# 第1章 JavaScript 教程

JavaScript 是属于 HTML 和 Web 的编程语言。

编程令计算机完成您需要它们做的工作。

JavaScript 很容易学习。

本教程涵盖 JavaScript 基础和高级教程。

为何学习 JavaScript？

JavaScript 是 web 开发者必学的三种语言之一：

HTML 定义网页的内容

CSS 规定网页的布局

JavaScript 对网页行为进行编程

本教程提供关于 JavaScript，以及 JavaScript 如何与 HTML 和 CSS 协同工作的知识。

您知道吗？

JavaScript 和 Java 是完全不同的语言，不论是概念还是设计。

JavaScript 在 1995 年由 Brendan Eich 发明，并于 1997 年成为一部 ECMA 标准。

ECMA-262 是其官方名称。ECMAScript 6 （发布于 2015 年）是最新的 JavaScript 版本。

学习速度

在本教程中，您可以自由控制学习速度。

一切都取决于您。

如果您遇到困难，请休息一会儿，然后重新读一读材料。

请始终确保您理解了“亲自试一试”中的实例和练习题。

JavaScript 参考手册

W3School 提供并维护了完整的 JavaScript 参考手册，包括所有 HTML DOM 以及浏览器对象。

本参考手册包含了所有对象、属性及方法的示例，并根据最新 web 标准持续更新。

# 第2章 JavaScript 简介

JavaScript 能够改变 HTML 内容

getElementById() 是多个 JavaScript HTML 方法之一。

本例使用该方法来“查找” id="demo" 的 HTML 元素，并把元素内容（innerHTML）更改为 "Hello JavaScript"：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 能做什么</h2>

<p id="demo">JavaScript 能够改变 HTML 内容。</p>

<button type="button" onclick='document.getElementById("demo").innerHTML = "Hello JavaScript!"'>点击我！</button>

</body>

</html>

提示：JavaScript 同时接受双引号和单引号：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 能做什么？</h1>

<p id="demo">JavaScript 可以更改 HTML 内容。</p>

<button type="button" onclick="document.getElementById('demo').innerHTML = 'Hello JavaScript!'">单击这里</button>

</body>

</html>

JavaScript 能够改变 HTML 属性

本例通过改变 <img> 标签的 src 属性（source）来改变一张 HTML 图像：

灯泡

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 能做什么？</h2>

<p>JavaScript 能够改变 HTML 属性值。</p>

<p>在本例中，JavaScript 改变了图像的 src 属性值。</p>

<button onclick="document.getElementById('myImage').src='/i/eg\_bulbon.gif'">开灯</button>

<img id="myImage" border="0" src="/i/eg\_bulboff.gif" style="text-align:center;">

<button onclick="document.getElementById('myImage').src='/i/eg\_bulboff.gif'">关灯</button>

</body>

</html>

JavaScript 能够改变 HTML 样式 (CSS)

改变 HTML 元素的样式，是改变 HTML 属性的一种变种：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 能够做什么</h2>

<p id="demo">JavaScript 能够改变 HTML 元素的样式。</p>

<button type="button" onclick="document.getElementById('demo').style.fontSize='35px'">

点击我！

</button>

</body>

</html>

JavaScript 能够隐藏 HTML 元素

可通过改变 display 样式来隐藏 HTML 元素：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 能够做什么</h2>

<p id="demo">JavaScript 能够隐藏 HTML 元素。</p>

<button type="button" onclick="document.getElementById('demo').style.display='none'">

点击我！

</button>

</body>

</html>

JavaScript 能够显示 HTML 元素

可通过改变 display 样式来显示隐藏的 HTML 元素：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 能够做什么</h2>

<p>JavaScript 能够显示隐藏的 HTML 元素。</p>

<p id="demo" style="display:none">Hello JavaScript!</p>

<button type="button" onclick="document.getElementById('demo').style.display='block'">

点击我！

</button>

</body>

</html>

# 第3章 JavaScript 使用

<script> 标签

在 HTML 中，JavaScript 代码必须位于 <script> 与 </script> 标签之间。

实例

<!DOCTYPE html>

<html>

<body>

<h2>Body 中的 JavaScript</h2>

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

<script>

document.getElementById("demo").innerHTML = "我的第一段 JavaScript";

</script>

</body>

</html>

注释：旧的 JavaScript 例子也许会使用 type 属性：<script type="text/javascript">。

注释：type 属性不是必需的。JavaScript 是 HTML 中的默认脚本语言。

JavaScript 函数和事件

JavaScript 函数是一种 JavaScript 代码块，它可以在调用时被执行。

例如，当发生事件时调用函数，比如当用户点击按钮时。

提示：您将在稍后的章节学到更多有关函数和事件的知识。

<head> 或 <body> 中的 JavaScript

您能够在 HTML 文档中放置任意数量的脚本。

脚本可被放置与 HTML 页面的 <body> 或 <head> 部分中，或兼而有之。

<head> 中的 JavaScript

在本例中，JavaScript 函数被放置于 HTML 页面的 <head> 部分。

该函数会在按钮被点击时调用：

实例

<!DOCTYPE html>

<html>

<head>

<script>

function myFunction() {

document.getElementById("demo").innerHTML = "段落已被更改。";

}

</script>

</head>

<body>

<h2>Head 中的 JavaScript</h2>

<p id="demo">一个段落。</p>

<button type="button" onclick="myFunction()">试一试</button>

</body>

</html>

<body> 中的 JavaScript

在本例中，JavaScript 函数被放置于 HTML 页面的 <body> 部分。

该函数会在按钮被点击时调用：

实例

<!DOCTYPE html>

<html>

<body>

<h2>Body 中的 JavaScript</h2>

<p id="demo">一个段落。</p>

<button type="button" onclick="myFunction()">试一试</button>

<script>

function myFunction() {

document.getElementById("demo").innerHTML = "段落已被更改。";

}

</script>

</body>

</html>

提示：把脚本置于 <body> 元素的底部，可改善显示速度，因为脚本编译会拖慢显示。

外部脚本

脚本可放置与外部文件中：

外部文件：myScript.js

function myFunction() {

document.getElementById("demo").innerHTML = "段落被更改。";

}

外部脚本很实用，如果相同的脚本被用于许多不同的网页。

JavaScript 文件的文件扩展名是 .js。

如需使用外部脚本，请在 <script> 标签的 src (source) 属性中设置脚本的名称：

实例

<!DOCTYPE html>

<html>

<body>

<h2>外部 JavaScript</h2>

<p id="demo">一个段落。</p>

<button type="button" onclick="myFunction()">试一试</button>

<p>（myFunction 存储在名为 "myScript.js" 的外部文件中。）</p>

<script src="/demo/myScript.js"></script>

</body>

</html>

您可以在 <head> 或 <body> 中放置外部脚本引用。

该脚本的表现与它被置于 <script> 标签中是一样的。

注释：外部脚本不能包含 <script> 标签。

外部 JavaScript 的优势

在外部文件中放置脚本有如下优势：

分离了 HTML 和代码

使 HTML 和 JavaScript 更易于阅读和维护

已缓存的 JavaScript 文件可加速页面加载

如需向一张页面添加多个脚本文件 - 请使用多个 script 标签：

实例

<script src="myScript1.js"></script>

<script src="myScript2.js"></script>

外部引用

可通过完整的 URL 或相对于当前网页的路径引用外部脚本：

本例使用完整的 URL 来链接至脚本：

实例

<!DOCTYPE html>

<html>

<body>

<h1>外部 JavaScript</h1>

<p id="demo">一个段落。</p>

<button type="button" onclick="myFunction()">试一试</button>

<p>（myFunction 存储在名为 "myScript.js" 的外部文件中。）</p>

<script src="https://www.w3school.com.cn/demo/myScript.js"></script>

</body>

</html>

本例使用了位于当前网站上指定文件夹中的脚本：

实例

<!DOCTYPE html>

<html>

<body>

<h2>外部 JavaScript</h2>

<p id="demo">一个段落。</p>

<button type="button" onclick="myFunction()">试一试</button>

<p>（myFunction 存储在名为 "myScript.js" 的外部文件中。）</p>

<script src="/demo/myScript.js"></script>

</body>

</html>

本例链接了与当前页面相同文件夹的脚本：

实例

<!DOCTYPE html>

<html>

<body>

<h2>外部 JavaScript</h2>

<p id="demo">一个段落。</p>

<button type="button" onclick="myFunction()">试一试</button>

<p>（myFunction 存储在名为 "myScript.js" 的外部文件中。）</p>

<script src="/demo/myScript.js"></script>

</body>

</html>

# 第4章 JavaScript 输出

JavaScript 不提供任何内建的打印或显示函数。

JavaScript 显示方案

JavaScript 能够以不同方式“显示”数据：

使用 window.alert() 写入警告框

使用 document.write() 写入 HTML 输出

使用 innerHTML 写入 HTML 元素

使用 console.log() 写入浏览器控制台

使用 innerHTML

如需访问 HTML 元素，JavaScript 可使用 document.getElementById(id) 方法。

id 属性定义 HTML 元素。innerHTML 属性定义 HTML 内容：

实例

<!DOCTYPE html>

<html>

<body>

<h2>我的第一张网页</h2>

<p>我的第一个段落。</p>

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

<script>

document.getElementById("demo").innerHTML = 5 + 6;

</script>

</body>

</html>

提示：更改 HTML 元素的 innerHTML 属性是在 HTML 中显示数据的常用方法。

使用 document.write()

出于测试目的，使用 document.write() 比较方便：

实例

<!DOCTYPE html>

<html>

<body>

<h2>我的第一张网页</h2>

<p>我的第一个段落。</p>

<script>

document.write(5 + 6);

</script>

</body>

</html>

注意：在 HTML 文档完全加载后使用 document.write() 将删除所有已有的 HTML ：

实例

<!DOCTYPE html>

<html>

<body>

<h2>我的第一张网页</h2>

<p>我的第一个段落。</p>

<button type="button" onclick="document.write(5 + 6)">试一试</button>

</body>

</html>

提示：document.write() 方法仅用于测试。

使用 window.alert()

您能够使用警告框来显示数据：

实例

<!DOCTYPE html>

<html>

<body>

<h2>我的第一张网页</h2>

<p>我的第一个段落。</p>

<script>

window.alert(5 + 6);

</script>

</body>

</html>

使用 console.log()

在浏览器中，您可使用 console.log() 方法来显示数据。

请通过 F12 来激活浏览器控制台，并在菜单中选择“控制台”。

实例

<!DOCTYPE html>

<html>

<body>

<h2>按 F12 启动</h2>

<p>在 debugger 中选择 "Console"。然后再次点击运行按钮。</p>

<script>

console.log(5 + 6);

</script>

</body>

</html>

# 第5章 JavaScript 语句

在 HTML 中，JavaScript 语句是由 web 浏览器“执行”的“指令”。

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 语句</h2>

<p><b>JavaScript 程序</b> 是一系列由计算机执行的 <b>语句</b>。</p>

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

<script>

var x, y, z; // 语句 1

x = 22; // 语句 2

y = 11; // 语句 3

z = x + y; // 语句 4

document.getElementById("demo").innerHTML =

"z 的值是" + z + "。";

</script>

</body>

</html>

JavaScript 程序

计算机程序是由计算机“执行”的一系列“指令”。

在编程语言中，这些编程指令被称为语句。

JavaScript 程序就是一系列的编程语句。

注释：在 HTML 中，JavaScript 程序由 web 浏览器执行。

JavaScript 语句

JavaScript 语句由以下构成：

值、运算符、表达式、关键词和注释。

这条语句告诉浏览器在 id="demo" 的 HTML 元素中输出 "Hello Kitty."：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 语句</h2>

<p>在 HTML 中，JavaScript 语句由浏览器执行。</p>

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

<script>

document.getElementById("demo").innerHTML = "Hello Kitty.";

</script>

</body>

</html>

大多数 JavaScript 程序都包含许多 JavaScript 语句。

这些语句会按照它们被编写的顺序逐一执行。

注释：JavaScript 程序（以及 JavaScript 语句）常被称为 JavaScript 代码。

分号 ;

分号分隔 JavaScript 语句。

请在每条可执行的语句之后添加分号：

a = 5;

b = 6;

c = a + b;

如果有分号分隔，允许在同一行写多条语句：

a = 5; b = 6; c = a + b;

您可能在网上看到不带分号的例子。

提示：以分号结束语句不是必需的，但我们仍然强烈建议您这么做。

JavaScript 空白字符

JavaScript 会忽略多个空格。您可以向脚本添加空格，以增强可读性。

下面这两行是相等的：

var person = "Bill";

var person="Bill";

在运算符旁边（ = + - \* / ）添加空格是个好习惯：

var x = y + z;

JavaScript 行长度和折行

为了达到最佳的可读性，程序员们常常喜欢把代码行控制在 80 个字符以内。

如果 JavaScript 语句太长，对其进行折行的最佳位置是某个运算符：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 语句</h2>

<p>

折行的最佳位置是运算符或逗号之后。

</p>

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

<script>

document.getElementById("demo").innerHTML =

"Hello Kitty.";

</script>

</body>

</html>

JavaScript 代码块

JavaScript 语句可以用花括号（{...}）组合在代码块中。

代码块的作用是定义一同执行的语句。

您会在 JavaScript 中看到成块组合在一起的语句：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 语句</h2>

<p>JavaScript 代码块在 { 与 } 之间编写。</p>

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

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

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

<script>

function myFunction() {

document.getElementById("demo1").innerHTML = "Hello Kitty.";

document.getElementById("demo2").innerHTML = "How are you?";

}

</script>

</body>

</html>

注释：在本教程中我们为代码块用了 4 个空格的缩进。

提示：您将稍后在本教程中学到更多有关函数的内容。

JavaScript 关键词

JavaScript 语句常常通过某个关键词来标识需要执行的 JavaScript 动作。

下面的表格列出了一部分将在教程中学到的关键词：

|  |  |
| --- | --- |
| 关键词 | 描述 |
| break | 终止 switch 或循环。 |
| continue | 跳出循环并在顶端开始。 |
| debugger | 停止执行 JavaScript，并调用调试函数（如果可用）。 |
| do ... while | 执行语句块，并在条件为真时重复代码块。 |
| for | 标记需被执行的语句块，只要条件为真。 |
| function | 声明函数。 |
| if ... else | 标记需被执行的语句块，根据某个条件。 |
| return | 退出函数。 |
| switch | 标记需被执行的语句块，根据不同的情况。 |
| try ... catch | 对语句块实现错误处理。 |
| var | 声明变量。 |

