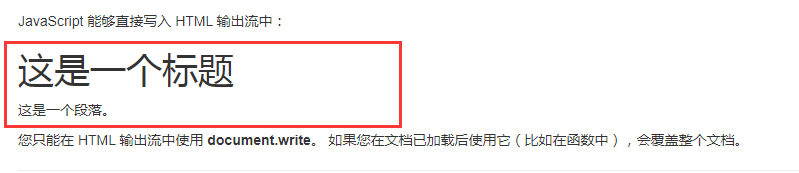
## JavaScript 教程

JavaScript 是 Web 的编程语言。所有现代的 HTML 页面都使用 JavaScript。

1. HTML 定义了网页的内容
2. CSS 描述了网页的布局
3. JavaScript 网页的行为

#### 实例：直接写入 HTML 输出流



<p>

JavaScript 能够直接写入 HTML 输出流中：

</p>

<script>

document.write("<h1>这是一个标题</h1>");

document.write("<p>这是一个段落。</p>");

</script>

<p>

您只能在 HTML 输出流中使用 <strong>document.write</strong>。

如果您在文档已加载后使用它（比如在函数中），会覆盖整个文档。

</p>

注：只能在 HTML 输出中使用 document.write。如果您在文档加载后使用该方法，会覆盖整个文档。

#### 实例：对事件的反应

<h1>我的第一个 JavaScript</h1>

<p>JavaScript 能够对事件作出反应。比如对按钮的点击：</p>

<button type="button" onclick="alert('欢迎!')">点我!</button>



#### 实例：改变 HTML 内容

<h1>我的第一段 JavaScript</h1>

<p id="demo">

JavaScript 能改变 HTML 元素的内容。

</p>

<script>

function myFunction()

{

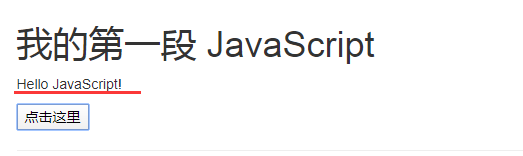
x=document.getElementById("demo"); // 找到元素

x.innerHTML="Hello JavaScript!"; // 改变内容

}

</script>

<button type="button" onclick="myFunction()">点击这里</button>

document.getElementById("some id")。这个方法是 HTML DOM 中定义的。

DOM (Document Object Model)（文档对象模型）是用于访问 HTML 元素的正式 W3C 标准。

#### 实例：改变 HTML 图像

{% load static %}

<script>

function changeImage()

{

element = document.getElementById('myimage')

if (element.src.match("bulbon"))

{

element.src = "/static/app/img/pic\_bulboff.gif";

}

else

{

element.src = "/static/app/img/pic\_bulbon.gif";

}

}

</script>

<img id="myimage" onclick="changeImage()" src="{% static 'app/img/pic\_bulboff.gif' %}" width="100" height="180">

<p>点击灯泡就可以打开或关闭这盏灯</p>

注意图像路径问题：

1{% load static %}解决Django静态文件引用

2在JS代码中使用路径/static/...指向静态文件



实例中代码 element.src.match("bulbon") 的作用意思是：检索 <img id="myimage" onclick="changeImage()" src="/images/pic\_bulboff.gif" width="100" height="180"> 里面 src 属性的值有没有包含 bulbon 这个字符串，如果存在字符串 bulbon，图片 src 更新为 bulboff.gif，若匹配不到 bulbon 字符串，src 则更新为 bulbon.gif

#### 实例：改变 HTML 样式

<p id="demo">

JavaScript 能改变 HTML 元素的样式。

</p>

<script>

function myFunction()

{

x = document.getElementById("demo") // 找到元素

x.style.color = "#ff0000"; // 改变样式

}

</script>

<button type="button" onclick="myFunction()">点击这里</button>



#### 实例：验证输入

<h1>我的第一段 JavaScript</h1>

<p>请输入数字。如果输入值不是数字，浏览器会弹出提示框。</p>

<input id="demo" type="text">

<script>

function myFunction()

{

var x = document.getElementById("demo").value;

if (x == "" || isNaN(x))

{

alert("不是数字");

}

}

</script>

<button type="button" onclick="myFunction()">点击这里</button>



## JavaScript 用法

HTML 中的脚本必须位于 <script> 与 </script> 标签之间，可被放置在 HTML 页面的 <body> 和 <head> 部分中。

脚本可位于 HTML 的 <body> 或 <head> 部分中，或者同时存在于两个部分中。通常的做法是把函数放入 <head> 部分中，或者放在页面底部。这样就可以把它们安置到同一处位置，不会干扰页面的内容。

也可以把脚本保存到外部文件中。外部文件通常包含被多个网页使用的代码。外部 JavaScript 文件的文件扩展名是 .js。如需使用外部文件，请在 <script> 标签的 "src" 属性中设置该 .js 文件：

## JavaScript 输出

JavaScript 没有任何打印或者输出的函数。

### JavaScript 显示数据

JavaScript 可以通过不同的方式来输出数据：

* 使用 window.alert() 弹出警告框。
* 使用 document.write() 方法将内容写到 HTML 文档中。
* 使用 innerHTML 写入到 HTML 元素。
* 使用 console.log() 写入到浏览器的控制台。

### 使用 window.alert()

弹出警告框来显示数据

#### 实例：警告框显示数据

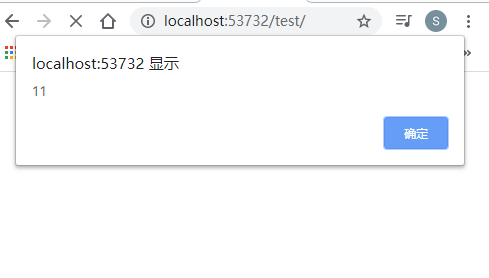
<h1>我的第一个页面</h1>

<p>我的第一个段落。</p>

<script>

window.alert(5 + 6);

</script>



### 操作 HTML 元素

如需从 JavaScript 访问某个 HTML 元素，您可以使用 document.getElementById(id) 方法。请使用 "id" 属性来标识 HTML 元素，并 innerHTML 来获取或插入元素内容：

#### 实例：操作HTML元素

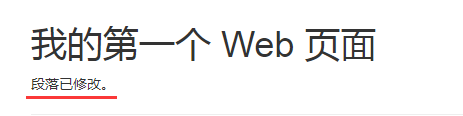
<h1>我的第一个 Web 页面</h1>

<p id="demo">我的第一个段落</p>

<script>

document.getElementById("demo").innerHTML = "段落已修改。";

</script>



注：document.getElementById("demo") 是使用 id 属性来查找 HTML 元素的 JavaScript 代码 。innerHTML = "段落已修改。" 是用于修改元素的 HTML 内容(innerHTML)的 JavaScript 代码。

### 写到 HTML 文档

#### 实例：写到 HTML 文档

<h1>我的第一个 Web 页面</h1>

<p>我的第一个段落。</p>

<script>

document.write(Date());

</script>



注：请使用 document.write() 仅仅向文档输出写内容。如果在文档已完成加载后执行 document.write，整个 HTML 页面将被覆盖。

### 写到控制台

如果您的浏览器支持调试，你可以使用 console.log() 方法在浏览器中显示 JavaScript 值。浏览器中使用 F12 来启用调试模式， 在调试窗口中点击 "Console" 菜单。

#### 实例：写到控制台

<h1>我的第一个 Web 页面</h1>

<script>

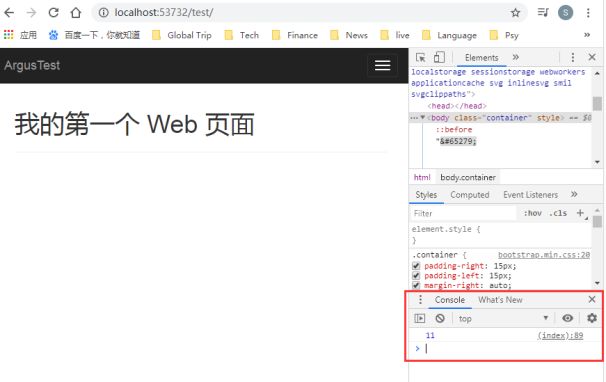
a = 5;

b = 6;

c = a + b;

console.log(c);

</script>



## 语法

### 字面量

在编程语言中，一般固定值称为字面量，如 3.14。

* **数字（Number）字面量** ：可以是整数或者是小数，或者是科学计数(e)。
* **字符串（String）字面量**：可以使用单引号或双引号:

"John Doe"

'John Doe'

* **表达式字面量**：用于计算：

5 + 6

5 \* 10

* **数组（Array）字面量**定义一个数组：

[40, 100, 1, 5, 25, 10]

* **对象（Object）字面量** 定义一个对象：

{firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"}

* **函数（Function）字面量**定义一个函数**：**

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

### 变量

* 变量用于存储数据值。JavaScript 使用关键字 var 来定义变量， 使用等号来为变量赋值：

var x, length

x = 5

length = 6

* 变量可以通过变量名访问。在指令式语言中，变量通常是可变的。字面量是一个恒定的值。变量是一个名称。字面量是一个值。

### 操作符

JavaScript语言有多种类型的运算符：

|  |  |  |
| --- | --- | --- |
| **类型** | **实例** | **描述** |
| 赋值，算术和位运算符 | =  +  -  \*  / | 在 JS 运算符中描述 |
| 条件，比较及逻辑运算符 | ==  != <  > | 在 JS 比较运算符中描述 |

### 语句

语句是用分号分隔：

x = 5 + 6;

y = x \* 10;

### 关键字

以下是 JavaScript 中最​​重要的保留字（按字母顺序）：

|  |  |  |  |
| --- | --- | --- | --- |
| abstract | else | instanceof | super |
|  |  |  |  |
| boolean | enum | int | switch |
|  |  |  |  |
| break | export | interface | synchronized |
|  |  |  |  |
| byte | extends | let | this |
|  |  |  |  |
| case | false | long | throw |
|  |  |  |  |
| catch | final | native | throws |
|  |  |  |  |
| char | finally | new | transient |
|  |  |  |  |
| class | float | null | true |
|  |  |  |  |
| const | for | package | try |
|  |  |  |  |
| continue | function | private | typeof |
|  |  |  |  |
| debugger | goto | protected | var |
|  |  |  |  |
| default | if | public | void |
|  |  |  |  |
| delete | implements | return | volatile |
|  |  |  |  |
| do | import | short | while |
|  |  |  |  |
| double | in | static | with |

### 注释

双斜杠 **//** 后的内容将会被浏览器忽略：

// 我不会执行

### 数据类型

var length = 16; // Number 通过数字字面量赋值

var points = x \* 10; // Number 通过表达式字面量赋值

var lastName = "Johnson"; // String 通过字符串字面量赋值

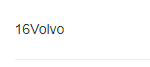
var cars = ["Saab", "Volvo", "BMW"]; // Array 通过数组字面量赋值

var person = { firstName: "John", lastName: "Doe" }; // Object 通过对象字面量赋值

**在运算中注意数据类型：**

var x = 16 + "Volvo" //不同数据类型的运算

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



### 函数

**引用一个函数** = 调用函数(执行函数内的语句)。

function myFunction(a, b) {

return a \* b; // 返回 a 乘以 b 的结果

}

### 字母大小写

JavaScript 对大小写是敏感的。

当编写 JavaScript 语句时，请留意是否关闭大小写切换键。

函数 **getElementById** 与 **getElementbyID** 是不同的。

同样，变量 **myVariable** 与 **MyVariable** 也是不同的。

### 字符集

JavaScript 使用 Unicode 字符集。

Unicode 覆盖了所有的字符，包含标点等字符。

## 语句

### 代码

JavaScript 代码是 JavaScript 语句的序列。

浏览器按照编写顺序依次执行每条语句。

### 代码块

JavaScript 可以分批地组合起来。代码块**以左花括号开始，以右花括号结束**。

代码块的作用是一并地执行语句序列。

### 语句标识符

JavaScript 语句通常以一个 **语句标识符** 为开始，并执行该语句。语句标识符是保留关键字不能作为变量名使用。下表列出了 JavaScript 语句标识符 (关键字) ：

|  |  |
| --- | --- |
| **语句** | **描述** |
| break | 用于跳出循环。 |
| catch | 语句块，在 try 语句块执行出错时执行 catch 语句块。 |
| continue | 跳过循环中的一个迭代。 |
| do ... while | 执行一个语句块，在条件语句为 true 时继续执行该语句块。 |
| for | 在条件语句为 true 时，可以将代码块执行指定的次数。 |
| for ... in | 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）。 |
| function | 定义一个函数 |
| if ... else | 用于基于不同的条件来执行不同的动作。 |
| return | 退出函数 |
| switch | 用于基于不同的条件来执行不同的动作。 |
| throw | 抛出（生成）错误 。 |
| try | 实现错误处理，与 catch 一同使用。 |
| var | 声明一个变量。 |
| while | 当条件语句为 true 时，执行语句块。 |

### 空格

JavaScript 会忽略多余的空格。您可以向脚本添加空格，来提高其可读性。下面的两行代码是等效的：

var person="Hege";

var person = "Hege";

### 对代码行进行折行

可以在文本字符串中使用反斜杠对代码行进行换行。

document.write("你好 \

世界!");

不过，您不能像这样折行：

document.write \

("你好世界!");

## 注释

JavaScript 不会执行注释。我们可以添加注释来对 JavaScript 进行解释，或者提高代码的可读性。

* 单行注释：以 // 开头。

// 输出标题：

document.getElementById("myH1").innerHTML="欢迎来到我的主页";

// 输出段落：

document.getElementById("myP").innerHTML="这是我的第一个段落。";

* 多行注释：以 /\* 开始，以 \*/ 结尾。

/\*

下面的这些代码会输出

一个标题和一个段落

并将代表主页的开始

\*/

document.getElementById("myH1").innerHTML="欢迎来到我的主页";

document.getElementById("myP").innerHTML="这是我的第一个段落。";

* 行末使用注释

var x=5; // 声明 x 并把 5 赋值给它

var y=x+2; // 声明 y 并把 x+2 赋值给它

## 变量

变量必须以字母开头

变量也能以 $ 和 \_ 符号开头（不过我们不推荐这么做）

变量名称对大小写敏感（y 和 Y 是不同的变量）

声明（创建）变量：使用关键字var声明

var carname; //只声明变量，没有赋值，值实际上是 undefined

var carname="Volvo"; //声明变量并赋值

注：一个好的编程习惯是，在代码开始处，统一对需要的变量进行声明。

一条语句，多个变量：

可以在一条语句中声明很多变量。该语句以 var 开头，并使用逗号分隔变量即可：

var lastname="Doe", age=30, job="carpenter";

声明也可横跨多行：

var lastname="Doe",

age=30,

job="carpenter";

注：一条语句中声明的多个不可以赋同一个值: var x,y,z=1;，此时x,y 为 undefined， z 为 1。

Value = undefined ：未使用值来声明的变量，其值实际上是 undefined。当输出未赋值的变量到html时会显示为字符串"undefined"

重新声明 JavaScript 变量：如果重新声明 JavaScript 变量，该变量的值不会丢失：

var carname="Volvo";

var carname; //变量 carname 的值依然是 "Volvo"

## 数据类型

**值类型(基本类型)**：字符串（String）、数字(Number)、布尔(Boolean)、对空（Null）、未定义（Undefined）、Symbol。

**引用数据类型**：对象(Object)、数组(Array)、函数(Function)。

### 拥有动态类型

JavaScript 拥有动态类型。这意味着相同的变量可用作不同的类型：

var x; // x 为 undefined

x = 5; // 现在 x 为数字

x = "John"; // 现在 x 为字符串

### 字符串

字符串可以是引号中的任意文本，可以使用单引号或双引号。

可以在字符串中使用引号，只要不匹配包围字符串的引号即可。

var answer="It's alright";

var answer="He is called 'Johnny'";

var answer='He is called "Johnny"';

### 数字

数字可以带小数点，也可以不带。极大或极小的数字可以通过科学（指数）计数法来书写：

var y=123e5; // 12300000

var z=123e-5; // 0.00123

### 布尔

布尔（逻辑）只能有两个值：true 或 false。

var x=true;

var y=false;

### 数组

例子：

var cars=new Array(); //数组下标是基于零的，所以第一个项目是 [0]，第二个是 [1]

cars[0]="Saab";

cars[1]="Volvo";

cars[2]="BMW";

var cars=new Array("Saab","Volvo","BMW");

var cars=["Saab","Volvo","BMW"];

### 对象

对象由花括号分隔。在括号内部，对象的属性以名称和值对的形式 (name : value) 来定义。属性由逗号分隔：

var person={firstname:"John", lastname:"Doe", id:5566};

声明可横跨多行：

var person = {

firstname: "John",

lastname: "Doe",

id: 5566

};

对象属性有两种寻址方式：

name=person.lastname;

name=person["lastname"];

### Undefined 和 Null

Undefined 这个值表示变量不含有值。

可以通过将变量的值设置为 null 来清空变量：

cars=null;

person=null;

### 声明变量类型

当您声明新变量时，可以使用关键词 "new" 来声明其类型：

var carname=new String;

var x= new Number;

var y= new Boolean;

var cars= new Array;

var person= new Object;

## 对象

JavaScript 对象是拥有属性和方法的数据。

"JavaScript 对象是变量的容器"。

但是，我们通常认为 "JavaScript 对象是键值对的容器"。

键值对通常写法为 **name : value** (键与值以冒号分割)。

键值对在 JavaScript 对象通常称为 **对象属性**。

### 访问对象属性

你可以通过两种方式访问对象属性:

var person = { lastName: 'Python', firstName: 'Terry' };

document.getElementById('demo1').innerHTML = person.lastName;

document.getElementById('demo2').innerHTML = person["lastName"];

### 对象方法

对象的方法定义了一个函数，并作为对象的属性存储。

对象方法通过添加 () 调用 (作为一个函数)。

var person = {

firstName: "John",

lastName: "Doe",

id: 5566,

fullName: function () {

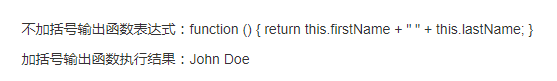
return this.firstName + " " + this.lastName;

}

};

document.getElementById("demo1").innerHTML = "不加括号输出函数表达式：" + person.fullName;

document.getElementById("demo2").innerHTML = "加括号输出函数执行结果：" + person.fullName();



## 函数

### 函数语法

函数就是包裹在花括号中的代码块，前面使用了关键词 function：

function functionname() {

// 执行代码

}

JavaScript 对大小写敏感。关键词 function 必须是小写的，并且必须以与函数名称相同的大小写来调用函数。

### 调用带参数的函数

在调用函数时，您可以向其传递值，这些值被称为参数。这些参数可以在函数中使用。您可以发送任意多的参数，由逗号 (,) 分隔：

function myFunction(var1, var2) {

//代码

}

变量和参数必须以一致的顺序出现。第一个变量就是第一个被传递的参数的给定的值，以此类推：

<p>点击这个按钮，来调用带参数的函数。</p>

<button onclick="myFunction('Harry Potter', 'Wizard')">点击这里</button>

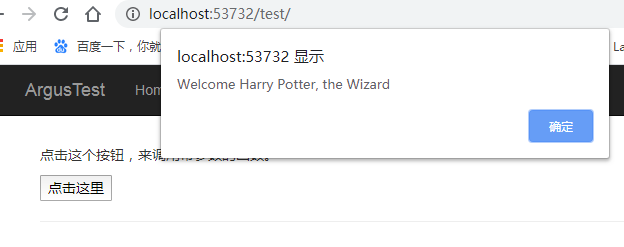
<script>

function myFunction(name, job) {

alert("Welcome " + name + ", the " + job);

}

</script>



### 带有返回值的函数

通过使用 return 语句就可以实现。

在使用 return 语句时，函数会停止执行，并返回指定的值。

实例：

<p>本例调用的函数会执行一个计算，然后返回结果：</p>

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

<script>

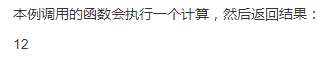
function myFunction(a, b) {

return a \* b;

}

document.getElementById("demo").innerHTML = myFunction(4, 3); //12

</script>



在您仅仅希望退出函数时 ，也可使用 return 语句。返回值是可选的：

function myFunction(a, b) {

if (a > b) {

return;

}

x = a + b

}

如果 a 大于 b，则上面的代码将退出函数，并不会计算 a 和 b 的总和。

### 局部 JavaScript 变量

在 JavaScript 函数内部声明的变量（使用 var）是局部变量，所以只能在函数内部访问它。（该变量的作用域是局部的）。

您可以在不同的函数中使用名称相同的局部变量，因为只有声明过该变量的函数才能识别出该变量。

只要函数运行完毕，本地变量就会被删除。

### 全局 JavaScript 变量

在函数外声明的变量是全局变量，网页上的所有脚本和函数都能访问它。

### JavaScript 变量的生存期

JavaScript 变量的生命期从它们被声明的时间开始。

局部变量会在函数运行以后被删除。

全局变量会在页面关闭后被删除。

### 向未声明的 JavaScript 变量分配值

如果您把值赋给尚未声明的变量，该变量将被自动作为 window 的一个属性。

carname = "Volvo"; //将声明 window 的一个属性 carname。

非严格模式下给未声明变量赋值创建的全局变量，是全局对象的可配置属性，可以删除。

var var1 = 1; // 不可配置全局属性

var2 = 2; // 没有使用 var 声明，可配置全局属性

console.log(this.var1); // 1

console.log(window.var1); // 1

delete var1; // false 无法删除

console.log(var1); //1

delete var2;

console.log(delete var2); // true

console.log(var2); // 已经删除 报错变量未定义

## 作用域

作用域是可访问变量的集合。

局部变量：在函数中通过var声明的变量。

全局变量：在函数外通过var声明的变量。

没有声明就使用的变量，默认为全局变量，不论这个变量在哪被使用。

### JavaScript 作用域

在 JavaScript 中, 对象和函数同样也是变量。

**在 JavaScript 中, 作用域为可访问变量，对象，函数的集合。**

JavaScript 函数作用域: 作用域在函数内修改。

### JavaScript 局部作用域

变量在函数内声明，变量为局部作用域。

局部变量：只能在函数内部访问。

// 此处不能调用 carName 变量

function myFunction() {

var carName = "Volvo";

// 函数内可调用 carName 变量

}

因为局部变量只作用于函数内，所以不同的函数可以使用相同名称的变量。

局部变量在函数开始执行时创建，函数执行完后局部变量会自动销毁。

### JavaScript 全局变量

变量在函数外定义，即为全局变量。

全局变量有 **全局作用域**: 网页中所有脚本和函数均可使用。

var carName = " Volvo";

// 此处可调用 carName 变量

function myFunction() {

// 函数内可调用 carName 变量

}

如果变量在函数内没有声明（没有使用 var 关键字），该变量为全局变量。以下实例中 carName 在函数内，但是为全局变量。

<p>

如果你的变量没有声明，它将自动成为全局变量：

</p>

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

<script>

myFunction();

document.getElementById("demo").innerHTML =

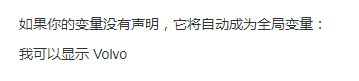
"我可以显示 " + carName;

function myFunction() {

carName = "Volvo";

}

</script>



### 函数参数

函数参数只在函数内起作用，是局部变量。

### HTML 中的全局变量

在 HTML 中, 全局变量是 window 对象: 所有数据变量都属于 window 对象。

<p>

在 HTML 中, 所有全局变量都会成为 window 变量。

</p>

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

<script>

myFunction();

document.getElementById("demo").innerHTML =

"我可以显示 " + window.carName;

function myFunction() {

carName = "Volvo";

}

</script>

注：全局变量，或者函数，可以覆盖 window 对象的变量或者函数。局部变量，包括 window 对象可以覆盖全局变量和函数。

## 事件

HTML 事件是发生在 HTML 元素上的事情。当在 HTML 页面中使用 JavaScript 时， JavaScript 可以触发这些事件。

### HTML 事件

HTML 事件可以是浏览器行为，也可以是用户行为。

以下是 HTML 事件的实例：

* HTML 页面完成加载
* HTML input 字段改变时
* HTML 按钮被点击

通常，当事件发生时，你可以做些事情。

在事件触发时 JavaScript 可以执行一些代码。

HTML 元素中可以添加事件属性，使用 JavaScript 代码来添加 HTML 元素。

单引号:

<some-HTML-element some-event='JavaScript 代码'>

双引号:

<some-HTML-element some-event="JavaScript 代码">

实例：

<button onclick="getElementById('demo').innerHTML=Date()">现在的时间是?</button>

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

### 常见的HTML事件

下面是一些常见的HTML事件的列表:

|  |  |
| --- | --- |
| **事件** | **描述** |
| onchange | HTML 元素改变 |
| onclick | 用户点击 HTML 元素 |
| onmouseover | 用户在一个HTML元素上移动鼠标 |
| onmouseout | 用户从一个HTML元素上移开鼠标 |
| onkeydown | 用户按下键盘按键 |
| onload | 浏览器已完成页面的加载 |

### JavaScript 可以做什么?

事件可以用于处理表单验证，用户输入，用户行为及浏览器动作:

* 页面加载时触发事件
* 页面关闭时触发事件
* 用户点击按钮执行动作
* 验证用户输入内容的合法性
* 等等 ...

