则表达式的字符串表示，按照字面量形式而非传入构造函数中的字符串模式返回。

RegExp实例方法

exec()：该方法为RegExp对象的主要方法，专门为捕获组而设计的。exec()接收一个参数，即要应用模式的字符串，然后返回

包含第一个匹配项信息的数组；或者没有匹配项的情况下返回null。返回的数组虽然是Array的实例，但是包含两个额外的属性：index和input。其中index表示匹配项在字符串中的位置，而input表示应用正则表达式的字符串。在数组中，，第一项是与整个模式匹配的字符串，其他项是与模式中的捕获组匹配的字符串（如果模式中没有捕获组，则该数组只包含一项）



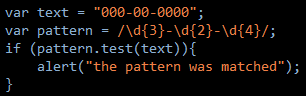
对于exec()方法，即使在模式中设置了全局标志（g），它每次也只会返回一个匹配项。在不设置全局标志的情况下，在

同一个字符串上多次调用exec()将始终返回第一个匹配项的信息。而在设置了全局标志的情况下，每次调用exec()则都

会在字符串中继续查找新匹配项。

test()：该方法接收一个字符串参数。在模式与该参数匹配的情况下返回true，否则返回false。在只想知道目标字符串与某个模

式是否匹配，但不需要知道其文本内容的情况下，使用这个方法非常方便。因此test()方法经常被用在if语句中。



RegExp实例继承的toLocaleString()和toString()方法都会返回正则表达式的字面量，与创建正则表达式的方式无关。valueOf()

方法返回正则表达式本身。

RegExp构造函数属性

|  |  |
| --- | --- |
| 长属性名 | 说明 |
| input | 最近一次要匹配的字符串 |
| lastMatch | 最近一次的匹配项 |
| lastParen | 最近一次匹配的捕获组 |
| leftContext | input字符串中lastMatch之前的文本 |
| multiline | 布尔值，表示是否所有表达式都使用多行模式 |
| rightContext | input字符串中lastMatch之后的文本 |

### Function类型

ECMAScript中每个函数都是对象，都是Function类型的实例，而且都与其他引用类型一样具有属性和方法。因为每个函数是对象，因此函数名实际上也是一个指向函数对象的指针，不会与某个函数绑定。所以将一个函数作为值复制给一个变量，相当于给该函数增加了一个指向该函数的别名。

创建函数

函数声明：



函数表达式：表达式末尾有一个分号



使用Function构造函数（不推荐）



没有重载

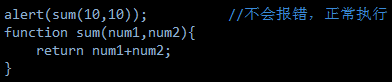
ECMAScript中的函数是没有重的，同时定义了多个同名的函数，最后定义的函数将覆盖之前定义的函数。

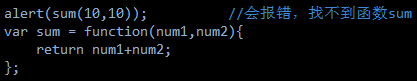
函数声明和函数表达式的区别

在同一个文档中，解析器在向执行环境中加载数据时，解析器会率先读取函数声明，并使其在执行任何代码之前可用（可以访

问）；而函数表达式定义的函数，只有解析器执行到它所在的代码行，才会真正被解析执行。除此之外，两种方式定义的函数都

是一样的。





函数作为值使用

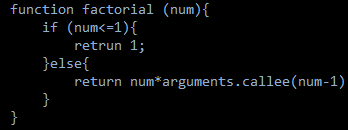
ECMAScript中的函数名本身就是变量，所以函数也可以作为值来使用。也就是说，不仅可以像传递参数一样把一个函数传递给

另一个函数，而且可以将一个函数作为另一个函数的结果返回

函数内部属性

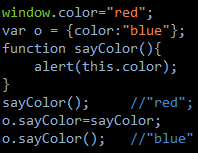
arguments：该属性是一个类数组对象，包含着传入函数中的所有参数，用于保存函数参数。arguments对象有一个名为callee

属性，该属性是一个指针，指向拥有这个arguments对象的函数。

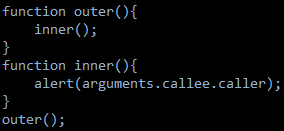


this：在函数中还有一个特殊对象this，该属性引用的是函数据以执行的环境对象----this值（当在网页的全局作用域中调用函数

时，this对象引用的就是window）



caller：该属性中保存着当前函数被调用的函数的引用，如果是在全局作用域中调用当前函数，它的值为null。

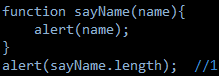


函数属性和方法

ECMAScript中所有的函数是对象，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。

length：表示函数希望接收的命名参数的个数，也就是在定义函数的时候预先定义的参数个数。



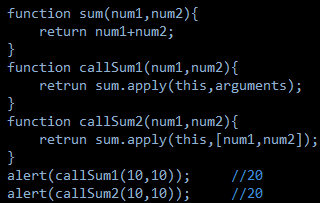
prototype：保存所有引用类型的实例方法。诸如：toString()、valueOf()等。prototype属性是不可枚举的。

每个函数都包含两个非继承而来的方法：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

函数内体this对象的值。

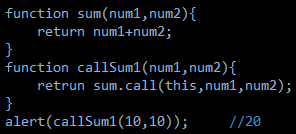
apply()：该方法接收两个参数，一个是在其中运行函数的作用域，另一个是参数数组（可以是Array的实例，也可以是

arguments对象）。

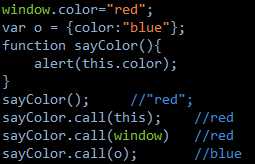


call()：该方法与apply()方法的作用相同，它们的区别仅在于接受参数的方式不同。对应call()方法，第一个参数是this值

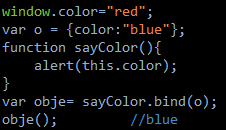
没有变化，变化的是其余参数都直接传递给函数。



apply()和call()这两个方法的真正强大之处在于能够扩充函数赖以运行的作用域，对象不需要与方法有任何耦合关系



bind()：该方法会创建一个函数的实例，其this值会被绑定到传给bind()函数的值。



每个函数继承的toLocaleString()、valueOf()和toString()方法始终都会返回函数的代码

### 基本包装类型

为了便于操作基本类型值，ECMAScript还提供了3个特殊的引用类型：Boolean、Number、String。

Boolean类型

Boolean类型是与布尔值对应的引用类型。要创建Boolean对象，可以调用Boolean构造函数并传入true或false值



Boolean类型的实例重写了valueOf()方法，返回基本类型值true或false；重写了toString()方法，返回字符串”true”和”false”。

基本类型与引用类型的布尔值有两个区别。首先typeof操作符对基本类型返回”boolean”，而对引用类型返回”object”。其次，

由于Boolean对象是Boolean类型的实例，所以用instanceof操作符测试Boolean对象返回true，而测试基本类型的布尔值返回false。

Numder类型

Numder类型是与数字值对应的引用类型，可以调用Number构造函数向其中传递相应的值来创建Number对象。



Numder类型也重写了valueOf()、toString()和toLocaleString()方法。valueOf()方法返回对象表示的基本类型的数值，toString()

和toLocaleString()方法则返回字符串形式的数值。toString()方法传递一个表示基数的参数，告知返回几进制数值的字符串形式。



toFixed()：该方法会按照指定的小数位返回数值的字符串表示，多的部分则四舍五入处理。

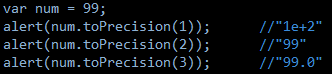


toExponential()：该方法返回以指数表示法表示的数值的字符串形式，该方法接收一个参数用于指定输出结果的小数位数。



toPrecision()：该方法可能返回固定大小格式，也可能返回指数格式的数值字符串表示。具体规则看那种格式合适。该方法接收

一个参数，表示数值的所有数字位数（不包含指数部分）。



String类型

String类型是字符串的对象包装类型，可以使用构造函数来创建String对象



String对象的方法也可以在所有基本的字符串值中访问到。其中继承的valueOf()、toString()和toLocaleString()方法，都返回对

象所表示的基本字符串值。String类型的每个实例都有一个length属性，表示字符串中包含多个字符。

字符方法

charAt()：该方法返回字符串中指定位置的字符



charCodeAt()：该方法返回字符串中指定位置字符对应的字符编码



使用方括号表示法返回指定位置的字符



字符串操作方法

concat()：用于将一个或多个字符串拼接起来，返回拼接得到的新的字符串。（与加号操作符一样）。

ECMAScript中还定义了截取原字符串，返回截取部分的新字符串的方法：slice()、substr()、substring()。其中slice()和

substring()类似，都是接收两个参数，第一个参数指定子字符串的开始位置，第二个参数指定结束位置。返回截取后的新字符串。而substr()则第一个参数指定子字符串的开始位置，第二个参数指定需要截取的字符串个数。返回截取后的新字符串。如果这三个方法传递的参数为负数，则slice()方法会将传入的负数与字符串长度相加，substr()方法将符的第一个参数加上字符串的长度，第二个负的参数转换为0 ，substring()方法会将所有的负值参数都转换为0

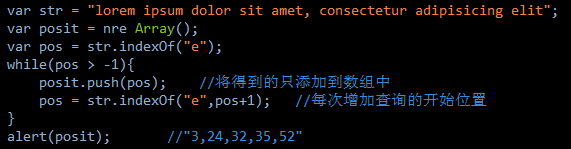
字符串位置方法

ECMAScript定义了两个用于查找子字符串的方法：indexOf()和lastIndexOf()。这两个方法都是行一个字符串中搜索给定的

子字符串，然后返回子字符串的位置（如果没有找到，则返回-1）。该方法接收两个参数，第一个为要查询的子字符串，第二个为查询的开始位置。

indexOf()：该方法会从字符串的开头向后搜索子字符串。

lastIndexOf()：该方法会从字符串的末尾开始向前搜索子字符串



trim()：该方法会创建一个字符串的副本，删除前置及后缀的所有空格，然后返回结果。

字符串大小写转换方法

ECMAScript中定义了4中大小写转换有关的方法，其中toLowerCase()和toUpperCase()方法是两个经典的方法。而

toLocaleLowerCase()和toLocaleUpperCase()方法则是针对特定地区的实现。根据地区的不一样而针对性的转换

字符串的模式匹配方法

String类型定义了几个用于在字符串中匹配模式的方法

match()：在字符串上调用这个方法，本质上与调用RegExp的exec()方法相同。该方法只接收一个参数，要么是一个正则

表达式，要么是一个RegExp对象。match()方法返回一个数组，数组的第一项是与整个模式匹配的字符串，之后的每一项保存着与正则表达式中的捕获组匹配的字符串。

search()：该方法只接收一个参数，要么是一个正则表达式，要么是一个RegExp对象。返回字符串中第一个匹配项的索引。

如果没有找到匹配，则返回-1。

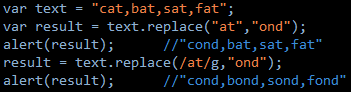
search()方法始终是从字符串开头向后查找模式。

replace()：该方法用于简化替换子字符串的操作。接收两个参数：第一个参数可以是一个RegExp对象或者一个字符串（这

个字符串不会被转换成正则表达式），第二个参数可以是一个字符串或者一个函数。如果第一个参数是字符串，

那么只会替换第一个子字符串。想要替换所有子字符串，唯一的办法就是提供一个正则表达式，而且要指定全局

（g）标志。此方法会改变元字符串

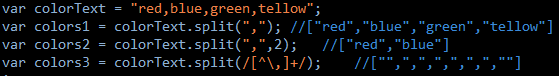


replace()方法的第二个参数也可以是一个函数。在只有一个匹配项（即与模式匹配的字符串）的情况下，会向这个函数传递3个参数：模式的匹配项、模式匹配项在字符串中的位置和原始字符串。在正则表达式中定义了多个捕获组的情况下，传递给函数的参数一次是模式的匹配项、第一个捕获组的匹配项、第二个捕获组的匹配项…，但是最后两个参数仍然分别是模式的匹配项在字符串中的位置和原始字符串。这个函数应该返回一个字符串，表示应该被替换的匹配项使用函数作为replace()方法的第二个参数可以实现更加精细的替换操作。



split()：该方法可以基于指定的分隔符将一个字符串分隔成多个子字符串，并将结果放在一个数组中。分隔符可以是字符串，

也可以是RegExp对象。该方法的第二个参数，用于指定数组的大小，以便确保返回的数组不会超过既定大小

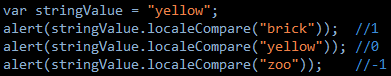


localeCompare()：该方法用于比较两个字符串，并返回下列值中的一个

如果字符串中字母表中应该排在字符串参数之前，则返回一个负数

如果字符串等于字符串参数，则返回0

如果字符串在字母表中应该排在字符串参数之后，则返回一个正数。



fromCharcode()：该方法为Strng构造函数本身的静态方法。该方法主要接收一个或多个字符编码，然后将它们转化成一个字

符串。与charCodeAt()方法执行相反的操作



### 单体内置对象

ECMAScript对内置对象的定义：由ECMAScript实现提供的、不依赖宿主环境的对象，这些对象在ECMAScript程序执行之前就已经存在了。不必显示的实例化内置对象。

Global对象

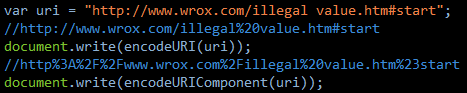
Global（全局）对象在某种意义上是作为一个终极的“兜底儿对象”来定义的。不属于任何其他对象的属性和方法，最终都是它

的属性和方法。

URI编码的方法

Global对象的encodeURI()和encodeURIComponent()方法可以对URI（通用资源标识符）进行编码，以便发送给浏览器。

它们用特殊的UTF-8编码替换所有无效的字符，从而让浏览器能够接受和理解。其中，encodeURI()用于整个URI（例如：http://www/wrox.com/illegal value.htm）,而encodeURIComponent()主要用于对URI中的某一段（例如：illegal value.htm）进行编码。它们的主要区别在于，encodeURI()不会对本身属于URI的特殊字符进行编码、例如冒号、正斜杠、问号和井号；而encodeURIComponent()则会对它们发现的任何非标准字符进行编码