注释：JavaScript 关键词指的是保留的单词。保留词无法用作变量名。

# 第6章 JavaScript 语法

JavaScript 语法是一套规则，它定义了 JavaScript 的语言结构。

var x, y; // 如何声明变量

x = 7; y = 8; // 如何赋值

z = x + y; // 如何计算值

JavaScript 值

JavaScript 语句定义两种类型的值：混合值和变量值。

混合值被称为字面量（literal）。变量值被称为变量。

JavaScript 字面量

书写混合值最重要的规则是：

写数值有无小数点均可：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数值</h2>

<p>写数值有无小数点均可。</p>

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

<script>

document.getElementById("demo").innerHTML = 15.90;

</script>

</body>

</html>

字符串是文本，由双引号或单引号包围：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 字符串</h2>

<p>字符串可由双引号或单引号包围。</p>

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

<script>

document.getElementById("demo").innerHTML = 'Bill Gates';

</script>

</body>

</html>

JavaScript 变量

在编程语言中，变量用于存储数据值。

JavaScript 使用 var 关键词来声明变量。

= 号用于为变量赋值。

在本例中，x 被定义为变量。然后，x 被赋的值是 7：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 变量</h2>

<p>在本例中，x 被定义为变量。然后 x 被赋值 7：</p>

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

<script>

var x;

x = 7;

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

</script>

</body>

</html>

JavaScript 运算符

JavaScript 使用算数运算符（+ - \* /）来计算值：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 运算符</h2>

<p>JavaScript 使用算数运算符来计算值（类似代数）。</p>

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

<script>

document.getElementById("demo").innerHTML = (7 + 8) \* 10;

</script>

</body>

</html>

JavaScript 使用赋值运算符（=）向变量赋值：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 赋值</h2>

<p>在 JavaScript 中，= 运算符用于向变量赋值。</p>

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

<script>

var x, y;

x = 7;

y = 8;

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

</script>

</body>

</html>

JavaScript 表达式

表达式是值、变量和运算符的组合，计算结果是值。

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 表达式</h2>

<p>表达式计算为值。</p>

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

<script>

document.getElementById("demo").innerHTML = 6 \* 10;

</script>

</body>

</html>

表达式也可包含变量值：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 表达式</h2>

<p>表达式计算为值。</p>

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

<script>

var x;

x = 6;

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

</script>

</body>

</html>

值可以是多种类型，比如数值和字符串。

例如，"Bill" + " " + "Gates"，计算为 "Bill Gates"：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 表达式</h2>

<p>表达式计算为值。</p>

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

<script>

document.getElementById("demo").innerHTML = "Bill" + " " + "Gates";

</script>

</body>

</html>

JavaScript 关键词

JavaScript 关键词用于标识被执行的动作。

var 关键词告知浏览器创建新的变量：

<!DOCTYPE html>

<html>

<body>

<h2>var 关键词创建变量</h2>

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

<script>

var x, y;

x = 7 + 8;

y = x \* 10;

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

</script>

</body>

</html>

JavaScript 注释

并非所有 JavaScript 语句都被“执行”。

双斜杠 // 或 /\* 与 \*/ 之间的代码被视为注释。

注释会被忽略，不会被执行：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 注释<b>不会</b>被执行。</h2>

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

<script>

var x;

x = 7;

// x = 8; 我不会被执行

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

</script>

</body>

</html>

JavaScript 标识符

标识符是名称。

在 JavaScript 中，标识符用于命名变量（以及关键词、函数和标签）。

在大多数编程语言中，合法名称的规则大多相同。

在 JavaScript 中，首字符必须是字母、下划线（-）或美元符号（$）。

连串的字符可以是字母、数字、下划线或美元符号。

提示：数值不可以作为首字符。这样，JavaScript 就能轻松区分标识符和数值。

JavaScript 对大小写敏感

所有 JavaScript 标识符对大小写敏感。

变量 lastName 和 lastname，是两个不同的变量。

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 对大小写敏感</h2>

<p>请试着把 lastName 改为 lastname。</p>

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

<script>

var lastname, lastName;

lastName = "Gates";

lastname = "Jobs";

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

</script>

</body>

</html>

JavaScript 不会把 VAR 或 Var 译作关键词 var。

JavaScript 与驼峰式大小写

历史上，程序员曾使用三种把多个单词连接为一个变量名的方法：

连字符：

first-name, last-name, master-card, inter-city.

注释：JavaScript 中不能使用连字符。它是为减法预留的。

下划线：

first\_name, last\_name, master\_card, inter\_city.

驼峰式大小写（Camel Case）：

FirstName, LastName, MasterCard, InterCity.

JavaScript 程序员倾向于使用以小写字母开头的驼峰大小写：

firstName, lastName, masterCard, interCity

JavaScript 字符集

JavaScript 使用 Unicode 字符集。

Unicode 覆盖世界上几乎所有的字符、标点和符号。

如需了解详情，请访问我们完整的 Unicode 参考手册。

# 第7章 JavaScript 注释

JavaScript 注释用于解释 JavaScript 代码，增强其可读性。

JavaScript 注释也可以用于在测试替代代码时阻止执行。

单行注释

单行注释以 // 开头。

任何位于 // 与行末之间的文本都会被 JavaScript 忽略（不会执行）。

本例在每条代码行之前使用单行注释：

<!DOCTYPE html>

<html>

<body>

<h1 id="myH"></h1>

<p id="myP"></p>

<script>

// 改变标题：

document.getElementById("myH").innerHTML = "JavaScript 注释";

// 改变段落：

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

</script>

</body>

</html>

多行注释

多行注释以 /\* 开头，以 \*/ 结尾。

任何位于 /\* 和 \*/ 之间的文本都会被 JavaScript 忽略。

本例使用多行注释（注释块）来解释代码：

实例

<!DOCTYPE html>

<html>

<body>

<h1 id="myH"></h1>

<p id="myP"></p>

<script>

/\*

下面的代码会改变

id = "myH" 的标题

以及 id = "myp" 的段落

\*/

document.getElementById("myH").innerHTML = "JavaScript 注释";

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

</script>

</body>

</html>

注释：使用单行注释最为常见。

提示：注释块常用于官方声明。

使用注释来阻止执行

使用注释来防止代码执行很适合代码测试。

在代码行之前添加 // 会把可执行的代码行更改为注释。

本例使用 // 来阻止代码行的执行：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 注释</h2>

<h1 id="myH"></h1>

<p id="myP"></p>

<script>

//document.getElementById("myH").innerHTML = "我的首页";

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

</script>

<p>// 开头的代码行不会被执行。</p>

</body>

</html>

本例使用注释块来阻止多行代码的执行：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 注释</h2>

<h1 id="myH"></h1>

<p id="myP"></p>

<script>

/\*

document.getElementById("myH").innerHTML = "欢迎访问我的首页";

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

\*/

document.getElementById("myP").innerHTML = "注释块不会被执行。";

</script>

</body>

</html>

# 第8章 JavaScript 变量

JavaScript 变量

JavaScript 变量是存储数据值的容器。

在本例中，x、y 和 z 是变量：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 变量</h1>

<p>在本例中，x、y 以及 z 都是变量。</p>

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

<script>

var x = 7;

var y = 8;

var z = x + y;

document.getElementById("demo").innerHTML =

"z 的值是：" + z;

</script>

</body>

</html>

从上例中，您可获得：

x 存储值 7

y 存储值 8

z 存储值 15

类似代数

在本例中，price1、price2 以及 total 是变量：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 变量</h1>

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

<script>

var price1 = 7;

var price2 = 8;

var price3 = 12;

var total = price1 + price2 + price3;

document.getElementById("demo").innerHTML = "总计：" + total;

</script>

</body>

</html>

在编程中，类似代数，我们使用变量（比如 price1）来存放值。

在编程中，类似代数，我们在表达式中使用变量（total = price1 + price2）。

从上例中，您可以算出 total 的值是 27。

提示：JavaScript 变量是存储数据值的容器。

JavaScript 标识符

所有 JavaScript 变量必须以唯一的名称的标识。

这些唯一的名称称为标识符。

标识符可以是短名称（比如 x 和 y），或者更具描述性的名称（age、sum、totalVolume）。

构造变量名称（唯一标识符）的通用规则是：

名称可包含字母、数字、下划线和美元符号

名称必须以字母开头

名称也可以 $ 和 \_ 开头（但是在本教程中我们不会这么做）

名称对大小写敏感（y 和 Y 是不同的变量）

保留字（比如 JavaScript 的关键词）无法用作变量名称

提示：JavaScript 标识符对大小写敏感。

赋值运算符

在 JavaScript 中，等号（=）是赋值运算符，而不是“等于”运算符。

这一点与代数不同。下面的代码在代数中是不合理的：

x = x + 5

然而在 JavaScript 中，它非常合理：把 x + 5 的值赋给 x。

（计算 x + 5 的值并把结果放入 x 中。x 的值递增 5。）

注释：JavaScript 中的“等于”运算符是 ==。

JavaScript 数据类型

JavaScript 变量可存放数值，比如 100，以及文本值，比如 "Bill Gates"。

在编程中，文本值被称为字符串。

JavaScript 可处理多种数据类型，但是现在，我们只关注数值和字符串值。

字符串被包围在双引号或单引号中。数值不用引号。

如果把数值放在引号中，会被视作文本字符串。

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 变量</h1>

<p>字符串用引号包围。</p>

<p>数值不用引号包围。</p>

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

<script>

var pi = 3.14;

var person = "Bill Gates";

var answer = 'How are you!';

document.getElementById("demo").innerHTML =

pi + "<br>" + person + "<br>" + answer;

</script>

</body>

</html>

声明（创建） JavaScript 变量

在 JavaScript 中创建变量被称为“声明”变量。

您可以通过 var 关键词来声明 JavaScript 变量：

var carName;

声明之后，变量是没有值的。（技术上，它的值是 undefined。）

如需赋值给变量，请使用等号：

carName = "porsche";

您可以在声明变量时向它赋值：

var carName = "porsche";

在上面的例子中，我们创建了名为 carName 的变量，并向其赋值 "porsche"。

然后，我们在 id="demo" 的 HTML 段落中“输出”该值：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 变量</h1>

<p>创建一个变量，向其赋值，然后显示它：</p>

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

<script>

var carName = "porsche";

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

</script>

</body>

</html>

提示：在脚本的开头声明所有变量是个好习惯！

一条语句，多个变量

您可以在一条语句中声明许多变量。

以 var 作为语句的开头，并以逗号分隔变量：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 变量</h1>

<p>您可以在一条语句中声明许多变量。</p>

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

<script>

var person = "Bill Gates", carName = "porsche", price = 150000;

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

</script>

</body>

</html>

声明可横跨多行：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 变量</h1>

<p>您可以在一条语句中声明多个变量。</p>

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

<script>

var person = "Bill Gates",

carName = "porsche",

price = 150000;

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

</script>

</body>

</html>

Value = undefined

在计算机程序中，被声明的变量经常是不带值的。值可以是需被计算的内容，或是之后被提供的数据，比如数据输入。

不带有值的变量，它的值将是 undefined。

变量 carName 在这条语句执行后的值是 undefined：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 变量</h1>

<p>如果变量声明时没有赋值，那么其值是 undefined。</p>

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

<script>

var carName;

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

</script>

</body>

</html>

重复声明 JavaScript 变量

如果再次声明某个 JavaScript 变量，将不会丢它的值。

在这两条语句执行后，变量 carName 的值仍然是 "porsche"：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 变量</h1>

<p>如果您再次声明某个 JavaScript 变量，那么它的值不会丢失。</p>

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

<script>

var carName = "porsche";

var carName;

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

</script>

</body>

</html>

JavaScript 算术

与代数类似，您能够通过 JavaScript 变量进行算术运算，使用 = 和 + 之类的运算符：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 变量</h1>

<p>相加 5 + 2 + 3 的结果：</p>

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

<script>

var x = 3 + 5 + 8;

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

</script>

</body>

</html>

字符串也可以使用加号，但是字符串将被级联：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 变量</h1>

<p>相加 "Bill" + " " + "Gates" 的结果是：</p>

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

<script>

var x = "Bill" + " " + "Gates";

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

</script>

</body>

</html>

还可以试试这个：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 变量</h1>

<p>相加 "8" + 3 + 5 的结果是：</p>

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

<script>

x = "8" + 3 + 5;

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

</script>

</body>

</html>

提示：如果把要给数值放入引号中，其余数值会被视作字符串并被级联。

现在试试这个：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 变量</h1>

<p>相加 3 + 5 + "8" 的结果是：</p>

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

<script>

var x = 3 + 5 + "8"

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

</script>

</body>

</html>

# 第9章 JavaScript Let

ECMAScript 2015

ES2015 引入了两个重要的 JavaScript 新关键词：let 和 const。

这两个关键字在 JavaScript 中提供了块作用域（Block Scope）变量（和常量）。

在 ES2015 之前，JavaScript 只有两种类型的作用域：全局作用域和函数作用域。

全局作用域

全局（在函数之外）声明的变量拥有全局作用域。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 全局作用域</h1>

<p>可以从任何脚本或函数中访问全局变量。</p>

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

<script>

var carName = "porsche";

myFunction();

function myFunction() {

document.getElementById("demo").innerHTML =

"I can display " + carName;

}

</script>

</body>

</html>

全局变量可以在 JavaScript 程序中的任何位置访问。

函数作用域

局部（函数内）声明的变量拥有函数作用域。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 局部作用域</h1>

<p>myFunction() 之外的 carName 未定义。</p>

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

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

<script>

myFunction();

function myFunction() {

var carName = "porsche";

document.getElementById("demo1").innerHTML =

typeof carName + " " + carName;

}

document.getElementById("demo2").innerHTML =

typeof carName;

</script>

</body>

</html>

局部变量只能在它们被声明的函数内访问。

JavaScript 块作用域

通过 var 关键词声明的变量没有块作用域。

在块 {} 内声明的变量可以从块之外进行访问。

实例

{

var x = 10;

}

// 此处可以使用 x

在 ES2015 之前，JavaScript 是没有块作用域的。

可以使用 let 关键词声明拥有块作用域的变量。

在块 {} 内声明的变量无法从块外访问：

实例

{

let x = 10;

}

// 此处不可以使用 x

重新声明变量

使用 var 关键字重新声明变量会带来问题。

在块中重新声明变量也将重新声明块外的变量：

实例

<!DOCTYPE html>

<html>

<body>

<h1>使用 var 声明变量</h1>

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

<script>

var x = 10;

// Here x is 10

{

var x = 2;