可以使用多种方法来执行 JavaScript 事件代码：

* HTML 事件属性可以直接执行 JavaScript 代码
* HTML 事件属性可以调用 JavaScript 函数
* 你可以为 HTML 元素指定自己的事件处理程序
* 你可以阻止事件的发生。
* 等等 ...

### 字符串

JavaScript 字符串用于存储和处理文本。字符串可以是插入到引号中的任何字符。你可以使用单引号或双引号：

var carname = "Volvo XC60";

var carname = 'Volvo XC60';

可以使用索引位置来访问字符串中的每个字符，字符串的索引从 0 开始，这意味着第一个字符索引值为 [0],第二个为 [1], 以此类推。

<p id="test"></p>

<script>

var carname = "Volvo XC60";

var carname = 'Volvo XC60';

var character = carname[7];

document.getElementById('test').innerText = character; //C

</script>

注：当索引位置超出字符串上标时，将显示为undefined。

可以在字符串添加转义字符来使用引号：

var x = 'It\'s alright'; //It's alright

var y = "He is called \"Johnny\""; //He is called "Johnny"

### 字符串长度

可以使用内置属性 **length** 来计算字符串的长度：

<script>

var txt = "Hello World!";

document.write("<p>" + txt.length + "</p>");

var txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

document.write("<p>" + txt.length + "</p>");

</script>

### 特殊字符

转义字符将特殊字符转换为字符串字符，转义字符 (\) 可以用于转义撇号，换行，引号，等其他特殊字符。

下表中列举了在字符串中可以使用转义字符转义的特殊字符：

|  |  |
| --- | --- |
| **代码** | **输出** |
| \' | 单引号 |
| \" | 双引号 |
| \\ | 反斜杠 |
| \n | 换行 |
| \r | 回车 |
| \t | tab(制表符) |
| \b | 退格符 |
| \f | 换页符 |

### 字符串可以是对象

通常， JavaScript 字符串是原始值，可以使用字符创建： **var firstName = "John"**

但我们也可以使用 new 关键字将字符串定义为一个对象： **var firstName = new String("John")**

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

<script>

var x = "John"; // x是一个字符串

var y = new String("John"); // y是一个对象

document.getElementById("demo").innerHTML = typeof x + " " + typeof y; //string object

</script>

注：不要创建 String 对象。它会拖慢执行速度，并可能产生其他副作用。

var x = "John"; // x 是字符串

var y = new String("John"); // y 是一个对象

document.getElementById("demo").innerHTML = x === y; //false

//=== 为绝对相等，即数据类型与值都必须相等。

### 字符串属性和方法

### 字符串属性

|  |  |
| --- | --- |
| **属性** | **描述** |
| constructor | 返回创建字符串属性的函数 |
| length | 返回字符串的长度 |
| prototype | 允许您向对象添加属性和方法 |

### 字符串方法

|  |  |
| --- | --- |
| **方法** | **描述** |
| charAt() | 返回指定索引位置的字符 |
| charCodeAt() | 返回指定索引位置字符的 Unicode 值 |
| concat() | 连接两个或多个字符串，返回连接后的字符串 |
| fromCharCode() | 将 Unicode 转换为字符串 |
| indexOf() | 返回字符串中检索指定字符第一次出现的位置 |
| lastIndexOf() | 返回字符串中检索指定字符最后一次出现的位置 |
| localeCompare() | 用本地特定的顺序来比较两个字符串 |
| match() | 找到一个或多个正则表达式的匹配 |
| replace() | 替换与正则表达式匹配的子串 |
| search() | 检索与正则表达式相匹配的值 |
| slice() | 提取字符串的片断，并在新的字符串中返回被提取的部分 |
| split() | 把字符串分割为子字符串数组 |
| substr() | 从起始索引号提取字符串中指定数目的字符 |
| substring() | 提取字符串中两个指定的索引号之间的字符 |
| toLocaleLowerCase() | 根据主机的语言环境把字符串转换为小写，只有几种语言（如土耳其语）具有地方特有的大小写映射 |
| toLocaleUpperCase() | 根据主机的语言环境把字符串转换为大写，只有几种语言（如土耳其语）具有地方特有的大小写映射 |
| toLowerCase() | 把字符串转换为小写 |
| toString() | 返回字符串对象值 |
| toUpperCase() | 把字符串转换为大写 |
| trim() | 移除字符串首尾空白 |
| valueOf() | 返回某个字符串对象的原始值 |

var x = "JohnJohn"; // x 是字符串

y = x.charAt(2); // h

y = x.charCodeAt(2); // 104

y = x.concat(y, y); // JohnJohn104104, x+y+y

y = x.indexOf('h'); // 2, 索引从0开始

y = x.lastIndexOf('h'); // 6

y = x.slice();

y = x.split('o'); //J,hnJ,hn

y = x.substr(2); // hnJohn

y = x.substring(2, 4) // hn，[2,3]

y = x.toLocaleLowerCase(); // johnjohn,小写

y = x.toLocaleUpperCase(); // JOHNJOHN,大写

y = x.toString(); // 转成Stirng

y = x.toUpperCase(); // JOHNJOHN,大写

y = x.trim(); // JohnJohn,去除两端的空格

y = x.valueOf(); // 返回某个字符串对象的原始值

## 运算符

运算符 = 用于给 JavaScript 变量赋值。

### 算术运算符

y=5，下面的表格解释了这些算术运算符：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **运算符** | **描述** | **例子** | **x 运算结果** | **y 运算结果** |
| + | 加法 | x=y+2 | 7 | 5 |
| - | 减法 | x=y-2 | 3 | 5 |
| \* | 乘法 | x=y\*2 | 10 | 5 |
| / | 除法 | x=y/2 | 2.5 | 5 |
| % | 取模（余数） | x=y%2 | 1 | 5 |
| ++ | 自增 | x=++y | 6 | 6 |
| x=y++ | 5 | 6 |
| -- | 自减 | x=--y | 4 | 4 |
| x=y-- | 5 | 4 |

实例：取模运算的结果符号只与左边值的符号有关：

var x = 7 % 3; // 结果为 1

var y = 7 % (-3); // 结果为 1

var z = (-7) % 3; // 结果为 -1

* 如果 % 左边的操作数是正数，则模除的结果为正数或零；
* 如果 % 左边的操作数是负数，则模除的结果为负数或零。

数字与 null(空值) 相加，null 转化为数字 0：

var car = null + 3 + 4; // 结果为7

字符串与 null(空值) 相加，null 转化为字符串：

var car = null + "a"; // 结果为 nulla

### 赋值运算符

赋值运算符用于给 JavaScript 变量赋值。

给定 **x=10**和**y=5**，下面的表格解释了赋值运算符：

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **例子** | **等同于** | **运算结果** |
| = | x=y |  | x=5 |
| += | x+=y | x=x+y | x=15 |
| -= | x-=y | x=x-y | x=5 |
| \*= | x\*=y | x=x\*y | x=50 |
| /= | x/=y | x=x/y | x=2 |
| %= | x%=y | x=x%y | x=0 |

### 用于字符串的 + 运算符

+ 运算符用于把文本值或字符串变量加起来（连接起来）。如需把两个或多个字符串变量连接起来，请使用 + 运算符。

txt1 = "What a very";

txt2 = "nice day";

txt3 = txt1 + txt2; //What a very nice day

## 对字符串和数字进行加法运算

两个数字相加，返回数字相加的和，如果数字与字符串相加，返回字符串

x = 5 + 5; //10

y = "5" + 5; //55

z = "Hello" + 5; //Hello5

var result1 = 5 + 5 + "abc"; //结果将是"10abc"

var result2 = "" + 5 + 5 + "abc"; //结果将是"55abc"

### 其他：

1、字符串一个很能强大的数据类型；在执行加 **+** 时,将被加的对象统一处理为字符串。

2、bool 类型在与数字类型进行相加时，视为 0 或者 1 处理。

3、null 类型与数字类型进行累加时，视为 0 处理。

4、bool 类型与 null 类型进行累加时，视为其与整数类型累加处理。

5、undefined 除了与字符串进行累加时有效（undefined 视为字符串"undefined"处理），其他情况皆返回 NaN。

6、求模 **%** 运算，运算结果只取决于第一个数字的正负。

7、查看 String 数据类型支持的方法和属性：

打开网页后 F12 进行开发调试环境，找到 console 标签页面，执行：



## 比较和逻辑运算符

比较和逻辑运算符用于测试 true 或者 false。

### 比较运算符

比较运算符在逻辑语句中使用，以测定变量或值是否相等。

|  |
| --- |
|  |
| **运算符** | **描述** | **比较** | **返回值** |
| == | 等于 | x==8 | *false* |
| x==5 | *true* |
| === | 绝对等于（值和类型均相等） | x==="5" | *false* |
| x===5 | *true* |
| != | 不等于 | x!=8 | *true* |
| !== | 不绝对等于（值和类型有一个不相等，或两个都不相等） | x!=="5" | *true* |
| x!==5 | *false* |
| > | 大于 | x>8 | *false* |
| < | 小于 | x<8 | *true* |
| >= | 大于或等于 | x>=8 | *false* |
| <= | 小于或等于 | x<=8 | *true* |

### 逻辑运算符

逻辑运算符用于测定变量或值之间的逻辑。

给定 x=6 以及 y=3，下表解释了逻辑运算符：

|  |  |  |
| --- | --- | --- |
| **运算符** | **描述** | **例子** |
| && | and | (x < 10 && y > 1) 为 true |
| || | or | (x==5 || y==5) 为 false |
| ! | not | !(x==y) 为 true |

### 条件运算符

JavaScript 还包含了基于某些条件对变量进行赋值的条件运算符。

### 条件运算符

JavaScript 还包含了基于某些条件对变量进行赋值的条件运算符。

语法：

variablename=(condition)?value1:value2

实例：

voteable = (age < 18) ? "年龄太小" : "年龄已达到";

## 条件语句

通常在写代码时，您总是需要为不同的决定来执行不同的动作。您可以在代码中使用条件语句来完成该任务。

在 JavaScript 中，我们可使用以下条件语句：

* **if 语句** - 只有当指定条件为 true 时，使用该语句来执行代码
* **if...else 语句** - 当条件为 true 时执行代码，当条件为 false 时执行其他代码
* **if...else if....else 语句**- 使用该语句来选择多个代码块之一来执行
* **switch 语句** - 使用该语句来选择多个代码块之一来执行

### if 语句

只有当指定条件为 true 时，该语句才会执行代码。

语法：

if (condition) {

当条件为 true 时执行的代码

}

注：使用小写的**if**。使用大写字母（IF）会生成 JavaScript 错误！

### if...else 语句

请使用 if....else 语句在条件为 true 时执行代码，在条件为 false 时执行其他代码。

语法：

if (condition) {

//当条件为 true 时执行的代码

}

else {

//当条件不为 true 时执行的代码

}

### if...else if...else 语句

使用 if....else if...else 语句来选择多个代码块之一来执行。

if (condition1) {

//当条件 1 为 true 时执行的代码

}

else if (condition2) {

//当条件 2 为 true 时执行的代码

}

else {

//当条件 1 和 条件 2 都不为 true 时执行的代码

}

### switch 语句

switch 语句用于基于不同的条件来执行不同的动作。使用 switch 语句来选择要执行的多个代码块之一。

语法：

switch (n) {

case 1:

//执行代码块 1

break;

case 2:

//执行代码块 2

break;

default:

//与 case 1 和 case 2 不同时执行的代码

}

实例：

<p>点击下面的按钮来显示今天是周几：</p>

<button onclick="myFunction()">点击这里</button>

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

<script>

function myFunction(){

var x;

var d=new Date().getDay();

switch (d){

case 0:x="今天是星期日";

break;

case 1:x="今天是星期一";

break;

case 2:x="今天是星期二";

break;

case 3:x="今天是星期三";

break;

case 4:x="今天是星期四";

break;

case 5:x="今天是星期五";

break;

case 6:x="今天是星期六";

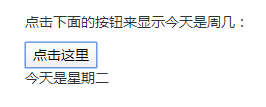
break;

}

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

}

</script>



default 关键词：使用 default 关键词来规定匹配不存在时做的事情：

## 循环

循环可以将代码块执行指定的次数。

实例：for循环

<script>

cars = ["BMW", "Volvo", "Saab", "Ford"];

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

document.write(cars[i] + "<br>");

}

</script>

### 不同类型的循环

JavaScript 支持不同类型的循环：

* **for** - 循环代码块一定的次数
* **for/in** - 循环遍历对象的属性
* **while** - 当指定的条件为 true 时循环指定的代码块
* **do/while** - 同样当指定的条件为 true 时循环指定的代码块

### For 循环

语法：

for (语句 1; 语句 2; 语句 3)

{

//被执行的代码块

}

**语句 1** （代码块）开始前执行

**语句 2** 定义运行循环（代码块）的条件

**语句 3** 在循环（代码块）已被执行之后执行

实例：

<button onclick="btn\_clk()">点击</button>

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

<script>

function btn\_clk() {

var x = "";

for (i = 0; i < 5; i++) {

//在循环开始之前设置变量 (var i=0);

//定义循环运行的条件（i 必须小于 5）;

//在每次代码块已被执行后增加一个值 (i++)

x = x + "The number is " + i + "<br>";

}

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

}

</script>

语句 1：通常我们会使用语句 1 初始化循环中所用的变量 (var i=0)。

语句 1 是可选的，也就是说不使用语句 1 也可以：

<script>

cars = ["BMW", "Volvo", "Saab", "Ford"];

var i = 2, len = cars.length;

for (; i < len; i++) {

document.write(cars[i] + "<br>");

}

</script>

语句2：用于评估初始变量的条件。

如果语句 2 返回 true，则循环再次开始，如果返回 false，则循环将结束。

语句 2 同样是可选的，如果您省略了语句 2，那么必须在循环内提供 break。否则循环就无法停下来。这样有可能令浏览器崩溃。

语句3：通常语句 3 会增加初始变量的值。语句 3 有多种用法。增量可以是负数 (i--)，或者更大 (i=i+15)。

语句 3 也是可选的，语句 3 也可以省略（比如当循环内部有相应的代码时）：

实例：

<script>

cars = ["BMW", "Volvo", "Saab", "Ford"];

var i = 0, len = cars.length; //初始设置

for (; i < len;) {

document.write(cars[i] + "<br>");

i++; //步增

}

</script>

### For/In 循环

for/in 语句循环遍历对象的属性：

实例：

<p>点击下面的按钮，循环遍历对象 "person" 的属性。</p>

<button onclick="myFunction()">点击这里</button>

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

<script>

function myFunction() {

var x;

var txt = "";

var person = { fname: "Bill", lname: "Gates", age: 56 };

for (x in person) {

txt = txt + person[x];

}

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

}

</script>

### while 循环

while 循环会在指定条件为真时循环执行代码块。

语法：

while (条件) {

//需要执行的代码

}

实例：while循环

<p>点击下面的按钮，只要 i 小于 5 就一直循环代码块。</p>

<button onclick="myFunction()">点击这里</button>

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

<script>

function myFunction(){

var x="",i=0;

while (i<5){

x=x + "该数字为 " + i + "<br>";

i++; //如果没有该代码，将陷入无限循环，造成页面崩溃

}

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

}

</script>

### do/while 循环

do/while 循环是 while 循环的变体。该循环会在检查条件是否为真之前执行一次代码块，然后如果条件为真的话，就会重复这个循环。

do {

//需要执行的代码

}

while (//条件);

实例：do...while

<p>点击下面的按钮，只要 i 小于 5 就一直循环代码块。</p>

<button onclick="myFunction()">点击这里</button>

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

<script>

function myFunction() {

var x = "", i = 0;

do {

x = x + "该数字为 " + i + "<br>";

i++;

}

while (i < 5)

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

}

</script>

### 总结：

4 种循环的适用场景吧:

* **for** : 比较适合遍历数组,字符串等等。
* **for in** : 比较适合遍历对象，遍历对象时使用这个再合适不过了。
* **while** : while 的话，与 for 的使用场景差不多。唯一不同的是，遍历的条件必须满足。
* **do while** : 至少执行一边的循环，遍历数组和字符串也是很方便。

## break 和 continue 语句

break 语句用于跳出循环。

continue 语句跳出循环后，会继续执行该循环之后的代码（如果有的话）

### break 语句

for (i = 0; i < 10; i++) {

if (i == 3) break; //自增到3时，跳出循环

x = x + "The number is " + i + "<br>";

}

### continue 语句

**continue 语句**中断循环中的迭代，如果出现了指定的条件，然后继续循环中的下一个迭代。 该例子跳过了值 3：

for (i = 0; i <= 10; i++) {

if (i == 3) continue; //跳过等于3的代码，继续循环下一个迭代

x = x + "The number is " + i + "<br>";

}

### 标签

标记 JavaScript 语句，请在语句之前加上冒号

label:

statements

break 和 continue 语句仅仅是能够跳出代码块的语句。

break labelname;

continue labelname;

continue 语句（带有或不带标签引用）只能用在循环中。

break 语句（不带标签引用），只能用在循环或 switch 中。

通过标签引用，break 语句可用于跳出任何 JavaScript 代码块：

cars = ["BMW", "Volvo", "Saab", "Ford"];

list: {

document.write(cars[0] + "<br>");

document.write(cars[1] + "<br>");

document.write(cars[2] + "<br>");

break list;

document.write(cars[3] + "<br>");

document.write(cars[4] + "<br>");

document.write(cars[5] + "<br>");

}

## typeof, null, 和 undefined

### typeof 操作符

可以使用 typeof 操作符来检测变量的数据类型。

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

<script>

document.getElementById("demo").innerHTML =

typeof "john" + "<br>" +

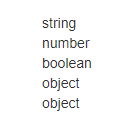
typeof 3.14 + "<br>" +

typeof false + "<br>" +

typeof [1, 2, 3, 4] + "<br>" +

typeof { name: 'john', age: 34 };

</script>



### null

在 JavaScript 中 null 表示 "什么都没有"。null是一个只有一个值的特殊类型。表示一个空对象引用。

注：用 typeof 检测 null 返回是object。

可以设置为 null 来清空对象:

var person = null; // 值为 null(空), 但类型为对象

可以设置为 undefined 来清空对象:

var person = undefined; // 值为 undefined, 类型为 undefined

当使用完一个比较大的对象时，需要对其进行释放内存时，设置为 null。

### undefined

**undefined** 是一个没有设置值的变量。**typeof** 一个没有值的变量会返回 **undefined**。

### undefined 和 null 的区别

document.getElementById("demo").innerHTML =

typeof undefined + "<br>" + //undefined

typeof null + "<br>" + //object

(null === undefined) + "<br>" + //false

(null == undefined); //true

## 类型转换

Number() 转换为数字， String() 转换为字符串， Boolean() 转化为布尔值。

### 数据类型

在 JavaScript 中有 6 种不同的数据类型：

* string
* number
* boolean
* object
* function
* symbol

3 种对象类型：

* Object
* Date
* Array

2 个不包含任何值的数据类型：

* null
* undefined

### typeof 操作符

可以使用 **typeof** 操作符来查看 JavaScript 变量的数据类型。

<p> typeof 操作符返回变量、对象、函数、表达式的类型。</p>

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

<script>

document.getElementById("demo").innerHTML =

typeof "John" + "<br>" + // 返回 string

typeof 3.14 + "<br>" + // 返回 number

typeof NaN + "<br>" + // 返回 number

typeof false + "<br>" + // 返回 boolean

typeof [1, 2, 3, 4] + "<br>" + // 返回 object

typeof { name: 'John', age: 34 } + "<br>" + // 返回 object

typeof new Date() + "<br>" + // 返回 object

typeof function () { } + "<br>" + // 返回 function

typeof myCar + "<br>" + // 返回 undefined (如果 myCar 没有声明)

typeof null; // 返回 object

</script>

注意：

* NaN 的数据类型是 number
* 数组(Array)的数据类型是 object
* 日期(Date)的数据类型为 object
* null 的数据类型是 object
* 未定义变量的数据类型为 undefined

注：如果对象是 JavaScript Array 或 JavaScript Date ，我们就无法通过 typeof 来判断他们的类型，因为都是 返回 object。

### constructor 属性

**constructor** 属性返回所有 JavaScript 变量的构造函数。

<p> constructor 属性返回变量或对象的构造函数。</p>

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

<script>

document.getElementById("demo").innerHTML =

"john".constructor + "<br>" + // 返回函数 String() { [native code] }

(3.14).constructor + "<br>" + // 返回函数 Number() { [native code] }

false.constructor + "<br>" + // 返回函数 Boolean() { [native code] }

[1, 2, 3, 4].constructor + "<br>" + // 返回函数 Array() { [native code] }

{ name: 'john', age: 34 }.constructor + "<br>" + // 返回函数 Object() { [native code] }

new Date().constructor + "<br>" + // 返回函数 Date() { [native code] }

function () { }.constructor; // 返回函数 Function(){ [native code] }

</script>

可以使用 constructor 属性来查看对象是否为数组 (包含字符串 "Array"):

#### 实例：判断是否数组

<p>判断是否为数组。</p>

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

<script>

var fruits = ["Banana", "Orange", "Apple", "Mango"];

document.getElementById("demo").innerHTML = isArray(fruits);

function isArray(myArray) {

return myArray.constructor.toString().indexOf("Array") > -1; //true

}

</script>

return myArray.constructor.toString().indexOf("Array") > -1;这句话怎么解释?

1、myArray 是函数 isArray 的参数，这里调用函数的时候，会传来数组 fruits。

2、constructor 是一个属性，构造函数属性，不同类型的数据，会得到相应不同的值。因为 myArray 是个数组，这里的 myArray.constructor 的值就是 **function Array() { [native code] }**。(如果 myArray 是个字符串，myArray.constructor 的值就是**function String() { [native code] }**。还有 number，boolean，object，等等。)

3、toString() 是个方法，变字符串的方法，这里把 **function Array() { [native code] }** 变成字符串，为了后面好检索。

4、indexOf("Array") 是个方法，检索字符串，这里看字符串 **function Array() { [native code] }** 里有没有Array，有就返回首次出现的位置，是一个数值，这里是 9。如果出现在第一个字符，会返回 0。空格参与计数。如果没有找到，就返回 -1。只要 >-1，就说明有 Array，就能判断原来那个函数调用传来的 fruits 是一个数组。如果 **myArray.constructor.toString().indexOf("Object")>-1**，那么 myArray 就是一个 Object 对象。不过那样地话，这个参数的名字就没取好了，应该叫做 myObject。

可以使用 constructor 属性来查看对象是否为日期 (包含字符串 "Date"):

#### 实例：判断是否日期

<p>判断是否为日期。</p>

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

<script>

var myDate = new Date();

document.getElementById("demo").innerHTML = isDate(myDate);

function isDate(myDate) {

return myDate.constructor.toString().indexOf("Date") > -1; //true

}

</script>

### ****instanceof****

判断某个变量是否是某个对象的实例。

语法：

var result = objectName instanceof objectType

返回布尔值，如果是指定类型返回 true，否则返回 false：

实例：

<p>可通过 instanceof 操作符来判断对象的具体类型，语法格式:</p>

<pre>var result = objectName instanceof objectType</pre>

<p>返回布尔值，如果是指定类型返回 true，否则返回 false。测试如下：</p>

<script>

arr = [1, 2, 3];

if (arr instanceof Array) {

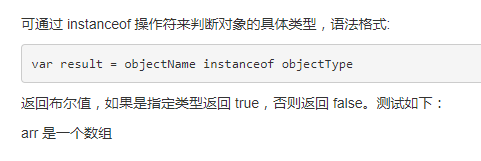
document.write("arr 是一个数组");

} else {

document.write("arr 不是一个数组");

}

</script>



### 其他：

我们可以使用 typeof 来获取一个变量是否存在，如 **if(typeof a!="undefined"){}**，而不要去使用 if(a) 因为如果 a 不存在（未声明）则会出错。

正因为 typeof 遇到 null,数组,对象时都会返回 object 类型，所以当我们要判断一个对象是否是数组时。

## 类型转换

JavaScript 变量可以转换为新变量或其他数据类型：

* 通过使用 JavaScript 函数
* 通过 JavaScript 自身自动转换

### 将数字转换为字符串

全局方法 String() 可以将数字转换为字符串。

该方法可用于任何类型的数字，字母，变量，表达式：

<script>

var x = 123;

document.getElementById("demo").innerHTML =

String(x) + "<br>" +

String(123) + "<br>" +

String(100 + 23);

</script>

Number 方法 **toString()** 也是有同样的效果。

var x = 123;

document.getElementById("demo").innerHTML =

x.toString() + "<br>" +

(123).toString() + "<br>" +

(100 + 23).toString();

在 Number 方法 章节中，你可以找到更多数字转换为字符串的方法：

|  |  |
| --- | --- |
| **方法** | **描述** |
| toExponential() | 把对象的值转换为指数计数法。 |
| toFixed() | 把数字转换为字符串，结果的小数点后有指定位数的数字。 |
| toPrecision() | 把数字格式化为指定的长度。 |

### 将布尔值转换为字符串

全局方法 **String()** 可以将布尔值转换为字符串。

String(false) // 返回 "false"

String(true) // 返回 "true"

Boolean 方法 **toString()** 也有相同的效果。

false.toString() // 返回 "false"

true.toString() // 返回 "true"

### 将日期转换为字符串

Date() 返回字符串。

var date = Date();

document.write(date); //返回当前时间

全局方法 String() 可以将日期对象转换为字符串。

var date = String(new Date());

document.write(date); //返回当前时间

Date 方法 toString() 也有相同的效果。

var date = new Date();

document.write(date.toString()); //返回当前时间

在 Date 方法 章节中，你可以查看更多关于日期转换为字符串的函数：

|  |  |
| --- | --- |
| **方法** | **描述** |
| getDate() | 从 Date 对象返回一个月中的某一天 (1 ~ 31)。 |
| getDay() | 从 Date 对象返回一周中的某一天 (0 ~ 6)。 |
| getFullYear() | 从 Date 对象以四位数字返回年份。 |
| getHours() | 返回 Date 对象的小时 (0 ~ 23)。 |
| getMilliseconds() | 返回 Date 对象的毫秒(0 ~ 999)。 |
| getMinutes() | 返回 Date 对象的分钟 (0 ~ 59)。 |
| getMonth() | 从 Date 对象返回月份 (0 ~ 11)。 |
| getSeconds() | 返回 Date 对象的秒数 (0 ~ 59)。 |
| getTime() | 返回 1970 年 1 月 1 日至今的毫秒数。 |

### 将字符串转换为数字

全局方法 **Number()** 可以将字符串转换为数字。

字符串包含数字(如 "3.14") 转换为数字 (如 3.14).