与encodeURI()和encodeURIComponent()方法对应的两个方法分别数decodeURI()和decodeURIComponent()。其中

decodeURI()只能对encodeURI()替换的字符进行解码。encodeURIComponent()也只能对encodeURIComponent()编码的所有字符进行解码

eval()方法

该方法像一个完整的ECMAScript解析器，它只接收一个参数，即要执行的字符串



当解析器发现代码中调用eval()方法时，它会将传入的参数当作实际的ECMAScript语句来解析，然后把执行结果插入到原

位置。通过eval()执行的代码被认为是包含该次调用的执行环境的一部分，因此被执行的代码具有与该执行环境相同的作

用域链。



在eval()中创建的任何变量或函数都不会被提升，因为在解析代码的时候，它们被包含在一个字符串中；它们只有在eval()\

执行的时候创建。

严格模式中，外部访问不到eval()中创建的任何变量或函数，为eval赋值也会导致错误。

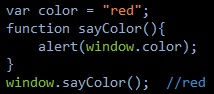
Global对象的属性

|  |  |  |  |
| --- | --- | --- | --- |
| 属性 | 说明 | 属性 | 说明 |
| undefined | 特殊值undefined | Date | 构造函数Date |
| NaN | 特殊值NaN | RegExp | 构造函数RegExp |
| Infinity | 特殊值Infinity | Error | 构造函数Error |
| Object | 构造函数Object | EvalError | 构造函数EvalError |
| Array | 构造函数Array | RangeError | 构造函数RangeError |
| Function | 构造函数Function | ReferenceError | 构造函数ReferenceError |
| Boolean | 构造函数Boolean | SyntaxError | 构造函数SyntaxError |
| String | 构造函数String | TypeError | 构造函数TypeError |
| Number | 构造函数Number | URIError | 构造函数URIError |

window对象

ECMAScript虽然没有指出如何直接访问Global对象，但是Web浏览器都是这个全局对象作为window对象的一部分加以

实现的。因此，在全局作用域中声明的所有变量和函数，就都成为了window对象的属性。



Math对象

ECMAScript中Math对象用于保存数学公式和信息，相比较在JavaScript中直接编写的计算功能相比，Math对象提供的计算功

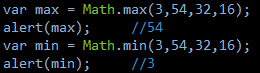
能执行起来要快得多。Math对象中还提供了辅助完成这些计算的属性和方法

Math对象的属性

|  |  |
| --- | --- |
| 属性 | 说明 |
| Math.E | 自然对数的底数，即常量e的值 |
| Math.LN10 | 10的自然对数 |
| Math.LN2 | 2的自然对数 |
| Math.LOG2E | 以2为底e的对数 |
| Math.LOG10E | 以10为底e的对数 |
| Math.PI | π的值 |
| Math.SQRT1\_2 | 1/2的平方根（即2的平方根的倒数） |
| Math.SQRT2 | 2的平方根 |

min()和max()方法

用于确定一组数值中的最小值和最大值。这连个方法可以接收任意多个数值参数



要找到数组中的最大值和最小值，可以使用apply()方法

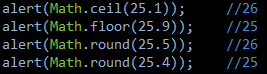


舍入方法

Math.ceil()：执行向上舍入，即它总是将数值向上舍入为最接近的整数；

Math.floor()：执行向下舍入，即它总是将数值向下舍入为最接近的整数；

Math.round()：执行标准舍入，即它总是将数值四舍五入为最接近的整数；



random()方法

Math.random()方法总是返回介于0和1之间一个随机数，不包括0和1。



其他方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 说明 | 方法 | 说明 |
| Math.abs(num) | 返回num的绝对值 | Math.asin(x) | 返回x的反正弦值 |
| Math.exp(num) | 返回Math.E的num次幂 | Math.atan(x) | 返回x的反正切值 |
| Math.log(num) | 返回num的自然对数 | Math.atan2(y,x) | 返回y/x的反正切值 |
| Math.pow(num,power) | 返回num的power次幂 | Math.cos(x) | 返回x的余弦值 |
| Math.sqrt(num) | 返回num的平方根 | Math.sin(x) | 返回x的正弦值 |
| Math.acos(x) | 返回x的反余弦值 | Math.tan(x) | 返回x的正切值 |

## 变量、作用域和内存问题

### 基本数据类型和引用数据类型的值

动态的属性

定义基本类型值和引用类型值的方式时类似的：创建一个变量并为该变量赋值。但对于引用类型的值，我们可以为其添加属性

和方法，也可以改变和删除其属性和方法。但是我们却不能为基本数据类型的值添加属性和方法。

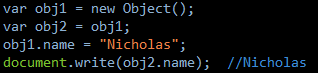
复制变量值

除了保存的方式不同之外，从一个变量向另一个变量复制值的时候，基本数据类型会在变量上创建一个新知，然后将该值复制

给新变量。两个变量的值相互之间是独立的，互不影响。但是在复制引用类型的值的时候，同样也会复制一份放到新变量分配

的空间中。但是这个值的副本实际上是一个指针，而这个指针指向存储在堆中的一个对象。所以两个变量实际上是指向内存中

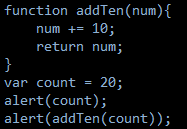
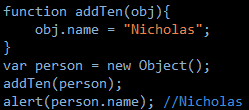
的同一个对象。如果改变其中一个变量，另一个变量也会发生变化。



传递参数

ECMAScript中所有函数的参数都是按照值传递的。也就是说，把函数外部的值复制给函数内部的参数，就和把值从一个变量

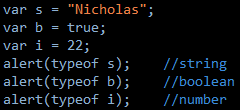
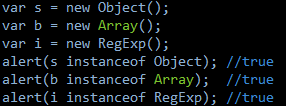
复制给另一个变量一样。基本数据类型在传递参数的时候是直接复制一份变量的值放到参数的空间中（互不影响），而引用类型的值在传递参数的时候是将对象的内存地址复制一份放到参数的空间中（同时指向内存中的一个变量）

检测类型

要检测一个变量是不是基本数据类型，可以使用typeof操作符。如果变量是一个对象或null则用typeof操作符检测会返回object。

在检测引用数据类型的变量的时候使用instanceof操作符。

### 执行环境及作用域链

执行环境

在ECMAScript中，执行环境定义了变量或函数是否有权访问其他数据，决定了它们各自的行为。在代码执行的时候，解析器会为每个执行环境创建一个变量对象（variable object），在执行环境中定义的变量和函数都是保存在这个对象中。该对象时解析器创建的，我们无法访问，但是解析器会在后台使用它。

执行环境分为：全局执行环境和局部执行环境。全局执行环境是最外围的执行环境。在web浏览器中，全局执行环境被认为是

window对象，所以在全局执行环境中定义的变量和函数，都是window对象的属性和方法。局部执行环境指的是定义的函数体，

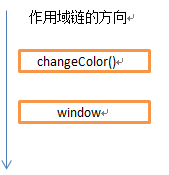
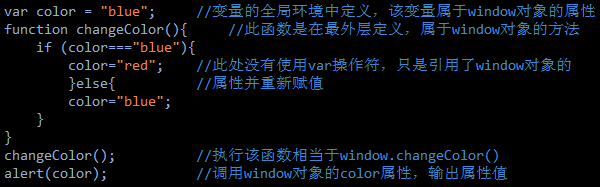
在函数体内部定义的变量和函数，都是属于该函数对象的属性和方法。（变量和属性只要在哪个执行环境中定义的，就属于那个

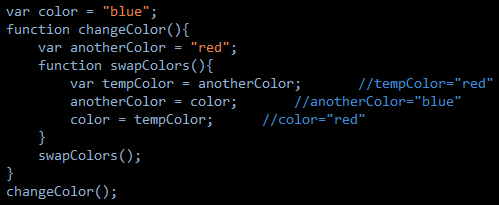
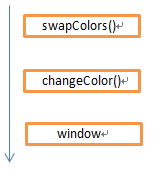
执行环境。却对赋值不限制）

作用域链

当代码在环境中执行的时候，会创建变量对象（variable object）的一个作用域链。用于执行环境中代码的有权及有序访问文档

中数据。作用域的前端是当前代码执行环境创建的变量对象（variable object）（如果该执行环境是一个函数，则将其活动对象作为变量对象，活动对象在最开始只包含一个变量，即arguments对象）。作用域中下一个变量对象是来自包含环境….以此类推，一直延续到全局执行环境，全局执行环境的变量对象始终都是最后一个对象。标识符解析就是这样一级一级的搜索的。（作用域限定了访问数据选择只能从内部往外部开始搜索数据并使用，不能从外部向内部搜索数据并使用）



延长作用域链

在ECMAScriot中有一些语句可以在作用域的前端零时增加一个变量对象（variable object），该变量对象（variable object）会在代码执行后被移除。

with语句：该语句会将指定的对象（参数）添加到作用域链中。将该对象添加到作用域的最前端并能访问到该对象的属性和方

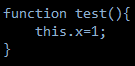
法。（大量使用with语句会降低性能）

try…catch语句的catch块：catch语句会创建一个新的变量对象（variable object），其中包含的是被抛出的错误对象的声明。

### this关键字

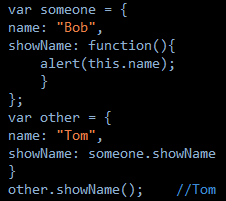
this是Javascript语言的一个关键字，它代表函数运行时自动生成的一个内部对象，只能在函数内部使用,下面分四种情况，详细讨论this的用法

例如：



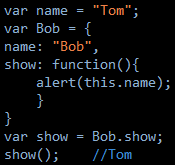
随着函数使用场合的不同，this的值会发生变化。但是有一个总的原则，那就是在Javascript中，this指向函数执行时的当前对象。值得注意，该关键字在Javascript中和执行环境，而非声明环境有关。

1. 函数被调用时this指向调用对象。

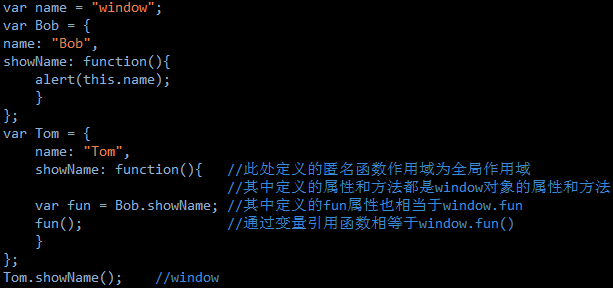


this关键字虽然是在someone.showName中声明的，但运行的时候是通过other对象的showName属性来调用someone对象的showName属性的方法，所以this指向other对象。this.name就直接找other对象中的name属性，故最后alert出来的是Tom

1. 在全局执行环境中定义的函数，并且不被调用，则this指向全局对象window。



1. 在函数中定义局部变量引用函数，其中this也是指向的window对象



其中Tom对象中的定义的shoeName方法为全局作用域，其中定义的局部变量相当于window对象的属性。所以这里用局部变量引用执行函数，也相当于window.fun()执行函数。执行对象还是window对象。

1. 通过构造函数创建的实例，其中的this指向的创建的实例对象

## 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。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。

### 创建函数

基本语法格式：



实例：



函数表达式语法格式：



实例：



其中函数表达式是将一个匿名函数赋值给一个变量，从而变量获取了函数在内存中的一个指向该函数的指针。

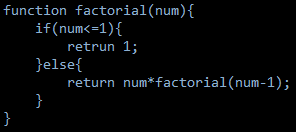
### 调用函数

在ECMAScript中调用函数与其他语言的方法一样，直接是函数名后跟一对圆括号。其中圆括号可以跟参数，也可以不用。

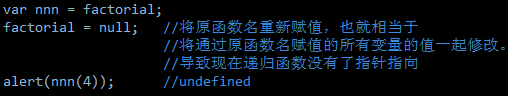


### 递归

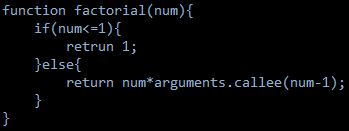
递归函数是指一个函数通过函数名调用自身的情况



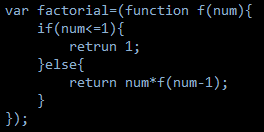
以上的代码没什么问题，但是如果修改了原函数名的值，就会出现了函数没有指针的情况，导致无法正常访问函数。



arguments.callee：该属性是一个指向正在执行的函数的指针，一般用于函数的递归调用，代替直接通过函数名来调用函数。



但是在严格模式下不能通过脚本访问arguments.callee，访问这个属性会导致错误，可以使用命名函数表达式来达成相同的结果。

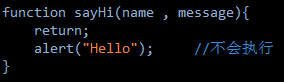


### 函数返回值

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语句后面跟要返回的值来实现返回值。但是代码在运行的时候遇到return语句的时候立刻停止语句的执行，并返回结果。return后面的代码是不会执行的。



如果return不带任何的返回值，函数在停止执行后将返回undefined值。这种用法一般用于需要提前停止函数执行而又不需要返回值的情况。



在严格模式中，对函数有一些限制：

不能把函数命名为eval或arguments；

不能把参数命名为eval或arguments；

不能出现两个命名参数同名的情况

### 函数的参数

ECMAScript函数的参数与大多数其他语言中的函数的参数有所不同。ECMAScript不限制传递参数的个数，也不限制传递参数的数据

类型。即便定义的函数只接收两个参数，在调用这个函数时也未必一定要传递两个参数。可以传递一个、三个甚至不传递参数。如

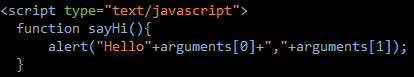
果没有传递值的参数将自动被赋予undefined值

在ECMAScript中的参数在内部是用一个数组来表示的。函数接收到的始终都是这个数组，而不关心数组中包含哪些参数。实际上，

在函数体内可以通过arguments对象来访问这个参数数组，从而获取传递给函数的每一个参数。其实arguments对象只是与数组类

型（它并不是Array的实例），因此可以使用方括号语法访问它的每个元素（第一个元素arguments[0]，….）。使用length属性来确

定传递进来多少个参数。



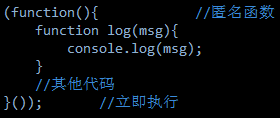
在严格模式中，对arguments对象做出了一些限制。首选，使用arguments[1]的方式为命名参数赋值是无效的。其次是重写arguments的值会导致语法错误。

ECMAScript中的所有参数传递的都是值，不可能通过引用传递参数。

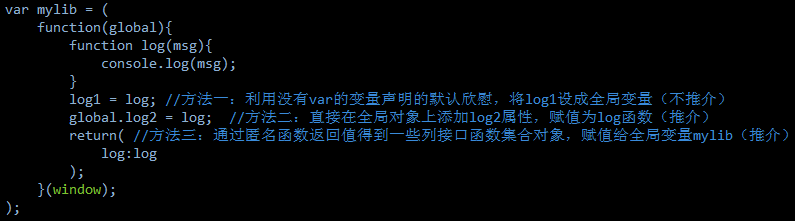
### 匿名函数和嵌套函数

在JavaScript可以声明一个没有名称的函数，称为匿名函数（Anonymouse Function）。同时JavaScript还允许在函数内部声明函数，称为嵌套函数（Nested Function）。由于匿名函数没有名称，不会引入新的变量污染上下文环境，而且会带来新的变量作用域，因此匿名函数常被用来防止全局环境污染。

JavaScript运行时中有一个特殊的全局环境（global object），这个对象上面存放全局的函数和变量，实际开发中经常会使用若干第三方的库或多个js文件，若不小心在全局对象引入重复的变量或函数声明，则会造成代码执行混乱。例如先后引入两个js文件，分别定义了自己的函数log作为内部使用，则第二引入的函数会覆盖第一个的定义且不会抛出任何错误，在后续的执行中调用log函数可能会造成错误。这时候使用一个匿名函数将整个js内的逻辑包装起来，就可以避免这种错误，这种方法已经被绝大多数开源js库使用。

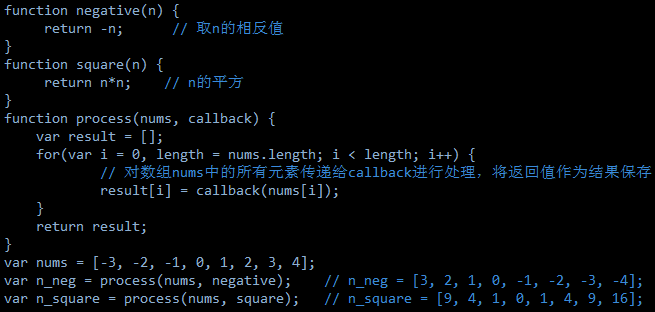


以上代码就是一个简单的示例，log函数的作用域被限制在这个匿名函数之内，而匿名函数则因为被外面一对小括号()包括起来，形成一个函数表达式，表达式的值是一个函数，紧接着一对小括号表示立即执行这个函数，让原有的代码正常执行一次。不过，这种方式声明的函数、通过var声明的变量等等都是内部的，不能被任何匿名函数以外的代码访问到。如果你需要对外暴露一些函数作为接口的话有如下几种方法：



### 高阶函数

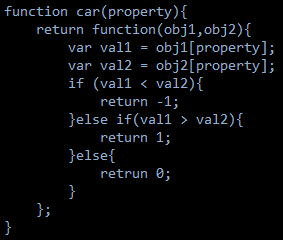
如果函数作为参数或返回值使用时，就称为高阶函数，JavaScript中的函数都可以作为高阶函数来使用