// Here x is 2

}

// Here x is 2

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

</script>

</body>

</html>

使用 let 关键字重新声明变量可以解决这个问题。

在块中重新声明变量不会重新声明块外的变量：

实例

<!DOCTYPE html>

<html>

<body>

<h1>使用 let 声明变量</h1>

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

<script>

var x = 10;

// Here x is 10

{

let x = 2;

// Here x is 2

}

// Here x is 10

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

</script>

</body>

</html>

循环作用域

在循环中使用 var：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript let</h1>

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

<script>

var i = 5;

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

// 一些语句

}

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

</script>

</body>

</html>

在循环中使用 let：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript let</h1>

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

<script>

let i = 5;

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

// 一些语句

}

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

</script>

</body>

</html>

在第一个例子中，在循环中使用的变量使用 var 重新声明了循环之外的变量。

在第二个例子中，在循环中使用的变量使用 let 并没有重新声明循环外的变量。

如果在循环中用 let 声明了变量 i，那么只有在循环内，变量 i 才是可见的。

函数作用域

在函数内声明变量时，使用 var 和 let 很相似。

它们都有函数作用域：

function myFunction() {

var carName = "porsche"; // 函数作用域

}

function myFunction() {

let carName = "porsche"; // 函数作用域

}

全局作用域

如果在块外声明声明，那么 var 和 let 也很相似。

它们都拥有全局作用域：

var x = 10; // 全局作用域

let y = 6; // 全局作用域

HTML 中的全局变量

使用 JavaScript 的情况下，全局作用域是 JavaScript 环境。

在 HTML 中，全局作用域是 window 对象。

通过 var 关键词定义的全局变量属于 window 对象：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 全局变量</h1>

<p>在 HTML 中，通过 <b>var</b> 定义的全局变量会成为 window 变量。</p>

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

<script>

var carName = "Audi";

// 此处的代码能够使用 window.carName

document.getElementById("demo").innerHTML = "我可以显示 " + window.carName;

</script>

</body>

</html>

通过 let 关键词定义的全局变量不属于 window 对象：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 全局变量</h1>

<p>在 HTML 中，由 <b>let</b> 定义的全局变量，不会成为 window 变量。</p>

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

<script>

let carName = "Audi";

// 此处的代码能够使用 window.carName

document.getElementById("demo").innerHTML = "我不能显示 " + window.carName;

</script>

</body>

</html>

重新声明

允许在程序的任何位置使用 var 重新声明 JavaScript 变量：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript let</h1>

<p>在程序的任何位置都允许使用 <b>var</b> 重新声明JavaScript变量：</p>

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

<script>

var x = 2;

// 此时 x 是 2

var x = 3;

// 此时 x 是 3

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

</script>

</body>

</html>

在相同的作用域，或在相同的块中，通过 let 重新声明一个 var 变量是不允许的：

实例

var x = 10; // 允许

let x = 6; // 不允许

{

var x = 10; // 允许

let x = 6; // 不允许

}

在相同的作用域，或在相同的块中，通过 let 重新声明一个 let 变量是不允许的：

实例

let x = 10; // 允许

let x = 6; // 不允许

{

let x = 10; // 允许

let x = 6; // 不允许

}

在相同的作用域，或在相同的块中，通过 var 重新声明一个 let 变量是不允许的：

实例

let x = 10; // 允许

var x = 6; // 不允许

{

let x = 10; // 允许

var x = 6; // 不允许

}

在不同的作用域或块中，通过 let 重新声明变量是允许的：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript let</h1>

<p>使用 <b>let</b>，在另一个作用域或另一个块中重新声明变量是允许的：</p>

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

<script>

let x = 2; // 允许

{

let x = 3; // 允许

}

{

let x = 4; // 允许

}

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

</script>

</body>

</html>

提升

通过 var 声明的变量会提升到顶端。如果您不了解什么是提升（Hoisting），请学习我们的提升这一章。

您可以在声明变量之前就使用它：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 提升</h1>

<p>使用 <b>var</b>，您可以在声明之前使用变量：</p>

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

<script>

carName = "Audi";

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

var carName;

</script>

</body>

</html>

通过 let 定义的变量不会被提升到顶端。

在声明 let 变量之前就使用它会导致 ReferenceError。

变量从块的开头一直处于“暂时死区”，直到声明为止：

实例

// 在此处，您不可以使用 carName

let carName;

# 第10章 JavaScript Const

ECMAScript 2015

ES2015 引入了两个重要的 JavaScript 新关键词：let 和 const。

通过 const 定义的变量与 let 变量类似，但不能重新赋值：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript const</h2>

<p>您无法更改原始值。</p>

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

<script>

try {

const PI = 3.141592653589793;

PI = 3.14;

}

catch (err) {

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

}

</script>

</body>

</html>

块作用域

在块作用域内使用 const 声明的变量与 let 变量相似。

在本例中，x 在块中声明，不同于在块之外声明的 x：

实例

<!DOCTYPE html>

<html>

<body>

<h2>使用 const 声明变量</h2>

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

<script>

var x = 10;

// 此处 x 是 10

{

const x = 2;

// 此处 x 是 2

}

// 此处 x 是 10

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

</script>

</body>

</html>

在声明时赋值

JavaScript const 变量必须在声明时赋值：

不正确

const PI;

PI = 3.14159265359;

正确

const PI = 3.14159265359;

不是真正的常数

关键字 const 有一定的误导性。

它没有定义常量值。它定义了对值的常量引用。

因此，我们不能更改常量原始值，但我们可以更改常量对象的属性。

原始值

如果我们将一个原始值赋给常量，我们就不能改变原始值：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript const</h2>

<p>您无法更改原始值。</p>

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

<script>

try {

const PI = 3.141592653589793;

PI = 3.14;

}

catch (err) {

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

}

</script>

</body>

</html>

常量对象可以更改

您可以更改常量对象的属性：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript const</h2>

<p>声明常量对象不会使对象属性不可更改：</p>

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

<script>

// 创建对象：

const car = {type:"Porsche", model:"911", color:"white"};

// 更改属性：

car.color = "red";

// 添加属性：

car.owner = "Steve";

// 显示属性：

document.getElementById("demo").innerHTML = "车主是：" + car.owner;

</script>

</body>

</html>

但是您无法重新为常量对象赋值：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript const</h2>

<p>但是你不能重新赋值一个常量对象：</p>

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

<script>

try {

const car = {type:"Porsche", model:"911", color:"white"};

car = {type:"BMW", model:"M5", color:"red"};

}

catch (err) {

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

}

</script>

</body>

</html>

常量数组可以更改

您可以更改常量数组的元素：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript const</h2>

<p>声明常量数组不会使元素不可更改：</p>

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

<script>

// 创建数组：

const cars = ["Audi", "BMW", "Porsche"];

// 更改元素：

cars[0] = "Toyota";

// 添加元素：

cars.push("Honda");

// 显示数组：

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

</script>

</body>

</html>

但是您无法重新为常量数组赋值：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript const</h2>

<p>您无法重新赋值常量数组：</p>

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

<script>

try {

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

cars = ["Toyota", "Volvo", "Audi"];

}

catch (err) {

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

}

</script>

</body>

</html>

重新声明

在程序中的任何位置都允许重新声明 JavaScript var 变量：

实例

var x = 2; // 允许

var x = 3; // 允许

x = 4; // 允许

在同一作用域或块中，不允许将已有的 var 或 let 变量重新声明或重新赋值给 const：

实例

var x = 2; // 允许

const x = 2; // 不允许

{

let x = 2; // 允许

const x = 2; // 不允许

}

在同一作用域或块中，为已有的 const 变量重新声明声明或赋值是不允许的：

实例

const x = 2; // 允许

const x = 3; // 不允许

x = 3; // 不允许

var x = 3; // 不允许

let x = 3; // 不允许

{

const x = 2; // 允许

const x = 3; // 不允许

x = 3; // 不允许

var x = 3; // 不允许

let x = 3; // 不允许

}

在另外的作用域或块中重新声明 const 是允许的：

实例

const x = 2; // 允许

{

const x = 3; // 允许

}

{

const x = 4; // 允许

}

提升

通过 var 定义的变量会被提升到顶端。如果您不了解什么是提升（Hoisting），请学习提升这一章。

您可以在声明 var 变量之前就使用它：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 提升</h2>

<p>使用 <b>var</b>，您可以在声明变量之前使用它：</p>

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

<script>

carName = "Audi";

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

var carName;

</script>

</body>

</html>

通过 const 定义的变量不会被提升到顶端。

const 变量不能在声明之前使用：

实例

carName = "Volvo"; // 您不可以在此处使用 carName

const carName = "Volvo";

# 第11章 JavaScript 运算符

JavaScript 运算符

实例

向变量赋值，并把它们相加：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 运算符</h1>

<p>x = 7，y = 8，计算 z = x + y，然后显示 z：</p>

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

<script>

var x = 7;

var y = 8;

var z = x + y;

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

</script>

</body>

</html>

赋值运算符（=）把值赋给变量。

赋值

<!DOCTYPE html>

<html>

<body>

<h1>= 运算符</h1>

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

<script>

var x = 15;

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

</script>

</body>

</html>

加法运算符（+）对数字相加：

加法

<!DOCTYPE html>

<html>

<body>

<h1>+ 运算符</h1>

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

<script>

var x = 7;

var y = 8;

var z = x + y;

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

</script>

</body>

</html>

乘法运算符（\*）对数字相乘：

乘法

<!DOCTYPE html>

<html>

<body>

<h1>\* 运算符</h1>

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

<script>

var x = 7;

var y = 8;

var z = x \* y;

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

</script>

</body>

</html>

JavaScript 算数运算符

算数运算符用于对数字执行算数运算：

|  |  |
| --- | --- |
| 运算符 | 描述 |
| + | 加法 |
| - | 减法 |
| \* | 乘法 |
| / | 除法 |
| % | 取模（余数） |
| ++ | 递加 |
| -- | 递减 |

注释：JS 算数这一章对算数运算符进行了完整描述。

JavaScript 赋值运算符

赋值运算符向 JavaScript 变量赋值。

|  |  |  |
| --- | --- | --- |
| 运算符 | 例子 | 等同于 |
| = | x = y | x = y |
| += | x += y | x = x + y |
| -= | x -= y | x = x - y |
| \*= | x \*= y | x = x \* y |
| /= | x /= y | x = x / y |
| %= | x %= y | x = x % y |

加法赋值运算符（+=）向变量添加一个值。

赋值

<!DOCTYPE html>

<html>

<body>

<h1>+= 运算符</h1>

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

<script>

var x = 7;

x += 8;

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

</script>

</body>

</html>

注释：JS 赋值这一章中完整描述了赋值运算符。

JavaScript 字符串运算符

+ 运算符也可用于对字符串进行相加（concatenate，级联）。

实例

txt1 = "Bill";

txt2 = "Gates";

txt3 = txt1 + " " + txt2;

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 运算符</h1>

<p>+ 运算符串联（相加）字符串。</p>

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

<script>

var txt1 = "Bill";

var txt2 = "Gates";

document.getElementById("demo").innerHTML = txt1 + " " + txt2;

</script>

</body>

</html>

+= 赋值运算符也可用于相加（级联）字符串：

实例

txt1 = "Hello ";

txt1 += "Kitty!";

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 运算符</h1>

<p>赋值运算符 += 串联字符串。</p>

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

<script>

txt1 = "Hello ";

txt1 += "Kitty！";

document.getElementById("demo").innerHTML = txt1;

</script>

</body>

</html>

提示：在用于字符串时，+ 运算符被称为级联运算符。

字符串和数字的相加

相加两个数字，将返回和，但对一个数字和一个字符串相加将返回一个字符串：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 运算符</h1>

<p>对数字和字符串相加，会返回字符串。</p>

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

<script>

var x = 7 + 8;

var y = "7" + 8;

var z = "Hello" + 7;

document.getElementById("demo").innerHTML =

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

</script>

</body>

</html>

提示：如果您对数字和字符串相加，结果将是字符串！

JavaScript 比较运算符

|  |  |
| --- | --- |
| 运算符 | 描述 |
| == | 等于 |
| === | 等值等型 |
| != | 不相等 |
| !== | 不等值或不等型 |
| > | 大于 |
| < | 小于 |
| >= | 大于或等于 |
| <= | 小于或等于 |
| ? | 三元运算符 |

注释：JS 比较这一章中完整描述了比较运算符。

JavaScript 逻辑运算符

|  |  |
| --- | --- |
| 运算符 | 描述 |
| && | 逻辑与 |
| || | 逻辑或 |
| ! | 逻辑非 |

JavaScript 类型运算符

|  |  |
| --- | --- |
| 运算符 | 描述 |
| typeof | 返回变量的类型。 |
| instanceof | 返回 true，如果对象是对象类型的实例。 |

JavaScript 位运算符

位运算符处理 32 位数。

该运算中的任何数值运算数都会被转换为 32 位的数。结果会被转换回 JavaScript 数。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 运算符 | 描述 | 例子 | 等同于 | 结果 | 十进制 |
| & | 与 | 5 & 1 | 0101 & 0001 | 0001 | 1 |
| | | 或 | 5 | 1 | 0101 | 0001 | 0101 | 5 |
| ~ | 非 | ~ 5 | ~0101 | 1010 | 10 |
| ^ | 异或 | 5 ^ 1 | 0101 ^ 0001 | 0100 | 4 |
| << | 零填充左位移 | 5 << 1 | 0101 << 1 | 1010 | 10 |
| >> | 有符号右位移 | 5 >> 1 | 0101 >> 1 | 0010 | 2 |
| >>> | 零填充右位移 | 5 >>> 1 | 0101 >>> 1 | 0010 | 2 |

上例使用 4 位无符号的例子。但是 JavaScript 使用 32 位有符号数。

因此，在 JavaScript 中，~ 5 不会返回 10，而是返回 -6。

~00000000000000000000000000000101 将返回 11111111111111111111111111111010。

注释：我们将在 JS 位运算这一章为您详解位运算符。

# 第12章 JavaScript 算数

处理数值的典型情景是算数。

JavaScript 算数运算符

算术运算符对数值（文字或变量）执行算术运算。

|  |  |
| --- | --- |
| 运算符 | 描述 |
| + | 加法 |
| - | 减法 |
| \* | 乘法 |
| \*\* | 幂（ES2016） |
| / | 除法 |
| % | 系数 |
| ++ | 递增 |
| -- | 递减 |

算数运算符

典型的算术运算会操作两个数值。

这两个数可以是字面量：

实例

<!DOCTYPE html>

<html>

<body>

<p>典型的算术运算会操作两个数（或表达式）并返回一个新的数。</p>

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

<script>

var x = 7 + 8;