空字符串转换为 0。

其他的字符串会转换为 NaN (不是个数字)。

var date = new Date();

document.getElementById('demo').innerHTML =

Number("3.14") + '<br/>' + // 返回 3.14

Number(" ") + '<br/>' + // 返回 0

Number("") + '<br/>' + // 返回 0

Number("99 88"); // 返回 NaN

在 Number 方法 章节中，你可以查看到更多关于字符串转为数字的方法：

|  |  |
| --- | --- |
| **方法** | **描述** |
| parseFloat() | 解析一个字符串，并返回一个浮点数。 |
| parseInt() | 解析一个字符串，并返回一个整数。 |

### 一元运算符 +

**Operator +**可用于将变量转换为数字：

<p> typeof 操作符返回变量或表达式的类型。</p>

<button onclick="myFunction()">点我</button>

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

<script>

function myFunction() {

var y = "5";

var x = + y;

document.getElementById("demo").innerHTML =

typeof y + "<br>" + typeof x;

}

</script>

如果变量不能转换，它仍然会是一个数字，但值为 NaN (不是一个数字):

<p> typeof 操作符返回变量或表达式的类型。</p>

<button onclick="myFunction()">点我</button>

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

<script>

function myFunction() {

var y = "John";

var x = + y;

document.getElementById("demo").innerHTML =

typeof x + "<br>" + x; //number NaN

}

</script>

### 将布尔值转换为数字

全局方法 **Number()** 可将布尔值转换为数字。

Number(false) // 返回 0

Number(true) // 返回 1

### 将日期转换为数字

全局方法 **Number()** 可将日期转换为数字。

d = new Date();

Number(d) // 返回 1404568027739

日期方法 **getTime()** 也有相同的效果。

d = new Date();

d.getTime() // 返回 1404568027739

### 自动转换类型

当 JavaScript 尝试操作一个 "错误" 的数据类型时，会自动转换为 "正确" 的数据类型。

5 + null // 返回 5 null 转换为 0

"5" + null // 返回"5null" null 转换为 "null"

"5" + 1 // 返回 "51" 1 转换为 "1"

"5" - 1 // 返回 4 "5" 转换为 5

### 自动转换为字符串

当你尝试输出一个对象或一个变量时 JavaScript 会自动调用变量的 toString() 方法：

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

myVar = { name: "Fjohn" } // toString 转换为 "[object Object]"

myVar = [1, 2, 3, 4] // toString 转换为 "1,2,3,4"

myVar = new Date() // toString 转换为 "Fri Jul 18 2014 09:08:55 GMT+0200"

数字和布尔值也经常相互转换：

myVar = 123 // toString 转换为 "123"

myVar = true // toString 转换为 "true"

myVar = false // toString 转换为 "false"

使用不同的数值转换为数字(Number), 字符串(String), 布尔值(Boolean):

|  |  |  |  |
| --- | --- | --- | --- |
| **原始值** | **转换为数字** | **转换为字符串** | **转换为布尔值** |
| false | 0 | "false" | false |
| true | 1 | "true" | true |
| 0 | 0 | "0" | false |
| 1 | 1 | "1" | true |
| "0" | 0 | "0" | **true** |
| "000" | 0 | "000" | **true** |
| "1" | 1 | "1" | true |
| NaN | NaN | "NaN" | false |
| Infinity | Infinity | "Infinity" | true |
| -Infinity | -Infinity | "-Infinity" | true |
| "" | **0** | "" | **false** |
| "20" | 20 | "20" | true |
| "Runoob" | NaN | "Runoob" | true |
| [ ] | **0** | "" | true |
| [20] | **20** | "20" | true |
| [10,20] | NaN | "10,20" | true |
| ["Runoob"] | NaN | "Runoob" | true |
| ["Runoob","Google"] | NaN | "Runoob,Google" | true |
| function(){} | NaN | "function(){}" | true |
| { } | NaN | "[object Object]" | true |
| null | **0** | "null" | false |
| undefined | NaN | "undefined" | false |

## 正则表达式

正则表达式（英语：Regular Expression，在代码中常简写为regex、regexp或RE）使用单个字符串来描述、匹配一系列符合某个句法规则的字符串搜索模式。

搜索模式可用于文本搜索和文本替换。

语法：

/正则表达式主体/修饰符(可选)

实例解析：正则表达式

var patt = /runoob/i

// /runoob/i 是一个正则表达式。

// runoob 是一个正则表达式主体(用于检索)

// i 是一个修饰符 (搜索不区分大小写)

### 使用字符串方法

在 JavaScript 中，正则表达式通常用于两个字符串方法 : search() 和 replace()。

**search() 方法**用于检索字符串中指定的子字符串，或检索与正则表达式相匹配的子字符串，并返回子串的起始位置。

**replace() 方法** 用于在字符串中用一些字符替换另一些字符，或替换一个与正则表达式匹配的子串。

### search() 方法使用正则表达式

实例：使用正则表达式搜索 "Runoob" 字符串，且不区分大小写

<p>搜索字符串 "runoob", 并显示匹配的起始位置：</p>

<button onclick="myFunction()">点我</button>

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

<script>

function myFunction() {

var str = "Visit Runoob!";

var n = str.search(/Runoob/i);

document.getElementById("demo").innerHTML = n; //6

}

</script>

### search() 方法使用字符串

search 方法可使用字符串作为参数。字符串参数会转换为正则表达式：

<p>搜索字符串 "Runoob", 并显示匹配的起始位置：</p>

<button onclick="myFunction()">点我</button>

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

<script>

function myFunction() {

var str = "Visit Runoob!";

var n = str.search("Runoob");

document.getElementById("demo").innerHTML = n; //6

}

</script>

### replace() 方法使用正则表达式

使用正则表达式且不区分大小写将字符串中的 Microsoft 替换为 Runoob :

<p>替换 "microsoft" 为 "Runoob" :</p>

<button onclick="myFunction()">点我</button>

<p id="demo">Visit Microsoft!</p>

<script>

function myFunction() {

var str = document.getElementById("demo").innerHTML;

var txt = str.replace(/microsoft/i, "Runoob");

document.getElementById("demo").innerHTML = txt; //Visit Runoob!

}

</script>

### replace() 方法使用字符串

<p>替换 "Microsoft" 为 "Runoob" :</p>

<button onclick="myFunction()">点我</button>

<p id="demo">请访问 Microsoft!</p>

<script>

function myFunction() {

var str = document.getElementById("demo").innerHTML;

var txt = str.replace("Microsoft", "Runoob");

document.getElementById("demo").innerHTML = txt; //请访问 Runoob!

}

</script>

### 正则表达式修饰符

**修饰符** 可以在全局搜索中不区分大小写:

|  |  |
| --- | --- |
| **修饰符** | **描述** |
| i | 执行对大小写不敏感的匹配。 |
| g | 执行全局匹配（查找所有匹配而非在找到第一个匹配后停止）。 |
| m | 执行多行匹配。 |

### 正则表达式模式

方括号用于查找某个范围内的字符：

|  |  |
| --- | --- |
| **表达式** | **描述** |
| [abc] | 查找方括号之间的任何字符。 |
| [0-9] | 查找任何从 0 至 9 的数字。 |
| (x|y) | 查找任何以 | 分隔的选项。 |

元字符是拥有特殊含义的字符：

|  |  |
| --- | --- |
| **元字符** | **描述** |
| \d | 查找数字。 |
| \s | 查找空白字符。 |
| \b | 匹配单词边界。 |
| \uxxxx | 查找以十六进制数 xxxx 规定的 Unicode 字符。 |

量词:

|  |  |
| --- | --- |
| **量词** | **描述** |
| n+ | 匹配任何包含至少一个 *n* 的字符串。 |
| n\* | 匹配任何包含零个或多个 *n* 的字符串。 |
| n? | 匹配任何包含零个或一个 *n* 的字符串。 |

### 使用 RegExp 对象

在 JavaScript 中，RegExp 对象是一个预定义了属性和方法的正则表达式对象。

### 使用 test()

test() 方法是一个正则表达式方法。

test() 方法用于检测一个字符串是否匹配某个模式，如果字符串中含有匹配的文本，则返回 true，否则返回 false。

var patt = /e/;

patt.test("The best things in life are free!");

可以不用设置正则表达式的变量，以上两行代码可以合并为一行：

var result = /e/.test("The best things in life are free!"); //返回true

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

### 使用 exec()

exec() 方法是一个正则表达式方法。exec() 方法用于检索字符串中的正则表达式的匹配。

该函数返回一个数组，其中存放匹配的结果。如果未找到匹配，则返回值为 null。

var patt1 = new RegExp("e");

document.write(patt1.exec("The best things in life are free")); // e

### 实用实例：

/\*是否带有小数\*/

function isDecimal(strValue) {

var objRegExp = /^\d+\.\d+$/;

return objRegExp.test(strValue);

}

/\*校验是否中文名称组成 \*/

function ischina(str) {

var reg = /^[\u4E00-\u9FA5]{2,4}$/; /\*定义验证表达式\*/

return reg.test(str); /\*进行验证\*/

}

/\*校验是否全由8位数字组成 \*/

function isStudentNo(str) {

var reg = /^[0-9]{8}$/; /\*定义验证表达式\*/

return reg.test(str); /\*进行验证\*/

}

/\*校验电话码格式 \*/

function isTelCode(str) {

var reg = /^((0\d{2,3}-\d{7,8})|(1[3584]\d{9}))$/;

return reg.test(str);

}

/\*校验邮件地址是否合法 \*/

function IsEmail(str) {

var reg = /^\w+@[a-zA-Z0-9]{2,10}(?:\.[a-z]{2,4}){1,3}$/;

return reg.test(str);

}

## 错误 - throw、try 和 catch

**try** 语句测试代码块的错误。

**catch** 语句处理错误。

**throw** 语句创建自定义错误。

**finally** 语句在 try 和 catch 语句之后，无论是否有触发异常，该语句都会执行。

### JavaScript 错误

当 JavaScript 引擎执行 JavaScript 代码时，会发生各种错误。

可能是语法错误，通常是程序员造成的编码错误或错别字。

可能是拼写错误或语言中缺少的功能（可能由于浏览器差异）。

可能是由于来自服务器或用户的错误输出而导致的错误。

当然，也可能是由于许多其他不可预知的因素。

### 抛出（throw）错误

当错误发生时，当事情出问题时，JavaScript 引擎通常会停止，并生成一个错误消息。

描述这种情况的技术术语是：JavaScript 将**抛出**一个错误。

### try 和 catch

**try** 语句允许我们定义在执行时进行错误测试的代码块。

**catch** 语句允许我们定义当 try 代码块发生错误时，所执行的代码块。

JavaScript 语句 **try** 和 **catch** 是成对出现的。

语法：

try {

//异常的抛出

} catch (e) {

//异常的捕获与处理

} finally {

//结束处理

}

实例：

<input type="button" value="查看消息" onclick="message()" />

<script>

var txt = "";

function message() {

try {

adddlert("Welcome guest!");

}

catch (err) {

txt = "本页有一个错误。\n\n";

txt += "错误描述：" + err.message + "\n\n";

txt += "点击确定继续。\n\n";

alert(txt);

}

}

</script>

### finally 语句

finally 语句不论之前的 try 和 catch 中是否产生异常都会执行该代码块。

<p>不管输入是否正确，输入框都会再输入后清空。</p>

<p>请输入 5 ~ 10 之间的数字：</p>

<input id="demo" type="text">

<button type="button" onclick="myFunction()">点我</button>

<p id="p01"></p>

<script>

function myFunction() {

var message, x;

message = document.getElementById("p01");

message.innerHTML = "";

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

try {

if (x == "") throw "值是空的";

if (isNaN(x)) throw "值不是一个数字";

x = Number(x);

if (x > 10) throw "太大";

if (x < 5) throw "太小";

}

catch (err) {

message.innerHTML = "错误: " + err + ".";

}

finally {

document.getElementById("demo").value = "";

}

}

</script>

### Throw 语句

throw 语句允许我们创建自定义错误。

正确的技术术语是：创建或**抛出异常**（exception）。

如果把 throw 与 try 和 catch 一起使用，那么您能够控制程序流，并生成自定义的错误消息。

语句： throw exception

异常可以是 JavaScript 字符串、数字、逻辑值或对象。

## 调试

有了调试工具，我们就可以设置断点 (代码停止执行的位置), 且可以在代码执行时检测变量。

浏览器启用调试工具一般是按下 F12 键，并在调试菜单中选择 "Console" 。

### console.log() 方法

如果浏览器支持调试，你可以使用 console.log() 方法在调试窗口上打印 JavaScript 值：

<h1>我的第一个 Web 页面</h1>

<p>

浏览器中(Chrome, IE, Firefox) 使用 F12 来启用调试模式， 在调试窗口中点击 "Console" 菜单。

</p>

<script>

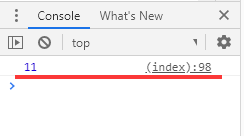
a = 5;

b = 6;

c = a + b;

console.log(c);

</script>



### 设置断点

在调试窗口中，你可以设置 JavaScript 代码的断点。

在每个断点上，都会停止执行 JavaScript 代码，以便于我们检查 JavaScript 变量的值。

在检查完毕后，可以重新执行代码（如播放按钮）。

### debugger 关键字

**debugger** 关键字用于停止执行 JavaScript，并调用调试函数。

这个关键字与在调试工具中设置断点的效果是一样的。

如果没有调试可用，debugger 语句将无法工作。

实例：开启 debugger ，代码在第三行前停止执行。

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

<p>开启调试工具，在代码执行到第三行前会停止执行。</p>

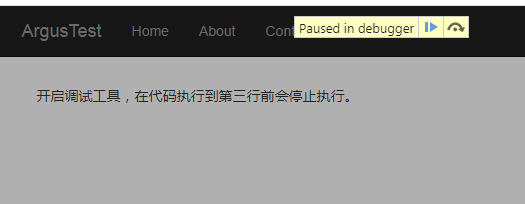
<script>

var x = 15 \* 5;

debugger;

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

</script>



## 变量提升

JavaScript 中，函数及变量的声明都将被提升到函数的最顶部。

JavaScript 中，变量可以在使用后声明，也就是变量可以先使用再声明。

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

<script>

x = 5; // 变量 x 设置为 5

elem = document.getElementById("demo"); // 查找元素

elem.innerHTML = x; // 在元素中显示 x

var x; // 声明 x

</script>

要理解以上实例就需要理解 "hoisting(变量提升)"。变量提升：函数声明和变量声明总是会被解释器悄悄地被"提升"到方法体的最顶部。

其实主要理解 js 的解析机制就行。

遇到 script 标签的话 js 就进行预解析，将变量 var 和 function 声明提升，但不会执行 function，然后就进入上下文执行，上下文执行还是执行预解析同样操作，直到没有 var 和 function，就开始执行上下文。如:

### 匿名函数

语法：

声明：

var 变量名称 = function (形参列表) {

//函数体

}

调用：

变量名称(实参列表)

实例解析：

var getName = function () {

console.log(2);

}

function getName() {

console.log(1);

}

getName();//结果为2

可能会有人觉得最后输出的结果是 1。但是 getName 是一个变量，因此这个变量的声明也将提升到顶部，而变量的赋值依然保留在原来的位置。需要注意的是，函数优先，虽然函数声明和变量声明都会被提升，但是函数会首先被提升，然后才是变量。

提升后的代码顺序：

//函数、变量声明提升后

function getName() { //函数声明提升到顶部

console.log(1);

}

var getName; //变量声明提升

getName = function () { //变量赋值依然保留在原来的位置

console.log(2);

}

getName(); // 最终输出：2

### JavaScript 初始化不会提升

JavaScript 只有声明的变量会提升，初始化的不会。

实例：对比

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

<script>

var x = 5; // 初始化 x

var y = 7; // 初始化 y

elem = document.getElementById("demo"); // 查找元素

elem.innerHTML = x + " " + y; // 显示 x 和 y

</script>



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

<script>

var x = 5; // 初始化 x

elem = document.getElementById("demo"); // 查找元素

elem.innerHTML = "x 为：" + x + "，y 为：" + y; // 显示 x 和 y

var y = 7; // 初始化 y

</script>

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

<script>

var x = 5; // 初始化 x

elem = document.getElementById("demo"); // 查找元素

elem.innerHTML = "x 为：" + x + "，y 为：" + y; // 显示 x 和 y

var y = 7; // 初始化 y

</script>



注：实例 2 的 y 输出了 undefined，这是因为变量声明 (var y) 提升了，但是初始化(y = 7) 并不会提升，所以 y 变量是一个未定义的变量。

### 在头部声明你的变量

对于大多数程序员来说并不知道 JavaScript 变量提升。

如果程序员不能很好的理解变量提升，他们写的程序就容易出现一些问题。

为了避免这些问题，通常我们在每个作用域开始前声明这些变量，这也是正常的 JavaScript 解析步骤，易于我们理解。

## 严格模式(use strict)

JavaScript 严格模式（strict mode）即在严格的条件下运行。

### 使用 "use strict" 指令

"use strict" 指令在 JavaScript 1.8.5 (ECMAScript5) 中新增。它不是一条语句，但是是一个字面量表达式。"use strict" 的目的是指定代码在严格条件下执行。严格模式下你不能使用未声明的变量。

### 严格模式声明

严格模式通过在脚本或函数的头部添加 "use strict"; 表达式来声明。

实例中我们可以在浏览器按下 **F12 (或点击"工具>更多工具>开发者工具")** 开启调试模式，查看报错信息。

实例：

<h1>使用 "use strict":</h1>

<h3>不允许使用未定义的变量。</h3>

<p>浏览器按下 F12 开启调试模式，查看报错信息。</p>

<script>

"use strict";

x = 3.14; // 报错 (x 未定义)

</script>

实例：

<p>在函数内使用 "use strict" 只在函数内报错。

</p>

<p>浏览器按下 F12 开启调试模式，查看报错信息。</p>

<script>

x = 3.14; // 不报错

myFunction();

function myFunction() {

"use strict";

y = 3.14; // 报错 (y 未定义)

}

</script>

目的：

消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为;

* 消除代码运行的一些不安全之处，保证代码运行的安全；
* 提高编译器效率，增加运行速度；
* 为未来新版本的Javascript做好铺垫。

同样的代码，在"严格模式"中，可能会有不一样的运行结果；一些在"正常模式"下可以运行的语句，在"严格模式"下将不能运行。掌握这些内容，有助于更细致深入地理解Javascript，让你变成一个更好的程序员。

### 严格模式的限制

* 不允许使用未声明的变量：
* 不允许删除变量或对象。

<h1>使用 "use strict":</h1>

<h3>不允许删除变量或对象。</h3>

<p>浏览器按下 F12 开启调试模式，查看报错信息。</p>

<script>

"use strict";

var x = 3.14;

delete x; //浏览器会报错：Uncaught SyntaxError: Delete of an unqualified identifier in strict mode.

</script>

* 不允许删除函数。

<h1>使用 "use strict":</h1>

<h3>不允许删除函数。</h3>

<p>浏览器按下 F12 开启调试模式，查看报错信息。</p>

<script>

"use strict";

function x(p1, p2) { };

delete x; //浏览器报错：Uncaught SyntaxError: Delete of an unqualified identifier in strict mode.

</script>

* 不允许变量重名:

<h1>使用 "use strict":</h1>

<h3>不允许变量重名。</h3>

<p>浏览器按下 F12 开启调试模式，查看报错信息。</p>

<script>

"use strict";

function x(p1, p1) { }; // 报错Uncaught SyntaxError: Duplicate parameter name not allowed in this context

</script>

* 不允许使用八进制:

"use strict";

var x = 010; // 报错

* 不允许使用转义字符:

"use strict";

var x = \010; // 报错

* 不允许对只读属性赋值:

"use strict";

var obj = {};

Object.defineProperty(obj, "x", { value: 0, writable: false });

obj.x = 3.14; // 报错

* 不允许对一个使用getter方法读取的属性进行赋值

"use strict";

var obj = { get x() { return 0 } };

obj.x = 3.14; // 报错

* 不允许删除一个不允许删除的属性：

"use strict";

delete Object.prototype; // 报错

* 变量名不能使用 "eval" 字符串:

"use strict";

var eval = 3.14; // 报错

* 变量名不能使用 "arguments" 字符串:

"use strict";

var arguments = 3.14; // 报错

* 不允许使用以下这种语句:

"use strict";

with (Math) { x = cos(2) }; // 报错

* 由于一些安全原因，在作用域 eval() 创建的变量不能被调用：

"use strict";

eval("var x = 2");

alert(x); // 报错

* 禁止this关键字指向全局对象。

function f() {

return !this;

}

// 返回false，因为"this"指向全局对象，"!this"就是false

function f() {

"use strict";

return !this;

}

// 返回true，因为严格模式下，this的值为undefined，所以"!this"为true。

### 保留关键字

严格模式新增了一些保留关键字：

* implements
* interface
* let
* package
* private
* protected
* public
* static
* yield
* "use strict" 指令只允许出现在脚本或函数的开头。

## 使用误区

### 赋值运算符应用错误

在 JavaScript 程序中如果你在 if 条件语句中使用赋值运算符的等号 (=) 将会产生一个错误结果, 正确的方法是使用比较运算符的两个等号 (==)。

### 比较运算符常见错误

在常规的比较中，数据类型是被忽略的，以下 if 条件语句返回 true：

var x = 10;

var y = "10";

document.getElementById("demo").innerHTML = Boolean(x == y); //true

在严格的比较运算中，=== 为恒等计算符，同时检查表达式的值与类型，以下 if 条件语句返回 false：

var x = 10;

var y = "10";

document.getElementById("demo").innerHTML = Boolean(x === y); //false

这种错误经常会在 switch 语句中出现，switch 语句会使用恒等计算符(===)进行比较:

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

<script>

var x = 10;

switch (x) {

case "10": alert("Hello"); //由于数据类型不匹配，将不会弹出

}

</script>

### 加法与连接注意事项

**加法**是两个**数字**相加。**连接**是两个**字符串**连接。

var x = 10;

var y = 5;

var z = x + y; // z 的结果为 15

var x = 10;

var y = "5";

var z = x + y; // z 的结果为 "105"

### 浮点型数据使用注意事项

JavaScript 中的所有数据都是以 64 位**浮点型数据(float)** 来存储。所有的编程语言，包括 JavaScript，对浮点型数据的精确度都很难确定：

var x = 0.1;

var y = 0.2;

var z = x + y;

var result = (z == 0.3)

document.getElementById("demo").innerHTML = result; //false

为解决以上问题，可以用整数的乘除法来解决：

var x = 0.1;

var y = 0.2;

var z = (x \* 10 + y \* 10) / 10; // z 的结果为 0.3

var result = (z == 0.3)

document.getElementById("demo").innerHTML = result; //true

### 字符串分行

JavaScript 允许我们在字符串中使用断行语句:

document.getElementById("demo").innerHTML =

"Hello World!";

在字符串中直接使用回车换行是会报错的：

var x = "Hello

World!";

字符串断行需要使用反斜杠(\)：

var x = "Hello \

World!";

### 错误的使用分号

if (x == 19);

{

// code block

}

### return 语句使用注意事项

JavaScript 默认是在代码的最后一行自动结束。

function myFunction(a) {

var power = 10

return a \* power

}

function myFunction(a) {

var power = 10;

return a \* power;

}

以上两个代码都可以运行，不会报错。

### 数组中使用名字来索引

许多程序语言都允许使用名字来作为数组的索引。使用名字来作为索引的数组称为关联数组(或哈希)。JavaScript 不支持使用名字来索引数组，只允许使用数字索引。

在 JavaScript 中, **对象** 使用 **名字作为索引**。

如果你使用名字作为索引，当访问数组时，JavaScript 会把数组重新定义为标准对象。

执行这样操作后，数组的方法及属性将不能再使用，否则会产生错误:

var person = [];

person["firstName"] = "John";

person["lastName"] = "Doe";

person["age"] = 46;

var x = person.length; // person.length 返回 0

var y = person[0]; // person[0] 返回 undefined

### 定义数组元素，最后不能添加逗号

正确的定义方式：

var colors = [5, 6, 7,]; //这样数组的长度可能为3 也可能为4。

### 定义对象，最后不能添加逗号

错误的定义方式： var websites = {site:"菜鸟教程", url:"www.runoob.com", like:460,}

### Undefined 不是 Null

**null** 用于对象, **undefined** 用于变量，属性和方法。

对象只有被定义才有可能为 null，否则为 undefined。如果我们想测试对象是否存在，在对象还没定义时将会抛出一个错误。

正确的方式是我们需要先使用 typeof 来检测对象是否已定义：

if (typeof myObj !== "undefined" && myObj !== null) {

}

### 程序块作用域

在每个代码块中 JavaScript 不会创建一个新的作用域，一般各个代码块的作用域都是全局的。

## 表单

### 表单验证

HTML 表单验证可以通过 JavaScript 来完成。

实例：

<form name="myForm" action="#" onsubmit="return validateForm()" method="post">

{% csrf\_token %}

名字: <input type="text" name="fname">

<input type="submit" value="提交">

</form>

<script>

function validateForm() {

var x = document.forms["myForm"]["fname"].value;

if (x == null || x == "") {

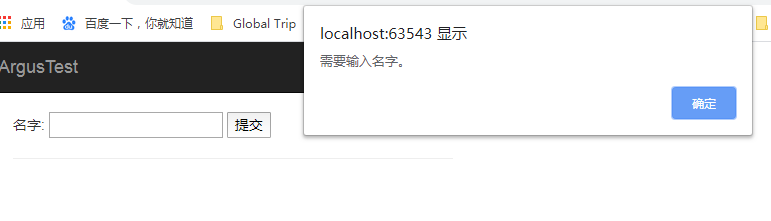
alert("需要输入名字。");

return false;

}

}

</script>



### HTML 表单自动验证

HTML 表单验证也可以通过浏览器来自动完成。

如果表单字段 (fname) 的值为空, **required** 属性会阻止表单提交：

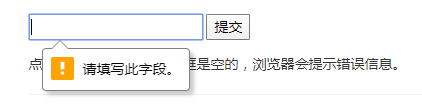
<form action="demo\_form.php" method="post">

<input type="text" name="fname" required="required">

<input type="submit" value="提交">

</form>

<p>点击提交按钮，如果输入框是空的，浏览器会提示错误信息。</p>



### 数据验证

数据验证用于确保用户输入的数据是有效的。

典型的数据验证有：

* 必需字段是否有输入?
* 用户是否输入了合法的数据?
* 在数字字段是否输入了文本?