以上代码展示了把函数作为参数传入另一个函数process调用的示例，在process函数的实现中，把callback作为一个黑盒子看待，负责把参数传给它，然后获取返回值，在调用之前并不清楚callback的具体实现。只有当执行到20行和22行时，callback才被分别代表negative或square，分别对每个元素进行取相反值或平方值的操作。

### 闭包

闭包(Closure)并不是一个新鲜的概念，很多函数式语言中都使用了闭包。在JavaScript中，当你在内嵌函数中使用外部函数作用域内的变量时，就是使用了闭包。用一个常用的类比来解释闭包和类（Class）的关系：类是带函数的数据，闭包是带数据的函数。闭包只得是有权访问另一个函数作用域中变量的函数。创建闭包的常见方式时在一个函数内部创建另一个函数。

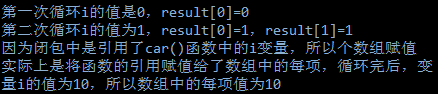
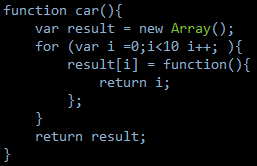


说明：上例中定义了car()函数，并传入参数property。在是在执行代码的时候遇到return关键字后car()函数的执行环境会被销毁，参数也会被销毁。但是在car()函数内部却再次定义了一个函数，并且调用了car()函数中的property，导致property变量依然存在，内部函数的作用域依然有效并存在，且可以正常访问car()函数的变量，直到内部函数代码执行完毕后，内部函数执行环境及便利被销毁后，外部函数中的变量和执行环境才会被彻底销毁。

闭包的作用通过ECMAScript中作用域链的定义，全局执行环境无法访问局部执行环境内部的数据。但是我们可以通过在函数内部定义一个函数，用于访问第一个函数内部的数据，并将其值返回除来，实现全局执行环境中获得局部执行环境中的数据。（这样也实现了使用同一的一个接口来访问局部数据，实现代码的封装）函数内部再定义一个函数，内部函数可以直接调用外部函数的变量或值，这样外部函数因为有调用数据的存在而不至于被销毁数据。这样我们可以通过这种机制来实现将外部函数中的数据缓存，用于内部函数的使用，这样有助于提高运行效率

闭包与变量

作用域链的这种配置机制引出了一个值得注意的副作用，即闭包只能取得包含函数中任何变量的最后一个值。



### 没有重载

ECMAScript中，函数不能像传统意义上那样实现重载。因为在ECMAScript中函数没有签名，其参数是由包含零或多个值的数组来表

示的。所以没有函数签名是不能实现函数的重载。如果同时定义了两个相同函数名的函数，则第二个函数会替换掉之前定义的函数。

关于this对象

this对象时在运行时基于函数的执行环境绑定的：在全局函数中，this等于window，而当函数被作为某个对象的方法调用时，

this等于那个对象。不过匿名函数的执行环境具有全局性，因此其this对象通常指向window。

## 对象

### 创建对象

对象是一个包含相关数据和方法的集合（通常由一些变量和函数组成，我们称之为对象里面的属性和方法），让我们通过一个例子来了解它们。

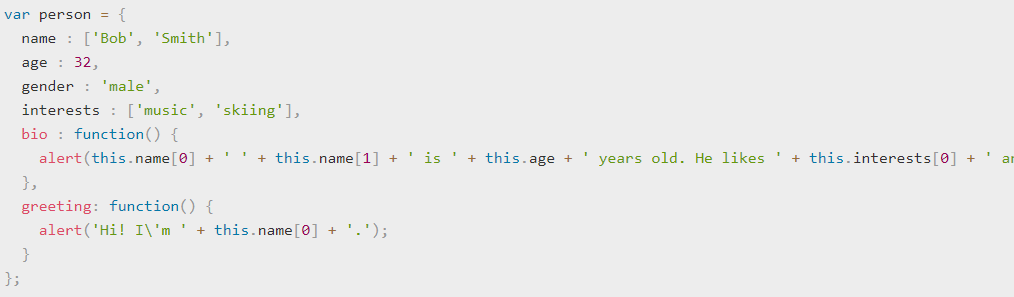
如同Javascript中的很多东西一样，创建一个对象通常先定义初始化变量。 尝试在您已有的文件中JavaScript代码下面输入以下内容, 保存刷新页面:



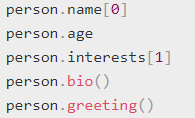
如果你在浏览器控制台输入person，然后按下Enter(确认)键，你会得到如下结果：



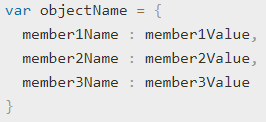
恭喜, 你刚创建了你的第一个对象. 干的漂亮! 但这是一个空对象，所以我们做不了更多的事情。像下面一样更新下我们的对象:



保存刷新后, 尝试在你的浏览器控制台输入下面的内容:



一个对象由许多的成员组成，每一个成员都拥有一个名字（像上面的name、age），和一个值（如['Bob', 'Smith']、32）。每一个名字/值（name/value）对被逗号分隔开，并且名字和值之间由冒号（:）分隔，语法规则如下所示：



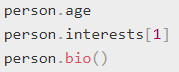
对象成员的值可以是任意的，在我们的person对象里有字符串(string)，数字(number)，两个数组(array)，两个函数(function)。前4个成员是资料项目，被称为对象的属性(property)，后两个成员是函数，允许对象对资料做一些操作，被称为对象的方法(method)

### 访问属性和方法

访问对象的属性和方法有两种方式，一种是使用点表示法，另一种是括号表示法

**点表示法**

在上面的例子中，你使用了点表示法(dot notation)来访问对象的属性和方法。对象的名字表现为一个命名空间(namespace)，它必须写在第一位——当你想访问对象内部的属性或方法时，然后是一个点(.)，紧接着是你想要访问的项目，标识可以是简单属性的名字(name)，或者是数组属性的一个子元素，又或者是对象的方法调用。如下所示：

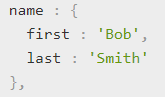


子命名空间

可以用一个对象来做另一个对象成员的值。例如将name成员



改成



这样，我们实际上创建了一个子命名空间，听起来有点复杂，但用起来很简单，你只需要链式的再使用一次点表示法，像这样：



**括号表示法**

另外一种访问属性的方式是使用括号表示法(bracket notation)，替代这样的代码



使用如下所示的代码：



这看起来很像访问一个数组的元素，从根本上来说是一回事儿，你使用了关联了值的名字，而不是索引去选择元素。难怪对象有时被称之为关联数组(associative array)了——对象做了字符串到值的映射，而数组做的是数字到值的映射。

### 设置对象成员

目前我们仅仅看到了如何访问对象的成员，而你其实也可以设置对象成员的值，通过声明你要设置的成员，像这样：



尝试这些代码，然后再查看这些成员是否已经被改变了



设置成员并不意味着你只能更新已经存在的属性的值，你完全可以创建新的成员，尝试以下代码：



现在你可以测试你新创建的成员



括号表示法一个有用的地方是它不仅可以动态的去设置对象成员的值，还可以动态的去设置成员的名字。

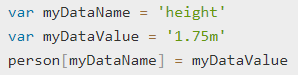
比如说，我们想让用户能够在他们的数据里存储自己定义的值类型，通过两个input框来输入成员的名字和值，通过以下代码获取用户输入的值：



我们可以这样把这个新的成员的名字和值加到person对象里：



为了测试这个功能，尝试在你的代码里添加以下几行，就在person对象的右花括号的下面：

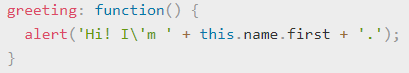


现在，保存并刷新，在输入框里输入以下代码：



这是使用点表示法无法做到的，点表示法只能接受字面量的成员的名字，不接受变量作为名字。

### this的含义



在上述代码中关键字"this"指向了当前代码运行时的对象( 原文：the current object the code is being written inside )——这里即指person对象，它保证了当代码的上下文(context)改变时变量的值的正确性（比如：不同的person对象拥有不同的name这个属性，很明显greeting这个方法需要使用的是它们自己的name）。

让我们以两个简单的person对象来说明：



在这里，person1.greeting()会输出："Hi! I'm Chris."；person2.greeting()会输出："Hi! I'm Brain."，即使greeting这个方法的代码是一样的。就像我们之前说的，this 指向了代码所在的对象(其实代码运行时所在的对象)。在字面量的对象里this看起来不是很有用，但是当你动态创建一个对象（例如使用构造器）时它是非常有用的，之后你会更清楚它的用途。

## 面向对象的程序设计

面向对象的语言有一个标志，就是它们都有类的概念，而通过类可以创建任意多个具有相同属性和方法的对象。但是在ECMAScript中是没有类的概念，因袭它的对象与基于类的语言中的对象有所不同。

ECMAScript中把对象定义为：无序属性的集合，其属性可以包含基本值、对象或者函数。对象的每个属性或方法都有一个名字，而名字都映射到一个值。

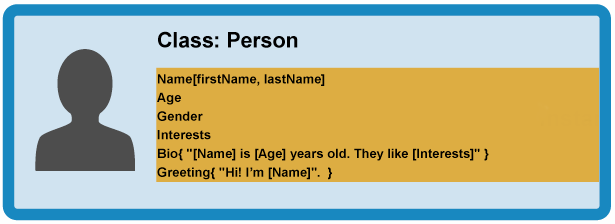
### 理解面向对象的程序设计

首先, 让我们给您定义一个面向对象的程序(OOP) 的极其简单的概念. 我们说的极其简单, 是因为 OOP 很快就变得非常复杂,如果现在给您一个完整的定义可能会让您产生迷惑而非帮助. 最基本的 OOP 思想就是我们想要在我们的程序中使用对象来表示现实世界模型, 并提供一个简单的方式来访问它的功能,否则很难甚至不能实现.

对象可以包含相关的数据和代码,这些代表现实世界模型的一些信息或者功能,或者它特有的一些行为. 对象数据(也经常称为函数) 可以有结构的存储 (官方术语为 封装) 在对象包内 (也可以给一个特殊的名字来表示,有时候也叫做命名空间), 可以使它容易组织和访问; 对象也通常用于存储数据,这样就可以很容易的在网络上传输.

**定义一个对象模板**

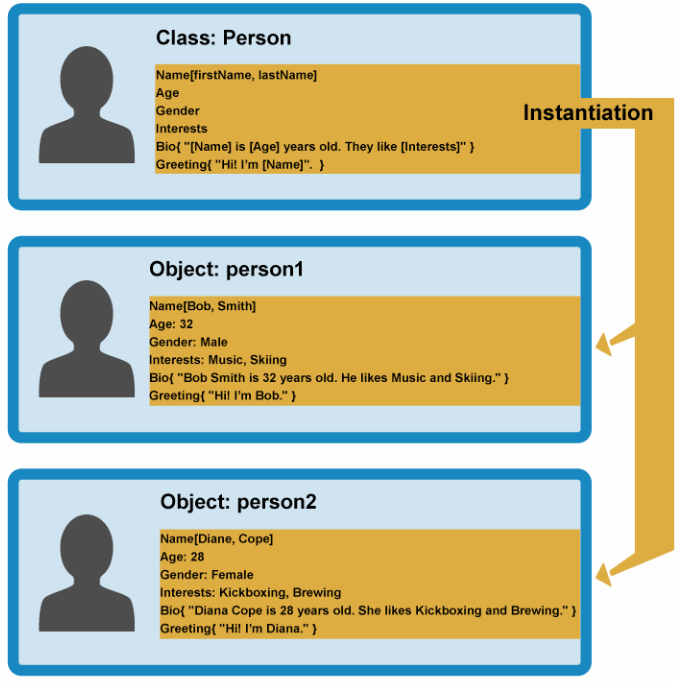
首先，我们可以回到上一节拿到定义好属性和方法的Person对象。对于一个人（person）来说，我们能在他们身上获取到很多信息（他们的住址，身高，鞋码，基因图谱，护照信息，显著的性格特征等等），然而，我们仅仅需要他们的名字，年龄，性别，兴趣 这些信息，然后，我们会基于他们的这些信息写一个简短的介绍关于他们自己，在最后我们还需要教会他们打招呼。以上的方式被称为抽象-为了我们编程的目标而利用事物的一些重要特性去把复杂的事物简单化



在一些面向对象的语言中，我们用类（class）的概念去描述一个对象（您在下面就能看到JavaScript使用了一个完全不同的术语）-类并不完全是一个对象，它更像是一个定义对象特质的模板。

**创造一个真正的对象**

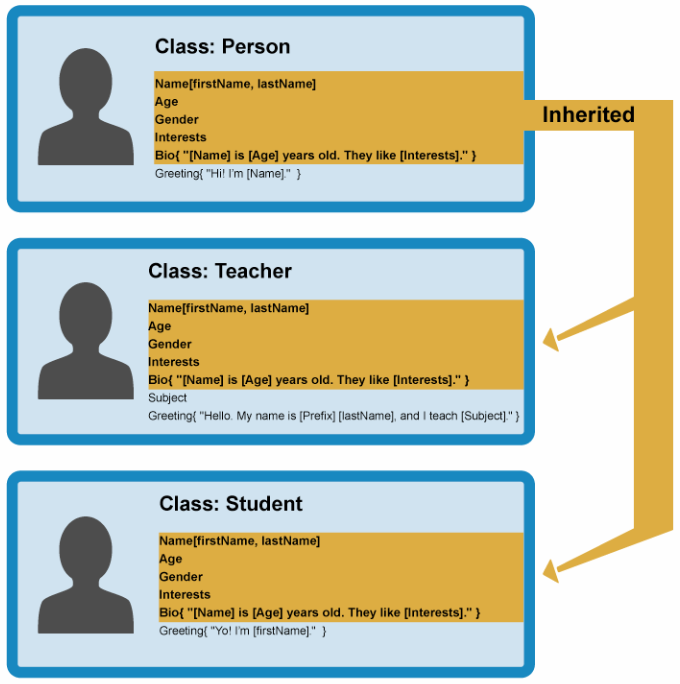
从上面我们创建的class中， 我们能够基于它创建出一些对象实例 - 一些拥有class中属性及方法的对象。基于我们的Person类，我们可以创建出许许多多的真实的人：



当一个对象实例需要从类中创建出来时，类的构造函数就会运行来创建这个实例。这种创建对象实例的过程我们称之为实例化-实例对象被类实例化。

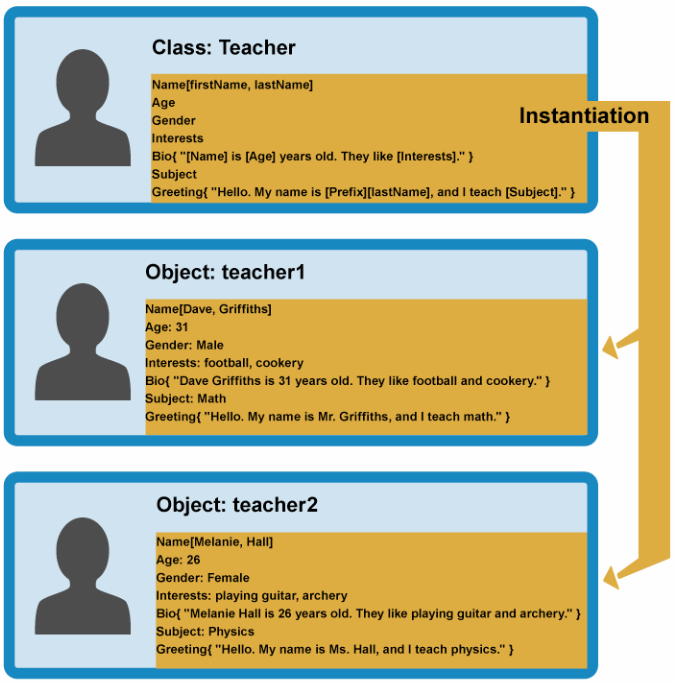
**具体的对象**

在这个例子里，我们不想要泛指的人，我们想要向老师和学生这样类型更为具体的人。在 OOP 里，我们可以创建基于其它类的新类，这些新的子类可以继承它们父类的数据和功能。比起复制来说这样能够使用父对象共有的功能。