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

</script>

</body>

</html>

或变量：

实例

<!DOCTYPE html>

<html>

<body>

<p>典型的算术运算会操作两个数（或表达式）并返回一个新的数。</p>

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

<script>

var a = 7;

var b = 8;

var x = a + b;

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

</script>

</body>

</html>

或表达式：

实例

<!DOCTYPE html>

<html>

<body>

<p>典型的算术运算会操作两个数（或表达式）并返回一个新的数。</p>

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

<script>

var a = 2;

var x = (7 + 8) \* a;

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

</script>

</body>

</html>

运算符和操作数

在算术运算中，数被称为操作数。

（两个操作数之间执行的）运算由运算符定义

加法

加法运算符（+）加数：

<!DOCTYPE html>

<html>

<body>

<h1>+ 运算符</h1>

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

<script>

var x = 7;

var y = 8;

var z = x + y;

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

</script>

</body>

</html>

减法

减法运算符（-）减数。

var x = 7;

var y = 8;

var z = x - y;

亲自试一试

乘法

乘法运算符（\*）乘数。

var x = 7;

var y = 8;

var z = x \* y;

亲自试一试

除法

除法运算符（/）除数。

var x = 7;

var y = 2;

var z = x / y;

亲自试一试

系数

系数运算符（%）返回除法的余数。

var x = 7;

var y = 2;

var z = x % y;

亲自试一试

注释：在算术中，两个整数的除法产生商和余数。

注释：在数学中，模运算的结果是算术除法的余数。

递增

递增运算符（++）对数值进行递增。

var x = 7;

x++;

var z = x;

亲自试一试

递减

递减运算符（--）对数值进行递减。

var x = 7;

x--;

var z = x;

亲自试一试

幂

取幂运算符（\*\*）将第一个操作数提升到第二个操作数的幂。

实例

var x = 5;

var z = x \*\* 2; // 结果是 25

亲自试一试

x \*\* y 产生的结果与 Math.pow(x,y) 相同:

实例

var x = 5;

var z = Math.pow(x,2); // 结果是 25

亲自试一试

运算符优先级

运算符优先级（Operator precedence）描述了在算术表达式中所执行操作的顺序。

实例

var x = 200 + 50 \* 2;

亲自试一试

上例的结果是 250 \* 2 还是 200 + 100 呢？

是加法还是乘法优先呢？

在传统的学校数学中，乘法是优先的。

乘法（\*）和除法（%）比加法（+）和减法（-）拥有更高的优先级。

同时，（就像在学校的数学中那样）能够通过使用括号来改变优先级：

实例

var x = (200 + 50) \* 2;

亲自试一试

当使用括号时，括号中的运算符会首先被计算。

当多个运算拥有相同的优先级时（比如加法和减法），对它们的计算是从左向右的：

实例

var x = 200 + 50 - 2;

JavaScript 运算符优先级值

|  |  |  |  |
| --- | --- | --- | --- |
| 值 | 运算符 | 描述 | 实例 |
| 20 | ( ) | 表达式分组 | (3 + 4) |
|  |  |  |  |
| 19 | . | 成员 | person.name |
| 19 | [] | 成员 | person["name"] |
| 19 | () | 函数调用 | myFunction() |
| 19 | new | 创建 | new Date() |
|  |  |  |  |
| 17 | ++ | 后缀递增 | i++ |
| 17 | -- | 后缀递减 | i-- |
|  |  |  |  |
| 16 | ++ | 前缀递增 | ++i |
| 16 | -- | 前缀递减 | --i |
| 16 | ! | 逻辑否 | !(x==y) |
| 16 | typeof | 类型 | typeof x |
|  |  |  |  |
| 15 | \*\* | 求幂 (ES7) | 10 \*\* 2 |
|  |  |  |  |
| 14 | \* | 乘 | 10 \* 5 |
| 14 | / | 除 | 10 / 5 |
| 14 | % | 模数除法 | 10 % 5 |
|  |  |  |  |
| 13 | + | 加 | 10 + 5 |
| 13 | - | 减 | 10 - 5 |
|  |  |  |  |
| 12 | << | 左位移 | x << 2 |
| 12 | >> | 右位移 | x >> 2 |
| 12 | >>> | 右位移（无符号） | x >>> 2 |
|  |  |  |  |
| 11 | < | 小于 | x < y |
| 11 | <= | 小于或等于 | x <= y |
| 11 | > | 大于 | x > y |
| 11 | >= | 大于或等于 | x >= y |
| 11 | in | 对象中的属性 | "PI" in Math |
| 11 | instanceof | 对象的实例 | instanceof Array |
|  |  |  |  |
| 10 | == | 相等 | x == y |
| 10 | === | 严格相等 | x === y |
| 10 | != | 不相等 | x != y |
| 10 | !== | 严格不相等 | x !== y |
|  |  |  |  |
| 9 | & | 按位与 | x & y |
| 8 | ^ | 按位 XOR | x ^ y |
| 7 | | | 按位或 | x | y |
| 6 | && | 逻辑与 | x && y |
| 5 | || | 逻辑否 | x || y |
| 4 | ? : | 条件 | ? "Yes" : "No" |
|  |  |  |  |
| 3 | = | 赋值 | x = y |
| 3 | += | 赋值 | x += y |
| 3 | -= | 赋值 | x -= y |
| 3 | \*= | 赋值 | x \*= y |
| 3 | %= | 赋值 | x %= y |
| 3 | <<= | 赋值 | x <<= y |
| 3 | >>= | 赋值 | x >>= y |
| 3 | >>>= | 赋值 | x >>>= y |
| 3 | &= | 赋值 | x &= y |
| 3 | ^= | 赋值 | x ^= y |
| 3 | |= | 赋值 | x |= y |
|  |  |  |  |
| 2 | yield | 暂停函数 | yield x |
| 1 | , | 逗号 | 7 , 8 |

注意：淡红色指示实验性或建议性的技术（ECMASScript 2016 或 ES7）

提示：括号中的表达式会在值在表达式的其余部分中被使用之前进行完全计算。

# 第13章 JavaScript 赋值

JavaScript 赋值运算符

赋值运算符向 JavaScript 变量赋值。

|  |  |  |
| --- | --- | --- |
| 运算符 | 例子 | 等同于 |
| = | x = y | x = y |
| += | x += y | x = x + y |
| -= | x -= y | x = x - y |
| \*= | x \*= y | x = x \* y |
| /= | x /= y | x = x / y |
| %= | x %= y | x = x % y |
| <<= | x <<= y | x = x << y |
| >>= | x >>= y | x = x >> y |
| >>>= | x >>>= y | x = x >>> y |
| &= | x &= y | x = x & y |
| ^= | x ^= y | x = x ^ y |
| |= | x |= y | x = x | y |
| \*\*= | x \*\*= y | x = x \*\* y |
|  |  |  |
|  |  |  |
|  |  |  |

提示：\*\*= 运算符属于 ECMAScript 2016 proposal (ES7) 的实验部分。它的跨浏览器表现并不稳定。请勿使用。

# 第14章 JavaScript 数据类型

字符串值，数值，布尔值，数组，对象。

JavaScript 数据类型

JavaScript 变量能够保存多种数据类型：数值、字符串值、数组、对象等等：

var length = 7; // 数字

var lastName = "Gates"; // 字符串

var cars = ["Porsche", "Volvo", "BMW"]; // 数组

var x = {firstName:"Bill", lastName:"Gates"}; // 对象

数据类型的概念

在编程过程中，数据类型是重要的概念。

为了能够操作变量，了解数据类型是很重要的。

如果没有数据类型，计算机就无法安全地解决这道题：

var x = 911 + "Porsche";

给 "Volvo" 加上 911 有意义吗？这么做会发生错误还是会产生一个结果？

JavaScript 会这样处理上面的例子：

var x = "911" + "Porsche";

当数值和字符串相加时，JavaScript 将把数值视作字符串。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数据类型</h2>

<p>当数值和字符串相加时，JavaScript 将把数值视作字符串。</p>

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

<script>

var x = 911 + "Porsche";

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

</script>

</body>

</html>

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数据类型</h2>

<p>当字符串和数值相加时，JavaScript 将把数值视作字符串。</p>

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

<script>

var x = "Porsche" + 911;

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

</script>

</body>

</html>

JavaScript 从左向右计算表达式。不同的次序会产生不同的结果：

JavaScript：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数据类型</h2>

<p>JavaScript 从左向右计算表达式。不同的次序会产生不同的结果：</p>

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

<script>

var x = 911 + 7 + "Porsche";

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

</script>

</body>

</html>

JavaScript 从左向右计算表达式。不同的次序会产生不同的结果：

JavaScript：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数据类型</h2>

<p>JavaScript 从左向右计算表达式。不同的次序会产生不同的结果：</p>

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

<script>

var x = 911 + 7 + "Porsche";

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

</script>

</body>

</html>

JavaScript：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数据类型</h2>

<p>JavaScript 从左向右计算表达式。不同的次序会产生不同的结果：</p>

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

<script>

var x = "Porsche" + 911 + 7;

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

</script>

</body>

</html>

在第一个例子中，JavaScript 把 911 和 7 视作数值，直到遇见 "Porsche"。

在第二个例子中，由于第一个操作数是字符串，因此所有操作数都被视为字符串。

JavaScript 拥有动态类型

JavaScript 拥有动态类型。这意味着相同变量可用作不同类型：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数据类型</h2>

<p>JavaScript 拥有动态类型。这意味着相同变量可用于保存不同类型：</p>

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

<script>

var x; // 现在 x 是 undefined

var x = 7; // 现在 x 是数值

var x = "Bill"; // 现在 x 是字符串值

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

</script>

</body>

</html>

JavaScript 字符串值

字符串（或文本字符串）是一串字符（比如 "Bill Gates"）。

字符串被引号包围。您可使用单引号或双引号：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 字符串</h2>

<p>字符串被引号包围。您可使用单引号或双引号：</p>

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

<script>

var carName1 = "Porsche 911"; // 使用双引号

var carName2 = 'Porsche 911'; // 使用单引号

document.getElementById("demo").innerHTML =

carName1 + "<br>" +

carName2;

</script>

</body>

</html>

您可以在字符串内使用引号，只要这些引号与包围字符串的引号不匹配：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 字符串</h2>

<p>您可以在字符串内使用引号，只要这些引号与包围字符串的引号不匹配：</p>

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

<script>

var answer1 = "It's alright"; // 双引号内的单引号

var answer2 = "He is called 'Bill'"; // 双引号内的单引号

var answer3 = 'He is called "Bill"'; // 单引号内的双引号

document.getElementById("demo").innerHTML =

answer1 + "<br>" +

answer2 + "<br>" +

answer3;

</script>

</body>

</html>

JavaScript 数值

JavaScript 只有一种数值类型。

写数值时用不用小数点均可：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数值</h2>

<p>写数值时用不用小数点均可：</p>

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

<script>

var x1 = 34.00;

var x2 = 34;

var x3 = 3.14;

document.getElementById("demo").innerHTML =

x1 + "<br>" + x2 + "<br>" + x3;

</script>

</body>

</html>

超大或超小的数值可以用科学计数法来写：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数值</h2>

<p>超大或超小的数值可以用科学计数法来写：</p>

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

<script>

var y = 123e5;

var z = 123e-5;

document.getElementById("demo").innerHTML =

y + "<br>" + z;

</script>

</body>

</html>

JavaScript 布尔值

布尔值只有两个值：true 或 false。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Booleans</h2>

<p>布尔值只有两个值：true 或 false。</p>

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

<script>

var x = 7;

var y = 7;

var z = 8;

document.getElementById("demo").innerHTML =

(x == y) + "<br>" + (x == z);

</script>

</body>

</html>

布尔值经常用在条件测试中。

您将在本教程中学到更多有关条件测试的知识。

JavaScript 数组

JavaScript 数组用方括号书写。

数组的项目由逗号分隔。

下面的代码声明（创建）了名为 cars 的数组，包含三个项目（汽车品牌）：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数组</h2>

<p>数组索引基于零，这意味着第一个项目是 [0]，第二个项目是 [1]，以此类推。</p>

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

<script>

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

document.getElementById("demo").innerHTML = cars[0];

</script>

</body>

</html>

数组索引基于零，这意味着第一个项目是 [0]，第二个项目是 [1]，以此类推。

您将在本教程中学到更多有关数组的知识。

JavaScript 对象

JavaScript 对象用花括号来书写。

对象属性是 name:value 对，由逗号分隔。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 对象</h2>

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

<script>

var person = {

firstName : "Bill",

lastName : "Gates",

age : 62,

eyeColor : "blue"

};

document.getElementById("demo").innerHTML =

person.firstName + " is " + person.age + " years old.";

</script>

</body>

</html>

上例中的对象（person）有四个属性：firstName、lastName、age 以及 eyeColor。

您将在本教程中学到更多有关对象的知识。

typeof 运算符

您可使用 JavaScript 的 typeof 来确定 JavaScript 变量的类型：

typeof 运算符返回变量或表达式的类型：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript typeof</h2>

<p>typeof 运算符返回变量或表达式的类型：</p>

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

<script>

document.getElementById("demo").innerHTML =

typeof "" + "<br>" +

typeof "Bill" + "<br>" +

typeof "Bill Gates";

</script>

</body>

</html>

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript typeof</h2>

<p>typeof 运算符返回变量或表达式的类型：</p>

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

<script>

document.getElementById("demo").innerHTML =

typeof 0 + "<br>" +

typeof 314 + "<br>" +

typeof 3.14 + "<br>" +

typeof (3) + "<br>" +

typeof (3 + 4);

</script>

</body>

</html>

typeof 运算符对数组返回 "object"，因为在 JavaScript 中数组属于对象。

Undefined

在 JavaScript 中，没有值的变量，其值是 undefined。typeof 也返回 undefined。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数据类型</h2>

<p>无值变量的值和数据类型是 <b>undefined</b>。</p>

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

<script>

var car;

document.getElementById("demo").innerHTML =

car + "<br>" + typeof car;

</script>

</body>

</html>

任何变量均可通过设置值为 undefined 进行清空。其类型也将是 undefined。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数据类型</h2>

<p>任何变量均可通过设置值为 undefined 进行清空。其类型也将是 undefined。</p>

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

<script>

var car = "Porsche";

car = undefined;

document.getElementById("demo").innerHTML =

car + "<br>" + typeof car;