大多数情况下，数据验证用于确保用户正确输入数据。

数据验证可以使用不同方法来定义，并通过多种方式来调用。

**服务端数据验证**是在数据提交到服务器上后再验证。

**客户端数据验证 side validation**是在数据发送到服务器前，在浏览器上完成验证。

### HTML 约束验证

HTML5 新增了 HTML 表单的验证方式：约束验证（constraint validation）。

约束验证是表单被提交时浏览器用来实现验证的一种算法。

HTML 约束验证基于：

* **HTML 输入属性**
* **CSS 伪类选择器**
* **DOM 属性和方法**

### 约束验证 HTML 输入属性

|  |  |
| --- | --- |
| **属性** | **描述** |
| disabled | 规定输入的元素不可用 |
| max | 规定输入元素的最大值 |
| min | 规定输入元素的最小值 |
| pattern | 规定输入元素值的模式 |
| required | 规定输入元素字段是必需的 |
| type | 规定输入元素的类型 |

### 约束验证 CSS 伪类选择器

|  |  |
| --- | --- |
| **选择器** | **描述** |
| :disabled | 选取属性为 "disabled" 属性的 input 元素 |
| :invalid | 选取无效的 input 元素 |
| :optional | 选择没有"required"属性的 input 元素 |
| :required | 选择有"required"属性的 input 元素 |
| :valid | 选取有效值的 input 元素 |

## 表单验证

JavaScript 可用来在数据被送往服务器前对 HTML 表单中的这些输入数据进行验证。

表单数据经常需要使用 JavaScript 来验证其正确性：

* 验证表单数据是否为空？
* 验证输入是否是一个正确的email地址？
* 验证日期是否输入正确？
* 验证表单输入内容是否为数字型？

### 必填（或必选）项目

实例：

<form name="myForm" action="#" onsubmit="return validateForm()" method="post">

姓: <input type="text" name="fname">

<input type="submit" value="提交">

</form>

<script>

function validateForm() {

var x = document.forms["myForm"]["fname"].value;

if (x == null || x == "") {

alert("姓必须填写");

return false;

}

}

</script>

### E-mail 验证

<form name="myForm" action="#" onsubmit="return validateForm();" method="post">

{% csrf\_token %}

Email: <input type="text" name="email">

<input type="submit" value="提交">

</form>

<script>

function validateForm() {

var x = document.forms["myForm"]["email"].value;

var atpos = x.indexOf("@");

var dotpos = x.lastIndexOf(".");

if (atpos < 1 || dotpos < atpos + 2 || dotpos + 2 >= x.length) {

alert("不是一个有效的 e-mail 地址");

return false;

}

}

</script>

## 验证 API

### 约束验证 DOM 方法

|  |  |
| --- | --- |
| **Property** | **Description** |
| checkValidity() | 如果 input 元素中的数据是合法的返回 true，否则返回 false。 |
| setCustomValidity() | 设置 input 元素的 validationMessage 属性，用于自定义错误提示信息的方法。  使用 setCustomValidity 设置了自定义提示后，validity.customError 就会变成true，则 checkValidity 总是会返回false。如果要重新判断需要取消自定义提示，方式如下：  setCustomValidity('')  setCustomValidity(null)  setCustomValidity(undefined) |

#### 实例：如果输入信息不合法，则返回错误信息

<p>输入数字并点击验证按钮:</p>

<input id="id1" type="number" min="100" max="300" required>

<button onclick="myFunction()">验证</button>

<p>如果输入的数字小于 100 或大于300，会提示错误信息。</p>

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

<script>

function myFunction() {

var inpObj = document.getElementById("id1");

if (inpObj.checkValidity() == false) {

document.getElementById("demo").innerHTML = inpObj.validationMessage;

} else {

document.getElementById("demo").innerHTML = "输入正确";

}

}

</script>



解析：

**HTMLSelectElement.checkValidity()**会检查元素是否有任何输入约束条件，并且检查值是否符合约束条件。 如果值是不符合约束条件的，浏览器就会在该元素上触发一个可以撤销的 invalid 事件。

### 实例：

<p>输入数字并点击验证按钮:</p>

<input id="nu" type="number" min="100" required>

<button onclick="myFunction()">验证</button>

<p>如果输入的数字小于 100 ( input 的 min 属性), 会显示错误信息。</p>

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

<script>

function myFunction() {

var x = document.getElementById("nu");

x.setCustomValidity("");//使用前先取消自定义，否则下次点击checkValidity总返false

if (x.checkValidity() == false) {

x.setCustomValidity("错误");

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

}

else {

x.setCustomValidity("正确");

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

}

}

</script>

### 约束验证 DOM 属性

|  |  |
| --- | --- |
| **属性** | **描述** |
| validity | 布尔属性值，返回 input 输入值是否合法 |
| validationMessage | 浏览器错误提示信息 |
| willValidate | 指定 input 是否需要验证 |

### Validity 属性

input 元素的 **validity 属性**包含一系列关于 validity 数据属性:

|  |  |
| --- | --- |
| **属性** | **描述** |
| customError | 设置为 true, 如果设置了自定义的 validity 信息。 |
| patternMismatch | 设置为 true, 如果元素的值不匹配它的模式属性。 |
| rangeOverflow | 设置为 true, 如果元素的值大于设置的最大值。 |
| rangeUnderflow | 设置为 true, 如果元素的值小于它的最小值。 |
| stepMismatch | 设置为 true, 如果元素的值不是按照规定的 step 属性设置。 |
| tooLong | 设置为 true, 如果元素的值超过了 maxLength 属性设置的长度。 |
| typeMismatch | 设置为 true, 如果元素的值不是预期相匹配的类型。 |
| valueMissing | 设置为 true，如果元素 (required 属性) 没有值。 |
| valid | 设置为 true，如果元素的值是合法的。 |

#### 实例：rangeOverflow 属性

如果输入的值大于 100，显示一个信息：

<p>输入数字并点击验证按钮:</p>

<input id="id1" type="number" max="100">

<button onclick="myFunction()">验证</button>

<p>如果输入的数字大于 100 ( input 的 max 属性), 会显示错误信息。</p>

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

<script>

function myFunction() {

var txt = "";

if (document.getElementById("id1").validity.rangeOverflow) {

txt = "输入的值太大了";

} else {

txt = "输入正确";

}

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

}

</script>

#### 实例：rangeUnderflow 属性

如果输入的值小于 100，显示一个信息：

<p>输入数字并点击验证按钮:</p>

<input id="id1" type="number" min="100" required>

<button onclick="myFunction()">验证</button>

<p>如果输入的数字小于 100 ( input 的 min 属性), 会显示错误信息。</p>

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

<script>

function myFunction() {

var txt = "";

var inpObj = document.getElementById("id1");

if (!isNumeric(inpObj.value)) {

txt = "你输入的不是数字";

} else if (inpObj.validity.rangeUnderflow) {

txt = "输入的值太小了";

} else {

txt = "输入正确";

}

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

}

// 判断输入是否为数字

function isNumeric(n) {

return !isNaN(parseFloat(n)) && isFinite(n);

}

</script>

## 保留关键字

在 JavaScript 中，一些标识符是保留关键字，不能用作变量名或函数名。

|  |
| --- |
|  |
| abstract | arguments | boolean | break | byte |
| case | catch | char | class\* | const |
| continue | debugger | default | delete | do |
| double | else | enum\* | eval | export\* |
| extends\* | false | final | finally | float |
| for | function | goto | if | implements |
| import\* | in | instanceof | int | interface |
| let | long | native | new | null |
| package | private | protected | public | return |
| short | static | super\* | switch | synchronized |
| this | throw | throws | transient | true |
| try | typeof | var | void | volatile |
| while | with | yield |  |  |

### 对象、属性和方法

也应该避免使用 JavaScript 内置的对象、属性和方法的名称作为 Javascript 的变量或函数名：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array | Date | eval | function | hasOwnProperty |
| Infinity | isFinite | isNaN | isPrototypeOf | length |
| Math | NaN | name | Number | Object |
| prototype | String | toString | undefined | valueOf |

### Java 保留关键字

avaScript 经常与 Java 一起使用。您应该避免使用一些 Java 对象和属性作为 JavaScript 标识符：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| getClass | java | JavaArray | javaClass | JavaObject | JavaPackage |

### Windows 保留关键字

JavaScript 可以在 HTML 外部使用。它可在许多其他应用程序中作为编程语言使用。

在 HTML 中，您必须（为了可移植性，您也应该这么做）避免使用 HTML 和 Windows 对象和属性的名称作为 Javascript 的变量及函数名：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| alert | all | anchor | anchors | area |
| assign | blur | button | checkbox | clearInterval |
| clearTimeout | clientInformation | close | closed | confirm |
| constructor | crypto | decodeURI | decodeURIComponent | defaultStatus |
| document | element | elements | embed | embeds |
| encodeURI | encodeURIComponent | escape | event | fileUpload |
| focus | form | forms | frame | innerHeight |
| innerWidth | layer | layers | link | location |
| mimeTypes | navigate | navigator | frames | frameRate |
| hidden | history | image | images | offscreenBuffering |
| open | opener | option | outerHeight | outerWidth |
| packages | pageXOffset | pageYOffset | parent | parseFloat |
| parseInt | password | pkcs11 | plugin | prompt |
| propertyIsEnum | radio | reset | screenX | screenY |
| scroll | secure | select | self | setInterval |
| setTimeout | status | submit | taint | text |
| textarea | top | unescape | untaint | window |

### HTML 事件句柄

除此之外，您还应该避免使用 HTML 事件句柄的名称作为 Javascript 的变量及函数名。

|  |  |  |  |
| --- | --- | --- | --- |
| onblur | onclick | onerror | onfocus |
| onkeydown | onkeypress | onkeyup | onmouseover |
| onload | onmouseup | onmousedown | onsubmit |

### 非标准 JavaScript

除了保留关键字，在 JavaScript 实现中也有一些非标准的关键字。

一个实例是 **const** 关键字，用于定义变量。 一些 JavaScript 引擎把 const 当作 var 的同义词。另一些引擎则把 const 当作只读变量的定义。

Const 是 JavaScript 的扩展。JavaScript 引擎支持它用在 Firefox 和 Chrome 中。但是它并不是 JavaScript 标准 ES3 或 ES5 的组成部分。**建议：不要使用它**。

## this 关键字

面向对象语言中 this 表示当前对象的一个引用。

但在 JavaScript 中 this 不是固定不变的，它会随着执行环境的改变而改变。

* 在方法中，this 表示该方法所属的对象。
* 如果单独使用，this 表示全局对象。
* 在函数中，this 表示全局对象。
* 在函数中，在严格模式下，this 是未定义的(undefined)。
* 在事件中，this 表示接收事件的元素。
* 类似 call() 和 apply() 方法可以将 this 引用到任何对象。

### 方法中的 this

在对象方法中， this 指向调用它所在方法的对象。

实例：

<h2>JavaScript <b>this</b> 关键字</h2>

<p>实例中，<b>this</b> 指向了 <b>person</b> 对象。</p>

<p>因为 person 对象是 fullName 方法的所有者。</p>

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

<script>

// 创建一个对象

var person = {

firstName: "John",

lastName: "Doe",

id: 5566,

fullName: function () {

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

}

};

// 显示对象的数据

document.getElementById("demo").innerHTML = person.fullName(); //John Doe

</script>

解析：this 表示 person 对象。fullName 方法所属的对象就是 person。

### 单独使用 this

单独使用 this，则它指向全局(Global)对象。在浏览器中，window 就是该全局对象为 [**object Window**]:

实例：

<h2>JavaScript <b>this</b> 关键字</h2>

<p>实例中，<b>this</b> 指向了 window 对象:</p>

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

<script>

var x = this;

document.getElementById("demo").innerHTML = x; //返回[object Window]

</script>

严格模式下，如果单独使用，this 也是指向全局(Global)对象。

<h2>JavaScript <b>this</b> 关键字</h2>

<p>实例中，<b>this</b> 指向了 window 对象:</p>

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

<script>

"use strict";

var x = this;

document.getElementById("demo").innerHTML = x; //返回[object Window]

</script>

### 函数中使用 this（默认）

在函数中，函数的所属者默认绑定到 this 上。

在浏览器中，window 就是该全局对象为 [**object Window**]:

<h2>JavaScript <b>this</b> 关键字</h2>

<p>实例中，<b>this</b> 表示 myFunction 函数的所有者：</p>

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

<script>

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

function myFunction() {

return this; //[object Window]

}

</script>

### 函数中使用 this（严格模式）

严格模式下函数是没有绑定到 this 上，这时候 this 是 **undefined**。

<h2>JavaScript <b>this</b> 关键字</h2>

<p>函数中，默认情况下，<b>this</b> 指向全局对象。</p>

<p>严格模式下，<b>this</b> 为 <b>undefined</b>，因为严格模式下不允许默认绑定:</p>

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

<script>

"use strict";

document.getElementById("demo").innerHTML = myFunction(); //undefined

function myFunction() {

return this;

}

</script>

### 事件中的 this

在 HTML 事件句柄中，this 指向了接收事件的 HTML 元素：

<h2>JavaScript <b>this</b> 关键字</h2>

<button onclick="this.style.display='none'">点我后我就消失了</button>

### 对象方法中绑定

实例：

<h2>JavaScript <b>this</b> 关键字</h2>

<p>在实例中，<b>this</b> 指向了 fullName 方法所属的对象 person。</p>

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

<script>

// 创建一个对象

var person = {

firstName: "John",

lastName: "Doe",

id: 5566,

myFunction: function () {

return this;

}

};

// 显示表单数据

document.getElementById("demo").innerHTML = person.myFunction(); //[object Object]

</script>

### 显式函数绑定

在 JavaScript 中函数也是对象，对象则有方法，apply 和 call 就是函数对象的方法。这两个方法异常强大，他们允许切换函数执行的上下文环境（context），即 this 绑定的对象。

<h2>JavaScript this 关键字</h2>

<p>实例中 <strong>this</strong> 指向了 person2，即便它是 person1 的方法:</p>

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

<script>

var person1 = {

fullName: function () {

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

}

}

var person2 = {

firstName: "John",

lastName: "Doe",

}

var x = person1.fullName.call(person2);

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

</script>

解析：当我们使用 person2 作为参数来调用 person1.fullName 方法时, **this** 将指向 person2, 即便它是 person1 的方法：

## let 和 const

let 声明的变量只在 let 命令所在的代码块内有效。

const 声明一个只读的常量，一旦声明，常量的值就不能改变。

在 ES6 之前，JavaScript 只有两种作用域： **全局变量** 与 **函数内的局部变量**。

### 全局变量

在函数外声明的变量作用域是全局的，全局变量在 JavaScript 程序的任何地方都可以访问。

### 局部变量

在函数内声明的变量作用域是局部的（函数内）：

<p>局部变量在声明的函数外不可以访问。</p>

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

<script>

myFunction();

document.getElementById("demo").innerHTML = "carName 的类型是：" + typeof carName;

//carName 的类型是：undefined

function myFunction() {

var carName = "Volvo";

}

</script>

注：函数内使用 var 声明的变量只能在函数内容访问，如果不使用 var 则是全局变量。

### JavaScript 块级作用域(Block Scope)

使用 var 关键字声明的变量不具备块级作用域的特性，它在 {} 外依然能被访问到。

{

var x = 2;

}

// 这里可以使用 x 变量

在 ES6 之前，是没有块级作用域的概念的。ES6 可以使用 let 关键字来实现块级作用域。let 声明的变量只在 let 命令所在的代码块 **{}** 内有效，在 **{}** 之外不能访问。

{

let x = 2;

}

// 这里不能使用 x 变量

### 重新定义变量

使用 var 关键字重新声明变量可能会带来问题。

在块中重新声明变量也会重新声明块外的变量：

<h2>使用 var 声明变量</h2>

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

<script>

var x = 10;

// 这里输出 x 为 10

{

var x = 2;

// 这里输出 x 为 2

}

// 这里输出 x 为 2

document.getElementById("demo").innerHTML = x; //2

</script>

let 关键字就可以解决这个问题，因为它只在 let 命令所在的代码块 **{}** 内有效。

<h2>使用 let 声明变量</h2>

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

<script>

var x = 10;

// 这里输出 x 为 10

{

let x = 2;

// 这里输出 x 为 2

}

// 这里输出 x 为 10

document.getElementById("demo").innerHTML = x; //10

</script>

### 循环作用域

使用 var 关键字：

<h2>JavaScript let</h2>

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

<script>

var i = 5;

for (var i = 0; i < 10; i++) {

// 一些代码...

}

document.getElementById("demo").innerHTML = i; //10

</script>

解析：使用了 **var** 关键字，它声明的变量是全局的，包括循环体内与循环体外。

使用 let 关键字：

<h2>JavaScript let</h2>

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

<script>

var i = 5;

for (let i = 0; i < 10; i++) {

// 一些代码...

}

document.getElementById("demo").innerHTML = i; //5

</script>

解析：使用 **let** 关键字， 它声明的变量作用域只在循环体内，循环体外的变量不受影响。

### 局部变量

在函数体内使用 **var** 和 **let** 关键字声明的变量有点类似。

它们的作用域都是 **局部的**:

// 使用 var

function myFunction() {

var carName = "Volvo"; // 局部作用域

}

// 使用 let

function myFunction() {

let carName = "Volvo"; // 局部作用域

}

### 全局变量

在函数体外或代码块外使用 **var** 和 **let** 关键字声明的变量也有点类似。它们的作用域都是 **全局的**:

// 使用 var

var x = 2; // 全局作用域

// 使用 let

let x = 2; // 全局作用域

### HTML 代码中使用全局变量

在 JavaScript 中, 全局作用域是针对 JavaScript 环境。

在 HTML 中, 全局作用域是针对 window 对象。

使用 **var** 关键字声明的全局作用域变量属于 window 对象:

var carName = "Volvo";

// 可以使用 window.carName 访问变量

使用 **let** 关键字声明的全局作用域变量不属于 window 对象:

let carName = "Volvo";

// 不能使用 window.carName 访问变量，若要访问，会返回undefined

### 重置变量

使用 **var** 关键字声明的变量在任何地方都可以修改：

在相同的作用域或块级作用域中，不能使用 **let** 关键字来重置 **var** 关键字声明的变量:

var x = 2; // 先用var声明x, 合法

let x = 3; // 再用let重置，不合法

{

var x = 4; //在代码块中，先用var声明x, 合法合法

let x = 5 // 在代码块中，再用let重置x，不合法

}

在相同的作用域或块级作用域中，不能使用 **let** 关键字来重置 **let** 关键字声明的变量:

let x = 2; // 先用let声明x,合法

let x = 3; //再用let重置，不合法

{

let x = 4; // 先用let声明x,合法

let x = 5; //再用let重置，不合法

}

在相同的作用域或块级作用域中，不能使用 **var** 关键字来重置 **let** 关键字声明的变量:

let x = 2; // 先用let声明x,合法

var x = 3; // 再用var声明x, 不合法

{

let x = 4; // 先用let声明x,合法

var x = 5; // 再用var声明x, 不合法

}

**let** 关键字在不同作用域，或不同块级作用域中是可以重新声明赋值的:

let x = 2; // 合法

{

let x = 3; // 合法

}

{

let x = 4; // 合法

}

### 变量提升

var 关键字定义的变量可以在使用后声明，也就是变量可以先使用再声明

let 关键字定义的变量则不可以在使用后声明，也就是变量需要先声明再使用。

### const 关键字

const 用于声明一个或多个常量，声明时必须进行初始化，且初始化后值不可再修改：

<h2>JavaScript const</h2>

<p>const 用于声明一个或多个常量，声明时必须进行初始化，且初始化后值不可再修改。</p>

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

<script>

try {

const PI = 3.141592653589793;

PI = 3.14;

}

catch (err) {

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

//TypeError: Assignment to constant variable.