这是非常有用的，老师和学生具有一些相同的特征比如姓名、性别、年龄，因此只需要定义这些特征一次就可以了。您可以在不同的类里分开定义这些相同的特征，这样该特征会有一个不同的命名空间。比如，一个学生的 greeting 可以是 "Yo, I'm [firstName]" (例子 Yo, I'm Sam)，老师的可能会正式一些，比如"Hello, my name is [Prefix] [lastName]" (例子 Hello, My name is Mr Griffiths)。

现在可以根据子类创建对象实例。如：



下面我们来看看 OOP 理论如何应用到 JavaScript 实践中去的。

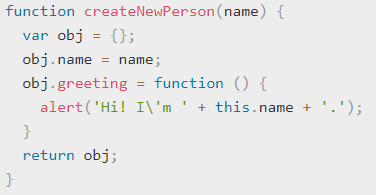
### 构建函数和对象实例

有些人认为 JavaScript 不是真正的面向对象的语言，比如它没有像许多面向对象的语言一样有用于创建class类的声明。JavaScript 用一种称为构建函数的特殊函数来定义对象和它们的特征。构建函数非常有用，因为很多情况下您不知道实际需要多少个对象（实例）。构建函数提供了创建您所需对象（实例）的有效方法，将对象的数据和特征函数按需联结至相应对象。

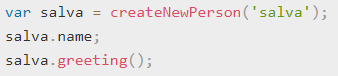
不像“经典”的面向对象的语言，从构建函数创建的新实例的特征并非全盘复制，而是通过一个叫做原形链的参考链链接过去的。（参见 Object prototypes），所以这并非真正的实例，严格的讲， JavaScript 在对象间使用和其它语言的共享机制不同。

1. **一个简单的例子**

让我们看看如何通过一个普通的函数定义一个”人“。在您的文件中添加以下代码:

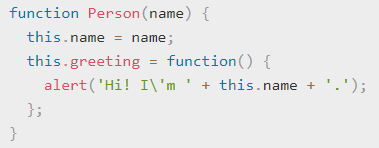


您现在可以通过调用这个函数创建一个新的叫 salva 的人，在您浏览器的JavaScript console 试试 ：



上述代码运行良好，但是有点冗长；如果我们知道如何创建一个对象，就没有必要创建一个新的空对象并且返回它。幸好 JavaScript 通过构建函数提供了一个便捷的方法，方法如下：

将之前的代码用如下代码代替：



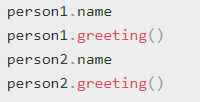
这个构建函数是 JavaScript 版本的类。您会发现，它只定义了对象的属性和方法，除了没有明确创建一个对象和返回任何值和之外，它有了您期待的函数所拥有的全部功能。这里使用了this关键词，即无论是该对象的哪个实例被这个构建函数创建，它的 name 属性就是传递到构建函数形参name的值，它的 greeting() 方法中也将使用相同的传递到构建函数形参name的值。

那如何调用构建函数创建新的实例呢？

将下面的代码加在您之前的代码下面：



保存并刷新浏览器，在 console 里输入如下代码：

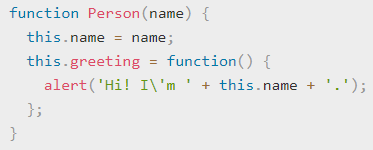


您现在看到页面上有两个对象，每一个保存在不同的命名空间里，当您访问它们的属性和方法时，您需要使用person1或者person2来调用它们。尽管它们有着相同的name属性和 greeting()方法它们是各自独立的，所以相互的功能不会冲突。注意它们使用的是自己的 name 值，这也是使用 this 关键字的原因，它们使用的从实参传入形参的自己的值，而不是其它的什么值。

再看看这个构造对象的语法：



上述代码中，关键字 new 跟着一个含参函数，用于告知浏览器我们想要创建一个对象实例，非常类似函数调用，并把结果保存到变量中。每个示例类都是根据下面的方式定义的。



当新的对象被创立, 变量person1与person2有效地包含了以下值：

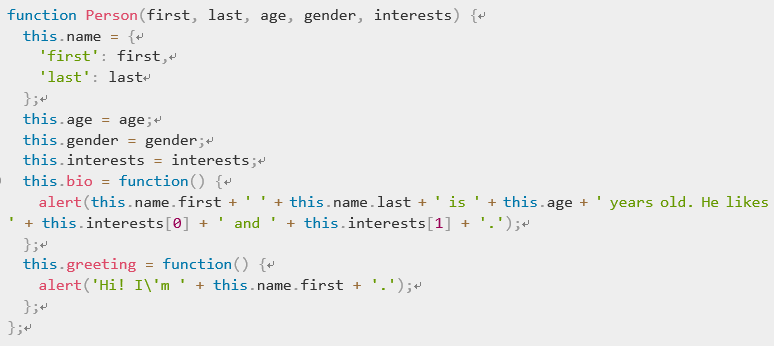


之所以说是“有效”， 是因为实际的方法仍然是定义在类里面， 而不是在对象实例里面, 这与我们之前说的字母意义上的对象还是有所不同的。

1. **创建我们最终的构造函数**

上面的例子仅仅是简单地介绍如何开始。让我们现在开始创建Person()构造函数。

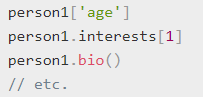
移除掉您之前写的所有代码， 用如下构造函数替代 —— 实现原理上，这与我们之前的例子并无二致， 只是变得稍稍复杂了些：



接下来加上这样一行代码， 用来创建它的一个对象实例：



这样，您就可以像我们定义第一个对象一样访问它的属性和方法了：



1. **创建对象实例的其他方式**

到现在为止，我们了解到了两种不同的创建对象实例的方式 —— 声明一个对象的语法， 与使用构造函数(回顾上面)。

这些方法都是很有用的， 但仍有其他的方法 —— 我们希望您能熟悉这些，以免您在Web世界的旅行中碰到它们。

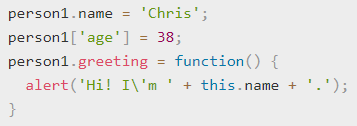
**Object()构造函数**

首先, 您能使用Object()构造函数来创建一个新对象。 是的， 一般对象都有构造函数，它创建了一个空的对象。

尝试在您浏览器中的Javascript控制台中输入以下代码：



这样就在person1变量中存储了一个空对象。然后, 可以根据需要, 使用点或括号表示法向此对象添加属性和方法；试试这个例子：



还可以将对象文本传递给Object() 构造函数作为参数， 以便用属性/方法填充它。请尝试以下操作：



**使用create()方法**

JavaScript有个内嵌的方法create(), 它允许您基于现有对象创建新的对象实例。

在 JavaScript 控制台中尝试此操作：



现在尝试这个：



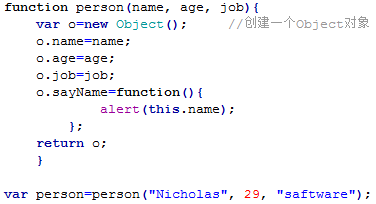
您可以看到，person2是基于person1创建的， 它们具有相同的属性和方法。这非常有用， 因为它允许您创建新的对象实例而无需定义构造函数。缺点是比起构造函数，浏览器在更晚的时候才支持create()方法（IE9, IE8 或甚至以前相比）， 加上一些人认为构造函数让您的代码看上去更整洁 —— 您可以在一个地方创建您的构造函数， 然后根据需要创建实例， 这让您能很清楚地知道它们来自哪里。

但是, 如果您不太担心对旧浏览器的支持， 并且您只需要一个对象的一些副本， 那么创建一个构造函数可能会让您的代码显得过度繁杂。这取决于您的个人爱好。有些人发现create() 更容易理解和使用。

我们在创建对象的时候可以使用Object构造函数或者是对象字面量，但是如果我们需要同时创建多个对象，就会产生大量的重复代码。为此人们开始使用工厂模式的一种变体。

1. **工厂模式**

利用函数封装以特定接口创建对象的细节



工厂模式实现了创建多个相似对象的问题，可以多次调用函数来返回三个属性和一个方法的对象。但是工厂模式却没有解决对象识别的问题（怎样知道一个对象的类型）

### Javascript中的原型

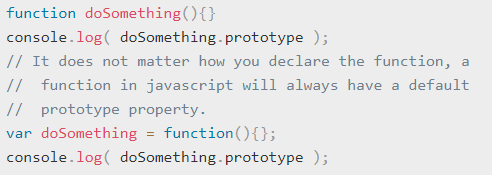
JavaScript 常被描述为一种基于原型的语言 (prototype-based language)——每个对象拥有一个原型对象，对象以其原型为模板、从原型继承方法和属性。原型对象也可能拥有原型，并从中继承方法和属性，一层一层、以此类推。这种关系常被称为原型链 (prototype chain)，它解释了为何一个对象会拥有定义在其他对象中的属性和方法。

准确地说，这些属性和方法定义在Object的构造器函数(constructor functions)之上的prototype属性上，而非对象实例本身。

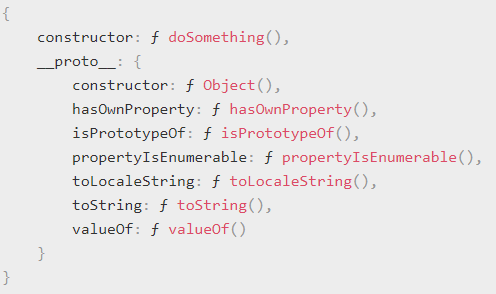
在传统的 OOP 中，首先定义“类”，此后创建对象实例时，类中定义的所有属性和方法都被复制到实例中。在 JavaScript 中并不如此复制——而是在对象实例和它的构造器之间建立一个链接（它是\_\_proto\_\_属性，是从构造函数的prototype属性派生的），之后通过上溯原型链，在构造器中找到这些属性和方法。

**使用Javascript中的原型**

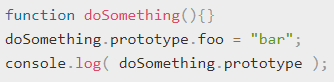
在javascript中，函数可以有属性。 每个函数都有一个特殊的属性叫作原型（prototype） ，正如下面所展示的。请注意，下面的代码是独立的一段(在网页中没有其他代码的情况下，这段代码是安全的)。为了最好的学习体验，你最好打开一个控制台 (在Chrome和Firefox中，可以按Ctrl+Shift+I来打开)切换到"控制台" 选项卡, 复制粘贴下面的JavaScript代码，然后按回车来运行.



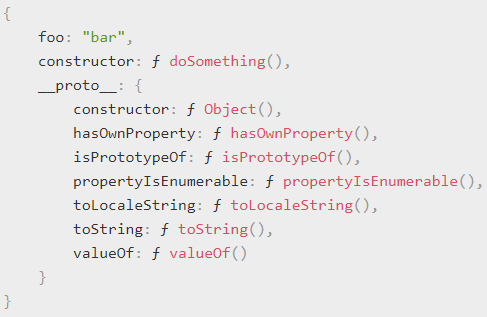
正如上面所看到的, doSomething 函数有一个默认的原型属性，它在控制台上面呈现了出来. 运行这段代码之后，控制台上面应该出现了像这样的一个对象.



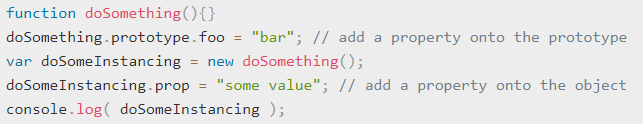
现在，我们可以添加一些属性到 doSomething 的原型上面，如下所示.



结果:



然后，我们可以使用 new 运算符来在现在的这个原型基础之上，创建一个 doSomething 的实例。正确使用 new 运算符的方法就是在正常调用函数时，在函数名的前面加上一个 new 前缀. 通过这种方法，在调用函数前加一个 new ，它就会返回一个这个函数的实例化对象. 然后，就可以在这个对象上面添加一些属性. 看.



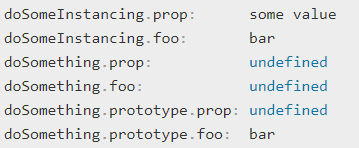
结果:



就像上面看到的, doSomeInstancing 的 \_\_proto\_\_ 属性就是doSomething.prototype. 但是这又有什么用呢? 好吧,当你访问 doSomeInstancing 的一个属性, 浏览器首先查找 doSomeInstancing 是否有这个属性. 如果 doSomeInstancing 没有这个属性, 然后浏览器就会在 doSomeInstancing 的 \_\_proto\_\_ 中查找这个属性(也就是 doSomething.prototype). 如果 doSomeInstancing 的 \_\_proto\_\_ 有这个属性, 那么 doSomeInstancing 的 \_\_proto\_\_ 上的这个属性就会被使用. 否则, 如果 doSomeInstancing 的 \_\_proto\_\_ 没有这个属性, 浏览器就会去查找 doSomeInstancing 的 \_\_proto\_\_ 的 \_\_proto\_\_ ，看它是否有这个属性. 默认情况下, 所有函数的原型属性的 \_\_proto\_\_ 就是 window.Object.prototype. 所以 doSomeInstancing 的 \_\_proto\_\_ 的 \_\_proto\_\_ (也就是 doSomething.prototype 的 \_\_proto\_\_ (也就是 Object.prototype)) 会被查找是否有这个属性. 如果没有在它里面找到这个属性, 然后就会在 doSomeInstancing 的 \_\_proto\_\_ 的 \_\_proto\_\_ 的 \_\_proto\_\_ 里面查找. 然而这有一个问题: doSomeInstancing 的 \_\_proto\_\_ 的 \_\_proto\_\_ 的 \_\_proto\_\_ 不存在. 最后, 原型链上面的所有的 \_\_proto\_\_ 都被找完了, 浏览器所有已经声明了的 \_\_proto\_\_ 上都不存在这个属性，然后就得出结论，这个属性是 undefined.