</script>

</body>

</html>

空值

空值与 undefined 不是一回事。

空的字符串变量既有值也有类型。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数据类型</h2>

<p>空的字符串变量既有值也有类型。</p>

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

<script>

var car = "";

document.getElementById("demo").innerHTML =

"其值是：" +

car + "<br>" +

"类型是：" + typeof car;

</script>

</body>

</html>

Null

在 JavaScript 中，null 是 "nothing"。它被看做不存在的事物。

不幸的是，在 JavaScript 中，null 的数据类型是对象。

您可以把 null 在 JavaScript 中是对象理解为一个 bug。它本应是 null。

您可以通过设置值为 null 清空对象：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数据类型</h2>

<p>您可以通过设置值为 null 来清空对象：</p>

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

<script>

var person = {firstName:"Bill", lastName:"Gates", age:62, eyeColor:"blue"};

person = null;

document.getElementById("demo").innerHTML = typeof person;

</script>

</body>

</html>

您也可以通过设置值为 undefined 清空对象：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数据类型</h2>

<p>任何变量均可通过设置值为 undefined 进行清空。其类型也将是 undefined。</p>

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

<script>

var person = {firstName:"Bill", lastName:"Gates", age:62, eyeColor:"blue"};

person = undefined;

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

</script>

</body>

</html>

Undefined 与 Null 的区别

Undefined 与 null 的值相等，但类型不相等：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数据类型</h2>

<p>Undefined and null are equal in value but different in type:</p>

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

<script>

document.getElementById("demo").innerHTML =

typeof undefined + "<br>" +

typeof null + "<br><br>" +

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

(null == undefined);

</script>

</body>

</html>

原始数据

原始数据值是一种没有额外属性和方法的单一简单数据值。

typeof 运算符可返回以下原始类型之一：

string

number

boolean

undefined

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript typeof</h2>

<p>typeof 运算符返回变量或表达式的类型：</p>

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

<script>

document.getElementById("demo").innerHTML =

typeof "Bill" + "<br>" +

typeof 3.14 + "<br>" +

typeof true + "<br>" +

typeof false + "<br>" +

typeof x;

</script>

</body>

</html>

复杂数据

typeof 运算符可返回以下两种类型之一：

function

object

typeof 运算符把对象、数组或 null 返回 object。

typeof 运算符不会把函数返回 object。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript typeof</h2>

<p>typeof 运算符把对象、数组或 null 返回 object。</p>

<p>typeof 运算符不会把函数返回 object。</p>

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

<script>

document.getElementById("demo").innerHTML =

typeof {name:'Bill', age:62} + "<br>" +

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

typeof null + "<br>" +

typeof function myFunc(){};

</script>

</body>

</html>

typeof 运算符把数组返回为 "object"，因为在 JavaScript 中数组即对象。

# 第15章 JavaScript 函数

JavaScript 函数是被设计为执行特定任务的代码块。

JavaScript 函数会在某代码调用它时被执行。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 函数</h2>

<p>本例调用了一个执行计算的函数，然后返回结果：</p>

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

<script>

function myFunction(p1, p2) {

return p1 \* p2;

}

document.getElementById("demo").innerHTML = myFunction(7, 8);

</script>

</body>

</html>

JavaScript 函数语法

JavaScript 函数通过 function 关键词进行定义，其后是函数名和括号 ()。

函数名可包含字母、数字、下划线和美元符号（规则与变量名相同）。

圆括号可包括由逗号分隔的参数：

(参数 1, 参数 2, ...)

由函数执行的代码被放置在花括号中：{}

function name(参数 1, 参数 2, 参数 3) {

要执行的代码

}

函数参数（Function parameters）是在函数定义中所列的名称。

函数参数（Function arguments）是当调用函数时由函数接收的真实的值。

在函数中，参数是局部变量。

在其他编程语言中，函数近似程序（Procedure）或子程序（Subroutine）。

函数调用

函数中的代码将在其他代码调用该函数时执行：

当事件发生时（当用户点击按钮时）

当 JavaScript 代码调用时

自动的（自调用）

您将在本教程中学到更多有关函数调用的知识。

函数返回

当 JavaScript 到达 return 语句，函数将停止执行。

如果函数被某条语句调用，JavaScript 将在调用语句之后“返回”执行代码。

函数通常会计算出返回值。这个返回值会返回给调用者：

实例

计算两个数的乘积，并返回结果：

var x = myFunction(7, 8); // 调用函数，返回值被赋值给 x

function myFunction(a, b) {

return a \* b; // 函数返回 a 和 b 的乘积

}

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 函数</h2>

<p>本例调用了一个执行计算的函数，然后返回结果：</p>

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

<script>

var x = myFunction(7, 8);

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

function myFunction(a, b) {

return a \* b;

}

</script>

</body>

</html>

为何使用函数？

您能够对代码进行复用：只要定义一次代码，就可以多次使用它。

您能够多次向同一函数传递不同的参数，以产生不同的结果。

实例

把华氏度转换为摄氏度：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 函数</h2>

<p>本例调用函数把华氏度转换为摄氏度：</p>

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

<script>

function toCelsius(f) {

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

}

document.getElementById("demo").innerHTML = toCelsius(86);

</script>

</body>

</html>

() 运算符调用函数

使用上面的例子，toCelsius 引用的是函数对象，而 toCelsius() 引用的是函数结果。

实例

访问没有 () 的函数将返回函数定义：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 函数</h2>

<p>不使用 () 访问函数将返回函数声明而不是函数结果：</p>

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

<script>

function toCelsius(f) {

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

}

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

</script>

</body>

</html>

用作变量值的函数

函数的使用方法与变量一致，在所有类型的公式、赋值和计算中。

实例

使用变量来存储函数的值：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 函数</h2>

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

<script>

document.getElementById("demo").innerHTML =

"温度是：" + toCelsius(86) + " 摄氏度。";

function toCelsius(fahrenheit) {

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

}

</script>

</body>

</html>

局部变量

在 JavaScript 函数中声明的变量，会成为函数的局部变量。

局部变量只能在函数内访问。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 函数</h2>

<p>myFunction() 之外的 carName 未定义。</p>

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

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

<script>

myFunction();

function myFunction() {

var carName = "Volvo";

document.getElementById("demo1").innerHTML =

typeof carName + " " + carName;

}

document.getElementById("demo2").innerHTML =

typeof carName;

</script>

</body>

</html>

由于局部变量只能被其函数识别，因此可以在不同函数中使用相同名称的变量。

局部变量在函数开始时创建，在函数完成时被删除。

# 第16章 JavaScript 对象

真实生活中的对象、属性和方法

在真实生活中，汽车是一个对象。

汽车有诸如车重和颜色等属性，也有诸如启动和停止的方法：

所有汽车都拥有同样的属性，但属性值因车而异。

所有汽车都拥有相同的方法，但是方法会在不同时间被执行。

JavaScript 对象

您之前已经学到，JavaScript 变量是数据值的容器。

这段代码把一个单一值（porsche）赋给名为 car 的变量：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 变量</h2>

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

<script>

// 创建并显示变量：

var car = "porsche";

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

</script>

</body>

</html>

对象也是变量。但是对象包含很多值。

这段代码把多个值（porsche, 911, white）赋给名为 car 的变量：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象</h1>

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

<script>

// 创建对象：

var car = {type:"porsche", model:"911", color:"white"};

// 显示对象中的数据：

document.getElementById("demo").innerHTML = car.type;

</script>

</body>

</html>

值以名称:值对的方式来书写（名称和值由冒号分隔）。

JavaScript 对象是被命名值的容器。

对象属性

（JavaScript 对象中的）名称:值对被称为属性。

var person = {firstName:"Bill", lastName:"Gates", age:62, eyeColor:"blue"};

对象方法

对象也可以有方法。

方法是在对象上执行的动作。

方法以函数定义被存储在属性中。

方法是作为属性来存储的函数。

实例

var person = {

firstName: "Bill",

lastName : "Gates",

id : 678,

fullName : function() {

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

}

};

this 关键词

在函数定义中，this 引用该函数的“拥有者”。

在上面的例子中，this 指的是“拥有” fullName 函数的 person 对象。

换言之，this.firstName 的意思是 this 对象的 firstName 属性。

请在 JS this 关键词这一章学习更多有关 this 关键词的知识。

对象定义

我们定义（创建）了一个 JavaScript 对象：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象创建</h1>

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

<script>

// 创建对象：

var person = {firstName:"Bill", lastName:"Gates", age:62, eyeColor:"blue"};

// 显示对象中的数据：

document.getElementById("demo").innerHTML =

person.firstName + " 已经 " + person.age + " 岁了。";

</script>

</body>

</html>

空格和折行都是允许的。对象定义可横跨多行：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象创建</h1>

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

<script>

// 创建对象：

var person = {

firstName : "Bill",

lastName : "Gates",

age : 62,

eyeColor : "blue"

};

// 显示对象中的数据：

document.getElementById("demo").innerHTML =

person.firstName + " 已经 " + person.age + " 岁了。";

</script>

</body>

</html>

访问对象属性

您能够以两种方式访问属性：

objectName.propertyName

或者

objectName["propertyName"]

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象属性</h1>

<p>有两种不同的方法来访问对象属性。</p>

<p>您可以使用 person.property 或 person["property"]。</p>

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

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

id : 12345

};

// 显示对象中的数据：

document.getElementById("demo").innerHTML =

person.firstName + " " + person.lastName;

</script>

</body>

</html>

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象属性</h1>

<p>有两种不同的方法来访问对象属性。</p>

<p>您可以使用 person.property 或 person["property"]。</p>

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

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

id : 12345

};

// 显示对象中的数据：

document.getElementById("demo").innerHTML =

person["firstName"] + " " + person["lastName"];

</script>

</body>

</html>

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象方法</h1>

<p>对象方法是一种函数定义，存储为属性值。</p>

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

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

id : 12345,

fullName : function() {

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

}

};

// 显示对象中的数据：

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

</script>

</body>

</html>

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象方法</h1>

<p>对象方法是一种函数定义，存储为属性值。</p>

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

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

id : 12345,

fullName : function() {

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

}

};

// 显示对象中的数据：

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

</script>

</body>

</html>

如果您不使用 () 访问 fullName 方法，则将返回函数定义：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象方法</h1>

<p>如果您不使用 () 访问对象，则返回函数定义：</p>

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

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

id : 12345,

fullName : function() {

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

}

};

// 显示对象中的数据：

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

</script>

</body>

</html>

方法实际上是以属性值的形式存储的函数定义。

请不要把字符串、数值和布尔值声明为对象！

如果通过关键词 "new" 来声明 JavaScript 变量，则该变量会被创建为对象：

var x = new String(); // 把 x 声明为 String 对象

var y = new Number(); // 把 y 声明为 Number 对象

var z = new Boolean(); // 把 z 声明为 Boolean 对象

请避免字符串、数值或逻辑对象。他们会增加代码的复杂性并降低执行速度。

您将在本教程的稍后章节学到更多有关对象的知识。

# 第17章 JavaScript 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 能够“应对”这些事件。

HTML 事件

HTML 事件可以是浏览器或用户做的某些事情。

下面是 HTML 事件的一些例子：

HTML 网页完成加载

HTML 输入字段被修改

HTML 按钮被点击

通常，当事件发生时，用户会希望做某件事。

JavaScript 允许您在事件被侦测到时执行代码。

通过 JavaScript 代码，HTML 允许您向 HTML 元素添加事件处理程序。

使用单引号：

<element event='一些 JavaScript'>

使用双引号：

<element event="一些 JavaScript">

在下面的例子中，onclick 属性（以及代码）被添加到 <button> 元素：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 事件</h1>

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

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

</body>

</html>

在上面的例子中，JavaScript 代码改变了 id="demo" 的元素的内容。

在接下来的例子中，代码（使用 this.innerHTML）改变了其自身元素的内容：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 事件</h1>

<button onclick="this.innerHTML=Date()">时间是？</button>

</body>

</html>

JavaScript 代码通常有很多行。事件属性调用函数更为常见：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 事件</h1>

<p>点击按钮来显示日期。</p>

<button onclick="displayDate()">时间是？</button>

<script>

function displayDate() {

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

}

</script>

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

</body>

</html>

常见的 HTML 事件

下面是一些常见的 HTML 事件：

|  |  |
| --- | --- |
| 事件 | 描述 |
| onchange | HTML 元素已被改变 |
| onclick | 用户点击了 HTML 元素 |
| onmouseover | 用户把鼠标移动到 HTML 元素上 |
| onmouseout | 用户把鼠标移开 HTML 元素 |
| onkeydown | 用户按下键盘按键 |
| onload | 浏览器已经完成页面加载 |

JavaScript 能够做什么？

事件处理程序可用于处理、验证用户输入、用户动作和浏览器动作：

每当页面加载时应该做的事情

当页面被关闭时应该做的事情

当用户点击按钮时应该被执行的动作

当用户输入数据时应该被验证的内容

等等

让 JavaScript 处理事件的不同方法有很多：

HTML 事件属性可执行 JavaScript 代码

HTML 事件属性能够调用 JavaScript 函数

您能够向 HTML 元素分配自己的事件处理函数

您能够阻止事件被发送或被处理

等等

您将在 HTML DOM 章节中学到更多有关事件和事件处理程序的知识。

# 第18章 JavaScript 字符串

JavaScript 字符串用于存储和操作文本。

JavaScript 字符串

JavaScript 字符串是引号中的零个或多个字符。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 字符串</h2>

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

<script>

var x = "Bill Gates"; // 用引号包围的字符串

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

</script>

</body>

</html>

您能够使用单引号或双引号：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 字符串</h2>

<p>字符串在引号中书写。您能够使用单引号或双引号：</p>

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

<script>

var carName1 = "Porsche 911"; // 双引号

var carName2 = 'Porsche 911'; // 单引号

document.getElementById("demo").innerHTML =

carName1 + " " + carName2;

</script>

</body>

</html>

您可以在字符串中使用引号，只要不匹配围绕字符串的引号即可：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 字符串</h2>

<p>您可以在字符串中使用引号，只要不匹配围绕字符串的引号即可：</p>

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

<script>

var answer1 = "It's good to see you again!";

var answer2 = "He is called 'Bill'";

var answer3 = 'He is called "Bill"';

document.getElementById("demo").innerHTML =

answer1 + "<br>" + answer2 + "<br>" + answer3;

</script>

</body>

</html>

字符串长度

内建属性 length 可返回字符串的长度：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串属性</h1>

<p>length 属性返回字符串的长度：</p>

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

<script>

var txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

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

</script>

</body>

</html>

特殊字符

由于字符串必须由引号包围，JavaScript 会误解这段字符串：

var y = "中国是瓷器的故乡，因此 china 与"China（中国）"同名。"