}

</script>

const定义常量与使用let 定义的变量相似：

* 二者都是块级作用域
* 都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

两者还有以下两点区别：

* const声明的常量必须初始化，而let声明的变量不用
* const 定义常量的值不能通过再赋值修改，也不能再次声明。而 let 定义的变量值可以修改。

// 错误写法

const PI;

PI = 3.14159265359;

// 正确写法

const PI = 3.14159265359;

### 并非真正的常量

const 的本质: const 定义的变量并非常量，并非不可变，它定义了一个常量引用一个值。使用 const 定义的对象或者数组，其实是可变的。

// 创建常量对象

const car = { type: "Fiat", model: "500", color: "white" };

// 修改属性:

car.color = "red";

// 添加属性

car.owner = "Johnson";

但是我们不能对常量对象重新赋值：

const car = { type: "Fiat", model: "500", color: "white" };

car = { type: "Volvo", model: "EX60", color: "red" }; // 错误

#### 实例：修改常量数组

<h2>JavaScript const</h2>

<p>以下实例修改常量数组:</p>

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

<script>

// 创建常量数组

const cars = ["Saab", "Volvo", "BMW"];

// 修改元素

cars[0] = "Toyota";

// 添加元素

cars.push("Audi");

// 显示数组

document.getElementById("demo").innerHTML = cars; //Toyota,Volvo,BMW,Audi

</script>

解析：可以修改常量数组，但不能对常量数组重新赋值：cars = ["Toyota", "Volvo", "Audi"]; // 错误

### 重置变量

在相同的作用域或块级作用域中，不能使用 **const** 关键字来重置 **var** 和 **let**关键字声明的变量:

在相同的作用域或块级作用域中，不能使用 **const** 关键字来重置 **const** 关键字声明的变量:

**const** 关键字在不同作用域，或不同块级作用域中是可以重新声明赋值的:

<h2>JavaScript const</h2>

<p>以下实例修改常量数组:</p>

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

<script>

var result = "";

const x = 2; // 合法

result = x + '<br/>';

{

const x = 3; // 合法

result = result + x + '<br/>';

}

{

const x = 4; // 合法

result = result + x + '<br/>';

}

result = result + x + '<br/>'; //回到代码块外，x的值为2

document.getElementById('demo').innerHTML = result; //2、3、4、2

</script>

### 变量提升

const 关键字定义的变量则不可以在使用后声明，也就是变量需要先声明再使用。

carName = "Volvo"; // 在这里不可以使用 carName 变量

const carName = "Volvo";

## JSON

JSON 是用于存储和传输数据的格式。JSON 通常用于服务端向网页传递数据 。

### 什么是 JSON?

* JSON 英文全称 **J**ava**S**cript **O**bject **N**otation
* JSON 是一种轻量级的数据交换格式。
* JSON是独立的语言 **\***
* JSON 易于理解。

注：\* JSON 使用 JavaScript 语法，但是 JSON 格式仅仅是一个文本。文本可以被任何编程语言读取及作为数据格式传递。

### 实例：JSON 实例

{

"sites": [

{ "name": "Runoob", "url": "www.runoob.com" },

{ "name": "Google", "url": "www.google.com" },

{ "name": "Taobao", "url": "www.taobao.com" }

]

}

### JSON 格式化后为 JavaScript 对象

JSON 格式在语法上与创建 JavaScript 对象代码是相同的。

由于它们很相似，所以 JavaScript 程序可以很容易的将 JSON 数据转换为 JavaScript 对象。

### JSON 语法规则

* 数据为 键/值 对。
* 数据由逗号分隔。
* 大括号保存对象
* 方括号保存数组

### JSON 数据 - 一个名称对应一个值

JSON 数据格式为 键/值 对，就像 JavaScript 对象属性。

键/值对包括字段名称（在双引号中），后面一个冒号，然后是值：

"name": "Runoob"

### JSON 对象

JSON 对象保存在大括号内。就像在 JavaScript 中, 对象可以保存多个 键/值 对：

{ "name": "Google", "url": "www.google.com" },

### JSON 数组

JSON 数组保存在中括号内。就像在 JavaScript 中, 数组可以包含对象：

"sites": [

{ "name": "Runoob", "url": "www.runoob.com" },

{ "name": "Google", "url": "www.google.com" },

{ "name": "Taobao", "url": "www.taobao.com" }

]

### JSON 字符串转换为 JavaScript 对象

通常我们从服务器中读取 JSON 数据，并在网页中显示数据。

简单起见，我们网页中直接设置 JSON 字符串。

首先，创建 JavaScript 字符串，字符串为 JSON 格式的数据：

var text = '{ "sites" : [' +

'{ "name":"Runoob" , "url":"www.runoob.com" },' +

'{ "name":"Google" , "url":"www.google.com" },' +

'{ "name":"Taobao" , "url":"www.taobao.com" } ]}';

然后，使用 JavaScript 内置函数 JSON.parse() 将字符串转换为 JavaScript 对象:

var obj = JSON.parse(text);

实例：

<h2>为 JSON 字符串创建对象</h2>

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

<script>

var text = '{ "sites" : [' +

'{ "name":"Runoob" , "url":"www.runoob.com" },' +

'{ "name":"Google" , "url":"www.google.com" },' +

'{ "name":"Taobao" , "url":"www.taobao.com" } ]}';

obj = JSON.parse(text);

document.getElementById("demo").innerHTML = obj.sites[1].name + " " + obj.sites[1].url; //Google www.google.com

</script>

### 相关函数

|  |  |
| --- | --- |
| **函数** | **描述** |
| JSON.parse() | 用于将一个 JSON 字符串转换为 JavaScript 对象。 |
| JSON.stringify() | 用于将 JavaScript 值转换为 JSON 字符串。 |

### void(0) 含义

javascript:void(0) 中最关键的是 void 关键字， void 是 JavaScript 中非常重要的关键字，该操作符指定要计算一个表达式但是不返回值。

实例：

<a href="javascript:void(0)">单此处什么也不会发生</a>

解析：当用户链接时，void(0) 计算为 0，但 Javascript 上没有任何效果。

实例：

<p>点击以下链接查看结果：</p>

<a href="javascript:void(alert('Warning!!!'))">点我!</a>

解析：在用户点击链接后显示警告信息

实例：

<p>点击以下按钮查看结果：</p>

<form>

<input type="button" value="点我" onclick="getValue();" />

</form>

<script type="text/javascript">

function getValue() {

var a, b, c;

a = void (b = 5, c = 7);

document.write('a = ' + a + ' b = ' + b + ' c = ' + c); //a = undefined b = 5 c = 7

}

</script>

解析：void ()返回undefined

### href="#"与href="javascript:void(0)"的区别

**#** 包含了一个位置信息，默认的锚是**#top** 也就是网页的上端。

而javascript:void(0), 仅仅表示一个死链接。

在页面很长的时候会使用 **#** 来定位页面的具体位置，格式为：**# + id**。

如果你要定义一个死链接请使用 javascript:void(0) 。

<p>点击以下链接查看不同效果：</p>

<a href="javascript:void(0);">点我没有反应的!</a>

<br/>

<a href="#pos">点我定位到指定位置!</a>

<br/>

<p id="pos">尾部定位点</p>

## 代码规范

所有的 JavaScript 项目适用同一种规范。

代码规范通常包括以下几个方面:

* 变量和函数的命名规则
* 空格，缩进，注释的使用规则。
* 其他常用规范……

规范的代码可以更易于阅读与维护。

代码规范一般在开发前规定，可以跟你的团队成员来协商设置。

### 变量名

变量名推荐使用驼峰法来命名(**camelCase**):

例子：

firstName = "John";

lastName = "Doe";

price = 19.90;

tax = 0.20;

fullPrice = price + (price \* tax);

### 空格与运算符

通常运算符 ( = + - \* / ) 前后需要添加空格:

var x = y + z;

var values = ["Volvo", "Saab", "Fiat"];

### 代码缩进

通常使用 4 个空格符号来缩进代码块：

function toCelsius(fahrenheit) {

return (5 / 9) \* (fahrenheit - 32);

}

注：不推荐使用 TAB 键来缩进，因为不同编辑器 TAB 键的解析不一样。

### 语句规则

简单语句的通用规则:

* 一条语句通常以分号作为结束符。

var values = ["Volvo", "Saab", "Fiat"];

var person = {

firstName: "John",

lastName: "Doe",

age: 50,

eyeColor: "blue"

};

复杂语句的通用规则:

* 将左花括号放在第一行的结尾。
* 左花括号前添加一空格。
* 将右花括号独立放在一行。
* 不要以分号结束一个复杂的声明。

函数：

function toCelsius(fahrenheit) {

return (5 / 9) \* (fahrenheit - 32);

}

循环：

for (i = 0; i < 5; i++) {

x += i;

}

条件语句：

if (time < 20) {

greeting = "Good day";

} else {

greeting = "Good evening";

}

### 对象规则

对象定义的规则:

* 将左花括号与类名放在同一行。
* 冒号与属性值间有个空格。
* 字符串使用双引号，数字不需要。
* 最后一个属性-值对后面不要添加逗号。
* 将右花括号独立放在一行，并以分号作为结束符号。

var person = {

firstName: "John",

lastName: "Doe",

age: 50,

eyeColor: "blue"

};

短的对象代码可以直接写成一行:

var person = { firstName: "John", lastName: "Doe", age: 50, eyeColor: "blue" };

### 每行代码字符小于 80

为了便于阅读每行字符建议小于数 80 个。如果一个 JavaScript 语句超过了 80 个字符，建议在 运算符或者逗号后换行。

### 命名规则

一般很多代码语言的命名规则都是类似的，例如:

* 变量和函数为小驼峰法标识, 即除第一个单词之外，其他单词首字母大写（ **lowerCamelCase**）
* 全局变量为大写 (**UPPERCASE**)
* 常量 (如 PI) 为大写 (**UPPERCASE**)

变量命名你是否使用这几种规则： hyp-hens, camelCase, 或 under\_scores ?

HTML 和 CSS 的横杠(-)字符:

* HTML5 属性可以以 data- (如：data-quantity, data-price) 作为前缀。
* CSS 使用 - 来连接属性名 (font-size)。

注：- 通常在 JavaScript 中被认为是减法，所以不允许使用。

**下划线:**

很多程序员比较喜欢使用下划线(如：date\_of\_birth), 特别是在 SQL 数据库中。

PHP 语言通常都使用下划线。

**帕斯卡拼写法(PascalCase):**

帕斯卡拼写法(PascalCase) 在 C 语言中语言较多。

**驼峰法：**

JavaScript 中通常推荐使用驼峰法，jQuery 及其他 JavaScript 库都使用驼峰法。

注：变量名不要以 $ 作为开始标记，会与很多 JavaScript 库冲突。

### HTML 载入外部 JavaScript 文件

使用简洁的格式载入 JavaScript 文件 ( type 属性不是必须的):

<script src="myscript.js">

### 使用 JavaScript 访问 HTML 元素

一个糟糕的 HTML 格式可能会导致 JavaScript 执行错误。

### 文件扩展名

HTML 文件后缀可以是 **.html** (或 **.htm**)。

CSS 文件后缀是 **.css** 。

JavaScript 文件后缀是 **.js** 。

### 使用小写文件名

大多 Web 服务器 (Apache, Unix) 对大小写敏感： london.jpg 不能通过 London.jpg 访问。

其他 Web 服务器 (Microsoft, IIS) 对大小写不敏感： london.jpg 可以通过 London.jpg 或 london.jpg 访问。

你必须保持统一的风格，我们建议统一使用小写的文件名。

## 函数

JavaScript 使用关键字 **function** 定义函数。函数可以通过声明定义，也可以是一个表达式。

### 函数声明

function functionName(parameters) {

//执行的代码

}

函数声明后不会立即执行，会在我们需要的时候调用到。分号是用来分隔可执行JavaScript语句。由于函数声明不是一个可执行语句，所以不以分号结束。

### 函数表达式

JavaScript 函数可以通过一个表达式定义。

**匿名函数** (函数没有名称)：函数存储在变量中，不需要函数名称，通常通过变量名来调用。

<p>函数存储在变量后，变量可作为函数使用：</p>

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

<script>

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

document.getElementById("demo").innerHTML = x(4, 3); //12

</script>

### Function() 构造函数

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

var myFunction = new Function("a", "b", "return a \* b");

等效于：

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

注：在 JavaScript 中，很多时候，你需要避免使用 **new** 关键字。

### 函数提升（Hoisting）

提升（Hoisting）是 JavaScript 默认将当前作用域提升到前面去的的行为。提升（Hoisting）应用在变量的声明与函数的声明。

函数可以在声明之前调用：

<script>

myFunction(5);

function myFunction(y) {

return y \* y;

}

</script>

使用表达式定义函数时无法提升。

### 自调用函数

函数表达式可以 "自调用"。自调用表达式会自动调用。如果表达式后面紧跟 () ，则会自动调用。

不能自调用声明的函数。

通过添加括号，来说明它是一个函数表达式：

<p>函数可以自动调用：</p>

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

<script>

(function () {

document.getElementById("demo").innerHTML = "Hello! 我是自己调用的";

})();

</script>

以上函数实际上是一个 **匿名自我调用的函数** (没有函数名)。

### 函数可作为一个值使用

作为一个值使用：

function myFunction(a, b) {

return a \* b;

}

var x = myFunction(4, 3);

 函数可作为表达式使用：

function myFunction(a, b) {

return a \* b;

}

var x = myFunction(4, 3) \* 2;

### 函数是对象

在 JavaScript 中使用 **typeof** 操作符判断函数类型将返回 "function" 。但是JavaScript 函数描述为一个对象更加准确。

JavaScript 函数有 **属性** 和 **方法**。

实例：arguments.length 属性返回函数调用过程接收到的参数个数：

<p> arguments.length 属性返回函数接收到参数的个数：</p>

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

<script>

function myFunction(a, b) {

return arguments.length;

}

document.getElementById("demo").innerHTML = myFunction(4, 3); //2

</script>

toString() 方法将函数作为一个字符串返回:

<p> toString() 将函数作为一个字符串返回：</p>

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

<script>

function myFunction(a, b) {

return a \* b;

}

document.getElementById("demo").innerHTML = myFunction.toString();

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

</script>

注：函数定义作为对象的属性，称之为对象方法。函数如果用于创建新的对象，称之为对象的构造函数。

### 箭头函数

ES6 新增了箭头函数。箭头函数表达式的语法比普通函数表达式更简洁。

(参数1, 参数2, …, 参数N) => { 函数声明 }

(参数1, 参数2, …, 参数N) => 表达式(单一)

// 相当于：(参数1, 参数2, …, 参数N) =>{ return 表达式; }

当只有一个参数时，圆括号是可选的：

(单一参数) => { 函数声明 }

单一参数 => { 函数声明 }

没有参数的函数应该写成一对圆括号:

() => { 函数声明 }

#### 实例：箭头函数

<h2>JavaScript 箭头函数</h2>

<p>箭头函数不需要使用 function、return 关键字及大括号 {}。</p>

<p>IE11 及更早 IE 版本不支持箭头函数。</p>

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

<script>

const x = (x, y) => x \* y;

document.getElementById("demo").innerHTML = x(5, 5); //25

</script>

解析：在ES5下相当于： var x = function (x, y) { return x \* y; }

有的箭头函数都没有自己的 **this**。 不适合顶一个 **对象的方法**。

当我们使用箭头函数的时候，箭头函数会默认帮我们绑定外层 this 的值，所以在箭头函数中 this 的值和外层的 this 是一样的。

箭头函数是不能提升的，所以需要在使用之前定义。

使用 **const** 比使用 **var** 更安全，因为函数表达式始终是一个常量。

如果函数部分只是一个语句，则可以省略 return 关键字和大括号 {}，这样做是一个比较好的习惯:

const x = (x, y) => { return x \* y };

## 函数参数

JavaScript 函数对参数的值没有进行任何的检查。

### 函数显式参数(Parameters)与隐式参数(Arguments)

函数的显式参数：

functionName(parameter1, parameter2, parameter3) {

// 要执行的代码……

}

函数显式参数在函数定义时列出。函数隐式参数在函数调用时传递给函数真正的值。

### 参数规则

JavaScript 函数定义显式参数时没有指定数据类型。JavaScript 函数对隐式参数没有进行类型检测。JavaScript 函数对隐式参数的个数没有进行检测。

### 默认参数

ES5 中如果函数在调用时未提供隐式参数，参数会默认设置为： **undefined**

有时这是可以接受的，但是建议最好为参数设置一个默认值：

<p>设置参数的默认值。</p>

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

<script>

function myFunction(x, y) {

if (y === undefined) {

y = 0;

}

return x \* y;

}

document.getElementById("demo").innerHTML = myFunction(4); //返回值为0。此时值调用1个参数，第2个参数将取默认值。

</script>

注：更简单的方式： function myFunction(x, y) { y = y || 0; return x \* y; }

如果y已经定义 ， y || 返回 y, 因为 y 是 true, 否则返回 0, 因为 undefined 为 false。

### ES6 函数可以自带参数

ES6 支持函数带有默认参数，就判断 undefined 和 || 的操作：

<p>设置参数的默认值。</p>

<p id="demo1"></p>

<p id="demo2"></p>

<script>

function myFunction(x, y = 10) {

// 如果不传入参数 y ，则其默认值为 10

return x + y;

}

// 输出 2

document.getElementById("demo1").innerHTML = myFunction(0, 2);

// 输出 15, y 参数的默认值

document.getElementById("demo2").innerHTML = myFunction(5);

</script>

### arguments 对象

JavaScript 函数有个内置的对象 arguments 对象。argument 对象包含了函数调用的参数数组。

#### 实例：通过这种方式你可以很方便的找到最大的一个参数的值

<p>查找最大的数。</p>

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

<script>

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

function findMax() {

var i, max = arguments[0];

if (arguments.length < 2) return max;

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

if (arguments[i] > max) {

max = arguments[i];

}

}

return max;

}

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

</script>

#### 实例：创建一个函数用来统计所有数值的和：

<p>计算所有参数之和：</p>

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

<script>

function sumAll() {

var i, sum = 0;

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

sum += arguments[i];

}

return sum;

}

document.getElementById("demo").innerHTML =

sumAll(1, 123, 500, 115, 44, 88); //871

</script>

### 通过值传递参数

在函数中调用的参数是函数的隐式参数。

JavaScript 隐式参数通过值来传递：函数仅仅只是获取值。

如果函数修改参数的值，不会修改显式参数的初始值（在函数外定义）。

隐式参数的改变在函数外是不可见的。

### 通过对象传递参数

在JavaScript中，可以引用对象的值。

因此我们在函数内部修改对象的属性就会修改其初始的值。

修改对象属性可作用于函数外部（全局变量）。

修改对象属性在函数外是可见的。

## 函数调用

JavaScript 函数有 4 种调用方式。每种方式的不同在于 **this** 的初始化。

### this 关键字

一般而言，在Javascript中，this指向函数执行时的当前对象。

### 调用 JavaScript 函数

函数中的代码在函数被调用后执行。

### 作为一个函数调用

<p>

全局函数 (myFunction) 返回参数参数相乘的结果：

</p>

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

<script>

function myFunction(a, b) {

return a \* b;

}

document.getElementById("demo").innerHTML = myFunction(10, 2); //20

</script>

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

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

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

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

document.getElementById("demo").innerHTML = window.myFunction(10, 2);

### 全局对象

当函数没有被自身的对象调用时 **this** 的值就会变成全局对象。在 web 浏览器中全局对象是浏览器窗口（window 对象）。

<p>在 HTML 中 <b>this</b> 的值, 在全局函数是一个 window 对象。</p>

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

<script>

function myFunction() {

return this;

}

document.getElementById("demo").innerHTML = myFunction(); //[object Window]

</script>

### 函数作为方法调用

在 JavaScript 中你可以将函数定义为对象的方法。

fullName: function () {

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

}

**fullName** 方法是一个函数。函数属于对象。 **myObject** 是函数的所有者。**this**对象，拥有 JavaScript 代码。实例中 **this** 的值为 **myObject** 对象。

<p>myObject.fullName() 返回 John Doe:</p>

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

<script>

var myObject = {

firstName: "John",

lastName: "Doe",

fullName: function () {

return this;

}

}

myObject.fullName(); // 返回 [object Object] (所有者对象)

document.getElementById("demo").innerHTML = myObject.fullName(); //[object Object]

</script>

函数作为对象方法调用，会使得 **this** 的值成为对象本身。

### 使用构造函数调用函数

如果函数调用前使用了 **new** 关键字, 则是调用了构造函数。

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

<p>该实例中， myFunction 是函数构造函数:</p>

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

<script>

function myFunction(arg1, arg2) {

this.firstName = arg1;

this.lastName = arg2;

}

var x = new myFunction("John", "Doe")

document.getElementById("demo").innerHTML = x.firstName; //John

</script>

构造函数的调用会创建一个新的对象。新对象会继承构造函数的属性和方法。

注：构造函数中 **this** 关键字没有任何的值。**this** 的值在函数调用实例化对象(new object)时创建。

### 作为函数方法调用函数

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

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

#### 实例：call方法

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

<script>

var myObject;

function myFunction(a, b) {

return a \* b;

}

myObject = myFunction.call(myObject, 10, 2);

document.getElementById("demo").innerHTML = myObject; // 返回 20

</script>

#### 实例：apply方法

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

<script>

var myObject, myArray;

function myFunction(a, b) {

return a \* b;

}

myArray = [10, 2]

myObject = myFunction.apply(myObject, myArray);

document.getElementById("demo").innerHTML = myObject; // 返回 20

</script>

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

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

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

#### 实例：apply方法

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

<script>

var myObject, myArray;

myObject = {

name: "hahaha ",

hsk: "en"

};

function myFunction(a, b) {

alert(this);

return this.name + this.hsk;

}

myArray = [10, 2]

myObject = myFunction.apply(myObject, myArray);

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

//会触发alert(object.object)，然后返回hahaha en

</script>

注意，其中myObject的属性名称与myFunction返回的this.属性名称一致，否则会返回undefined。

## 闭包

JavaScript 变量可以是局部变量或全局变量。私有变量可以用到闭包。

### 全局变量

函数可以访问由函数内部定义的变量，如：

function myFunction() {

var a = 4; //局部变量

return a \* a;

}

函数也可以访问函数外部定义的变量，如：

var a = 4; //全局变量

function myFunction() {

return a \* a;

}

局部变量只能用于定义它函数内部。对于其他的函数或脚本代码是不可用的。

全局和局部变量即便名称相同，它们也是两个不同的变量。修改其中一个，不会影响另一个的值。

注：变量声明时如果不使用 **var** 关键字，那么它就是一个全局变量，即便它在函数内定义。

### 变量生命周期

全局变量的作用域是全局性的，即在整个JavaScript程序中，全局变量处处都在。

而在函数内部声明的变量，只在函数内部起作用。这些变量是局部变量，作用域是局部性的；函数的参数也是局部性的，只在函数内部起作用。

#### 实例：计数器困境

设想下如果你想统计一些数值，且该计数器在所有函数中都是可用的。

你可以使用全局变量，函数设置计数器递增：

var counter = 0;

function add() {

return counter += 1;

}

add();

add();

add();

// 计数器现在为 3

计数器数值在执行 add() 函数时发生变化。

但问题来了，页面上的任何脚本都能改变计数器，即便没有调用 add() 函数。

如果我在函数内声明计数器，如果没有调用函数将无法修改计数器的值：

function add() {

var counter = 0;

return counter += 1;

}

add();

add();

add();

// 本意是想输出 3, 但事与愿违，输出的都是 1 !

以上代码将无法正确输出，每次我调用 add() 函数，计数器都会设置为 1。

**JavaScript 内嵌函数可以解决该问题。**

#### JavaScript 内嵌函数

所有函数都能访问全局变量。

实际上，在 JavaScript 中，所有函数都能访问它们上一层的作用域。

JavaScript 支持嵌套函数。嵌套函数可以访问上一层的函数变量。

该实例中，内嵌函数 **plus()** 可以访问父函数的 **counter** 变量：

function add() {

var counter = 0;

function plus() { counter += 1; }

plus();

return counter;

}

如果我们能在外部访问 **plus()** 函数，这样就能解决计数器的困境。

我们同样需要确保 **counter = 0** 只执行一次。

**我们需要闭包。**

#### 闭包

函数自我调用：

<p>局部变量计数。</p>

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

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

<script>

var add = (function () {

var counter = 0;

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

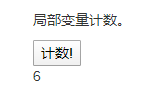
})();

function myFunction() {

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

}

</script>



#### 实例解析

变量 **add** 指定了函数自我调用的返回字值。

自我调用函数只执行一次。设置计数器为 0。并返回函数表达式。

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

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

计数器受匿名函数的作用域保护，只能通过 add 方法修改。

注：闭包是一种保护私有变量的机制，在函数执行时形成私有的作用域，保护里面的私有变量不受外界干扰。直观的说就是形成一个不销毁的栈环境。

即：在使用闭包的情况下，即可以使用函数内私有变量，而不像全局变量那样能被任何函数影响，同时又可以在函数内部进行计算，而不像纯私有变量那样不能进行计算。

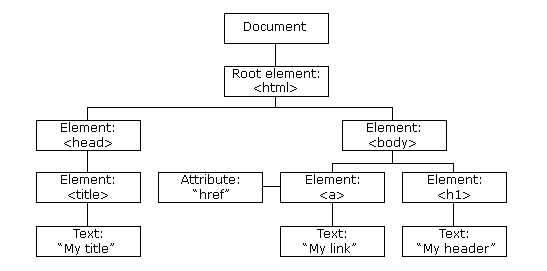
## HTML DOM

通过 HTML DOM，可访问 JavaScript HTML 文档的所有元素。

### HTML DOM (文档对象模型)

当网页被加载时，浏览器会创建页面的文档对象模型（Document Object Model）。

**HTML DOM** 模型被构造为**对象**的树：



通过可编程的对象模型，JavaScript 获得了足够的能力来创建动态的 HTML。

* JavaScript 能够改变页面中的所有 HTML 元素
* JavaScript 能够改变页面中的所有 HTML 属性
* JavaScript 能够改变页面中的所有 CSS 样式
* JavaScript 能够对页面中的所有事件做出反应

### 查找 HTML 元素

通常，通过 JavaScript，您需要操作 HTML 元素。为了做到这件事情，您必须首先找到该元素。有三种方法来做这件事：

* 通过 id 找到 HTML 元素
* 通过标签名找到 HTML 元素
* 通过类名找到 HTML 元素

### 通过 id 查找 HTML 元素

在 DOM 中查找 HTML 元素的最简单的方法，是通过使用元素的 id。

var x = document.getElementById("intro");

注：如果找到该元素，则该方法将以对象（在 x 中）的形式返回该元素。如果未找到该元素，则 x 将包含 null。

### 通过标签名查找 HTML 元素

本例查找 id="main" 的元素，然后查找 id="main" 元素中的所有 <p> 元素：

var x = document.getElementById("main");

var y = x.getElementsByTagName("p");

### 通过类名找到 HTML 元素

本例通过 getElementsByClassName 函数来查找 class="intro" 的元素：

var x = document.getElementsByClassName("intro");

## 改变 HTML

### 改变 HTML 输出流

JavaScript 能够创建动态的 HTML 内容：

**今天的日期是： Fri Feb 21 2020 12:43:44 GMT-0500 (北美东部标准时间)**

document.write() 可用于直接向 HTML 输出流写内容

<script>

document.write(Date());

</script>

注：绝对不要在文档(DOM)加载完成之后使用 document.write()。这会覆盖该文档。

### 改变 HTML 内容

修改 HTML 内容的最简单的方法是使用 innerHTML 属性。

如需改变 HTML 元素的内容，请使用这个语法：

document.getElementById(id).innerHTML=新的 HTML

#### 实例：改变了 <p>元素的内容

<p id="p1">Hello World!</p>

<script>

document.getElementById("p1").innerHTML = "新文本!";

</script>

#### 实例：改变了 <h1> 元素的内容

<h1 id="header">Old Header</h1>

<script>

var element = document.getElementById("header");

element.innerHTML = "新标题";

</script>

实例讲解：

* 上面的 HTML 文档含有 id="header" 的 <h1> 元素
* 我们使用 HTML DOM 来获得 id="header" 的元素
* JavaScript 更改此元素的内容 (innerHTML)

### 改变 HTML 属性

语法： document.getElementById(id).attribute = 新属性值

#### 实例：改变了 <img> 元素的 src 属性

<img id="image" src="smiley.gif">

<script>

document.getElementById("image").src = "landscape.jpg";