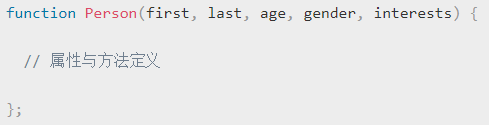


结果:



**理解原型对象**

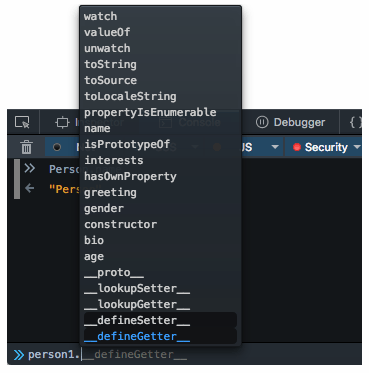
让我们回到 Person() 构造器的例子。本例中我们将定义一个构造器函数：



然后创建一个对象实例：



在 JavaScript 控制台输入 "person1."，你会看到，浏览器将根据这个对象的可用的成员名称进行自动补全：



在这个列表中，你可以看到定义在 person1 的原型对象、即 Person() 构造器中的成员—— name、age、gender、interests、bio、greeting。同时也有一些其他成员—— watch、valueOf 等等——这些成员定义在 Person() 构造器的原型对象、即 Object 之上。下图展示了原型链的运作机制。



那么，调用 person1 的“实际定义在 Object 上”的方法时，会发生什么？比如：



这个方法仅仅返回了被调用对象的值。在这个例子中发生了如下过程：

* + - * 浏览器首先检查，person1 对象是否具有可用的 valueOf() 方法。
      * 如果没有，则浏览器检查 person1 对象的原型对象（即 Person构造函数的prototype属性所指向的对象）是否具有可用的 valueof() 方法。
      * 如果也没有，则浏览器检查 Person() 构造函数的prototype属性所指向的对象的原型对象（即 Object构造函数的prototype属性所指向的对象）是否具有可用的 valueOf() 方法。这里有这个方法，于是该方法被调用。

**prototype 属性：继承成员被定义的地方**

那么，那些继承的属性和方法在哪儿定义呢？如果你查看 Object 参考页，会发现左侧列出许多属性和方法——大大超过我们在 person1 对象中看到的继承成员的数量。某些属性或方法被继承了，而另一些没有——为什么呢？

原因在于，继承的属性和方法是定义在 prototype 属性之上的（你可以称之为子命名空间 (sub namespace) ）——那些以 Object.prototype. 开头的属性，而非仅仅以 Object. 开头的属性。prototype 属性的值是一个对象，我们希望被原型链下游的对象继承的属性和方法，都被储存在其中。

于是 Object.prototype.watch()、Object.prototype.valueOf() 等等成员，适用于任何继承自 Object() 的对象类型，包括使用构造器创建的新的对象实例。

Object.is()、Object.keys()，以及其他不在 prototype 对象内的成员，不会被“对象实例”或“继承自 Object() 的对象类型”所继承。这些方法/属性仅能被 Object() 构造器自身使用。

你可以检查已有的 prototype 属性。回到先前的例子，在 JavaScript 控制台输入：



输出并不多，毕竟我们没有为自定义构造器的原型定义任何成员。缺省状态下，构造器的 prototype 属性初始为空白。现在尝试：



你会看到 Object 的 prototype 属性上定义了大量的方法；如前所示，继承自 Object 的对象都可以使用这些方法。

JavaScript 中到处都是通过原型链继承的例子。比如，你可以尝试从 String、Date、Number 和 Array 全局对象的原型中寻找方法和属性。它们都在原型上定义了一些方法，因此当你创建一个字符串时：



myString 立即具有了一些有用的方法，如 split()、indexOf()、replace() 等。

我们曾经讲过如何用 Object.create() 方法创建新的对象实例。

例如，在上个例子的 JavaScript 控制台中输入：



create() 实际做的是从指定原型对象创建一个新的对象。这里以 person1 为原型对象创建了 person2 对象。在控制台输入：



结果返回对象person1。

**constructor 属性**

每个实例对象都从原型中继承了一个constructor属性，该属性指向了用于构造此实例对象的构造函数。

例如，在控制台中尝试下面的指令：



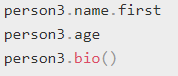
都将返回 Person() 构造器，因为该构造器包含这些实例的原始定义。

一个小技巧是，你可以在 constructor 属性的末尾添加一对圆括号（括号中包含所需的参数），从而用这个构造器创建另一个对象实例。毕竟构造器是一个函数，故可以通过圆括号调用；只需在前面添加 new 关键字，便能将此函数作为构造器使用。

在控制台中输入：



现在尝试访问新建对象的属性，例如：



正常工作。通常你不会去用这种方法创建新的实例；但如果你刚好因为某些原因没有原始构造器的引用，那么这种方法就很有用了。

此外，constructor 属性还有其他用途。比如，想要获得某个对象实例的构造器的名字，可以这么用：



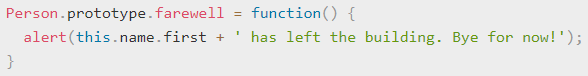
具体地，像这样：



**修改原型**

从我们从下面这个例子来看一下如何修改构造器的 prototype 属性。

回到之前的实例中，在JavaScript代码后面添加如下代码，这段代码将为构造器的 prototype 属性添加一个新的方法：

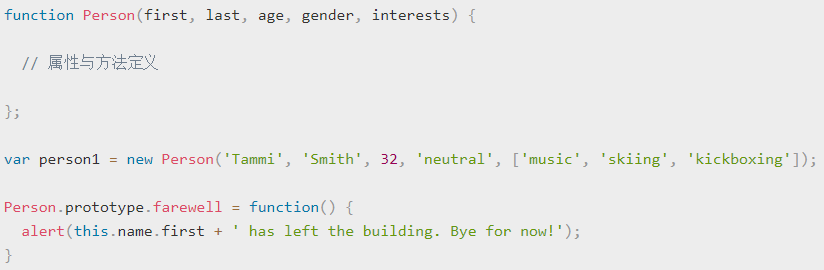


保存代码，在浏览器中加载页面，然后在控制台输入：



你会看到一条警告信息，其中还显示了构造器中定义的人名；这很有用。但更关键的是，整条继承链动态地更新了，任何由此构造器创建的对象实例都自动获得了这个方法。

再想一想这个过程。我们的代码中定义了构造器，然后用这个构造器创建了一个对象实例，此后向构造器的 prototype 添加了一个新的方法：



但是 farewell() 方法仍然可用于 person1 对象实例——旧有对象实例的可用功能被自动更新了。这证明了先前描述的原型链模型。这种继承模型下，上游对象的方法不会复制到下游的对象实例中；下游对象本身虽然没有定义这些方法，但浏览器会通过上溯原型链、从上游对象中找到它们。这种继承模型提供了一个强大而可扩展的功能系统。

你很少看到属性定义在 prototype 属性中，因为如此定义不够灵活。比如，你可以添加一个属性：



但这不够灵活，因为人们可能不叫这个名字。用 name.first 和 name.last 组成 fullName 会好很多：



然而，这么做是无效的，因为本例中 this 引用全局范围，而非函数范围。访问这个属性只会得到 undefined undefined。但这个语句若放在 先前定义在 prototype 上的方法中则有效，因为此时语句位于函数范围内，从而能够成功地转换为对象实例范围。你可能会在 prototype 上定义常属性 (constant property) （指那些你永远无需改变的属性），但一般来说，在构造器内定义属性更好。

事实上，一种极其常见的对象定义模式是，在构造器（函数体）中定义属性、在 prototype 属性上定义方法。如此，构造器只包含属性定义，而方法则分装在不同的代码块，代码更具可读性。例如：



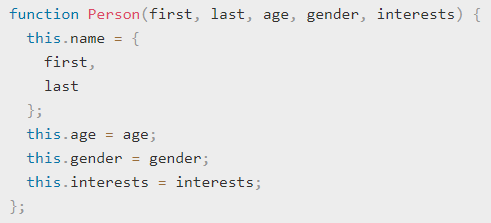
### Java​Script 中的继承

**原型式的继承**

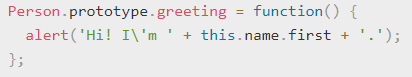
到目前为止我们已经了解了一些关于原型链的实现方式以及成员变量是如何通过它来实现继承，但是之前涉及到的大部分都是浏览器内置函数（比如 String、Date、Number 和 Array），那么我们如何创建一个继承自另一对象的JavaScript对象呢？

正如前面课程所提到的，有些人认为JavaScript并不是真正的面向对象语言，在经典的面向对象语言中，您可能倾向于定义类对象,然后您可以简单地定义哪些类继承哪些类（参考C++ inheritance里的一些简单的例子），JavaScript使用了另一套实现方式，继承的对象函数并不是通过复制而来，而是通过原型链继承（通常被称为 原型式继承 —— prototypal inheritance）。

如下代码其中您能看到一个只定义了一些属性的Person()构造器，与之前通过模块来实现所有功能的Person的构造器类似。



所有的方法都定义在构造器的原型上，比如：

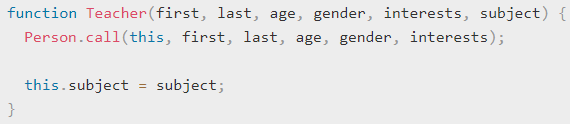


比如我们想要创建一个Teacher类，就像我们前面在面向对象概念解释时用的那个一样。这个类会继承Person的所有成员，同时也包括：

* 一个新的属性，subject——这个属性包含了教师教授的学科。
* 一个被更新的greeting()方法，这个方法打招呼听起来比一般的greeting()方法更正式一点——对于一个教授一些学生的老师来说。

**定义 Teacher() 构造器函数**

我们要做的第一件事是创建一个Teacher()构造器——将下面的代码加入到现有代码之下：

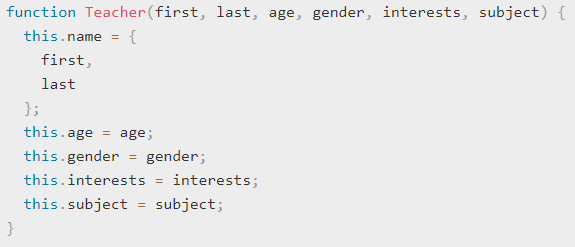


这在很多方面看起来都和Person的构造器很像，但是这里有一些我们从没见过的奇怪玩意——call()函数。基本上，这个函数允许您调用一个在这个文件里别处定义的函数。第一个参数指明了在您运行这个函数时想对“this”指定的值，也就是说，您可以重新指定您调用的函数里所有“this”指向的对象。其他的变量指明了所有目标函数运行时接受的参数。

所以在这个例子里，我们很有效的在Teacher()构造函数里运行了Person()构造函数（见上文），得到了和在Teacher()里定义的一样的属性，但是用的是传送给Teacher()，而不是Person()的值（我们简单使用这里的this作为传给call()的this，意味着this指向Teacher()函数）。

在构造器里的最后一行代码简单地定义了一个新的subject属性，这将是教师会有的，而一般人没有的属性。

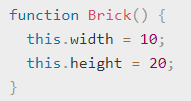
顺便提一下，我们本也可以这么做：



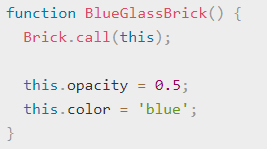
但是这只是重新定义了一遍属性，并不是将他们从Person()中继承过来的，所以这违背了我们的初衷。这样写也会需要更长的代码。

从无参构造函数继承

请注意，如果您继承的构造函数不从传入的参数中获取其属性值，则不需要在call()中为其指定其他参数。所以，例如，如果您有一些相当简单的东西：



您可以这样继承width和height属性（以及下面描述的其他步骤）：



请注意，我们仅传入了this到call()中 - 不需要其他参数，因为我们不会继承通过参数设置的父级的任何属性。

**设置 Teacher() 的原型和构造器引用**

到目前为止一切看起来都还行，但是我们遇到问题了。我们已经定义了一个新的构造器，这个构造器默认有一个空的原型属性。我们需要让Teacher()从Person()的原型对象里继承方法。我们要怎么做呢？

在您先前添加的代码的下面增加以下这一行：



这里我们的老朋友create()又来帮忙了——在这个例子里我们用这个函数来创建一个和Person.prototype一样的新的原型属性值（这个属性指向一个包括属性和方法的对象），然后将其作为Teacher.prototype的属性值。这意味着Teacher.prototype现在会继承Person.prototype的所有属性和方法。

接下来，在我们动工之前，还需要完成一件事 — 现在Teacher()的prototype的constructor属性指向的是Person(), 这是由我们生成Teacher()的方式决定的。将您写的页面在浏览器中打开，进入JavaScript控制台，输入以下代码来确认：



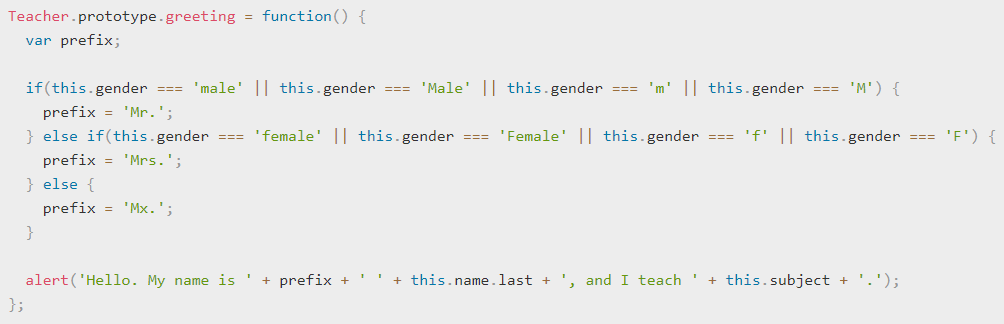
这或许会成为很大的问题，所以我们需要将其正确设置——您可以回到源代码，在底下加上这一行代码来解决：



当您保存并刷新页面以后，输入Teacher.prototype.constructor就会得到Teacher()。

**向 Teacher() 添加一个新的greeting()函数**

为了完善代码，您还需在构造函数Teacher()上定义一个新的函数greeting()。最简单的方法是在Teacher的原型上定义它—把以下代码添加到您代码的底部：



这样就会出现老师打招呼的弹窗，老师打招呼会使用条件结构判断性别从而使用正确的称呼。

## JavaScript错误

## JavaScript调试

# BOM

## 什么是BOM

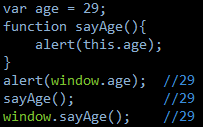
BOM（浏览器对象模型）、用于实现JavaScript对浏览器的各种操作，其中定义了很多的内置对象用于操作浏览器。

## window对象

BOM中的核心对象，表示浏览器的一个实例。在浏览器中，window对象有双重角色，即是是通过JavaScript访问浏览器窗口的一个接口，又是ECMAScript中规定的Global对象。这意味着在网页中定义的任何一个对象、变量和函数，都是以window作为其Global对象，因此有权访问parseInt()等方法

### 全局作用域

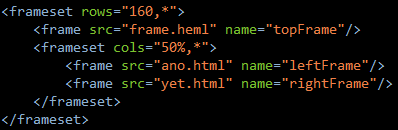
由于window对象同时扮演着ECMAScript中Global对象的角色，因此所有在全局作用域中声明的变量、函数都会变成window对象的属性和方法。



在全局作用域中定义的属性和直接在window对象上直接定义的属性还是有区别的。全局变量不能通过delete操作符删除，而直接在window对象上定义的属性是可以的。

### 窗口关系及框架

如果页面中包含框架，则每个框架都拥有自己的window对象，并且保存在frames集合中。在frames集合中，可以通过数值索引（从0开始）或者框架名称来访问相应的window对象。每个window对象都有一个name属性，其中包含框架的名称。



其中可以通过window.frames[0]或者window.frames[“topFrame”]来引用上方的框架。不过最好还是使用top来引用这些框架（top.frames[0]）。

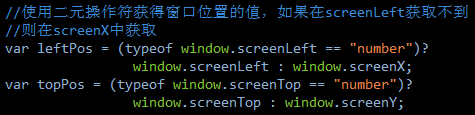
1. top始终指向最高（最外）层的框架，也就是浏览器窗口。使用它可以确保在一个框架中正确地访问另一个框架。因为对于在一个框架中编写的任何代码来说，其中的window对象指向的都是那个框架的特定实例，而非最高层的框架。
2. 与top相对的另一个window对象是parent。parent（父）对象始终指向当前框架的直接上层框架。在某些情况下，parent有可能等于top；但在没有框架的情况下，parent一定等于top（window）
3. 与框架有关的最后一个对象时self，它始终指向window；实际上，self和window对象可以互换使用。引入self对象的目的只是为了与top和parent对象对应起来，因此它不格外包含其他值。

这些所有的对象都是window对象的属性，可以通过window.parent、window.top等形式来访问。同时，也可以将不同层次的window对象连缀起来：window.parent.parent.frames[0]。

在使用框架的情况下，浏览器中会存在多个Global对象。在每个框架中定义的全局变量会自动成为框架中window对象的属性。由于每个window对象都包含原生类型的构造函数，因此每个框架都有一套自己的构造函数，这些构造函数一一对应，但并不相等。例如：top.Object并不等于top.frames[0].Object。这个问题会影响到对跨框架传递的对象使用instanceof操作符。