该字符串将被切为 "中国是瓷器的故乡，因此 china 与"。

避免此问题的解决方法是，使用 \ 转义字符。

反斜杠转义字符把特殊字符转换为字符串字符：

|  |  |  |
| --- | --- | --- |
| 代码 | 结果 | 描述 |
| \' | ' | 单引号 |
| \" | " | 双引号 |
| \\ | \ | 反斜杠 |

实例

序列 \" 在字符串中插入双引号：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串</h1>

<p>转义序列 \" 在字符串中插入双引号。</p>

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

<script>

var x = "中国是瓷器的故乡，因此 china 与\"China（中国）\"同名。";

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

</script>

</body>

</html>

序列 \' 在字符串中插入单引号：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串</h1>

<p>转义序列 \' 在字符串中插入单引号。</p>

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

<script>

var x = 'It\' good to see you!';

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

</script>

</body>

</html>

序列 \\ 在字符串中插入反斜杠：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串</h1>

<p>转义序列 \\ 在字符串中插入反斜杠。</p>

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

<script>

var x = "字符 \\ 被称为反斜杠。";

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

</script>

</body>

</html>

转义字符（\）也可用于在字符串中插入其他特殊字符。

其他六个 JavaScript 中有效的转义序列：

|  |  |
| --- | --- |
| 代码 | 结果 |
| \b | 退格键 |
| \f | 换页 |
| \n | 新行 |
| \r | 回车 |
| \t | 水平制表符 |
| \v | 垂直制表符 |

这六个转义字符最初设计用于控制打字机、电传打字机和传真机。它们在 HTML 中没有任何意义。

长代码行换行

为了最佳可读性， 程序员们通常会避免每行代码超过 80 个字符串。

如果某条 JavaScript 语句不适合一整行，那么最佳换行位置是某个运算符之后：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 语句</h2>

<p>

折行的最佳位置是运算符或逗号之后。

</p>

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

<script>

document.getElementById("demo").innerHTML =

"Hello Kitty.";

</script>

</body>

</html>

您也可以在字符串中换行，通过一个反斜杠即可：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串</h1>

<p>您可以用反斜杠在文本字符串中进行换行。</p>

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

<script>

document.getElementById("demo").innerHTML = "Hello \

Kitty";

</script>

</body>

</html>

\ 方法并不是 ECMAScript (JavaScript) 标准。

某些浏览器也不允许 \ 字符之后的空格。

对长字符串换行的最安全做法（但是有点慢）是使用字符串加法：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串</h1>

<p>在字符串中进行换行的最安全方法是使用字符串相加。</p>

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

<script>

document.getElementById("demo").innerHTML = "Hello " +

"Kitty";

</script>

</body>

</html>

您不能通过反斜杠对代码行进行换行：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 语句</h1>

<p id="demo">您不能用反斜杠对代码行进行折行。</p>

<script>

document.getElementById("demo").innerHTML = \

"Hello Kitty";

</script>

</body>

</html>

字符串可以是对象

通常，JavaScript 字符串是原始值，通过字面方式创建：

var firstName = "Bill"

但是字符串也可通过关键词 new 定义为对象：

var firstName = new String("Bill")

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串</h1>

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

<script>

var x = "Bill"; // x 是字符串

var y = new String("Bill"); // y 是对象

document.getElementById("demo").innerHTML =

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

</script>

</body>

</html>

请不要把字符串创建为对象。它会拖慢执行速度。

new 关键字使代码复杂化。也可能产生一些意想不到的结果：

当使用 == 相等运算符时，相等字符串是相等的：

实例

<!DOCTYPE html>

<html>

<body>

<h1>绝不要把字符串创建为对象</h1>

<p>字符串与对象无法安全地比较。</p>

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

<script>

var x = "Bill"; // x 是字符串

var y = new String("Bill"); // y 是对象

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

</script>

</body>

</html>

当使用 === 运算符时，相等字符串是不相等的，因为 === 运算符需要类型和值同时相等。

实例

<!DOCTYPE html>

<html>

<body>

<h1>绝不要把字符串创建为对象</h1>

<p>JavaScript 对象无法比较。</p>

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

<script>

var x = "Bill"; // x 是字符串

var y = new String("Bill"); // y 是对象

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

</script>

</body>

</html>

甚至更糟。对象无法比较：

实例

<!DOCTYPE html>

<html>

<body>

<h1>绝不要把字符串创建为对象</h1>

<p>JavaScript 对象无法比较。</p>

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

<script>

var x = new String("Bill"); // x 是对象

var y = new String("Bill"); // y 也是对象

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

</script>

</body>

</html>

<!DOCTYPE html>

<html>

<body>

<h1>绝不要把字符串创建为对象</h1>

<p>JavaScript 对象无法比较。</p>

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

<script>

var x = new String("Bill"); // x 是对象

var y = new String("Bill"); // y 也是对象

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

</script>

</body>

</html>

请注意 (x==y) 与 (x===y) 的区别。

JavaScript 对象无法进行对比，比较两个 JavaScript 将始终返回 false。

# 第19章 JavaScript 字符串方法

字符串方法帮助您处理字符串。

字符串方法和属性

原始值，比如“Bill Gates”，无法拥有属性和方法（因为它们不是对象）。

但是通过 JavaScript，方法和属性也可用于原始值，因为在执行方法和属性时 JavaScript 将原始值视为对象。

字符串长度

length 属性返回字符串的长度：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串属性</h1>

<p>length 属性返回字符串的长度：</p>

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

<script>

var txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

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

</script>

</body>

</html>

查找字符串中的字符串

indexOf() 方法返回字符串中指定文本首次出现的索引（位置）：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>indexOf() 方法返回指定文本首次出现的位置：</p>

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

<script>

var str = "The full name of China is the People's Republic of China.";

var pos = str.indexOf("China");

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

</script>

</body>

</html>

JavaScript 从零计算位置。

0 是字符串中的第一个位置，1 是第二个，2 是第三个 ...

lastIndexOf() 方法返回指定文本在字符串中最后一次出现的索引：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>lastIndexOf() 返回指定文本最后出现的位置：</p>

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

<script>

var str = "The full name of China is the People's Republic of China.";

var pos = str.lastIndexOf("China");

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

</script>

</body>

</html>

如果未找到文本， indexOf() 和 lastIndexOf() 均返回 -1。

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>如果未找到文本，则 indexOf() 和 lastIndexOf() 都返回 -1：</p>

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

<script>

var str = "The full name of China is the People's Republic of China.";

var pos = str.indexOf("USA");

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

</script>

</body>

</html>

两种方法都接受作为检索起始位置的第二个参数。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>indexOf() 方法接受第二个参数作为搜索的起始位置：</p>

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

<script>

var str = "The full name of China is the People's Republic of China.";

var pos = str.indexOf("China",18);

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

</script>

</body>

</html>

lastIndexOf() 方法向后进行检索（从尾到头），这意味着：假如第二个参数是 50，则从位置 50 开始检索，直到字符串的起点。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>lastIndexOf() 方法接受第二个参数作为搜索的起始位置：</p>

请记住，lastIndexOf（）方法向后搜索，因此位置 50 表示在第 50 位开始搜索，并搜索到开头。

<p>位置 50 指的是从开头算起的位置 50。</p>

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

<script>

var str = "The full name of China is the People's Republic of China.";

var pos = str.lastIndexOf("China", 50);

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

</script>

</body>

</html>

检索字符串中的字符串

search() 方法搜索特定值的字符串，并返回匹配的位置：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>search() 方法返回字符串中指定文本第一次出现的位置：</p>

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

<script>

var str = "The full name of China is the People's Republic of China.";

var pos = str.search("China");

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

</script>

</body>

</html>

您注意到了吗？

两种方法，indexOf() 与 search()，是相等的。

这两种方法是不相等的。区别在于：

search() 方法无法设置第二个开始位置参数。

indexOf() 方法无法设置更强大的搜索值（正则表达式）。

您将在正则表达式的章节学习到这些更强大的检索值。

提取部分字符串

有三种提取部分字符串的方法：

slice(start, end)

substring(start, end)

substr(start, length)

slice() 方法

slice() 提取字符串的某个部分并在新字符串中返回被提取的部分。

该方法设置两个参数：起始索引（开始位置），终止索引（结束位置）。

这个例子裁剪字符串中位置 7 到位置 13 的片段：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>slice() 提取字符串的某个部分，并在新字符串中返回被提取的部分。</p>

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

<script>

var str = "Apple, Banana, Mango";

var res = str.slice(7,13);

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

</script>

</body>

</html>

如果某个参数为负，则从字符串的结尾开始计数。

这个例子裁剪字符串中位置 -12 到位置 -6 的片段：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>slice() 提取字符串的某个部分，并在新字符串中返回被提取的部分。</p>

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

<script>

var str = "Apple, Banana, Mango";

var res = str.slice(-13,-7);

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

</script>

</body>

</html>

如果省略第二个参数，则该方法将裁剪字符串的剩余部分：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>slice() 提取字符串的某个部分，并在新字符串中返回被提取的部分。</p>

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

<script>

var str = "Apple, Banana, Mango";

var res = str.slice(7);

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

</script>

</body>

</html>

或者从结尾计数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>slice() 提取字符串的某个部分，并在新字符串中返回被提取的部分。</p>

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

<script>

var str = "Apple, Banana, Mango";

var res = str.slice(-13)

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

</script>

</body>

</html>

提示：负值位置不适用 Internet Explorer 8 及其更早版本。

substring() 方法

substring() 类似于 slice()。

不同之处在于 substring() 无法接受负的索引。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>substring() 提取字符串的某个部分，并在新字符串中返回被提取的部分。</p>

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

<script>

var str = "Apple, Banana, Mango";

var res = str.substring(7,13);

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

</script>

</body>

</html>

如果省略第二个参数，则该 substring() 将裁剪字符串的剩余部分。

substr() 方法

substr() 类似于 slice()。

不同之处在于第二个参数规定被提取部分的长度。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>substr() 提取字符串的某个部分，并在新字符串中返回被提取的部分。</p>

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

<script>

var str = "Apple, Banana, Mango";

var res = str.substr(7,6);

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

</script>

</body>

</html>

如果省略第二个参数，则该 substr() 将裁剪字符串的剩余部分。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>substr() 提取字符串的某个部分，并在新字符串中返回被提取的部分。</p>

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

<script>

var str = "Apple, Banana, Mango";

var res = str.substr(7);

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

</script>

</body>

</html>

如果首个参数为负，则从字符串的结尾计算位置。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>substr() 提取字符串的某个部分，并在新字符串中返回被提取的部分。</p>

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

<script>

var str = "Apple, Banana, Mango";

var res = str.substr(-5);

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

</script>

</body>

</html>

第二个参数不能为负，因为它定义的是长度。

替换字符串内容

replace() 方法用另一个值替换在字符串中指定的值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>请把下面的段落中的“Microsoft”替换为“W3School”：</p>

<button onclick="myFunction()">试一试</button>

<p id="demo">请访问 Microsoft！</p>

<script>

function myFunction() {

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

var txt = str.replace("Microsoft","W3School");

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

}

</script>

</body>

</html>

replace() 方法不会改变调用它的字符串。它返回的是新字符串。

默认地，replace() 只替换首个匹配：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>请把下面的段落中的“Microsoft”替换为“W3School”：</p>

<button onclick="myFunction()">试一试</button>

<p id="demo">请访问 Microsoft 和 Microsoft！</p>

<script>

function myFunction() {

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

var txt = str.replace("Microsoft","W3School");

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

}

</script>

</body>

</html>

默认地，replace() 对大小写敏感。因此不对匹配 MICROSOFT：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>请把下面的段落中的“Microsoft”替换为“W3School”：</p>

<button onclick="myFunction()">试一试</button>

<p id="demo">请访问 Microsoft！</p>

<script>

function myFunction() {

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

var txt = str.replace("MICROSOFT","W3School");

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

}

</script>

<p>此例无效，因为 MICROSOFT 为大写。</p>

</body>

</html>

如需执行大小写不敏感的替换，请使用正则表达式 /i（大小写不敏感）：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>请把下面的段落中的“Microsoft”替换为“W3School”：</p>

<button onclick="myFunction()">试一试</button>

<p id="demo">请访问 Microsoft！</p>

<script>

function myFunction() {

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

var txt = str.replace(/MICROSOFT/i,"W3School");

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

}

</script>

</body>

</html>

请注意正则表达式不带引号。

如需替换所有匹配，请使用正则表达式的 g 标志（用于全局搜索）：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>请把下面的段落中的“Microsoft”替换为“W3School”：</p>

<button onclick="myFunction()">试一试</button>

<p id="demo">请访问 Microsoft 和 Microsoft！</p>

<script>

function myFunction() {

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

var txt = str.replace(/Microsoft/g,"W3School");

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

}

</script>

</body>

</html>

您将在 JavaScript 正则表达式这一章学到更多有关正则表达式的内容。

转换为大写和小写

通过 toUpperCase() 把字符串转换为大写：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>把字符串转换为大写：</p>

<button onclick="myFunction()">试一试</button>

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

<script>

function myFunction() {

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

document.getElementById("demo").innerHTML = text.toUpperCase();

}

</script>

</body>

</html>

通过 toLowerCase() 把字符串转换为小写：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>把字符串转换为小写：</p>

<button onclick="myFunction()">试一试</button>

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

<script>

function myFunction() {

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

document.getElementById("demo").innerHTML = text.toLowerCase();

}

</script>

</body>

</html>

concat() 方法

concat() 连接两个或多个字符串：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>concat() 方法连接两个或多个字符串：</p>

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

<script>

var text1 = "Hello";

var text2 = "World!";

var text3 = text1.concat(" ",text2);

document.getElementById("demo").innerHTML = text3;

</script>

</body>

</html>

concat() 方法可用于代替加运算符。下面两行是等效的：

实例

var text = "Hello" + " " + "World!";

var text = "Hello".concat(" ","World!");

所有字符串方法都会返回新字符串。它们不会修改原始字符串。

正式地说：字符串是不可变的：字符串不能更改，只能替换。

String.trim()

trim() 方法删除字符串两端的空白符：

实例

var str = " Hello World! ";

alert(str.trim());

警告：Internet Explorer 8 或更低版本不支持 trim() 方法。

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript String.trim()</h1>

<p>点击这个按钮来输出已删除空白字符的字符串。</p>

<button onclick="myFunction()">试一试</button>

<p><strong>注释：</strong>Internet Explorer 8 或更低版本不支持 trim() 方法。</p>

<script>

function myFunction() {

var str = " Hello World! ";

alert(str.trim());