</script>

实例讲解：

* 上面的 HTML 文档含有 id="image" 的 <img> 元素
* 我们使用 HTML DOM 来获得 id="image" 的元素
* JavaScript 更改此元素的属性（把 "smiley.gif" 改为 "landscape.jpg"）

## 改变CSS

### 改变 HTML 样式

document.getElementById(id).style.property = 新样式

<p id="p1">Hello World!</p>

<p id="p2">Hello World!</p>

<script>

document.getElementById("p2").style.color = "blue";

document.getElementById("p2").style.fontFamily = "Arial";

document.getElementById("p2").style.fontSize = "larger";

</script>

<p>以上段落通过脚本修改。</p>

实例：改变了 id="id1" 的 HTML 元素的样式，当用户点击按钮时：

<h1 id="id1">我的标题 1</h1>

<button type="button"

onclick="document.getElementById('id1').style.color='red'">

点我!</button>

## DOM 事件

HTML DOM 使 JavaScript 有能力对 HTML 事件做出反应。

### 对事件做出反应

我们可以在事件发生时执行 JavaScript，比如当用户在 HTML 元素上点击时。

如需在用户点击某个元素时执行代码，请向一个 HTML 事件属性添加 JavaScript 代码：

onclick=JavaScript

HTML 事件的例子：

* 当用户点击鼠标时
* 当网页已加载时
* 当图像已加载时
* 当鼠标移动到元素上时
* 当输入字段被改变时
* 当提交 HTML 表单时
* 当用户触发按键时

#### 实例：点击文本

<h1 onclick="changetext(this)">点击文本!</h1>

<script>

function changetext(id) {

id.innerHTML = "Ooops!";

}

</script>

### 事件属性

实例：向 button 元素分配 onclick 事件

<p>点击按钮执行 <em>displayDate()</em> 函数.</p>

<button onclick="displayDate()">点这里</button>

<script>

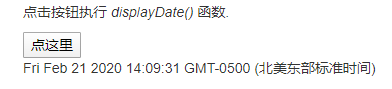
function displayDate() {

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

}

</script>

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



### 使用 HTML DOM 来分配事件

 允许您使用 JavaScript 来向 HTML 元素分配事件：

<p>点击按钮执行 <em>displayDate()</em> 函数.</p>

<button id="myBtn">点这里</button>

<script>

document.getElementById("myBtn").onclick = function () { displayDate() };

function displayDate() {

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

}

</script>

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

### onload 和 onunload 事件

onload 和 onunload 事件会在用户进入或离开页面时被触发。

onload 事件可用于检测访问者的浏览器类型和浏览器版本，并基于这些信息来加载网页的正确版本。

onload 和 onunload 事件可用于处理 cookie。

<body onload="checkCookies()">

<p>弹窗-提示浏览器 cookie 是否可用。</p>

<script>

function checkCookies() {

if (navigator.cookieEnabled == true) {

alert("Cookies 可用")

}

else {

alert("Cookies 不可用")

}

}

</script>

</body>

### onchange 事件

onchange 事件常结合对输入字段的验证来使用。

#### 实例：当用户改变输入字段的内容时，会调用 upperCase() 函数。

输入你的名字: <input type="text" id="fname" onchange="myFunction()">

<p>当你离开输入框后，函数将被触发，将小写字母转为大写字母。</p>

<script>

function myFunction() {

var x = document.getElementById("fname");

x.value = x.value.toUpperCase();

}

</script>

### onmouseover 和 onmouseout 事件

onmouseover 和 onmouseout 事件可用于在用户的鼠标移至 HTML 元素上方或移出元素时触发函数。

#### 实例：onmouseover 和 onmouseout 事件

<div onmouseover="mOver(this)" onmouseout="mOut(this)" style="background-color:#D94A38;width:120px;height:20px;padding:40px;">Mouse Over Me</div>

<script>

function mOver(obj) {

obj.innerHTML = "Thank You"

}

function mOut(obj) {

obj.innerHTML = "Mouse Over Me"

}

</script>

### onmousedown、onmouseup 以及 onclick 事件

onmousedown, onmouseup 以及 onclick 构成了鼠标点击事件的所有部分。首先当点击鼠标按钮时，会触发 onmousedown 事件，当释放鼠标按钮时，会触发 onmouseup 事件，最后，当完成鼠标点击时，会触发 onclick 事件。

#### 实例：onmousedown、onmouseup事件

<div onmousedown="mDown(this)" onmouseup="mUp(this)" style="width:120px;height:20px;padding:40px;">Click</div>

<script>

function mDown(obj) {

obj.innerHTML = "Release"; //鼠标放下时代码

}

function mUp(obj) {

obj.innerHTML = "Click" //鼠标释放时代码

}

</script>

#### 实例：onfocus事件

输入你的名字: <input type="text" onfocus="myFunction(this)">

<p>当输入框获取焦点时，修改背景色（background-color属性） 将被触发。</p>

<script>

function myFunction(x) {

x.style.background = "yellow";

}

</script>



## DOM EventListener

### addEventListener() 方法

<p>该实例使用 addEventListener() 方法在按钮中添加点击事件。 </p>

<button id="myBtn">点我</button>

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

<script>

document.getElementById("myBtn").addEventListener("click", displayDate);

function displayDate() {

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

}

</script>

addEventListener() 方法用于向指定元素添加事件句柄。addEventListener() 方法添加的事件句柄不会覆盖已存在的事件句柄。

你可以向一个元素添加多个事件句柄。你可以向同个元素添加多个同类型的事件句柄，如：两个 "click" 事件。

你可以向任何 DOM 对象添加事件监听，不仅仅是 HTML 元素。如： window 对象。

addEventListener() 方法可以更简单的控制事件（冒泡与捕获）。

当你使用 addEventListener() 方法时, JavaScript 从 HTML 标记中分离开来，可读性更强， 在没有控制HTML标记时也可以添加事件监听。

你可以使用 removeEventListener() 方法来移除事件的监听。

语法：

element.addEventListener(event, function, useCapture);

第一个参数是事件的类型 (如 "click" 或 "mousedown").

第二个参数是事件触发后调用的函数。

第三个参数是个布尔值用于描述事件是冒泡还是捕获。该参数是可选的。

注意：不要使用 "on" 前缀。 例如，使用 "click" ,而不是使用 "onclick"。

### 向原元素添加事件句柄

<p>该实例使用 addEventListener() 方法在按钮中添加点击事件。 </p>

<button id="myBtn">点我</button>

<script>

document.getElementById("myBtn").addEventListener("click", function () {

alert("Hello World!");

});

</script>

可以使用函数名，来引用外部函数:

<p>该实例使用 addEventListener() 方法在用户点击按钮时执行函数。</p>

<button id="myBtn">点我</button>

<script>

document.getElementById("myBtn").addEventListener("click", myFunction);

function myFunction() {

alert("Hello World!");

}

</script>

### 向同一个元素中添加多个事件句柄

addEventListener() 方法允许向同一个元素添加多个事件，且不会覆盖已存在的事件：

<p>该实例使用 addEventListener() 方法向同个按钮中添加两个点击事件。</p>

<button id="myBtn">点我</button>

<script>

var x = document.getElementById("myBtn");

x.addEventListener("click", myFunction); //先执行myFunction

x.addEventListener("click", someOtherFunction); //后执行someOtherFunction

function myFunction() {

alert("Hello World!")

}

function someOtherFunction() {

alert("函数已执行!")

}

</script>

可以向同个元素添加不同类型的事件：

<p>实例使用 addEventListener() 方法在同一个按钮中添加多个事件。</p>

<button id="myBtn">点我</button>

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

<script>

var x = document.getElementById("myBtn");

x.addEventListener("mouseover", myFunction);

x.addEventListener("click", mySecondFunction);

x.addEventListener("mouseout", myThirdFunction);

function myFunction() {

document.getElementById("demo").innerHTML += "Moused over!<br>"

}

function mySecondFunction() {

document.getElementById("demo").innerHTML += "Clicked!<br>"

}

function myThirdFunction() {

document.getElementById("demo").innerHTML += "Moused out!<br>"

}

</script>

### 向 Window 对象添加事件句柄

addEventListener() 方法允许你在 HTML DOM 对象添加事件监听， HTML DOM 对象如： HTML 元素, HTML 文档, window 对象。或者其他支出的事件对象如: xmlHttpRequest 对象。

#### 实例：当用户重置窗口大小时添加事件监听：

<p>实例在 window 对象中使用 addEventListener() 方法。</p>

<p>尝试重置浏览器的窗口触发 "resize" 事件句柄。</p>

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

<script>

window.addEventListener("resize", function () {

document.getElementById("demo").innerHTML = Math.random();

});

</script>

### 传递参数

当传递参数值时，使用"匿名函数"调用带参数的函数：

<p>实例演示了在使用 addEventListener() 方法时如何传递参数。</p>

<p>点击按钮执行计算。</p>

<button id="myBtn">点我</button>

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

<script>

var p1 = 5;

var p2 = 7;

document.getElementById("myBtn").addEventListener("click", function () {

myFunction(p1, p2);

});

function myFunction(a, b) {

var result = a \* b;

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

}

</script>

### 事件冒泡或事件捕获？

事件传递有两种方式：冒泡与捕获。

事件传递定义了元素事件触发的顺序。 如果你将 <p> 元素插入到 <div> 元素中，用户点击 <p> 元素, 哪个元素的 "click" 事件先被触发呢？

在 冒泡 中，内部元素的事件会先被触发，然后再触发外部元素，即： <p> 元素的点击事件先触发，然后会触发 <div> 元素的点击事件。

在 捕获 中，外部元素的事件会先被触发，然后才会触发内部元素的事件，即： <div> 元素的点击事件先触发 ，然后再触发 <p> 元素的点击事件。

addEventListener() 方法可以指定 "useCapture" 参数来设置传递类型：

addEventListener(event, function, useCapture);

### 实例：事件冒泡或事件捕获

<style>

div {

background-color: coral;

border: 1px solid;

padding: 50px;

}

</style>

<p>实例演示了在添加不同事件监听时，冒泡与捕获的不同。</p>

<div id="myDiv">

<p id="myP">点击段落，我是冒泡。</p>

</div><br>

<div id="myDiv2">

<p id="myP2">点击段落，我是捕获。 </p>

</div>

<script>

document.getElementById("myP").addEventListener("click", function () {

alert("你点击了 P 元素!");

}, false); //冒泡，参数为false

document.getElementById("myDiv").addEventListener("click", function () {

alert(" 你点击了 DIV 元素 !");

}, false); //冒泡，参数为false

document.getElementById("myP2").addEventListener("click", function () {

alert("你点击了 P2 元素!");

}, true); //捕获，参数为true

document.getElementById("myDiv2").addEventListener("click", function () {

alert("你点击了 DIV2 元素 !"); //捕获，参数为true

}, true);

</script>

### removeEventListener() 方法

removeEventListener() 方法移除由 addEventListener() 方法添加的事件句柄:

<style>

#myDIV {

background-color: coral;

border: 1px solid;

padding: 50px;

color: white;

}

</style>

<div id="myDIV"> div 元素添加了 onmousemove 事件句柄，鼠标在桔红色的框内移动时会显示随机数。

<p>点击按钮移除 DIV 的事件句柄。</p>

<button onclick="removeHandler()" id="myBtn">点我</button> <!-- 点击按钮后将释放事件句柄 -->

</div>

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

<script>

document.getElementById("myDIV").addEventListener("mousemove", myFunction);

function myFunction() {

document.getElementById("demo").innerHTML = Math.random();

}

function removeHandler() {

document.getElementById("myDIV").removeEventListener("mousemove", myFunction);

}

</script>

### 浏览器支持

实例：跨浏览器解决方法

<p> Internet Explorer 8 及更早IE版本不支持 addEventListener() 方法。</p>

<p>该实例演示了所有浏览器兼容的解决方法。</p>

<button id="myBtn">点我</button>

<script>

var x = document.getElementById("myBtn");

if (x.addEventListener) {

x.addEventListener("click", myFunction);

} else if (x.attachEvent) {

x.attachEvent("onclick", myFunction);

}

function myFunction() {

alert("Hello World!");

}

</script>

## DOM 元素 (节点)

### 创建新的 HTML 元素 (节点) - appendChild()

使用 **appendChild()** 方法，用于添加新元素到尾部。

#### 实例：创建新节点

<div id="div1">

<p id="p1">这是一个段落。</p>

<p id="p2">这是另外一个段落。</p>

</div>

<script>

var para = document.createElement("p");

var node = document.createTextNode("这是新的段落。");

para.appendChild(node);

var element = document.getElementById("div1");

element.appendChild(para);

</script>

解析：

* 用于创建 <p> 元素: var para = document.createElement("p");
* 为 <p> 元素创建一个新的文本节点：var node = document.createTextNode("这是新的段落。");
* 将文本节点添加到 <p> 元素中：para.appendChild(node);
* 查找已存在的元素：var element = document.getElementById("div1");
* 添加到已存在的元素中: element.appendChild(para);

### 创建新的 HTML 元素 (节点) - insertBefore()

如果我们需要将新元素添加到开始位置，可以使用 **insertBefore()** 方法:

<div id="div1">

<p id="p1">这是一个段落。</p>

<p id="p2">这是另外一个段落。</p>

</div>

<script>

var para = document.createElement("p"); //创建p元素

var node = document.createTextNode("这是一个新的段落。"); //创建文本节点

para.appendChild(node); //将文本节点添加到p元素

var element = document.getElementById("div1"); //查找到已存在的父元素

var child = document.getElementById("p1"); //查找已存在子元素

element.insertBefore(para, child); //将目标元素插入到父元素中，但在子元素前

</script>

### 移除已存在的元素

要移除一个元素，你需要知道该元素的父元素。

#### 实例：移除已存在的元素

<div id="div1">

<p id="p1">这是一个段落。</p>

<p id="p2">这是另外一个段落。</p>

</div>

<script>

var parent = document.getElementById("div1"); //查找 id="div1" 的元素:

var child = document.getElementById("p1"); //查找 id="p1" 的 <p> 元素:

parent.removeChild(child); //从父元素中移除子节点：

</script>

注：如果能够在不引用父元素的情况下删除某个元素，就太好了。不过很遗憾。DOM 需要清楚您需要删除的元素，以及它的父元素。

### 替换 HTML 元素 - replaceChild()

可以使用 replaceChild() 方法来替换 HTML DOM 中的元素。

<div id="div1">

<p id="p1">这是一个段落。</p>

<p id="p2">这是另外一个段落。</p>

</div>

<script>

var para = document.createElement("p");

var node = document.createTextNode("这是一个新的段落。");

para.appendChild(node); //创建新节点

var parent = document.getElementById("div1"); //查找父节点

var child = document.getElementById("p1"); //查找子节点

parent.replaceChild(para, child); //将子节点替换成新节点

</script>

## DOM 集合(Collection)

### HTMLCollection 对象

getElementsByTagName() 方法返回 HTMLCollection 对象。

HTMLCollection 对象类似包含 HTML 元素的一个数组。

<h2>JavaScript HTML DOM</h2>

<p>Hello World!</p>

<p>Hello Runoob!</p>

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

<script>

var myCollection = document.getElementsByTagName("p"); //获取所有p元素集合

document.getElementById("demo").innerHTML = "第二个段落的内容为:<span style='color:red;'> " + myCollection[1].innerHTML + '</span>';

//在demo的p元素中写入p元素集合myCollection[1]的内容

</script>

### HTML对象 length 属性

Collection 对象的 length 属性定义了集合中元素的数量。

#### 实例：HTMLCollection 对象 length 属性

<h2>JavaScript HTML DOM</h2>

<p>Hello World!</p>

<p>Hello Runoob!</p>

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

<script>

var myCollection = document.getElementsByTagName("p"); //获取 <p> 元素的集合

document.getElementById("demo").innerHTML = "文档包含" + myCollection.length + " 个段落。"; //显示集合元素个数

</script>

集合 length 属性常用于遍历集合中的元素。

<h2>JavaScript HTML DOM</h2>

<p>Hello World!</p>

<p>Hello Runoob!</p>

<p>点击按钮修改 p 元素的背景颜色。</p>

<button onclick="myFunction()">点我</button>

<script>

function myFunction() {

var myCollection = document.getElementsByTagName("p");

var i;

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

myCollection[i].style.color = "red";

}

}

</script>

注意：

**HTMLCollection 不是一个数组！**

HTMLCollection 看起来可能是一个数组，但其实不是。

你可以像数组一样，使用索引来获取元素。

HTMLCollection 无法使用数组的方法： valueOf(), pop(), push(), 或 join() 。

## HTML DOM 节点列表

NodeList 对象是一个从文档中获取的节点列表 (集合) 。

NodeList 对象类似 HTMLCollection 对象。

一些旧版本浏览器中的方法（如：getElementsByClassName()）返回的是 NodeList 对象，而不是 HTMLCollection 对象。

所有浏览器的 childNodes 属性返回的是 NodeList 对象。

大部分浏览器的 querySelectorAll() 返回 NodeList 对象。

<h2>JavaScript HTML DOM!</h2>

<p>Hello World!</p>

<p>Hello Runoob!</p>

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

<script>

var myNodelist = document.querySelectorAll("p"); //获取所有<p>元素集合

document.getElementById("demo").innerHTML = "第二个段落的内容为:<span style='color:red;'> " + myNodelist[1].innerHTML + '</span>';

//将集合中的第二个元素的文本写入demo中

</script>



### NodeList 对象 length 属性

NodeList 对象 length 属性定义了节点列表中元素的数量。

#### 实例：NodeList 对象 length 属性

<h2>JavaScript HTML DOM!</h2>

<p>Hello World!</p>

<p>Hello Runoob!</p>

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

<script>

var myNodelist = document.querySelectorAll("p"); //获取所有<p>元素集合

document.getElementById("demo").innerHTML = "文档包含 " + myNodelist.length + " 个段落。";

//获取集合元素数量

</script>

length 属性常用于遍历节点列表。

#### 实例：修改节点列表中所有 <p> 元素的背景颜色

<h2>JavaScript HTML DOM!</h2>

<p>Hello World!</p>

<p>Hello Runoob!</p>

<p>点击按钮修改所有 p 元素的背景颜色。</p>

<button onclick="myFunction()">点我</button>

<script>

function myFunction() {

var myNodelist = document.querySelectorAll("p"); //获取所有<p>元素集合

var i;

for (i = 0; i < myNodelist.length; i++) { //遍历所有集合的元素

myNodelist[i].style.color = "red";

}

}

</script>

### HTMLCollection 与 NodeList 的区别

HTMLCollection 是 HTML 元素的集合。

NodeList 是一个文档节点的集合。

NodeList 与 HTMLCollection 有很多类似的地方：

* NodeList 与 HTMLCollection 都与数组对象有点类似，可以使用索引 (0, 1, 2, 3, 4, ...) 来获取元素。
* NodeList 与 HTMLCollection 都有 length 属性。

不同：

* HTMLCollection 元素可以通过 name，id 或索引来获取。
* NodeList 只能通过索引来获取。
* 只有 NodeList 对象有包含属性节点和文本节点。

注意：

***节点列表不是一个数组！***

节点列表看起来可能是一个数组，但其实不是。

你可以像数组一样，使用索引来获取元素。

节点列表无法使用数组的方法： valueOf(), pop(), push(), 或 join() 。

## JavaScript 对象

JavaScript 中的所有事物都是对象：字符串、数值、数组、函数...。此外，JavaScript 允许自定义对象。

### JavaScript 对象

JavaScript 提供多个内建对象，比如 String、Date、Array 等等。对象只是带有属性和方法的特殊数据类型。对象只是一种特殊的数据。对象拥有**属性**和**方法**。

### 访问对象的属性

属性是与对象相关的值。

访问对象属性的语法是： objectName.propertyName

### 访问对象的方法

方法是能够在对象上执行的动作。

语法：objectName.methodName()

### 创建 JavaScript 对象

通过 JavaScript，您能够定义并创建自己的对象。

创建新对象有两种不同的方法：

* 定义并创建对象的实例
* 使用函数来定义对象，然后创建新的对象实例

### 创建直接的实例

实例：创建了对象的一个新实例，并向其添加了四个属性

<script>

var person = new Object();

person.firstname = "John";

person.lastname = "Doe";

person.age = 50;

person.eyecolor = "blue";

document.write(person.firstname + " is " + person.age + " years old."); //John is 50 years old.

</script>

或：

<script>

person = { firstname: "John", lastname: "Doe", age: 50, eyecolor: "blue" }

document.write(person.firstname + " is " + person.age + " years old."); //John is 50 years old.

</script>

### 使用对象构造器

<script>

//构造函数

function person(firstname, lastname, age, eyecolor) {

this.firstname = firstname;

this.lastname = lastname;

this.age = age;

this.eyecolor = eyecolor;

}

//创建实例

myFather = new person("John", "Doe", 50, "blue");

document.write(myFather.firstname + " is " + myFather.age + " years old.");

//John is 50 years old.

</script>

注：在JavaScript中，this通常指向的是我们正在执行的函数本身，或者是指向该函数所属的对象（运行时）

### 创建 JavaScript 对象实例

一旦您有了对象构造器，就可以创建新的对象实例，就像这样：

var myFather = new person("John", "Doe", 50, "blue");

var myMother = new person("Sally", "Rally", 48, "green");

### 把属性添加到 JavaScript 对象

您可以通过为对象赋值，向已有对象添加新属性。

### 把方法添加到 JavaScript 对象

方法只不过是附加在对象上的函数。

在构造器函数内部定义对象的方法：

function person(firstname, lastname, age, eyecolor) {

this.firstname = firstname;

this.lastname = lastname;

this.age = age;

this.eyecolor = eyecolor;

this.changeName = changeName;

function changeName(name) {

this.lastname = name;

}

}

调用：

myMother.changeName("Doe");

### JavaScript 类

JavaScript 是面向对象的语言，但 JavaScript 不使用类。

在 JavaScript 中，不会创建类，也不会通过类来创建对象（就像在其他面向对象的语言中那样）。

JavaScript 基于 prototype，而不是基于类的。

### 遍历对象的属性

使用for...in 循环

实例：遍历对象属性

<p>点击下面的按钮，循环遍历对象 "person" 的属性。</p>

<button onclick="myFunction()">点击这里</button>

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

<script>

function myFunction() {

var x;

var txt = "";

var person = { fname: "Bill", lname: "Gates", age: 56 };

for (x in person) {

txt = txt + person[x];

}

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

}

</script>

## prototype（原型对象）

所有的 JavaScript 对象都会从一个 prototype（原型对象）中继承属性和方法。

使用对象构造器（constructor）：

<h2>JavaScript 对象</h2>

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

<script>

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

var myFather = new Person("John", "Doe", 50, "blue");

var myMother = new Person("Sally", "Rally", 48, "green");

document.getElementById("demo").innerHTML =

"我的父亲年龄是 " + myFather.age + "。我的母亲年龄是 " + myMother.age;

//我的父亲年龄是 50。我的母亲年龄是 48

</script>

要添加一个新的属性需要在在构造器函数中添加。

### prototype 继承

所有的 JavaScript 对象都会从一个 prototype（原型对象）中继承属性和方法：

* Date 对象从 Date.prototype 继承。
* Array 对象从 Array.prototype 继承。
* Person 对象从 Person.prototype 继承。

所有 JavaScript 中的对象都是位于原型链顶端的 Object 的实例。

JavaScript 对象有一个指向一个原型对象的链。当试图访问一个对象的属性时，它不仅仅在该对象上搜寻，还会搜寻该对象的原型，以及该对象的原型的原型，依次层层向上搜索，直到找到一个名字匹配的属性或到达原型链的末尾。

Date 对象, Array 对象, 以及 Person 对象从 Object.prototype 继承。

### 添加属性和方法

有的时候我们想要在所有已经存在的对象添加新的属性或方法。另外，有时候我们想要在对象的构造函数中添加属性或方法。

## Number 对象

JavaScript 只有一种数字类型。可以使用也可以不使用小数点来书写数字。

## 数字

JavaScript 数字可以使用也可以不使用小数点来书写：

var pi = 3.14; // 使用小数点

var x = 34; // 不使用小数点

极大或极小的数字可通过科学（指数）计数法来写：

var y = 123e5; // 12300000

var z = 123e-5; // 0.00123

### 所有 JavaScript 数字均为 64 位

JavaScript 不是类型语言。与许多其他编程语言不同，JavaScript 不定义不同类型的数字，比如整数、短、长、浮点等等。

在JavaScript中，数字不分为整数类型和浮点型类型，所有的数字都是由 浮点型类型。JavaScript采用IEEE754标准定义的64位浮点格式表示数字，它能表示最大值为±1.7976931348623157 x 10308，最小值为±5 x 10 -324

### 精度

整数（不使用小数点或指数计数法）最多为 15 位。

<p>整数最多为 15 位。</p>

<button onclick="myFunction()">点我</button>

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

<script>

function myFunction() {

var x = 999999999999999; //15位，将返回

var y = 9999999999999999; //超过15位，将返回10000000000000000

document.getElementById("demo").innerHTML = x + "<br>" + y;

}

</script>

小数的最大位数是 17，但是浮点运算并不总是 100% 准确：

<script>

var x;

document.write("<p>仅显示17位: ");

x = 12345678901234567890; //将返回12345678901234567000

document.write(x + "</p>");

document.write("<p>0.2 + 0.1 = ");

x = 0.2 + 0.1; //将返回0.30000000000000004

document.write(x + "</p>");

document.write("<p>可以通过乘以10或除以10来计算: ");

x = (0.2 \* 10 + 0.1 \* 10) / 10; //将返回0.3

document.write(x + "</p>");