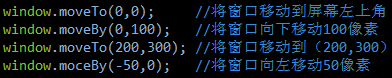
### 窗口位置

用来确定和修改window对象位置的属性和方法有很多。IE、Safari、Opera、Chrome都提供了screenLeft和screenTop属性，分别用于表示窗口相对于屏幕左边和上边的位置。Firefox则在screenX和screenY属性中提供相同的窗口位置信息，Safari和Chrome也同时支持这两个属性。使用下列代码可以跨浏览器取得窗口的位置



这几个属性由于浏览器的支持的不同，及保存的数值所表达的意思不同，导致不能获取到窗口左边和上边的精确坐标值。

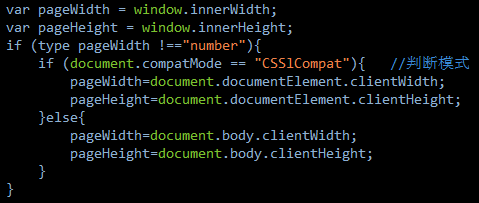
使用moveTo()和moveBy()方法可以将窗口精确地移动到一个新位置。这两个方法接收两个参数，其中moveTo()接收的是新位置的x和y坐标，而moveBy()接收的是水平和垂直方向上移动的像素数。这两个方法只对最外层的window对象使用，不适用于框架。



### 窗口的大小

跨浏览器确定一个窗口的大小不是一件简单的事。所有的浏览器为确定窗口的大小提供了innerWidth、innerHeight、outerWidth、outerHeight四个属性。在IE9+、Safari、Firefox中outerWidth、outerHeight返回浏览器窗口本身的尺寸（无论是从最外层的window对象还是从某个框架访问）。在Opera中，这两个属性的值表示页面视图容器大小。而innerWidth、innerHeight则表示该容器中页面视图区的大小（减去边框宽度）。在Chrome中，outerWidth、outerHeight与innerWidth、innerHeight返回相同的值，即视口（viewport）大小而非浏览器窗口大小。

在所有浏览器中，document.documentElement.clientWidth和document.documentElement.clientHeight中保存了页面视口的信息。在IE6中这些属性必须在标准模式下才有效。在混杂模式下就必须通过document.body.clientWidth和document.body.clientHeight获得相同的信息。而处于混杂模式下的Chrome这两种方式都可以获取到相同的信息。



resizeTo()和resizeBy()方法可以调整整个浏览器窗口的大小。这两个方法都接收两个参数，其中resizeTo()接收浏览器的新宽度和新高度，而resizeBy()接收新窗口与原窗口的宽度和高度之差。这两个方法不适用于框架，而只能对最外层的window对象使用。



### 导航和打开窗口

使用window.open()方法即可以导航到一个特定的URL，也可以打开一个新的浏览器窗口。这个方法可以接收4个参数：要加载的URL、窗口目标、一个特殊字符串、一个表示新页面是否取代浏览器历史记录中当前加载页面的布尔值。通常只需传递第一个参数，最后一个参数只在打开新窗口的情况下使用。

如果为window.open()传递了第二个参数，而且该参数是已有窗口或框架的名字，那么就会在具有该名称的窗口或框架中加载第一个参数指定的URL。如果没有找到窗口或框架名字，则会新建一个窗口加载。



1. 弹出窗口

如果给window.open()传递的第二个参数并不是一个已经存在的窗口或框架，那么该方法就会更具在第三个参数位置上传入的字符串创建一个新窗口或新标签页。如果没有传入第三个参数，那么就会打开一个带有全部默认设置（工具栏、地址和状态栏等）的新浏览器窗口（或者打开一个新标签页）。在不打开新窗口的情况下，会忽略第三个参数。

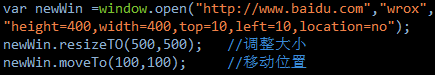
第三个参数是一个逗号分隔的设置字符串，表示在新窗口中都显示那些特性：

|  |  |  |
| --- | --- | --- |
| 设置 | 值 | 说明 |
| fullscreen | yes或no | 表示浏览器窗口是否最大化。仅限IE |
| height | 数值 | 表示新窗口的高度。不能小于100 |
| left | 数值 | 表示新窗口的左坐标。不能是负值 |
| location | yes或no | 表示是否在浏览器窗口中显示地址栏。不同的浏览器的默认值不同。如果设置为no，地址栏可能会隐藏，也可能会被禁用 |
| menubar | yes或no | 表示是否在浏览器窗口中显示菜单栏。默认值为no |
| resizeble | yes或no | 表示是否可以通过拖地浏览器窗口的边框改变其大小。默认值为no |
| scrollbars | yes或no | 表示如果内容在视口中显示不下，是否允许滚动。默认值为no。 |
| status | yes或no | 表示是否在浏览器窗口中显示状态栏。默认值为no |
| toolbar | yes或no | 表示是否在浏览器窗口中显示工具栏。默认值为no |
| top | 数值 | 表示新窗口的上坐标。不能是负值 |
| width | 数值 | 表示新窗口的宽度。不能小于100 |



注意：整个特殊字符串中不允许出现空格

1. window.open()方法会返回一个指向新窗口的引用。引用的对象与其他window对象大致相似，但是我们可以对其进行更多的控制。例如，有些浏览器在默认情况下可能不允许我们针对主浏览器窗口调整大小或移动位置，单却允许我们针对我们通过window.open()创建的窗口调整大小或移动位置。通过这个返回的对象，可以像操作其他窗口一样操作新打开的窗口。



1. 新创建的window对象有一个opener属性，其中保存着打开它的原属窗口对象。这个属性只在弹出窗口中的最外层window对象（top）中有定义，而且指向调用window.open()的窗口或框架。



虽然弹出窗口有一个指针指向打开它的原始窗口，但是原始窗口并没有这样的指针指向弹出窗口。因此我们只能在必要的时候自己来手动实现跟踪。

1. close()：该方法可以关闭通过window.open()打开的弹出窗口。对于浏览器的主窗口，如果没有得到用于的允许是不能关闭它的。

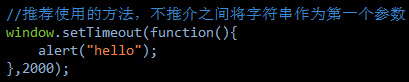
不过，弹出窗口倒是可以调用top.close()在不经用户允许的情况下关闭自己。弹出窗口关闭之后，窗口的引用仍然还在，但是除了像检测closed属性之外，已经没有其他用处了。



### 间歇调用和超时调用

JavaScript是单线程语言，但它允许通过设置超时值和间歇时间值来调度代码在特定的时刻执行。

1. setTimeout()：该方法实现超时调用。它属于window对象的方法。该方法接收两个参数：要执行的代码和以毫秒数表示的时间。

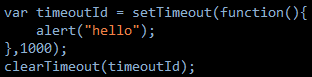


JavaScript是一个单线程序的解释器，因此一定时间内只能执行一段代码。为了控制要执行的代码，就有一个JavaScript任务队

列。这个写任务会按照它们添加到队列的顺序执行setTimeout()的第二个蚕食告诉JavaScript再过多长时间把当前任务添加到队列中。

调用setTimeout()之后，该方法会返回一个数值ID，表示超时调用。这个超时调用ID是计划执行代码的唯一标识符，可以通过它来取消超时调用。

1. clearTimeout()：用于取消设置了超时调用，但是还未执行的超时调用。

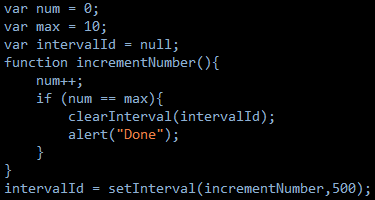


1. setInterval()：间歇调用与超时调用基本类似，只是间歇调用会在指定时间间隔重复执行代码，直到间歇调用被取消或页面被卸载。



调用setInterval()会返回一个间歇调用ID，该ID可用于在将来某个时刻取消间歇调用。

1. clearInterval()：用于取消间歇调用。

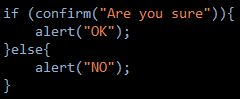


### 系统对话框

浏览器通过alert()、confirm()、prompt()方法可以调用系统对话框项用户显示消息。系统对话框与在浏览器中显示的网页没有关系，也不包含HTML。它们的外观由操作系统及浏览器设置决定，而不是由CSS决定。显示这个对话框的时候代码会停止执行，关闭这些对话框代码又恢复执行。

1. alert()：该方法向用户显示一个系统对话框，其中包含指定的文本和一个OK（确定）按钮。
2. confirm()：该方法向用于显示的对话框与alert()显示的对话框基本类似，只是多了一个Cancel（取消）按钮。一般用于让用户决定是否执行给定的操作。

为了确定用户是单击了OK还是Cancel，可以检测confirm()方法返回的布尔值，true表示单击了OK，false表示单击了Cancel或是点击了关闭。



1. prompt()：该方法生成一个提示框，用于提示用户输入一些文本。提示框除了显示OK和Cancel按钮外，还会I西安市一个文本输入域，以供用户在其中输入内容。该方法接收两个参数：要显示给用户的文本提示和文本输入域的默认值（可以是一个字符串）。如果点击OK，则prompt()返回文本输入域的值，如果点击Cancel，则返回null。

## location对象

location对象提供了与当前窗口中加载的文档有关的信息，还提供了一些导航功能。location对象既是window对象的属性，也是document对象的属性：window.location和document.location引用的是同一个对象。

location还可以将URL解析为独立的片段，让开发人员可以通过不同的属性访问这些片段。下表列出了location对象的所有属性

|  |  |  |
| --- | --- | --- |
| 属性名 | 例子 | 说明 |
| hash |  | 返回URL中的hash(#号后跟零或多个字符)，如果URL中不包含散列，则返回空字符串 |
| host | “www.wrox.com:80” | 返回服务器名称和端口号（如果有） |
| hostname | “www.wrox.com” | 返回不带端口号的服务器名称 |
| href | “http://www.wrox.com” | 返回当前加载页面的完整URL。而location对象的toString()方法也返回这个值 |
| pathname | “/WileyCDA/” | 返回URL中的目录和（或）文件名 |
| port | “8080” | 返回URL中指定的端口号。如果URL中不包含端口号，则这个属性返回空字符串 |
| protocol | “http:” | 返回页面使用的协议。通常是http:或https: |
| search | “?q=javascript” | 返回URL的查询字符串。这个字符串以问号开头 |

### 查询字符串参数

## navigator对象

navigator对象包含有关浏览器的信息

### navigator对象属性

|  |  |
| --- | --- |
| 属性 | 说明 |
| appCodeName | 该属性是一个只读字符串，声明了浏览器的代码名 |
| appMinorVersion | 返回浏览器的次要版本 |
| appName | 返回浏览器的名称。 |
| appVersion | 返回浏览器的平台和版本信息。 |
| browserLanguage | 返回当前浏览器的语言。 |
| cookieEnabled | 返回指明浏览器中是否启用 cookie 的布尔值 |
| cpuClass | 返回浏览器系统的 CPU 等级。 |
| onLine | 返回指明系统是否处于脱机模式的布尔值。 |
| platform | 返回运行浏览器的操作系统平台。 |
| systemLanguage | 返回 OS 使用的默认语言。 |
| userAgent | 返回由客户机发送服务器的 user-agent 头部的值。 |
| userLanguage | 返回 OS 的自然语言设置。 |

### navigator对象方法

|  |  |
| --- | --- |
| 方法 | 说明 |
| javaEnabled() | 规定浏览器是否启用 Java。 |
| taintEnabled() | 规定浏览器是否启用数据污点 (data tainting)。 |

## screen对象

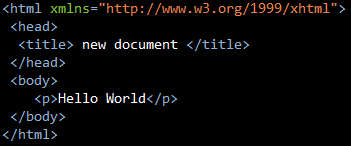
## history对象

# DOM

DOM（文档对象模型）是针对HTML和XML文档的一个API(应用程序编程接口)。DOM描绘了一个层次化的节点树，允许开发人员添加、移除、修改页面的某一部分。DOM不是ECMAScript中的一部分。

## 节点层次

DOM可以将任何HTML或XML文档描绘成一个由多层节点构成的结构。节点分为几种不同的类型，每种类型分别表示文档中不同的信息及（或）标记。每个节点都拥有各自的特点、数据和方法，也与其他节点存在某种关系。节点之间的关系构成了层次，而所有页面标记则表现为一个以特定节点为根节点的树形结构。



文档节点只有一个子节点，我们称之为文档元素。文档元素是文档的最外层元素，文档中的其他所有元素都包含在文档元素中。每个文档中只能有一个文档元素。在HTML中，文档元素始终是<html>元素。在XML中，没有预定义的元素，因此任何元素都可能成为文档元素。

每一段标记都可以通过树中的一个节点来表示：HTML元素通过元素节点表示，特性通过特性节点表示，文档类型通过文档类型节点表示，而注释则通过注释节点表示。总共有12种节点类型，这些类型都继承自一个基本类型。

### Node类型

DOM1级定义了一个Node接口，该接口在JavaScriot中作为Node类型实现的。Node是所有节点的基类型，所有节点都继承自它，所以所有节点都有一些共同的方法和属性。

**方法**

**属性**

nodeType：该属性用于表明节点的类型。

节点类型由在Node类型中定义的下列12个数值常量来表示，任何节点类型必居其一：

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 说明 | Node类型表示 | 常量值 |
| Element | 表示 element（元素标记） | Node.ELEMENT\_NODE | 1 |
| Attr | 表示属性。 | Node.ATTRIBUTE\_NODE | 2 |
| Text | 表示元素或属性中的文本内容。 | Node.TEXT\_NODE | 3 |
| CDATASection | 表示文档中的 CDATA 区段（文本不会被解析器解析） | Node.CDATA\_SECTION\_NODE | 4 |
| EntityReference | 表示实体引用元素。 | Node.ENTITY\_REFERENCE\_NODE | 5 |
| Entity | 表示实体。 | Node.ENTITY\_NODE | 6 |
| ProcessingInstruction | 表示处理指令。 | Node.PROCESSING\_INSTRUCTION\_NODE | 7 |
| Comment | 表示注释。 | Node.COMMENT\_NODE | 8 |
| Document | 表示整个文档（DOM 树的根节点） | Node.DOCUMENT\_NODE | 9 |
| DocumentType | 向为文档定义的实体提供接口。 | Node.DOCUMENT\_TYPE\_NODE | 10 |
| DocumentFragment | 表示轻量级的 Document 对象，其中容纳了一部分文档。 | Node.DOCUMENT\_FRAGMENT\_NODE | 11 |
| Notation | 表示在 DTD 中声明的符号。 | Node.NOTATION\_NODE | 12 |

通过比较上面的这些常量，可以很容易地确定节点的类型：



此例中如果表达式返回true，则表明someNode是一个元素。由于在IE浏览器中没有公开Node类型的构造函数，所以上例中的代码会在IE中报错。所以可以使用数值的方式进行比较达到同样的效果



nodeName和nodeValue：只读属性。该两个属性返回节点的具体信息，该两个属性的值完全取决于节点的类型。所以在使用值的时

候最好检测下节点的类型。



|  |  |  |
| --- | --- | --- |
| 节点类型 | nodeName返回的值 | nodeValue返回的值 |
| Document | #document | null |
| DocumentFragment | #document fragment | null |
| DocumentType | doctype 名称 | null |
| EntityReference | 实体引用名称 | null |
| Element | 元素的名称 | null |
| Attr | 属性名称 | 属性值 |
| ProcessingInstruction | target | 节点的内容 |
| Comment | #comment | 注释文本 |
| Text | #text | 节点内容 |
| CDATASection | #cdata-section | 节点内容 |
| Entity | 实体名称 | null |
| Notation | 符号名称 | null |

childNodes：该属性将返回一个数组，这个数组由给定元素节点的直接子节点构成。其中所有的子节点都具有节点对象常见的属性。

调用childNodes返回的是一个NodeList对象，它极其像数组，但是有一个最关键的地方，它是动态查询的，也就是说

每次调用它都会对DOM结构查询，所以对它的使用需要慎重，注意性能。访问数组中的子节点可以用下标法或者item()