}

</script>

</body>

</html>

如需支持 IE 8，您可搭配正则表达式使用 replace() 方法代替：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript String.trim()</h1>

<p>IE 8 不支持 String.trim()。您可搭配正则表达式使用 replace() 方法代替：</p>

<script>

var str = " Hello World! ";

alert(str.replace(/^[\s\uFEFF\xA0]+|[\s\uFEFF\xA0]+$/g, ''));

</script>

</body>

</html>

您还可以使用上面的 replace 方案把 trim 函数添加到 JavaScript String.prototype：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript String.trim()</h1>

<p>IE 8 不支持 String.trim()。您可以使用 replace 方案把 trim 函数添加到 JavaScript String.prototype：</p>

<script>

if (!String.prototype.trim) {

String.prototype.trim = function () {

return this.replace(/^[\s\uFEFF\xA0]+|[\s\uFEFF\xA0]+$/g, '');

};

};

var str = " Hello World! ";

alert(str.trim());

</script>

</body>

</html>

提取字符串字符

这是两个提取字符串字符的安全方法：

charAt(position)

charCodeAt(position)

charAt() 方法

charAt() 方法返回字符串中指定下标（位置）的字符串：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>charAt() 方法返回字符串中指定位置的字符串：</p>

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

<script>

var str = "HELLO WORLD";

document.getElementById("demo").innerHTML = str.charAt(0);

</script>

</body>

</html>

charCodeAt() 方法

charCodeAt() 方法返回字符串中指定索引的字符 unicode 编码：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>charCodeAt() 方法返回字符串中指定位置的字符 unicode 编码：</p>

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

<script>

var str = "HELLO WORLD";

document.getElementById("demo").innerHTML = str.charCodeAt(0);

</script>

</body>

</html>

属性访问（Property Access）

ECMAScript 5 (2009) 允许对字符串的属性访问 [ ]：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>ECMAScript 5 允许对字符串的属性访问：</p>

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

<script>

var str = "HELLO WORLD";

document.getElementById("demo").innerHTML = str[0];

</script>

</body>

</html>

使用属性访问有点不太靠谱：

不适用 Internet Explorer 7 或更早的版本

它让字符串看起来像是数组（其实并不是）

如果找不到字符，[ ] 返回 undefined，而 charAt() 返回空字符串。

它是只读的。str[0] = "A" 不会产生错误（但也不会工作！）

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>ECMAScript 5 允许对字符串的属性访问，不过是只读的：</p>

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

<script>

var str = "HELLO WORLD";

str[0] = "A"; // 无效

document.getElementById("demo").innerHTML = str[0];

</script>

</body>

</html>

提示：如果您希望按照数组的方式处理字符串，可以先把它转换为数组。

把字符串转换为数组

可以通过 split() 将字符串转换为数组：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>单击“试一试”以显示在字符串拆分后的第一个数组元素。</p>

<button onclick="myFunction()">试一试</button>

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

<script>

function myFunction() {

var str = "a,b,c,d,e,f";

var arr = str.split(",");

document.getElementById("demo").innerHTML = arr[0];

}

</script>

</body>

</html>

如果省略分隔符，被返回的数组将包含 index [0] 中的整个字符串。

如果分隔符是 ""，被返回的数组将是间隔单个字符的数组：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

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

<script>

var str = "Hello";

var arr = str.split("");

var text = "";

var i;

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

text += arr[i] + "<br>"

}

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

</script>

</body>

</html>

# 第20章 JavaScript 字符串搜索

用于搜索字符串的 JavaScript 方法：

String.indexOf()

String.lastIndexOf()

String.startsWith()

String.endsWith()

String.indexOf()

indexOf() 方法返回指定文本在字符串中第一次出现（的位置）的索引：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript String 方法</h2>

<p>indexOf() 方法返回指定文本第一次出现的位置：</p>

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

<script>

let str = "Please locate where 'locate' occurs!";

document.getElementById("demo").innerHTML = str.indexOf("locate");

</script>

</body>

</html>

avaScript 从零开始计算位置。

0 是字符串中的第一个位置，1 是第二个，2 是第三个 ......

String.lastIndexOf()

lastIndexOf() 方法返回指定文本在字符串中最后一次出现的索引：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript String 方法</h2>

<p>lastIndexOf() 方法返回指定文本最后一次出现的位置：</p>

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

<script>

let str = "Please locate where 'locate' occurs!";

document.getElementById("demo").innerHTML = str.lastIndexOf("locate");

</script>

</body>

</html>

如果未找到文本，indexOf() 和 lastIndexOf() 都返回 -1：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript String 方法</h1>

<p>如果未找到文本，indexOf() 和 lastIndexOf() 都返回 -1：</p>

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

<script>

let str = "Please locate where 'locate' occurs!";

document.getElementById("demo").innerHTML = str.indexOf("Bill");

</script>

</body>

</html>

这两种方法都接受第二个参数作为搜索的开始位置：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript String 方法</h2>

<p>indexOf() 方法接受第二个参数作为搜索的起始位置：</p>

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

<script>

let str = "Please locate where 'locate' occurs!";

document.getElementById("demo").innerHTML = str.indexOf("locate",15);

</script>

</body>

</html>

lastIndexOf() 方法向后搜索（从末尾到开头），意思是：如果第二个参数是 15，则从位置 15 开始搜索，一直搜索到字符串的开头。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript String 方法</h2>

<p>lastIndexOf() 方法接受第二个参数作为搜索的起始位置。</p>

<p>请记住 lastIndexOf() 方法向后搜索，因此位置 15 表示从位置 15 开始搜索，并搜索到开头。</p>

<p>位置 15 指的是从开头算起的位置 15。</p>

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

<script>

let str = "Please locate where 'locate' occurs!";

document.getElementById("demo").innerHTML = str.lastIndexOf("locate", 15);;

</script>

</body>

</html>

String.search()

search() 方法在字符串中搜索指定值并返回匹配的位置：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript String 方法</h2>

<p>search() 方法返回指定文本在字符串中第一次出现的位置：</p>

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

<script>

let str = "Please locate where 'locate' occurs!";

document.getElementById("demo").innerHTML = str.search("locate");

</script>

</body>

</html>

您注意到了吗？

indexOf() 和 search() 这两个方法，相等吗？

它们接受相同的参数，并返回相同的值？

这两种方法并不相等。差别如下：

search() 方法不能接受第二个起始位置参数。

indexOf() 方法不能采用强大的搜索值（正则表达式）。

您将在后面的章节中学到有关正则表达式的更多内容。

String.match()

match() 方法根据正则表达式在字符串中搜索匹配项，并将匹配项作为 Array 对象返回。

实例 1

在字符串中搜索 "ain"：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 字符串搜索</h2>

<p>在字符串中搜索 "ain"：</p>

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

<script>

let text = "The rain in SPAIN stays mainly in the plain";

document.getElementById("demo").innerHTML = text.match(/ain/g);

</script>

</body>

</html>

请在 JS RegExp 一章中学习有关正则表达式的更多内容。

如果正则表达式不包含 g 修饰符（执行全局搜索），match() 方法将只返回字符串中的第一个匹配项。

语法

string.match(regexp)

regexp 必需。需要搜索的值，为正则表达式。

返回： 数组，包含匹配项，每个匹配项对应一个项目，若未找到匹配项，则为 null。

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 字符串搜索</h2>

<p>对 "ain" 执行不区分大小写的全局搜索：</p>

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

<script>

let text = "The rain in SPAIN stays mainly in the plain";

document.getElementById("demo").innerHTML = text.match(/ain/gi);

</script>

</body>

</html>

String.includes()

如果字符串包含指定值，includes() 方法返回 true。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 字符串搜索</h2>

<p>检查字符串是否包含 "world"：</p>

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

<p>Internet Explorer 不支持 includes() 方法。</p>

<script>

let text = "Hello world, welcome to the universe.";

document.getElementById("demo").innerHTML = text.includes("world");

</script>

</body>

</html>

语法

string.includes(searchvalue, start)

|  |  |
| --- | --- |
| searchvalue | 必需。需要搜索的字符串。 |
| start | 可选。默认为 0. 开始搜索的位置。 |
| 返回： | 如果字符串包含该值则返回 true，否则返回 false。 |
| JS 版本： | ES6 (2015) |

检查字符串是否包含 "world"，从位置 12 开始搜索：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 字符串搜索</h2>

<p>检查字符串是否包含 "world"：</p>

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

<p>Internet Explorer 不支持 includes() 方法。</p>

<script>

let text = "Hello world, welcome to the universe.";

document.getElementById("demo").innerHTML = text.includes("world", 12);

</script>

</body>

</html>

String.startsWith()

如果字符串以指定值开头，则 startsWith() 方法返回 true，否则返回 false：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 字符串</h2>

<p>检查字符串是否以 "Hello" 开头：</p>

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

<p>Internet Explorer 不支持 startsWith() 方法。</p>

<script>

let text = "Hello world, welcome to the universe.";

document.getElementById("demo").innerHTML = text.startsWith("Hello");

</script>

</body>

</html>

语法

string.startsWith(searchvalue, start)

参数值

|  |  |
| --- | --- |
| 参数 | 描述 |
| searchvalue | 必需。需要搜索的值。 |
| start | 可选。默认为 0。开始搜索的位置。 |

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 字符串</h2>

<p>startsWith() 方法。</p>

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

<p>Internet Explorer 不支持 startsWith() 方法。</p>

<script>

let text = "Hello world, welcome to the universe.";

document.getElementById("demo").innerHTML = text.startsWith("world", 6);

</script>

</body>

</html>

注释：startsWith() 方法区分大小写。

Internet Explorer 不支持 startsWith() 方法。

String.endsWith()

如果字符串以指定值结尾，则 endsWith() 方法返回 true，否则返回 false：

实例

检查字符串是否以 "Gates" 结尾：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 字符串</h2>

<p>检查字符串是否以 "Gates" 结尾：</p>

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

<p>Internet Explorer 不支持 endsWith() 方法。</p>

<script>

let text = "Bill Gates";

document.getElementById("demo").innerHTML = text.endsWith("Gates");

</script>

</body>

</html>

语法

string.endsWith(searchvalue, length)

参数值

|  |  |
| --- | --- |
| 参数 | 描述 |
| searchvalue | 必需。需要搜索的值。 |
| length | 可选。要搜索的长度。 |

检索以 "world" 结尾的字符串的前 11 个字符：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 字符串</h2>

<p>检查以 "world" 结尾的字符串的前 11 个字符：</p>

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

<p>Internet Explorer 不支持endsWith() 方法。</p>

<script>

let text = "Hello world, welcome to the universe.";

document.getElementById("demo").innerHTML = text.endsWith("world", 11);

</script>

</body>

</html>

注释：endsWith() 方法区分大小写。

Internet Explorer 不支持 endsWith() 方法。

# 第21章 JavaScript 字符串模板

同义词：

Template Literals

Template Strings

String Templates

Back-Tics 语法

Back-Tics 语法

模板字面量使用反引号 (``) 而不是引号 ("") 来定义字符串：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 模板字面量</h2>

<p>模板字面量使用反引号 (``) 而不是引号 ("") 来定义字符串：</p>

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

<p>Internet Explorer 不支持模板字面量。</p>

<script>

let text = `Hello world!`;

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

</script>

</body>

</html>

字符串内的引号

通过使用模板字面量，您可以在字符串中同时使用单引号和双引号：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 模板字面量</h2>

<p>通过使用反引号，您可以在字符串中同时使用单引号和双引号：</p>

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

<p>Internet Explorer 不支持模板字面量。</p>

<script>

let text = `He's often called "Johnny"`;

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

</script>

</body>

</html>

多行字符串

模板字面量允许多行字符串：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 模板字面量</h2>

<p>模板字面量允许多行字符串：</p>

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

<p>Internet Explorer 不支持模板字面量。</p>

<script>

let text =

`The quick

brown fox

jumps over

the lazy dog`;

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

</script>

</body>

</html>

插值

模板字面量提供了一种将变量和表达式插入字符串的简单方法。

该方法称为字符串插值（string interpolation）。

语法

${...}

变量替换

模板字面量允许字符串中的变量：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 模板字面量</h2>

<p>模板字面量允许字符串中的变量：</p>

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

<p>Internet Explorer 不支持模板字面量。</p>

<script>

let firstName = "Bill";

let lastName = "Gates";

let text = `Welcome ${firstName}, ${lastName}!`;

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

</script>

</body>

</html>

用真实值自动替换变量称为字符串插值。

表达式替换

模板字面量允许字符串中的表达式：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 模板字面量</h2>

<p>模板字面量允许字符串中的表达式：</p>

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

<p>Internet Explorer 不支持模板字面量。</p>

<script>

let price = 10;

let VAT = 0.25;

let total = `Total: ${(price \* (1 + VAT)).toFixed(2)}`;

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

</script>

</body>

</html>

用真实值自动替换表达式称为字符串插值。

HTML 模板

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 模板字面量</h2>

<p>模板字面量允许字符串中的变量：</p>

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

<p>Internet Explorer 不支持模板字面量。</p>

<script>

let header = "Templates Literals";

let tags = ["template literals", "javascript", "es6"];

let html = `<h2>${header}</h2><ul>`;

for (const x of tags) {

html += `<li>${x}</li>`;

}

html += `</ul>`;

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

</script>

</body>

</html>

# 第22章 JavaScript 数字

JavaScript 只有一种数值类型。

书写数值时带不带小数点均可。

JavaScript 数值

书写 JavaScript 数值既可以带小数点，也可以不带：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>书写 JavaScript 数值既可以带小数点，也可以不带：</p>

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

<script>

var x = 3.14;

var y = 3;

document.getElementById("demo").innerHTML = x + "<br>" + y;

</script>

</body>

</html>

超大或超小的数可通过科学计数法来写：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>超大或超小的数可通过科学（指数）计数法来写：</p>

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

<script>

var x = 123e5;

var y = 123e-5;

document.getElementById("demo").innerHTML = x + "<br>" + y;

</script>

</body>

</html>

JavaScript 数值始终是 64 位的浮点数

与许多其他编程语言不同，JavaScript 不会定义不同类型的数，比如整数、短的、长的、浮点的等等。

JavaScript 数值始终以双精度浮点数来存储，根据国际 IEEE 754 标准。

此格式用 64 位存储数值，其中 0 到 51 存储数字（片段），52 到 62 存储指数，63 位存储符号：

|  |  |  |
| --- | --- | --- |
| 值(aka Fraction/Mantissa) | 指数 | 符号 |
| 52 bits(0 - 51) | 11 bits (52 - 62) | 1 bit (63) |