</script>

### 八进制和十六进制

如果前缀为 0，则 JavaScript 会把数值常量解释为八进制数，如果前缀为 0 和 "x"，则解释为十六进制数。

<script>

var y = 0377; //返回255

var z = 0xFF; //返回255

document.write(y + "<br>");

document.write(z + "<br>");

</script>

注：绝不要在数字前面写零，除非您需要进行八进制转换。

默认情况下，JavaScript 数字为十进制显示。但是你可以使用 toString() 方法 输出16进制、8进制、2进制。

<script>

var myNumber = 128;

document.write(myNumber + ' 十进制<br>'); //返回128

document.write(myNumber.toString(16) + ' 十六进制<br>'); //返回80

document.write(myNumber.toString(8) + ' 八进制<br>'); //返回200

document.write(myNumber.toString(2) + ' 二进制<br>'); //返回10000000

</script>

### 无穷大（Infinity）

当数字运算结果超过了JavaScript所能表示的数字上限（溢出），结果为一个特殊的无穷大（infinity）值，在JavaScript中以Infinity表示。同样地，当负数的值超过了JavaScript所能表示的负数范围，结果为负无穷大，在JavaScript中以-Infinity表示。无穷大值的行为特性和我们所期望的是一致的：基于它们的加、减、乘和除运算结果还是无穷大（当然还保留它们的正负号）。

<script>

myNumber = 2;

while (myNumber != Infinity) {

myNumber = myNumber \* myNumber;

document.write(myNumber + '<BR>');

}

//返回：

//4

//16

//256

//65536

//4294967296

//18446744073709552000

//3.402823669209385e+38

//1.157920892373162e+77

//1.3407807929942597e+154

//Infinity

</script>

除以0也产生了无限:

<script>

var x = 2 / 0;

var y = -2 / 0;

document.write(x + "<br>"); //Infinity

document.write(y + "<br>"); //-Infinity

</script>

### NaN - 非数字值

NaN 属性是代表非数字值的特殊值。该属性用于指示某个值不是数字。可以把 Number 对象设置为该值，来指示其不是数字值。

你可以使用 isNaN() 全局函数来判断一个值是否是 NaN 值。

<p>一个数字除以一个字符串结果不是一个数字</p>

<p>一个数字除以一个字符串数字结果是一个数字</p>

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

<script>

var x = 1000 / "Apple"; // isNaN(x)=true

var y = 1000 / "1000"; // isNaN(x)=false

document.getElementById("demo").innerHTML = isNaN(x) + "<br>" + isNaN(y);

</scrip

除以0是无穷大，无穷大是一个数字:

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

<script>

var y = 1000 / 0; //isNaN(y)=false

document.getElementById("demo").innerHTML = isNaN(y);

</script>

### 数字可以是数字或者对象

数字可以私有数据进行初始化，就像 x = 123;

JavaScript 数字对象初始化数据， var y = new Number(123);

<script>

var x = 123; // x 是一个数字

var y = new Number(123); // y 是一个对象

var txt = typeof (x) + " " + typeof (y);

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

</script>

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

<script>

var x = 123; // x 是一个数字

var y = new Number(123); // y 是一个对象

document.getElementById("demo").innerHTML = x === y; //false

</script>

### 数字属性

* MAX\_VALUE
* MIN\_VALUE
* NEGATIVE\_INFINITY
* POSITIVE\_INFINITY
* NaN
* prototype
* constructor

### 数字方法

* toExponential()
* toFixed()
* toPrecision()
* toString()
* valueOf()

## 字符串（String） 对象

String 对象用于处理已有的字符块。

## JavaScript 字符串

一个字符串用于存储一系列字符就像 "John Doe".一个字符串可以使用单引号或双引号。

你使用位置（索引）可以访问字符串中任何的字符：

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

<script>

var carname="Volvo XC60";

var character = carname[7];

document.getElementById('demo').innerHTML = character //返回C

</script>

字符串（String）使用长度属性**length**来计算字符串的长度：

### 在字符串中查找字符串

字符串使用 indexOf() 来定位字符串中某一个指定的字符首次出现的位置：

<p id="p1">Click the button to locate where "locate" first occurs.</p>

<p id="p2">0</p>

<button onclick="myFunction()">点我</button>

<script>

function myFunction() {

var str = document.getElementById("p1").innerHTML;

var n = str.indexOf("locate");

document.getElementById("p2").innerHTML = n + 1; //21

}

</script>

如果没找到对应的字符函数返回-1。

lastIndexOf() 方法在字符串末尾开始查找字符串出现的位置。

### 内容匹配

**match()**函数用来查找字符串中特定的字符，并且如果找到的话，则返回这个字符。

<script>

var str = "Hello world!";

document.write(str.match("world") + "<br>"); //world

document.write(str.match("World") + "<br>"); //null

document.write(str.match("world!")); //world!

</script>

### 替换内容

**replace()** 方法在字符串中用某些字符替换另一些字符。

<p>替换 "Microsoft" 为 "Runoob" :</p>

<button onclick="myFunction()">点我</button>

<p id="demo">请访问 Microsoft!</p>

<script>

function myFunction() {

var str = document.getElementById("demo").innerHTML;

var txt = str.replace("Microsoft", "Runoob");

document.getElementById("demo").innerHTML = txt; //请访问 Runoob!

}

</script>

### 字符串大小写转换

字符串大小写转换使用函数 **toUpperCase()**/**toLowerCase()**:

<script>

var txt = "Hello World!";

document.write("<p>" + txt.toUpperCase() + "</p>"); //HELLO WORLD!

document.write("<p>" + txt.toLowerCase() + "</p>"); //hello world!

document.write("<p>" + txt + "</p>");

</script>

<p>该方法返回一个新的字符串，源字符串没有被改变。</p>

### 字符串转为数组

字符串使用**split()**函数转为数组:

<p id="demo">单击按钮显示数组。</p>

<button onclick="myFunction()">点我</button>

<script>

function myFunction() {

var str = "a,b,c,d,e,f";

var n = str.split(",");

document.getElementById("demo").innerHTML = n[0]; //返回a

}

</script>

### 特殊字符

Javascript 中可以使用反斜线（\）插入特殊符号，如：撇号,引号等其他特殊符号。

下表列出其他特殊字符，可以使用反斜线转义特殊字符：

|  |  |
| --- | --- |
| **代码** | **输出** |
| \' | 单引号 |
| \" | 双引号 |
| \\ | 斜杆 |
| \n | 换行 |
| \r | 回车 |
| \t | tab |
| \b | 空格 |
| \f | 换页 |

### 字符串属性和方法

属性:

* length
* prototype
* constructor

方法:

* charAt()
* charCodeAt()
* concat()
* fromCharCode()
* indexOf()
* lastIndexOf()
* match()
* replace()
* search()
* slice()
* split()
* substr()
* substring()
* toLowerCase()
* toUpperCase()
* valueOf()

## Date（日期） 对象

日期对象用于处理日期和时间。

#### 实例：使用 Date() 方法获得当日的日期。

<script>

var d = new Date();

document.write(d);

</script>

#### 实例：获取年份

<p id="demo">点击按钮获取今年的年份。</p>

<button onclick="myFunction()">点我</button>

<script>

function myFunction() {

var d = new Date();

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

x.innerHTML = d.getFullYear();

}

</script>

#### 实例：返回从 1970 年 1 月 1 日至今的毫秒数

<p id="demo">单击按钮显示1970年1月1号至今的毫秒数。</p>

<button onclick="myFunction()">点我</button>

<script>

function myFunction() {

var d = new Date();

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

x.innerHTML = d.getTime();

}

</script>

#### 实例：设置具体的日期

<p id="demo">单击按钮显示修改后的年月日。</p>

<button onclick="myFunction()">点我</button>

<script>

function myFunction() {

var d = new Date();

d.setFullYear(2020, 10, 3); //设置日期为2020年11月3日

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

x.innerHTML = d;

}

</script>

<p>记住 JavaScript 月数是从0至11。10是11月。</p>

#### 实例：将当日的日期（根据 UTC）转换为字符串

<p id="demo">点击按钮把 utc 日期和时间转换成字符串。</p>

<button onclick="myFunction()">点我</button>

<script>

function myFunction() {

var d = new Date();

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

x.innerHTML = d.toUTCString();

}

</script>

#### 实例：显示星期

<p id="demo">单击按钮显示今天周几</p>

<button onclick="myFunction()">点我</button>

<script>

function myFunction() {

var d = new Date();

var weekday = new Array(7);

weekday[0] = "周日";

weekday[1] = "周一";

weekday[2] = "周二";

weekday[3] = "周三";

weekday[4] = "周四";

weekday[5] = "周五";

weekday[6] = "周六";

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

x.innerHTML = weekday[d.getDay()];

}

</script>

#### 实例：显示一个钟表

<body onload="startTime()">

<script>

function startTime() {

var today = new Date();

var h = today.getHours();

var m = today.getMinutes();

var s = today.getSeconds();// 在小于10的数字前加一个‘0’

m = checkTime(m);

s = checkTime(s);

document.getElementById('txt').innerHTML = h + ":" + m + ":" + s;

t = setTimeout(function () { startTime() }, 500);

}

function checkTime(i) {

if (i < 10) {

i = "0" + i;

}

return i;

}

</script>

<div id="txt"></div>

</body>

### 创建日期

Date 对象用于处理日期和时间。 可以通过 new 关键词来定义 Date 对象。以下代码定义了名为 myDate 的 Date 对象：

有四种方式初始化日期:

new Date() // 当前日期和时间

new Date(milliseconds) //返回从 1970 年 1 月 1 日至今的毫秒数

new Date(dateString)

new Date(year, month, day, hours, minutes, seconds, milliseconds)

上面的参数大多数都是可选的，在不指定的情况下，默认参数是0。

实例化一个日期的一些例子：

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

<script>

var today = new Date()

var d1 = new Date("October 13, 1975 11:13:00")

var d2 = new Date(79, 5, 24)

var d3 = new Date(79, 5, 24, 11, 33, 0)

document.getElementById('demo').innerHTML =

d1 + '<br/>' + //Mon Oct 13 1975 11:13:00 GMT-0400 (北美东部夏令时间)

d2 + '<br/>' + //Sun Jun 24 1979 00:00:00 GMT-0400 (北美东部夏令时间)

d3 + '<br/>'; //Sun Jun 24 1979 11:33:00 GMT-0400 (北美东部夏令时间)

</script>

### 设置日期

通过使用针对日期对象的方法，我们可以很容易地对日期进行操作。

为日期对象设置了一个特定的日期 (2010 年 1 月 14 日)：

var myDate = new Date();

myDate.setFullYear(2010, 0, 14);

将日期对象设置为 5 天后的日期：

var myDate = new Date();

myDate.setDate(myDate.getDate() + 5);

**注意:** 如果增加天数会改变月份或者年份，那么日期对象会自动完成这种转换。

### 两个日期比较

#### 实例：比较两个日期

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

<script>

var x = new Date();

x.setFullYear(2100, 0, 14);

var today = new Date();

if (x > today) {

alert("今天是2100年1月14日之前");

}

else {

alert("今天是2100年1月14日之后");

}

</script>

#### 实例：****把日期格式化为指定格式****

<p id="demo1"></p>

<p id="demo2"></p>

<p id="demo3"></p>

<script>

Date.prototype.format = function (fmt) {

var o = {

"M+": this.getMonth() + 1, //月份

"d+": this.getDate(), //日

"h+": this.getHours(), //小时

"m+": this.getMinutes(), //分

"s+": this.getSeconds(), //秒

"q+": Math.floor((this.getMonth() + 3) / 3), //季度

"S": this.getMilliseconds() //毫秒

};

if (/(y+)/.test(fmt)) {

fmt = fmt.replace(RegExp.$1, (this.getFullYear() + "").substr(4 - RegExp.$1.length));

}

for (var k in o) {

if (new RegExp("(" + k + ")").test(fmt)) {

fmt = fmt.replace(

RegExp.$1, (RegExp.$1.length == 1) ? (o[k]) : (("00" + o[k]).substr(("" + o[k]).length)));

}

}

return fmt;

}

document.getElementById("demo1").innerHTML = new Date(79, 5, 24, 11, 33, 0).format("MM月dd日");

var now = new Date();

var nowStr = now.format("yyyy-MM-dd hh:mm:ss");

document.getElementById("demo2").innerHTML = new Date().format("yyyy年MM月dd日"); //06月24日

var nowStr = now.format("yyyy-MM-dd hh:mm:ss"); //2020年02月23日

document.getElementById("demo3").innerHTML = new Date().format("yyyy年MM月dd日hh小时mm分ss秒"); //2020年02月23日23小时14分55秒

</script>

其他格式实例：

<script>

alert(new Date().format("yyyy年MM月dd日"));

alert(new Date().format("MM/dd/yyyy"));

alert(new Date().format("yyyyMMdd"));

alert(new Date().format("yyyy-MM-dd hh:mm:ss"));

</script>

## Array（数组） 对象

数组对象的作用是：使用单独的变量名来存储一系列的值。

数组对象是使用单独的变量名来存储一系列的值。

### 创建一个数组

创建一个数组，有三种方法：

1: 常规方式:

var myCars = new Array();

myCars[0] = "Saab";

myCars[1] = "Volvo";

myCars[2] = "BMW";

2: 简洁方式:

var myCars = new Array("Saab", "Volvo", "BMW");

3: 字面:

var myCars = ["Saab", "Volvo", "BMW"];

### 访问数组

通过指定数组名以及索引号码，你可以访问某个特定的元素。

访问myCars数组的第一个值：

var name = myCars[0];

修改了数组 myCars 的第一个元素

myCars[0] = "Opel";

注：[0] 是数组的第一个元素。[1] 是数组的第二个元素。

### 在一个数组中你可以有不同的对象

所有的JavaScript变量都是对象。数组元素是对象。函数是对象。

因此，你可以在数组中有不同的变量类型。

可以在一个数组中包含对象元素、函数、数组：

var myArray = new Array();

myArray[0] = Date.now;

myArray[1] = myFunction;

myArray[2] = myCars;

### 数组方法和属性

var x = myCars.length // myCars 中元素的数量

var y = myCars.indexOf("Volvo") // "Volvo" 值的索引值

### 创建新方法

原型是JavaScript全局构造函数。它可以构建新Javascript对象的属性和方法。

<p id="demo">单击按钮创建一个数组,调用 ucase（）方法, 并显示结果。</p>

<button onclick="myFunction()">点我</button>

<script>

Array.prototype.myUcase = function () {

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

this[i] = this[i].toUpperCase();

}

}

function myFunction() {

var fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.myUcase();

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

x.innerHTML = fruits;

}

</script>

#### 实例：使用数组对象结合 select 对象实现二级级联菜单。

<select id="a1" onchange="addOption()"></select>

<select id="a2"></select>

<script>

var city = new Array;

city['江苏'] = ['南京', '连云港', '苏州', '镇江'];

city['山东'] = ['青岛', '烟台', '济南'];

function allCity() {

var select1 = document.getElementById("a1");

for (var i in city) { //这里注意遍历数组的写法

select1.add(new Option(i, i), null);

}

addOption(); // 初始化选项

}

function addOption() {

var select2 = document.getElementById("a2");

var select1 = document.getElementById("a1").value;

select2.length = 0; //每次都先清空一下市级菜单

if (select1 != '请选择省份') {

for (var i in city[select1]) {

select2.add(new Option(city[select1][i], city[select1][i]), null);

}

} else if (sheng == '请选择省份') {

select2.length = 0;

select2.add(new Option("请选择城市", "请选择城市"), null);

}

}

window.onload = allCity();

</script>

遍历一维数组: 依次访问数组中每个元素，对每个元素执行相同的操作

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

arr[i]//当前正在遍历的元素

}

遍历关联数组: for in 循环

for (var key in hash) {

key//只是元素的下标名

hash[key]//当前元素值

}

for of 循环

const ary = ['a', 'b', 'c'];

var result = '';

for (let value of ary) {

result = result + value + '<br/>';

}

document.getElementById('demo').innerHTML = result; // a b c

遍历二维数组: 外层循环遍历行，内层循环遍历列

for (var r = 0; r < arr.length; r++) {

for (var c = 0; c < arr[r].length; c++) {

arr[r][c]//当前元素

}

}

Js还为数组对象内置了多种迭代方法, 处理数组时极为方便

<script>

const ary = ['a', 'b', 'c'];

/\*\*

\* forEach(function)

\*

\* 该方法接受一个函数作为参数, 该函数拥有两个参数, 分别为数组的值、键

\* 该方法没有返回值

\*/

ary.forEach((value, key) => {

console.log(key, value); // 0 a 1 b 2 c

})

/\*\*

\* map(function)

\*

\* 该方法接受一个函数作为参数, 该函数拥有两个参数, 分别为数组的值、键

\* 可以在传入的函数中返回相应的值,你每次迭代时返回的值会被map方法组装成一个新数组作为返回值返回

\*/

var result = ary.map((value, key) => {

return value + '\_runoob.com';

})

console.log(result); // ['a\_runoob.com', 'b\_runoob.com', 'c\_runoob.com']

</script>

## Boolean（布尔） 对象

Boolean（布尔）对象用于将非布尔值转换为布尔值（true 或者 false）。

### 创建 Boolean 对象

Boolean 对象代表两个值:"true" 或者 "false"

var myBoolean=new Boolean();

如果布尔对象无初始值或者其值为:

* 0
* -0
* null
* ""
* false
* undefined
* NaN

那么对象的值为 false。否则，其值为 true（即使当变量值为字符串 "false" 时）！

**Math（算数） 对象**

Math（算数）对象的作用是：执行常见的算数任务。

#### 实例：返回最接近的整数

<p id="demo">单击按钮舍入与“2.5”最接近的整数</p>

<button onclick="myFunction()">点我</button>

<script>

function myFunction() {

document.getElementById("demo").innerHTML = Math.round(2.5); //返回值为3

}

</script>

#### 实例：返回 0 到 1 之间的随机数

<p id="demo">点击按钮显示一个随机数</p>

<button onclick="myFunction()">点我</button>

<script>

function myFunction() {

document.getElementById("demo").innerHTML = Math.random();

}

</script>

#### 实例：返回最大值

<p id="demo">单击按钮返回5到10之间的最大值。</p>

<button onclick="myFunction()">点我</button>

<script>

function myFunction() {

document.getElementById("demo").innerHTML = Math.max(5, 10);

}

</script>

#### 实例：返回最小的值

<p id="demo">单击按钮返回5到10之间最小的值。</p>

<button onclick="myFunction()">点我</button>

<script>

function myFunction() {

document.getElementById("demo").innerHTML = Math.min(5, 10);

}

</script>

### Math 对象

Math（算数）对象的作用是：执行普通的算数任务。

Math 对象提供多种算数值类型和函数。无需在使用这个对象之前对它进行定义。

**使用Math的属性/方法的语法：**

var x = Math.PI;

var y = Math.sqrt(16);

### 算数值

JavaScript 提供 8 种可被 Math 对象访问的算数值：

Math.E

Math.PI

Math.SQRT2

Math.SQRT1\_2

Math.LN2

Math.LN10

Math.LOG2E

Math.LOG10E

### 算数方法

除了可被 Math 对象访问的算数值以外，还有几个函数（方法）可以使用。

round 方法对一个数进行四舍五入：

document.write(Math.round(4.7)); //5

random() 方法来返回一个介于 0 和 1 之间的随机数

document.write(Math.random());

使用了 Math 对象的 floor() 方法和 random() 来返回一个介于 0 和 11 之间的随机数：

document.write(Math.floor(Math.random() \* 11));

根据上下限生成随机数：

var rand = (min, max) => Math.round(Math.random() \* (max - min)) + min

//Max为最大值，Min为最小值

根据概率随机生成 bool 值：

function randBool(percent = 0.5) {

//percent为概率，默认0.5（50%）。

if (Math.random() < percent)//如果随机数小于概率值，返回true，否则返回false。

return true;

else

return false;

}

随机生成指定字符：

function randChar(length, characters = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789") {

//length为所需长度，characters为所包含的所有字符，默认为字母+数字。

characters = characters.split("");//分割字符。

result = "";//返回的结果。

while (result.length < length) result += characters[Math.round(Math.random() \* characters.length) - 1];

return result;

}

随机生成字符的另一种玩法：

function randCharAnother(length, rangeMin = 0x80, rangeMax = 0x7FF) {

//length长度，rangeMin为最小Unicode码，rangeMax为最大Unicode码。

result = "";

while (result.length < length) result += String.fromCharCode(Math.round(Math.random() \* (rangeMax - rangeMin)) - rangeMin);

return result;

}

随机从数组中取出一个东东：

Array.prototype.pick = function () {

//不能为 ()=>{/\*函数\*/}，否则this会指向Window。

return this.length ? Math.round(Math.random() \* (this.length - 1)) : undefined;//如果长度为0，返回undefined。

}

## RegExp 对象

正则表达式描述了字符的模式对象。

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。

简单的模式可以是一个单独的字符。

更复杂的模式包括了更多的字符，并可用于解析、格式检查、替换等等。

您可以规定字符串中的检索位置，以及要检索的字符类型，等等。

语法：

var patt=new RegExp(pattern,modifiers);

或更简单的方法

var patt=/pattern/modifiers;

* 模式描述了一个表达式模型。
* 修饰符(modifiers)描述了检索是否是全局，区分大小写等。

***注意：***当使用构造函数创造正则对象时，需要常规的字符转义规则（在前面加反斜杠 \）。

### RegExp 修饰符

修饰符用于执行不区分大小写和全文的搜索。

**i** - 修饰符是用来执行不区分大小写的匹配。

**g** - 修饰符是用于执行全文的搜索（而不是在找到第一个就停止查找,而是找到所有的匹配）。

#### 实例：不区分大小写查找

var str = "Visit RUnoob";

var patt1 = /runoob/i;

document.write(str.match(patt1)); //RUnoob

#### 实例：全文查找 "is"

var str = "Is this all there is?";

var patt1 = /is/g;

document.write(str.match(patt1)); //is,is

#### 实例：全文查找和不区分大小写搜索 "is"

var str = "Is this all there is?";

var patt1 = /is/gi;

document.write(str.match(patt1)); //Is,is,is

### test()

test()方法搜索字符串指定的值，根据结果并返回真或假。

var patt1 = new RegExp("e");

document.write(patt1.test("The best things in life are free")); //true

当使用构造函数创造正则对象时，需要常规的字符转义规则（在前面加反斜杠 \）

var str = 'runoob';

var patt1 = new RegExp('\\w', 'g'); // 有转义作为正则表达式处理

var patt2 = new RegExp('\w', 'g'); // 无转义作为字符串处理

var patt3 = /\w+/g; // 与 patt1 效果相同

document.write(patt1.test(str)) //输出 true

document.write("<br>")

document.write(patt2.test(str)) //输出 false

document.write("<br>")

document.write(patt3.test(str)) //输出 true

### exec()

exec() 方法检索字符串中的指定值。返回值是被找到的值。如果没有发现匹配，则返回 null。

var patt1 = new RegExp("e");

document.write(patt1.exec("The best things in life are free")); //e

## Window - 浏览器对象模型

浏览器对象模型 (BOM) 使 JavaScript 有能力与浏览器"对话"。

### 浏览器对象模型 (BOM)

浏览器对象模型（**B**rowser **O**bject **M**odel (BOM)）尚无正式标准。

由于现代浏览器已经（几乎）实现了 JavaScript 交互性方面的相同方法和属性，因此常被认为是 BOM 的方法和属性。

### Window 对象

所有浏览器都支持 window 对象。它表示浏览器窗口。

所有 JavaScript 全局对象、函数以及变量均自动成为 window 对象的成员。

全局变量是 window 对象的属性。

全局函数是 window 对象的方法。

HTML DOM 的 document 也是 window 对象的属性之一

window.document.getElementById("header");

等效于：

document.getElementById("header");

### Window 尺寸

有三种方法能够确定浏览器窗口的尺寸。

对于Internet Explorer、Chrome、Firefox、Opera 以及 Safari：

* window.innerHeight - 浏览器窗口的内部高度(包括滚动条)
* window.innerWidth - 浏览器窗口的内部宽度(包括滚动条)

对于 Internet Explorer 8、7、6、5：

* document.documentElement.clientHeight
* document.documentElement.clientWidth

或者

* document.body.clientHeight
* document.body.clientWidth

#### 实例：实用的 JavaScript 方案（涵盖所有浏览器）：

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

<script>

var w = window.innerWidth

|| document.documentElement.clientWidth

|| document.body.clientWidth;

var h = window.innerHeight

|| document.documentElement.clientHeight

|| document.body.clientHeight;

x = document.getElementById("demo");

x.innerHTML = "浏览器window宽度: " + w + ", 高度: " + h + "。" //返回浏览器宽度和高度

</script>

### 其他 Window 方法

一些其他方法：

* window.open() - 打开新窗口
* window.close() - 关闭当前窗口
* window.moveTo() - 移动当前窗口
* window.resizeTo() - 调整当前窗口的尺寸