方法



parentNode：该属性将返回一个给定节点的父节点。



firstChild：该属性返回一个给定元素节点的第一个节点。属性值为一个节点对象的引用指针。这个节点对象都有着nodeType 、

nodeName 、nodeValue 等常见的节点属性。



lastChild：该属性返回一个给定元素节点的最后一个节点。属性值为一个节点对象的引用指针。这个节点对象都有着nodeType 、

nodeName 、nodeValue 等常见的节点属性。

nextSibling：该属性返回指定节点之后紧跟的节点，在相同的树层级中。属性值为一个节点对象的引用指针。这个节点对象都有着

nodeType、nodeName、nodeValue等常见的节点属性。

previousSibling：该属性返回选定节点的上一个同级节点（在相同树层级中的前一个节点）。如果不存在这样的节点，则返回null。

### Docunment类型

JavaScript通过Document类型表示文档。在浏览器中document对象是HTMLDocument（继承自Document类型）的一个实例，表示整个HTML页面。document对象时window对象的一个属性，可以作为全局对象来访问。通过document对象可以取得页面的有关信息，还可以操作页面的外观及底层结构。

**属性**

documentElement：该属性直接指向HTML页面中的<html>元素。



body：该属性直接指向<body>元素。



doctype：该属性直接指向<!DOCTYPE>元素。

title：该属性包含着<title>元素中的文本。通过这个属性可以取得页面的标题信息，也可以修改当前页面的标题信息。



URL：该属性包含页面完整的URL

domain：该属性只包含页面的域名

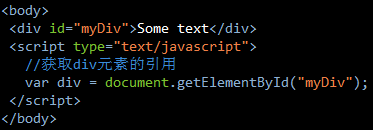
referrer：该属性保存着连接到当前页面的那个页面的URL。如果没有来源页面，则为空字符串。

**方法**

getElementById()：该方法接收一个参数：要取得的元素的ID值。该方法用于获取到特点的某个或某组元素的引用，然后再执行一

些操作。如果不存在带有相应的ID的元素，则返回null。如果页面中多个元素的ID值相同，则该方法只返回文

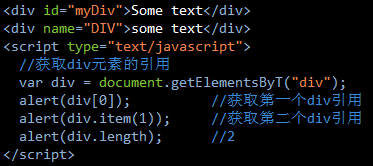
档中第一次出现的元素。



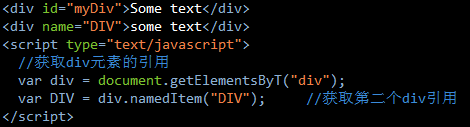
getElementsByTagName()：该方法接收一个参数，即要取得元素的标签名。该方法用于获取到特点的某个或某组元素的引用，然后

再执行一些操作。该方法返回一个HTMLCollection对象，作为一个“动态”集合。可以使用方括号语法

或item()方法来访问HTMLCollection对象中的项。而这个对象中元素的数量可以通过length属性获取。



HTMLCollection对象中的namedItem()方法，通过元素的name值取得集合中的项。



对于HTMLCollection对象来说，用方括号语法表示，如果方括号中传入的数值，则后台默认调用item()

方法，如果传入的是字符串，则后台默认调用namedItem()方法。



如果想取得整个文档的引用，则可以在方法中使用\*星号表示。

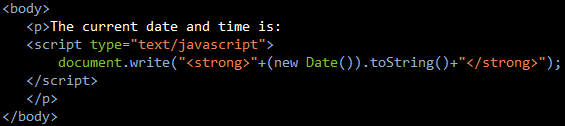


getElementsByName()：该方法接收一个参数，即元素的name属性值。该方法返回带有指定name值的元素引用。

getElementsByName()方法也会返回一个HTMLCollection对象。

write()：该方法接收一个字符串参数，用于将字符串原样写入文档流中。

writeln()：该方法接收一个字符串参数，用于将字符串写入文档流中，并在字符串后面加换行符（\n）。



open()：该方法用于打开网页的输出流。但是在加载期间使用write()和writeln()方法，则不需要使用该方法。

close()：该方法用于关闭网页的输出流。但是在加载期间使用write()和writeln()方法，则不需要使用该方法。

### Element类型

Element类型用于表现XML或HTML元素，提供了对元素标签名，子节点及特性的访问。

属性

nodeName：该属性返回元素的标签名

tagName：该属性返回元素的标签名



HTML元素

所有的HTML元素都是由HTMLElement类型或它的子类型来表示。HTMLElement类型直接继承自Element并添加了一些属性，并

分别对应于每个HTML元素中

id：元素在文档中的唯一标识符。

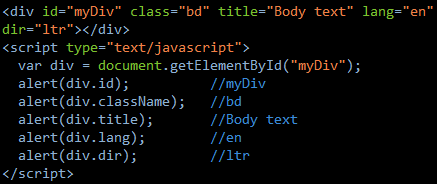
title：有关元素的附加说明信息，一般通过工具提示条显示出来。

lang：元素内容的语言代码，很少使用。

dir：语言的方法，也很少使用。

className：表示元素的class属性（因为class属性是关键字所以以className代替），即为元素指定的css类。

上述的这些属性都可以用来取得或修改相应的属性值。

  
 获取属性的值

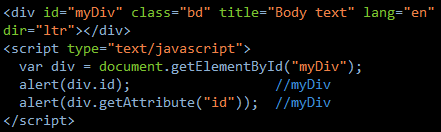
每个元素都有一个或多个属性，在DOM中可以使用DOM对象点属性名的方式获取属性值（除自定义属性外）。但也可以使用

getAttribute()来获取属性的值。

getAttribute()：该方法接收一个参数，即需要返回属性值对应的属性名称。（元素的class属性这里传入class）。该方法返回元素属

性名对应的属性值（包括自定义属性）。除了styple和onclick属性外（返回的属性值与直接使用属性法取得的值不一

致），其余的都与使用属性来取值得到的结果一样。



设置属性值

setAttribute()：该方法接收两个参数，第一个参数需要设置的属性名，第二个参数为设置的值。如果该属性已近存在，则直接修改

属性值。如果该属性不存在，直接添加属性及属性值。该方法即可以操作HTML元素的现有属性，也可以操作自定义

属性。通过该方法设置的属性名会同一转换为小写。

removeAttribute()：该方法接收一个参数，即需要删除的元素属性名。此方法用于彻底删除元素的属性值和属性。

遍历元素的属性及值

attributes：该属性中包含一个NamedNodeMap对象。该对象保存着元素的所有属性及属性值。都是以Attr节点的形式保存。每个

节点的nodeName就是属性的名称，每个节点的nodeValue就是属性的值。

NamedNodeMap对象用于以下方法：

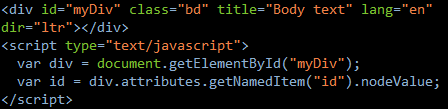
getNamedItem(name)：该方法返回nodeName属性等于name的节点

removeNamedItem(name)：该方法移除nodeName属性等于name的节点。

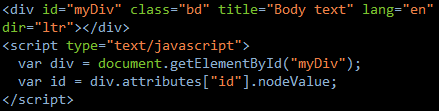
setNamedItem(node)：向列表中添加节点，以节点的nodeName属性为索引。

item(pos)：返回位于数字pos位置处的节点

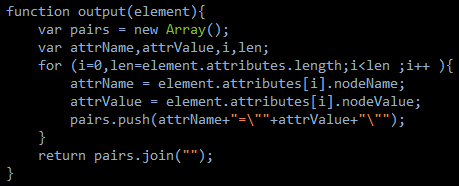
获取ID属性的值：

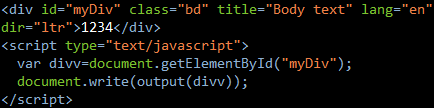


方括号表示法：



遍历元素的属性及值





创建元素

document.createElement()：该方法用于在网页文档中创建新元素。此方法接收一个参数，即需要创建的元素名。



该方法在创建新元素的同时，也为新元素设置了ownerDocemnt属性。此时可以给新的元素设置属性和属性值。

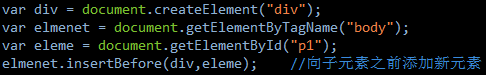


添加新标签到文档树

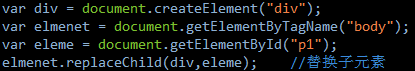
appendChild()：该方法在指定的元素中添加子节点元素，添加的新元素在子元素最后一位。



insertBefore()：该方法是在指定的子元素前面插入新的子元素。



replaceChild()：该方法是替换指定的子元素。



删除已有的HTML元素

removeChild()：删除指定的元素。



### Text类型

在HTML文档树中，所有的文本内容都是由Text类型表示。其中nodeValue和data属性可以获取文本内容。同样也可以修改文本内容。length属性保存着文本的字符数。文本节点必须要有文本内容才有文本节点（包括空格符），没有文本内容就不存在文本节点

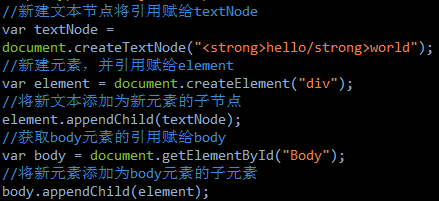
方法

document.createTextNode()：创建新的文本节点，作为参数的文本内容将按照HTML或XML的格式进行编码，如果参数中包含元素文

本则解析器会将其以元素的形式解析。



将新建的文本节点添加到文档树中



appendData(text)：将text添加到节点的末尾。

deleteData(offset,count)：从offset指定的位置开始删除count个字符。

insertData(offset,text)：在offset指定的位置插入text内容。

replaceData(offset,count,text)：用text替换从offset指定位置开始到count位置位置处的文本。

splitText(offset)：从offset指定的位置将文本节点分成两个文本节点。返回一个新的文本节点。

substringData(offset,count)：提取从offset指定的位置开始到count位置处的字符串。

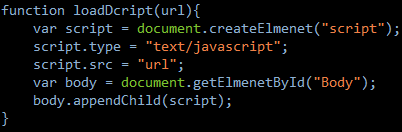
## DOM的操作

### 动态脚本

动态脚本指的是在页面加载的时候JavaScript代码不存在，而在将来通过修改DOM动态添加的脚本。

创建动态脚本有两种方式：插入外部文件和直接插入JavaScript代码。

动态加载外部JS文本



将JS代码加载到文档流中

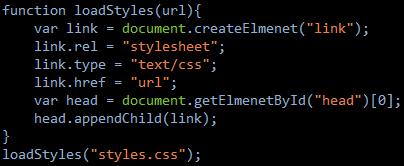


### 动态样式

动态样式是指在页面刚加载时不存在的样式，通过后期的一些操作该动态加载新的样式。

动态样式也分两种方式：加载外部样式文本和将样式代码写入文档中。

加载外部样式文本



将样式写入文档中



### 操作表格

<table>元素是HTML中最复杂的结构之一。要想创建表格，一般都必须涉及表示表格行、单元格、表头等方面的标签。在DOM中为<table>、<tbody>、<tr>元素添加了一些属性和方法以便减少DOM代码量。

为<table>元素添加的属性和方法：

caption：保存着对<caption>元素（如果有）的指针。

tBodies：是一个<tbody>元素的HTMLCollection。

tFoot：保存着对<tfoot>元素（如果有）的指针。

tHead：保存着对<thead>元素（如果有）的指针。

rows：是一个表格中所有行的HTMLCollection。

createTHead()：创建<thead>元素，将其放到表格中，返回引用。

createTFoot()：创建<tfoot>元素，将其放到表格中，返回引用。

createCaption()：创建<caption>元素，将其放到表格中，返回引用。

deleteTHead()：删除<thead>元素。

deleteTFoot()：删除<tfoot>元素。

deleteCaption()：删除<caption>元素。

deleteRow(pos)：删除指定位置的行。

insertRow(pos)：项rows几个中的指定位置插入一行。

为<tbody>元素添加的属性和方法：

rows：保存着<tbody>元素中行的HTMLCollection。

deleteRow(pos)：删除指定位置的行。

insertRow(pos)：向rows集合中的指定位置插入一行，返回对新插入行的引用。

为<tr>元素添加的属性和方法：

cells：保存着<tr>元素中单元格的HTMLCollection。

deleteCell(pos)：删除指定位置的单元格。

insertCell(pos)：项cells集合中的指定位置插入一个单元格，但会对新插入单元格的引用。

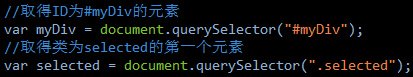


## DOM扩展

### 选择符API

Selectors API（选择符API）是由W3C发起制定的一个标准，致力于让浏览器原生支持CSS查询。通过CSS选择符查询DOM文档取得元素的引用，从而抛开了getElementById()和getElementsByTagName()。其中jQuery的核心就是这样。

querySelector()：该方法接收一个CSS选择符，返回与该选择符匹配的第一个元素，如果没有找到匹配的元素，返回null。



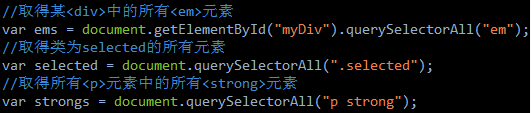
通过Document类型调用querySelector()方法是，会在文档元素的范围内查找匹配的元素。而通过Element类型调用

querySelector()方法是，只会在该元素后代元素的范围内查找匹配的元素

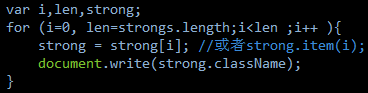
querySelectorAll()：该方法接收一个CSS选择符，返回的是所有原选择符匹配的元素。结果为NodeList的对象。如果没有找到匹配