#### 实例：open()和close()

<p>

<a href="javascript: popuponclick()">Open Popup Window</a>

</p>

<p>

<a href="javascript: closepopup()">Close the Popup Window</a>

</p>

<script>

function popuponclick() {

my\_window = window.open("",

"mywindow", "status=1,width=350,height=150"); //使用window.open创建一个window对象,并打开该对象

my\_window.document.write('<h1>The Popup Window</h1>'); //向新window对象写入内容

}

function closepopup() {

if (false == my\_window.closed) { //判断新创建的window对象是否关闭

my\_window.close(); //如果没有关闭，就关闭窗口

}

else {

alert('Window already closed!'); //如果已经关闭新的window，就弹出警告栏

}

}

</script>

定义全局变量与在 window 对象上直接定义属性差别。

1、全局变量不能通过 delete 操作符删除；而 window 属性上定义的变量可以通过 delete 删除

<script type="text/javascript">

var num = 123;

window.str = "string";

delete num;

delete str;

//console.log(num); //123

console.log(str); //会报错，因为str is not defined

//全局变量不能通过 delete 删除，因为通过 var 定义全局变量会有一个名为 [Configurable] 的属性，默认值为 false，所以这样定义的属性不可以通过 delete 操作符删除

</script>

2、访问未声明的变量会抛出错误，但是通过查询 window 对象，可以知道某个可能未声明的变量是否存在。

<script type="text/javascript">

var newValue = oldValue; // 报错：oldValue is not defined

var newValue = window.oldValue; // 不会报错

console.log(newValue); // undefined

</script>

3、有些自执行函数里面的变量，想要外部也访问到的话，在 window 对象上直接定义属性。

## Window Screen

window.screen 对象包含有关用户屏幕的信息。

**window.screen**对象在编写时可以不使用 window 这个前缀。

一些属性：

* screen.availWidth - 可用的屏幕宽度
* screen.availHeight - 可用的屏幕高度

### Screen 可用宽度

screen.availWidth 属性返回访问者屏幕的宽度，以像素计，减去界面特性，比如窗口任务栏。

#### 实例：返回您的屏幕的可用宽度：

<script>

document.write("可用宽度: " + screen.availWidth); //可用宽度: 1366

</script>

### Screen 可用高度

screen.availHeight 属性返回访问者屏幕的高度，以像素计，减去界面特性，比如窗口任务栏。

#### 实例：返回您的屏幕的可用高度：

<script>

document.write("可用高度: " + screen.availHeight); //可用高度: 728

</script>

#### 实例：所有screen 属性应用

<h3>你的屏幕:</h3>

<script>

document.write("总宽度/高度: ");

document.write(screen.width + "\*" + screen.height);

document.write("<br>");

document.write("可用宽度/高度: ");

document.write(screen.availWidth + "\*" + screen.availHeight);

document.write("<br>");

document.write("色彩深度: ");

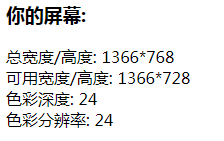
document.write(screen.colorDepth);

document.write("<br>");

document.write("色彩分辨率: ");

document.write(screen.pixelDepth);

</script>



## Window Location

window.location 对象用于获得当前页面的地址 (URL)，并把浏览器重定向到新的页面。

### Window Location

**window.location** 对象在编写时可不使用 window 这个前缀。 一些例子：

一些实例:

* location.hostname 返回 web 主机的域名
* location.pathname 返回当前页面的路径和文件名
* location.port 返回 web 主机的端口 （80 或 443）
* location.protocol 返回所使用的 web 协议（http: 或 https:）

#### 实例：获取当前web页面地址信息

<script>

document.write('所使用的 web 协议：' + location.protocol + '<br/>');

document.write('当前web主机的域名：' + location.hostname + '<br/>');

document.write('当前页面的路径和文件名：' + location.pathname + '<br/>');

document.write('web 主机的端口：' + location.port + '<br/>');

</script>

### Window Location Href

location.href 属性返回当前页面的 URL。

#### 实例：返回（当前页面的）整个 URL

<script>

document.write(location.href); //http://localhost:62132/test/

</script>

### Window Location Pathname

location.pathname 属性返回 URL 的路径名。

#### 实例：返回当前 URL 的路径名

<script>

document.write(location.pathname); ///test/

</script>

### Window Location Assign

location.assign() 方法加载新的文档。

#### 实例：加载一个新的文档

<input type="button" value="加载新文档" onclick="newDoc()">

<script>

function newDoc() {

window.location.assign("http://localhost:62132/")

}

</script>

window.location.assign(url) ： 加载 URL 指定的新的 HTML 文档。 就相当于一个链接，跳转到指定的url，当前页面会转为新页面内容，可以点击后退返回上一个页面。

window.location.replace(url) ： 通过加载 URL 指定的文档来替换当前文档 ，这个方法是替换当前窗口页面，前后两个页面共用一个窗口，所以是没有后退返回上一页的

## Window History

window.history 对象包含浏览器的历史。

**window.history**对象在编写时可不使用 window 这个前缀。

为了保护用户隐私，对 JavaScript 访问该对象的方法做出了限制。

一些方法：

* history.back() - 与在浏览器点击后退按钮相同
* history.forward() - 与在浏览器中点击向前按钮相同

### history.back()

history.back() 方法加载历史列表中的前一个 URL。这与在浏览器中点击后退按钮是相同的：

#### 实例：在页面上创建后退按钮

<input type="button" value="Back" onclick="goBack()">

<script>

function goBack() {

window.history.back()

}

</script>

### history.forward()

history forward() 方法加载历史列表中的下一个 URL。这与在浏览器中点击前进按钮是相同的：

<input type="button" value="Forward" onclick="goForward()">

<script>

function goForward() {

window.history.forward()

}

</script>

### history.go()

除此之外可以用 **history.go()** 这个方法来实现向前，后退的功能。

<input type="button" value="Forward" onclick="b()">

<script>

function a() {

history.go(1); // go() 里面的参数表示跳转页面的个数 例如 history.go(1) 表示前进一个页面

}

function b() {

history.go(-1); // go() 里面的参数表示跳转页面的个数 例如 history.go(-1) 表示后退一个页面

}

</script>

**history.go()** 这个方法来实现刷新的功能。

function a() {

history.go(0); // go() 里面的参数为0,表示刷新页面

}

## Window Navigator

window.navigator 对象包含有关访问者浏览器的信息。

#### 实例：获取浏览器信息

<div id="example"></div>

<script>

txt = "<p>浏览器代号: " + navigator.appCodeName + "</p>";

txt += "<p>浏览器名称: " + navigator.appName + "</p>";

txt += "<p>浏览器版本: " + navigator.appVersion + "</p>";

txt += "<p>启用Cookies: " + navigator.cookieEnabled + "</p>";

txt += "<p>硬件平台: " + navigator.platform + "</p>";

txt += "<p>用户代理: " + navigator.userAgent + "</p>";

txt += "<p>用户代理语言: " + navigator.systemLanguage + "</p>";

document.getElementById("example").innerHTML = txt;

</script>

### 警告!!!

来自 navigator 对象的信息具有误导性，不应该被用于检测浏览器版本，这是因为：

* navigator 数据可被浏览器使用者更改
* 一些浏览器对测试站点会识别错误
* 浏览器无法报告晚于浏览器发布的新操作系统

## JavaScript 弹窗

在 JavaScript 中创建三种消息框：警告框、确认框、提示框。

### 警告框

警告框经常用于确保用户可以得到某些信息。当警告框出现后，用户需要点击确定按钮才能继续进行操作。

语法： window.alert("sometext");

#### 实例：弹出警告框

<input type="button" onclick="myFunction()" value="显示警告框">

<script>

function myFunction() {

alert("你好，我是一个警告框！");

}

</script>

### 确认框

确认框通常用于验证是否接受用户操作。当确认卡弹出时，用户可以点击 "确认" 或者 "取消" 来确定用户操作。当你点击 "确认", 确认框返回 true， 如果点击 "取消", 确认框返回 false。

语法： window.confirm("sometext");

#### 实例：确认框并获取确认框中交互结果

<p>点击按钮，显示确认框。</p>

<button onclick="myFunction()">点我</button>

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

<script>

function myFunction() {

var x;

var r = confirm("按下按钮!"); //获取确认框中按下的按钮

if (r == true) { //如果按下“确认”按钮

x = "你按下了\"确定\"按钮!";

}

else { //如果按下“取消”按钮

x = "你按下了\"取消\"按钮!";

}

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

}

</script>

### 提示框

提示框经常用于提示用户在进入页面前输入某个值。

当提示框出现后，用户需要输入某个值，然后点击确认或取消按钮才能继续操纵。

如果用户点击确认，那么返回值为输入的值。如果用户点击取消，那么返回值为 null。

语法： window.prompt("sometext","defaultvalue");

#### 实例：提示框

<p>点击按钮查看输入的对话框。</p>

<button onclick="myFunction()">点我</button>

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

<script>

function myFunction() {

var x;

var person = prompt("请输入你的名字", "Harry Potter"); //在输入提示框中显示的文字和输入框的默认值，并将提示框中的信息返回为变量person

if (person != null && person != "") { //判断是否点击取消，输入框是否为空

x = "你好 " + person + "! 今天感觉如何?";

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

}

}

</script>

### 换行

弹窗使用 反斜杠 + "n"(\n) 来设置换行。

<p>点击按钮在弹窗总使用换行。</p>

<button onclick="myFunction()">点我</button>

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

<script>

function myFunction() {

alert("Hello\nHow are you?");

}

</script>

## 计时事件

计时事件：JavaScript 一个设定的时间间隔之后来执行代码，，而不是在函数被调用后立即执行。

在 JavaScritp 中使用计时事件是很容易的，两个关键方法是:

* setInterval() - 间隔指定的毫秒数不停地执行指定的代码。
* setTimeout() - 在指定的毫秒数后执行指定代码。

**注意:** setInterval() 和 setTimeout() 是 HTML DOM Window对象的两个方法。

### setInterval() 方法

setInterval() 间隔指定的毫秒数不停地执行指定的代码。

语法：

window.setInterval("javascript function",milliseconds);

setInterval() 第一个参数是函数（function）。

第二个参数间隔的毫秒数

**注意:** 1000 毫秒是一秒。

#### 实例：定时弹出警告框

<p>单击按钮每3秒出现一个“Hello”警告框。</p>

<p>再次点击警告框，经过3秒出现新的警告框，这将一直执行 ...</p>

<button onclick="myFunction()">点我</button>

<script>

function myFunction() {

setInterval(function () { alert("Hello") }, 3000);

}

</script>

#### 实例：每秒更新当前时间

<p>在页面显示一个时钟</p>

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

<script>

var myVar = setInterval(function () { myTimer() }, 1000);

function myTimer() {

var d = new Date();

var t = d.toLocaleTimeString();

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

}

</script>

### 如何停止执行?

clearInterval() 方法用于停止 setInterval() 方法执行的函数代码。

语法： window.clearInterval(intervalVariable)

#### ****实例：停止执行****setInterval()

<p>页面上显示时钟：</p>

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

<button onclick="myStopFunction()">停止</button>

<script>

var myVar = setInterval(function () { myTimer() }, 1000); //将定时执行的代码赋值到变量

function myTimer() {

var d = new Date();

var t = d.toLocaleTimeString();

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

}

function myStopFunction() {

clearInterval(myVar); //停止执行指定变量

}

</script>

### setTimeout() 方法

在指定时间间隔后执行代码。

语法： myVar= window.setTimeout("javascript function", milliseconds);

setTimeout() 的第一个参数是含有 JavaScript 语句的字符串。

第二个参数指示从当前起多少毫秒后执行第一个参数。

提示：1000 毫秒等于一秒。

#### 实例：在指定时间间隔后执行代码。

<p>点击按钮，在等待 3 秒后弹出 "Hello"。</p>

<button onclick="myFunction()">点我</button>

<script>

function myFunction() {

setTimeout(function () { alert("Hello") }, 3000); //点击按钮后3秒弹出警告框

}

</script>

### 如何停止执行?

clearTimeout() 方法用于停止执行setTimeout()方法的函数代码。

语法： window.clearTimeout(timeoutVariable)

要使用clearTimeout() 方法, 你必须在创建超时方法中（setTimeout）使用全局变量。如果函数还未被执行，你可以使用 clearTimeout() 方法来停止执行函数代码。

实例：阻止

<p>点击第一个按钮等待3秒后出现"Hello"弹框。</p>

<p>点击第二个按钮来阻止第一个函数运行。（你必须在3秒之前点击它）。</p>

<button onclick="myFunction()">点我</button>

<button onclick="myStopFunction()">停止弹框</button>

<script>

var myVar;

function myFunction() {

myVar = setTimeout(function () { alert("Hello") }, 3000);

}

function myStopFunction() {

clearTimeout(myVar);

}

</script>

#### 实例：在输入框中显示秒数

<form>

<input type="button" value="显示文本时间!" onclick="timedText()" />

<input type="text" id="txt" />

</form>

<p>点击上面的按钮，输出的文本将告诉你2秒，4秒，6秒已经过去了。</p>

<script>

function timedText() {

var x = document.getElementById('txt');

var t1 = setTimeout(function () { x.value = "2 秒" }, 2000);

var t2 = setTimeout(function () { x.value = "4 秒" }, 4000);

var t3 = setTimeout(function () { x.value = "6 秒" }, 6000);

}

</script>

## Cookie

Cookie 用于存储 web 页面的用户信息。

Cookie 是一些数据, 存储于你电脑上的文本文件中。

当 web 服务器向浏览器发送 web 页面时，在连接关闭后，服务端不会记录用户的信息。

Cookie 的作用就是用于解决 "如何记录客户端的用户信息":

* 当用户访问 web 页面时，他的名字可以记录在 cookie 中。
* 在用户下一次访问该页面时，可以在 cookie 中读取用户访问记录。

Cookie 以名/值对形式存储，如下所示:

username=John Doe

当浏览器从服务器上请求 web 页面时， 属于该页面的 cookie 会被添加到该请求中。服务端通过这种方式来获取用户的信息。

### 使用 JavaScript 创建Cookie

JavaScript 可以使用 **document.cookie** 属性来创建 、读取、及删除 cookie。

创建 cookie 如下所示：

document.cookie = "username=John Doe";

还可以为 cookie 添加一个过期时间（以 UTC 或 GMT 时间）。默认情况下，cookie 在浏览器关闭时删除：

document.cookie = "username=John Doe; expires=Thu, 18 Dec 2043 12:00:00 GMT";

可以使用 path 参数告诉浏览器 cookie 的路径。默认情况下，cookie 属于当前页面

document.cookie = "username=John Doe; expires=Thu, 18 Dec 2043 12:00:00 GMT; path=/";

### 使用 JavaScript 读取 Cookie

在 JavaScript 中, 可以使用以下代码来读取 cookie

var x = document.cookie;

注：document.cookie 将以字符串的方式返回所有的 cookie，类型格式： cookie1=value; cookie2=value; cookie3=value;

### 使用 JavaScript 修改 Cookie

在 JavaScript 中，修改 cookie 类似于创建 cookie：

document.cookie="username=John Smith; expires=Thu, 18 Dec 2043 12:00:00 GMT; path=/";

旧的 cookie 将被覆盖。

### 使用 JavaScript 删除 Cookie

删除 cookie 非常简单。您只需要设置 expires 参数为以前的时间即可，如下所示，设置为 Thu, 01 Jan 1970 00:00:00 GMT:

document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 GMT";

注意，当您删除时不必指定 cookie 的值。

### Cookie 字符串

document.cookie 属性看起来像一个普通的文本字符串，其实它不是。

即使您在 document.cookie 中写入一个完整的 cookie 字符串, 当您重新读取该 cookie 信息时，cookie 信息是以名/值对的形式展示的。

如果您设置了新的 cookie，旧的 cookie 不会被覆盖。 新 cookie 将添加到 document.cookie 中，所以如果您重新读取document.cookie，您将获得如下所示的数据：

cookie1=value; cookie2=value;

#### 实例：Cookie 实例

首先，访问者访问 web 页面, 他将被要求填写自己的名字。该名字会存储在 cookie 中。

访问者下一次访问页面时，他会看到一个欢迎的消息。

在这个实例中我们会创建 3 个 JavaScript 函数:

1. 设置 cookie 值的函数
2. 获取 cookie 值的函数
3. 检测 cookie 值的函数

**设置 cookie 值的函数**

首先，我们创建一个函数用于存储访问者的名字：

function setCookie(cname, cvalue, exdays) {

var d = new Date();

d.setTime(d.getTime() + (exdays \* 24 \* 60 \* 60 \* 1000));

var expires = "expires=" + d.toGMTString();

document.cookie = cname + "=" + cvalue + "; " + expires;

}

**函数解析：**

以上的函数参数中，cookie 的名称为 cname，cookie 的值为 cvalue，并设置了 cookie 的过期时间 expires。

该函数设置了 cookie 名、cookie 值、cookie过期时间。

**获取 cookie 值的函数**

然后，我们创建一个函数用户返回指定 cookie 的值：

function getCookie(cname) {

var name = cname + "=";

var ca = document.cookie.split(';');

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

var c = ca[i].trim();

if (c.indexOf(name) == 0) return c.substring(name.length, c.length);

}

return "";

}

函数解析**：**

cookie 名的参数为 cname。

创建一个文本变量用于检索指定 cookie :cname + "="。

使用分号来分割 document.cookie 字符串，并将分割后的字符串数组赋值给 ca (ca = document.cookie.split(';'))。

循环 ca 数组 (i=0;i<ca.length;i++)，然后读取数组中的每个值，并去除前后空格 (c=ca[i].trim())。

如果找到 cookie(c.indexOf(name) == 0)，返回 cookie 的值 (c.substring(name.length,c.length)。

如果没有找到 cookie, 返回 ""。

**检测 cookie 值的函数**

最后，我们可以创建一个检测 cookie 是否创建的函数。

如果设置了 cookie，将显示一个问候信息。

如果没有设置 cookie，将会显示一个弹窗用于询问访问者的名字，并调用 setCookie 函数将访问者的名字存储 365 天：

function checkCookie() {

var username = getCookie("username");

if (username != "") {

alert("Welcome again " + username);

}

else {

username = prompt("Please enter your name:", "");

if (username != "" && username != null) {

setCookie("username", username, 365);

}

}

}

## JavaScript 库

JavaScript 库 - jQuery、Prototype、MooTools。

### JavaScript 框架（库）

JavaScript 高级程序设计（特别是对浏览器差异的复杂处理），通常很困难也很耗时。

为了应对这些调整，许多的 **JavaScript (helper)** 库应运而生。

这些 JavaScript 库常被称为 **JavaScript 框架**。

在本教程中，我们将了解到一些广受欢迎的 JavaScript 框架：

* jQuery
* Prototype
* MooTools

所有这些框架都提供针对常见 JavaScript 任务的函数，包括动画、DOM 操作以及 Ajax 处理。

### jQuery

jQuery 是目前最受欢迎的 JavaScript 框架。

它使用 CSS 选择器来访问和操作网页上的 HTML 元素（DOM 对象）。

jQuery 同时提供 companion UI（用户界面）和插件。

许多大公司在网站上使用 jQuery：Google、Microsoft、IBM、Netflix

### Prototype

**Prototype** 是一种库，提供用于执行常见 web 任务的简单 API。

**API** 是应用程序编程接口（Application Programming Interface）的缩写。它是包含属性和方法的库，用于操作 HTML DOM。

Prototype 通过提供类和继承，实现了对 JavaScript 的增强。

### MooTools

**MooTools** 也是一个框架，提供了可使常见的 JavaScript 编程更为简单的 API。

MooTools 也含有一些轻量级的效果和动画函数。

### 其他框架

下面是其他一些在上面未涉及的框架：

**YUI** - Yahoo! User Interface Framework，涵盖大量函数的大型库，从简单的 JavaScript 功能到完整的 internet widget。

**Ext JS** - 可定制的 widget，用于构建富因特网应用程序（rich Internet applications）。

**Dojo** - 用于 DOM 操作、事件、widget 等的工具包。

**script.aculo.us** - 开源的 JavaScript 框架，针对可视效果和界面行为。

**UIZE** - Widget、AJAX、DOM、模板等等。

### CDN -内容分发网络

您总是希望网页可以尽可能地快。您希望页面的容量尽可能地小，同时您希望浏览器尽可能多地进行缓存。

如果许多不同的网站使用相同的 JavaScript 框架，那么把框架库存放在一个通用的位置供每个网页分享就变得很有意义了。

CDN (Content Delivery Network) 解决了这个问题。CDN 是包含可分享代码库的服务器网络。

Google 为一系列 JavaScript 库提供了免费的 CDN，包括：

* jQuery
* Prototype
* MooTools
* Dojo
* Yahoo! YUI

但是由于 Google 在中国经常被GFW（防火长城，英文名称Great Firewall of China，简写为Great Firewall，缩写GFW）屏蔽，造成访问不稳定，所以建议使用百度静态资源公共库。

如需在您的网页中使用 JavaScript 框架库，只需在 <script> 标签中引用该库即可：

<script src="https://cdn.staticfile.org/jquery/3.4.0/jquery.min.js">

</script>

### 使用框架

在您决定为网页使用 JavaScript 框架之前，首先对框架进行测试是明智的。

JavaScript 框架很容易进行测试。您无需在计算机上安装它们，同时也没有安装程序。

通常您只需从网页中引用一个库文件。

## 测试 jQuery

测试 JavaScript 框架库 - jQuery

### 引用 jQuery

如需测试 JavaScript 库，您需要在网页中引用它。

为了引用某个库，请使用 <script> 标签，其 src 属性设置为库的 URL：

<script src="https://cdn.staticfile.org/jquery/1.8.3/jquery.min.js">

</script>

### jQuery 描述

主要的 jQuery 函数是 $() 函数（jQuery 函数）。如果您向该函数传递 DOM 对象，它会返回 jQuery 对象，带有向其添加的 jQuery 功能。

jQuery 允许您通过 CSS 选择器来选取元素。

在 JavaScript 中，您可以分配一个函数以处理窗口加载事件：

* JavaScript 方式：

<p id="h01"></p>

<script>

function myFunction() {

var obj = document.getElementById("h01");

obj.innerHTML = "Hello jQuery";

}

onload = myFunction;

</script>

* jQuery 方式：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>菜鸟教程(runoob.com)</title>

<script src="https://cdn.staticfile.org/jquery/1.8.3/jquery.min.js"></script>

<!-- 注意需要将脚本的引用先在head中标示，否则若放在页面末尾，jQuery将无法运行 -->

</head>

<body>

<h1 id="h01"></h1>

<script>

function myFunction() {

$("#h01").html("Hello jQuery")

}

$(document).ready(myFunction); //在文档加载时运行函数

</script>

</body>

</html>

上面代码的最后一行，HTML DOM 文档对象被传递到 jQuery ：$(document)。

当您向 jQuery 传递 DOM 对象时，jQuery 会返回以 HTML DOM 对象包装的 jQuery 对象。

jQuery 函数会返回新的 jQuery 对象，其中的 ready() 是一个方法。

由于在 JavaScript 中函数就是变量，因此可以把 myFunction 作为变量传递给 jQuery 的 ready 方法。

注： jQuery 返回 jQuery 对象，与已传递的 DOM 对象不同。jQuery 对象拥有的属性和方法，与 DOM 对象的不同。您不能在 jQuery 对象上使用 HTML DOM 的属性和方法。

实例：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>菜鸟教程(runoob.com)</title>

<script src="https://cdn.staticfile.org/prototype/1.7.3/prototype.min.js"></script>

<script>

function myFunction(){

$("h01").writeAttribute("style", "color:red").insert("Hello Prototype!");

//该处使用了链式语法：在同一对象上执行多个任务的便捷方法。该处同时设置标签内容和样式

}

Event.observe(window,"load",myFunction);

</script>

</head>

<body>

<h1 id="h01"></h1>

</body>

</html>

注：正如您在上面的例子中看到的，jQuery 允许链接（链式语法）。链接（Chaining）是一种在同一对象上执行多个任务的便捷方法。

## 测试 Prototype

### 引用 Prototype

如需测试 JavaScript 库，您需要在网页中引用它。

为了引用某个库，请使用 <script> 标签，其 src 属性设置为库的 URL：

<script src="https://cdn.staticfile.org/prototype/1.7.3/prototype.min.js">

</script>

### Prototype 描述

Prototype 提供的函数可使 HTML DOM 编程更容易。

与 jQuery 类似，Prototype 也有自己的 $() 函数。

$() 函数接受 HTML DOM 元素的 id 值（或 DOM 元素），并会向 DOM 对象添加新的功能。

与 jQuery 不同，Prototype 没有用以取代 window.onload() 的 ready() 方法。相反，Prototype 会向浏览器及 HTML DOM 添加扩展。

* JavaScript 方式：

function myFunction() {

var obj = document.getElementById("h01");

obj.innerHTML = "Hello Prototype";

}

onload = myFunction;

* Prototype 方式：

function myFunction() {

$("h01").insert("Hello Prototype!");

}

Event.observe(window, "load", myFunction);

Event.observe() 接受三个参数：

* 您希望处理的 HTML DOM 或 BOM（浏览器对象模型）对象
* 您希望处理的事件
* 您希望调用的函数

与 jQuery 相同，Prototype 允许链式语法。链接（Chaining）是一种在同一对象上执行多个任务的便捷方法。

$("h01").writeAttribute("style", "color:red").insert("Hello Prototype!");

## end