的，则NodeList就是空的

能够调用querySelectorAll()对象的类型包括Document、DocumentFragment、Element类型。



要取得返回的NodeList中的每一个元素，可以使用item()方法，也可以使用方括号表示法。



matchesSelector()：该方法接收一个CSS选择符。为Element类型新增的方法。如果调用元素与该选择符匹配，则返回true，否则

返回fulse。

但是每个浏览器都各有各的支持方法。

### HTML5

getElementsByClassName()：该方法可以通过document对象及所有HTML元素调用获得元素的引用。此方法接收一个参数，即一

个包含一或多个类名的字符串，返回带有指定类的所有元素的NodeList。



document调用该方法，返回所有元素中类名与之相匹配的元素引用，元素调用该方法返回元素的子元素

中与之相匹配的元素。

classList：该属性是HTML5中新增加的操作类名的属性。通过给所有元素添加classList属性来实现对类名的修改

该属性是新集合类型DOMTokenList的实例，拥有表示自己包含多少元素的length属性和取得每个元素的item()方法或者

方括号法。

新类型中定义的方法：

add()：将给定的字符串值添加到列表中。如果值已经存在，就不添加。

contains()：表示列表中是否存在给定的值，如果存在则返回true，否则返回false。

remove()：从列表中删除给定的字符串。

toggle()：如果列表中已经存在给定的值，删除它；如果没有，添加它。

document.activeElement：该属性始终会引用DOM中当前获得了焦点的元素。

document.hasFocus()：该方法用于确定文档是否获得了焦点。

focus()：该方法用于获取焦点。

document.readySyaye：该属性用于检测文档加载的情况。属性值为：loading（正在加载文档）或complete（已经加载完文档）。

document.compatMode：该属性用于检测浏览器采用了那种渲染模式。属性值为：CSS1Compat（标准模式）或BackCompat（混杂

模式）。

document.head：该属性实现对文档的<head>元素的引用。

document.charset：该属性用于获取文档使用的字符集，也可以使用该属性来设置文档的字符集。

document.defaultCharset：该属性获取当前默认使用的字符集应该是什么（根据浏览器及操作系统的设置）。

在HTML5中规定了给元素添加非标准的属性时，都必须要添加前缀data-，目的是为元素提供与渲染无关的信息或者提供语义讯息。

innerHTML：该属性在读模式下，返回与调用元素的所有自己诶单对应的HTML标记。在写模式下，该属性会根据指定的值创建新的

DOM树，然后用这个DOM数完全替换调用元素原先的所有子节点。

outerHTML：该属性在读模式下，返回调用它的元素及所有子节点的HTML标签。在写模式下，该属性会根据指定的HTML字符串

创建新的DOM子树，然后用这个DOM子树完全替换调用元素。

insertAdjacentHTML()：该方法用于将指点的元素插入文档树中。该方法接收两个参数：插入的位置和插入的HTML文本。

第一个参数必须是下列值之一：

beforebegin：在当前元素之前插入一个紧邻的同辈元素。

afterbegin：在当前元素之下插入一个新的子元素或在第一个子元素之前再插入新的子元素。

beforeend：在当前元素之下插入一个新的子元素或在最后一个子元素之后再插入新的子元素。

afterend：在当前元素之后插入一个紧邻的同辈元素。

第二个元素是一个HTML字符串。

scrollIntoView()：该方法可以在所有的HTML元素上调用，通过滚动浏览器窗口或某个容器元素，调用元素就可以出现在视口中。

如果给这个方法传入true作为参数，或者不传入任何参数，那么窗口滚动之后会让调用原的顶部与视口顶部尽可

能平齐。如果传入false作为参数，调用元素会尽可能全部出现的视口中，不过顶部不一定平齐。

# DOM2和DOM3

# 事件

JavaScript与HTML之间的交互是通过事件来实现的。事件，就是文档或浏览器窗口中发生的一些特定的交互瞬间。可以使用侦听器（或处理程序）来预定时间，以便时间发生时执行相应的代码。

## 事件处理程序

事件就是用户或浏览器自身执行的某些动作。如click、load、mouseover都是事件的名字。而响应某个事件的函数就叫做事件处理程序（或事件侦听器）。事件处理程序的名字以“on”开头，因此click事件的事件处理程序就是onclick。

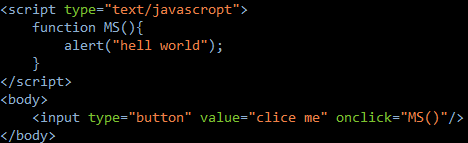
### 行内事件处理器

某个元素支持的每种事件，都可以使用一个与相应事件处理程序同名的HTML属性来指定。这个属性的值应是能够执行的JavaScript代码或者调用定义好的函数。

例如：要在按钮被单击时执行一些JavaScript



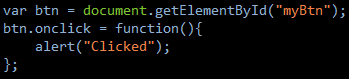
点击按钮调用函数。



这种通过THML事件属性与JavaScript执行程序绑定的方式是早期使用的方式，现在不建议频繁通过这种方式绑定事件。这会使得HTML代码和JavaScript代码变得混乱难以维护。

### 事件处理器属性

通过JavaScript指定事件处理程序的传统方式就是将一个函数赋值给一个事件处理程序属性。要使用JavaScript指定事件处理程序，首选必须取得一个要操作的DOM对像的引用。



需要删除标签上的事件，只需要将属性值设置为null则实现了删除事件处理程序的效果。

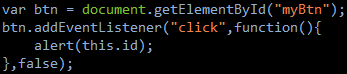
### 事件监听器

DOM2级事件规范给浏览器提供了两个方法，用于处理指定和删除事件处理程序的操作。所有DOM节点中都包含这个两个方法。

addEventListener()：该方法用于为元素添加事件监听器。该方法接收3个参数：要处理的事件名、作为事件处理程序的函数、一

个布尔值。其中布尔值如果是true，表示在捕获阶段调用事件处理程序；为false，表示在冒泡阶段调用事件处

理程序。



removeEventListener()：该方法用于移除通过addEventListener()方法创建的事件监听器。该方法所接受的参数与

addEventListener()方法一致，但是removeEventListener()无法删除定义了匿名函数的事件处理程序。

通过事件监听器的方式给节点绑定事件的好处在于可以同时给节点绑定多个事件处理程序。

您也可以给同一个监听器注册多个处理器，下面这种方式不能实现这一点：



第二行会覆盖第一行，但是下面这种方式就会正常工作了：



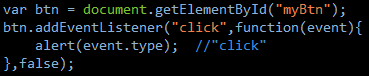
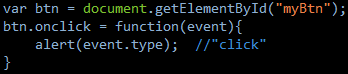
当元素被点击时两个函数都会工作

## 事件处理对象

在触发DOM上的某个事件时，会产生一个事件对象event，这个对象中包含着所有与事件有关的信息。包括导致事件的元素、事件的类型以及其他与特定事件相关的信息。它被自动传递给事件处理函数，以提供额外的功能和信息。

**DOM中的事件对象**

兼容DOM的浏览器会将一个event对象传入到事件处理程序中。无论指定事件处理程序时使用什么方法，都会传入event对象。

在通过HTML属性指定事件处理程序时，变量event中保存着event对象



以这种方式提供event对象，可以让HTML属性事件处理程序与JavaScript函数执行相同的操作。

event对象包含于创建它的特定事件有关的属性和方法。触发的事件类型不一样，可用的属性和方法也不一样。但是所有时间都会有

下表中列出的属性和方法

|  |  |  |  |
| --- | --- | --- | --- |
| 属性/方法 | 类型 | 读/写 | 说明 |
| bubbles | Boolean | 只读 | 表明事件是否冒泡 |
| cancelable | Boolean | 只读 | 表明是否可以取消事件的默认行为 |
| currentTarget | Element | 只读 | 其事件处理程序当前正在处理事件的那个元素 |
| defaultPrevented | Boolean | 只读 | 为true表示已经调用了preventDefault()（DOM3级事件中新增） |
| detail | Integer | 只读 | 与事件相关的细节信息 |
| eventPhase | Integer | 只读 | 调用事件处理程序的阶段：1表示捕获阶段，2表示“处于目标”，3表示冒泡阶段 |
| target | Element | 只读 | 返回触发此事件的元素（事件的目标节点） |
| trusted | Boolean | 只读 | 为true表示事件是浏览器生成。为false表示事件通过JavaScript创建的 |
| type | String | 只读 | 被触发的事件的类型 |
| view | AbstractView | 只读 | 与事件关联的抽象视图。等同于发生事件的window对象 |
| initEvent() | Function | 只读 | 初始化新创建的 Event 对象的属性。 |
| preventDefault() | Function | 只读 | 取消事件的默认行为。如果cancelable是true，则可以使用这个方法 |
| stopImmediatePropagation() | Function | 只读 | 取消事件的进一步捕获或冒泡，同时阻止任何事件处理程序被调用 |
| stopPropagation() | Function | 只读 | 取消事件的进一步捕获或冒泡。如果bubbles为true，则可以使用 |

## 阻止事件默认行为

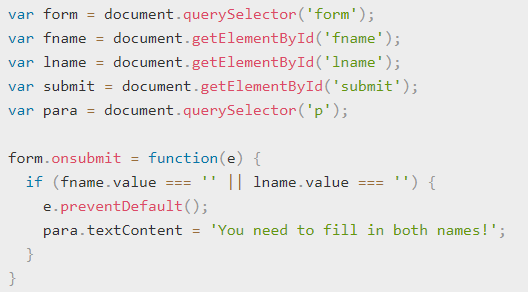
有时，你会遇到一些情况，你希望事件不执行它的默认行为。 最常见的例子是Web表单，例如自定义注册表单。 当你填写详细信息并按提交按钮时，自然行为是将数据提交到服务器上的指定页面进行处理，并将浏览器重定向到某种“成功消息”页面（或 相同的页面，如果另一个没有指定。）

当用户没有正确提交数据时，麻烦就来了 - 作为开发人员，你希望停止提交信息给服务器，并给他们一个错误提示，告诉他们什么做错了，以及需要做些什么来修正错误。 一些浏览器支持自动的表单数据验证功能，但由于许多浏览器不支持，因此建议你不要依赖这些功能，并实现自己的验证检查。 我们来看一个简单的例子。

首先，一个简单的HTML表单，需要你填入名（first name）和姓（last name）



这里我们用一个onsubmit事件处理程序（在提交的时候，在一个表单上发起submit事件）来实现一个非常简单的检查，用于测试文本字段是否为空。 如果是，我们在事件对象上调用preventDefault()函数，这样就停止了表单提交，然后在我们表单下面的段落中显示一条错误消息，告诉用户什么是错误的：



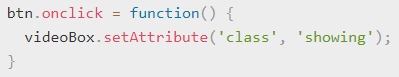
## 事件冒泡及捕获

事件冒泡和捕捉是两种机制，主要描述当在一个元素上有两个相同类型的事件处理器被激活会发生什么。让我们看一个例子让这些更简单点。

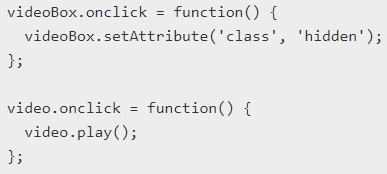
这是一个非常简单的例子，它显示和隐藏一个包含<video>元素的<div>元素：



当‘button’元素按钮被单击时，将显示视频，它是通过将改变<div>的class属性值从hidden变为showing(这个例子的CSS包含两个class，它们分别控制这个<div>盒子在屏幕上显示还是隐藏。)：



然后我们再添加几个onclick事件处理器，第一个添加在<div>元素上，第二个添加在<video>元素上。这个想法是当视频(<video>）外 <div>元素内这块区域被单击时，这个视频盒子应该再次隐藏；当单击视频(<video>）本身，这个视频将开始播放。



但是有一个问题 - 当您点击video开始播放的视频时，它会在同一时间导致<div>也被隐藏。 这是因为video在<div>之内 - video是<div>的一个子元素 - 所以点击video实际上是同时也运行<div>上的事件处理程序。

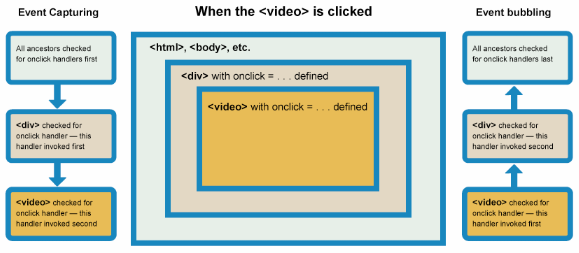
**对事件冒泡和捕捉的解释**

当一个事件发生在具有父元素的元素上(例如，在我们的例子中是<video>元素)时，现代浏览器运行两个不同的阶段 - 捕获阶段和冒泡阶段。 在捕获阶段：

* 浏览器检查元素的最外层祖先<html>，是否在捕获阶段中注册了一个onclick事件处理程序，如果是，则运行它。
* 然后，它移动到<html>中单击元素的下一个祖先元素，并执行相同的操作，然后是单击元素再下一个祖先元素，依此类推，直到到达实际点击的元素。

在冒泡阶段，恰恰相反:

* 浏览器检查实际点击的元素是否在冒泡阶段中注册了一个onclick事件处理程序，如果是，则运行它
* 然后它移动到下一个直接的祖先元素，并做同样的事情，然后是下一个，等等，直到它到达<html>元素。

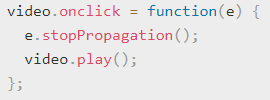


在现代浏览器中，默认情况下，所有事件处理程序都在冒泡阶段进行注册。因此，在我们当前的示例中，当您单击视频时，这个单击事件从 <video>元素向外冒泡直到<html>元素。沿着这个事件冒泡线路：

* 它发现了video.onclick...事件处理器并且运行它，因此这个视频<video>第一次开始播放。
* 接着它发现了（往外冒泡找到的） videoBox.onclick...事件处理器并且运行它，因此这个视频<video>也隐藏起来了。

**用stopPropagation()修复问题**

这是令人讨厌的行为，但有一种方法来解决它！标准事件对象具有可用的名为 stopPropagation()的函数, 当在事件对象上调用该函数时，它只会让当前事件处理程序运行，但事件不会在冒泡链上进一步扩大，因此将不会有更多事件处理器被运行(不会向上冒泡)。所以，我们可以通过改变前面代码块中的第二个处理函数来解决当前的问题:

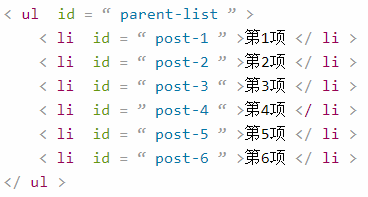


**事件委托**

冒泡还允许我们利用事件委托——这个概念依赖于这样一个事实,如果你想要在大量子元素中单击任何一个都可以运行一段代码，您可以将事件监听器设置在其父节点上，并将事件监听器气泡的影响设置为每个子节点，而不是每个子节点单独设置事件监听器。

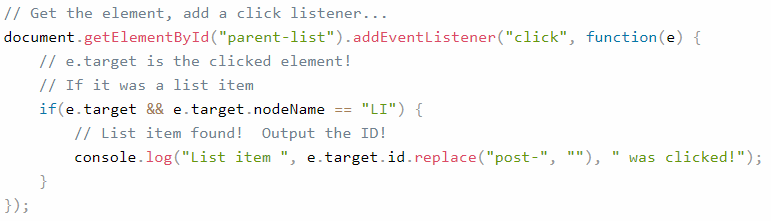
一个很好的例子是一系列列表项，如果你想让每个列表点击时弹出一条信息，您可以将click单击事件监听器设置在父元素<ul>上，它将会冒泡到列表项上。

假设我们有一个ul包含多个子元素的父元素：

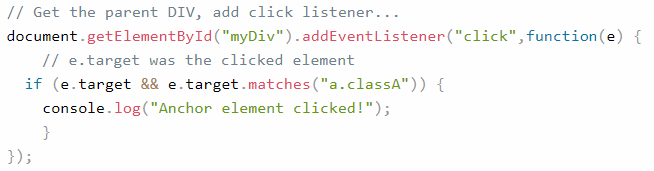


我们还要说，点击每个子元素时需要发生一些事情。您可以为每个单独的li元素添加单独的事件侦听器，但是如果li经常在列表中添加和删除元素会怎么样？添加和删​​除事件侦听器将是一场噩梦，尤其是如果添加和删除代码位于应用程序中的不同位置。更好的解决方案是向父ul元素添加事件侦听器。但是，如果将事件侦听器添加到父级，您将如何知道单击了哪个元素？

简单：当事件冒泡到ul元素时，检查事件对象的target属性以获取对实际单击节点的引用。这是一个非常基本的JavaScript代码段，用于说明事件委派：



首先向click父元素添加一个事件监听器。触发事件侦听器时，检查事件元素以确保它是要响应的元素类型。如果它是一个li元素。



由于大多数开发人员使用JavaScript库进行DOM元素和事件处理，因此我建议使用库的事件委派方法，因为它们能够进行高级委派和元素识别。

## 事件类型

### UI事件

UI事件指的是那些不一定与用户操作有关的事件。

1. load事件

当页面完成加载后(包括所有图像、JavaScript文件、CSS文件等外部资源)就会触发window上面的load事件。当所有框架都加

在完毕时在框架集上面触发，当图像加载完毕时在<img>元素上面触发，或者当嵌入的内容加载完毕时在<object>元素上面触

发。

定义onload事件的方式