精度

整数（不使用指数或科学计数法）会被精确到 15 位。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>整数（不使用指数或科学计数法）会被精确到 15 位：</p>

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

<script>

var x = 999999999999999;

var y = 9999999999999999;

document.getElementById("demo").innerHTML = x + "<br>" + y;

</script>

</body>

</html>

小数的最大数是 17 位，但是浮点的算数并不总是 100% 精准：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>浮点的算数并不总是 100% 精准：</p>

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

<script>

var x = 0.2 + 0.1;

document.getElementById("demo").innerHTML = "0.2 + 0.1 = " + x;

</script>

</body>

</html>

使用乘除法有助于解决上面的问题：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>浮点的算数并不总是 100% 精准：</p>

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

<p>但是乘法和除法是有用的：</p>

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

<script>

var x = 0.2 + 0.1;

document.getElementById("demo1").innerHTML = "0.2 + 0.1 = " + x;

var y = (0.2\*10 + 0.1\*10) / 10;

document.getElementById("demo2").innerHTML = "0.2 + 0.1 = " + y;

</script>

</body>

</html>

数字和字符串相加

警告！！

JavaScript 的加法和级联（concatenation）都使用 + 运算符。

数字用加法。字符串用级联。

如果您对两个数相加，结果将是一个数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>如果您对两个数相加，结果将是一个数：</p>

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

<script>

var x = 10;

var y = 20;

var z = x + y;

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

</script>

</body>

</html>

如果对两个字符串相加，结果将是一个字符串的级联：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>如果添加两个数字字符串，结果将是级联字符串：</p>

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

<script>

var x = "10";

var y = "20";

var z = x + y;

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

</script>

</body>

</html>

如果您对一个数和一个字符串相加，结果也是字符串级联：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>如果数字和数字字符串相加，结果将是级联字符串：</p>

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

<script>

var x = 10;

var y = "20";

var z = x + y;

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

</script>

</body>

</html>

如果您对一个字符串和一个数字相加，结果也是字符串级联：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>如果数字字符串和数字相加，结果将是级联字符串：</p>

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

<script>

var x = "10";

var y = 20;

document.getElementById("demo").innerHTML = "结果是：" + x + y;

</script>

</body>

</html>

常见的错误是认为结果应该是 30：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>一个常见的错误是期望这个结果为 30：</p>

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

<script>

var x = 10;

var y = 20;

document.getElementById("demo").innerHTML = "结果是：" + x + y;

</script>

</body>

</html>

常见的错误是认为结果应该是 102030：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>一个常见的错误是期望这个结果是 102030：</p>

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

<script>

var x = 10;

var y = 20;

var z = "30";

var result = x + y + z;

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

</script>

</body>

</html>

JavaScript 从左向右进行编译。

因为 x 和 y 都是数，10 + 20 将被相加。

因为 z 是字符串，30 + "30" 被级联。

数字字符串

JavaScript 字符串可以拥有数字内容：

var x = 100; // x 是数字

var y = "100"; // y 是字符串

在所有数字运算中，JavaScript 会尝试将字符串转换为数字：

该例如此运行：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>在相除时，JavaScript 会尝试把字符串转换为数字：</p>

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

<script>

var x = "100";

var y = "10";

var z = x / y;

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

</script>

</body>

</html>

该例也将如此运行：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>在相乘时，JavaScript 会尝试把字符串转换为数字：</p>

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

<script>

var x = "100";

var y = "10";

var z = x \* y;

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

</script>

</body>

</html>

该例如此运行：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>在相减时，JavaScript 会尝试把字符串转换为数字：</p>

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

<script>

var x = "100";

var y = "10";

var z = x - y;

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

</script>

</body>

</html>

但是该例不会如上例般运行：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>在相加时，JavaScript 不会把字符串转换为数字：</p>

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

<script>

var x = "100";

var y = "10";

var z = x + y;

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

</script>

</body>

</html>

在最后一个例子中，JavaScript 用 + 运算符对字符串进行了级联。

NaN - 非数值

NaN 属于 JavaScript 保留词，指示某个数不是合法数。

尝试用一个非数字字符串进行除法会得到 NaN（Not a Number）：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>用非数字字符串进行除法会得到 NaN（Not a Number）：</p>

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

<script>

document.getElementById("demo").innerHTML = 100 / "Apple";

</script>

</body>

</html>

不过，假如字符串包含数值，则结果将是数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>数字除以数字字符串将是数字：</p>

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

<script>

document.getElementById("demo").innerHTML = 100 / "10";

</script>

</body>

</html>

您可使用全局 JavaScript 函数 isNaN() 来确定某个值是否是数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>您可使用全局 JavaScript 函数 isNaN() 来确定某个值是否是数：</p>

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

<script>

var x = 100 / "Apple";

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

</script>

</body>

</html>

要小心 NaN。假如您在数学运算中使用了 NaN，则结果也将是 NaN：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>假如您在数学运算中使用了 NaN，则结果也将是 NaN：</p>

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

<script>

var x = NaN;

var y = 5;

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

</script>

</body>

</html>

结果也许是串连接：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>假如您在数学运算中使用了 NaN，则结果可能是级联：</p>

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

<script>

var x = NaN;

var y = "5";

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

</script>

</body>

</html>

NaN 是数，typeof NaN 返回 number：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>NaN 是数，typeof NaN 返回 number：</p>

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

<script>

var x = NaN;

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

</script>

</body>

</html>

Infinity

Infinity （或 -Infinity）是 JavaScript 在计算数时超出最大可能数范围时返回的值。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>Infinity （或 -Infinity）是 JavaScript 在计算数时超出最大可能数范围时返回的值：</p>

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

<script>

var myNumber = 2;

var txt = "";

while (myNumber != Infinity) {

myNumber = myNumber \* myNumber;

txt = txt + myNumber + "<br>";

}

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

</script>

</body>

</html>

除以 0（零）也会生成 Infinity：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>除以 0（零）也会生成 Infinity：</p>

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

<script>

var x = 2/0;

var y = -2/0;

document.getElementById("demo").innerHTML = x + "<br>" + y;

</script>

</body>

</html>

Infinity 是数：typeOf Infinity 返回 number。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>Infinity 是数字：</p>

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

<script>

var x = Infinity;

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

</script>

</body>

</html>

十六进制

JavaScript 会把前缀为 0x 的数值常量解释为十六进制。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>前缀为 0x 的数值常量会被解释为十六进制：</p>

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

<script>

var x = 0xFF;

document.getElementById("demo").innerHTML = "0xFF = " + x;

</script>

</body>

</html>

绝不要用前导零写数字（比如 07）。

一些 JavaScript 版本会把带有前导零的数解释为八进制。

默认地，Javascript 把数显示为十进制小数。

但是您能够使用 toString() 方法把数输出为十六进制、八进制或二进制。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>toString() 方法以基数 2 到 36 输出字符串：</p>

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

<script>

var myNumber = 32;

document.getElementById("demo").innerHTML =

"32 = " + "<br>" +

" Decimal " + myNumber.toString(10) + "<br>" +

" Hexadecimal " + myNumber.toString(16) + "<br>" +

" Octal " + myNumber.toString(8) + "<br>" +

" Binary " + myNumber.toString(2);

</script>

</body>

</html>

数值可以是对象

通常 JavaScript 数值是通过字面量创建的原始值：var x = 123

但是也可以通过关键词 new 定义为对象：var y = new Number(123)

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>数字可以是对象，但是没有必要把数字创建为对象。</p>

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

<script>

var x = 123;

var y = new Number(123);

document.getElementById("demo").innerHTML = typeof x + "<br>" + typeof y;

</script>

</body>

</html>

请不要创建数值对象。这样会拖慢执行速度。

new 关键词使代码复杂化，并产生某些无法预料的结果：

当使用 == 相等运算符时，相等的数看上去相等：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>绝对不要把数字创建为对象。</p>

<p>数字和对象无法安全地比较。</p>

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

<script>

var x = 500; // x 是数字

var y = new Number(500); // y 是对象

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

</script>

</body>

</html>

当使用 === 相等运算符后，相等的数变为不相等，因为 === 运算符需要类型和值同时相等。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>绝对不要把数字创建为对象。</p>

<p>数字和对象无法安全地比较。</p>

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

<script>

var x = 500; // x 是数字

var y = new Number(500); // y 是对象

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

</script>

</body>

</html>

甚至更糟。对象无法进行对比：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>绝对不要把数字创建为对象。</p>

<p>数字和对象无法安全地比较。</p>

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

<script>

var x = new Number(500); // x 是对象

var y = new Number(500); // y 是对象

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

</script>

</body>

</html>

# 第23章 JavaScript 数字方法

Number 方法帮助您处理数值。

Number 方法和属性

原始值（比如 3.14 或 2016），无法拥有属性和方法（因为它们不是对象）。

但是通过 JavaScript，方法和属性也可用于原始值，因为 JavaScript 在执行方法和属性时将原始值视作对象。

toString() 方法

toString() 以字符串返回数值。

所有数字方法可用于任意类型的数字（字面量、变量或表达式）：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>toString() 方法以基数 2 到 36 输出字符串：</p>

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

<script>

var myNumber = 32;

document.getElementById("demo").innerHTML =

"32 = " + "<br>" +

" Decimal " + myNumber.toString(10) + "<br>" +

" Hexadecimal " + myNumber.toString(16) + "<br>" +

" Octal " + myNumber.toString(8) + "<br>" +

" Binary " + myNumber.toString(2);

</script>

</body>

</html>

toExponential() 方法

toExponential() 返回字符串值，它包含已被四舍五入并使用指数计数法的数字。

参数定义小数点后的字符数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字方法</h1>

<p>toExponential() 方法返回一个字符串，包含进行舍入的数字并使用指数表示法。</p>

<p>可选参数定义小数点后面的位数。</p>

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

<script>

var x = 9.656;

document.getElementById("demo").innerHTML =

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

x.toExponential(2) + "<br>" +

x.toExponential(4) + "<br>" +

x.toExponential(6);

</script>

</body>

</html>

该参数是可选的。如果您没有设置它，JavaScript 不会对数字进行舍入。

toFixed() 方法

toFixed() 返回字符串值，它包含了指定位数小数的数字：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字方法</h1>

<p>toFixed() 方法将数字四舍五入为给定的位数。</p>

<p>对于处理金钱，toFixed(2) 是完美的。</p>

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

<script>

var x = 9.656;

document.getElementById("demo").innerHTML =

x.toFixed(0) + "<br>" +

x.toFixed(2) + "<br>" +

x.toFixed(4) + "<br>" +

x.toFixed(6);

</script>

</body>

</html>

toFixed(2) 非常适合处理金钱。

toPrecision() 方法

toPrecision() 返回字符串值，它包含了指定长度的数字：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字方法</h1>

<p>toPrecision() 方法返回一个字符串，其中包含一个用指定长度写的数字：</p>

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

<script>

var x = 9.656;

document.getElementById("demo").innerHTML =

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

x.toPrecision(2) + "<br>" +

x.toPrecision(4) + "<br>" +

x.toPrecision(6);

</script>

</body>

</html>

valueOf() 方法

valueOf() 以数值返回数值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字方法</h1>

<p>valueOf() 把数字返回为数字：</p>

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

<script>

var x = 123;

document.getElementById("demo").innerHTML =

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

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

(100 + 23).valueOf();

</script>

</body>

</html>

在 JavaScript 中，数字可以是原始值（typeof = number）或对象（typeof = object）。

在 JavaScript 内部使用 valueOf() 方法可将 Number 对象转换为原始值。

没有理由在代码中使用它。

所有 JavaScript 数据类型都有 valueOf() 和 toString() 方法。

把变量转换为数值

这三种 JavaScript 方法可用于将变量转换为数字：

Number() 方法

parseInt() 方法

parseFloat() 方法

这些方法并非数字方法，而是全局 JavaScript 方法。

全局方法

JavaScript 全局方法可用于所有 JavaScript 数据类型。

这些是在处理数字时最相关的方法：

|  |  |
| --- | --- |
| 方法 | 描述 |
| Number() | 返回数字，由其参数转换而来。 |
| parseFloat() | 解析其参数并返回浮点数。 |
| parseInt() | 解析其参数并返回整数。 |

Number() 方法

Number() 可用于把 JavaScript 变量转换为数值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 全局方法</h1>

<p>Number() 方法把变量转换为数字：</p>

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

<script>

document.getElementById("demo").innerHTML =

Number(true) + "<br>" +

Number(false) + "<br>" +

Number("10") + "<br>" +

Number(" 10") + "<br>" +

Number("10 ") + "<br>" +

Number(" 10 ") + "<br>" +

Number("10.33") + "<br>" +

Number("10,33") + "<br>" +

Number("10 33") + "<br>" +

Number("Bill");

</script>

</body>

</html>

如果无法转换数字，则返回 NaN。

用于日期的 Number() 方法

Number() 还可以把日期转换为数字：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 全局方法</h1>

<p>Number() 方法可以把日期转换为数字：</p>

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

<script>

var x = new Date("2017-09-30");

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

</script>

</body>

</html>

上面的 Number() 方法返回 1970 年 1 月 1 日至今的毫秒数。

parseInt() 方法

parseInt() 解析一段字符串并返回数值。允许空格。只返回首个数字：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 全局函数</h1>

<p>全局 JavaScript 函数 parseInt() 把字符串转换为数字：</p>

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

<script>

document.getElementById("demo").innerHTML =

parseInt("10") + "<br>" +

parseInt("10.33") + "<br>" +

parseInt("10 6") + "<br>" +

parseInt("10 years") + "<br>" +

parseInt("years 10");

</script>

</body>

</html>

如果无法转换为数值，则返回 NaN (Not a Number)。

parseFloat() 方法

parseFloat() 解析一段字符串并返回数值。允许空格。只返回首个数字：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 全局方法</h1>

<p>parseFloat() 方法把字符串转换为数字：</p>

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

<script>

document.getElementById("demo").innerHTML =

parseFloat("10") + "<br>" +

parseFloat("10.33") + "<br>" +

parseFloat("10 6") + "<br>" +

parseFloat("10 years") + "<br>" +

parseFloat("years 10");

</script>

</body>

</html>

如果无法转换为数值，则返回 NaN (Not a Number)。

数值属性

|  |  |
| --- | --- |
| 属性 | 描述 |
| MAX\_VALUE | 返回 JavaScript 中可能的最大数。 |
| MIN\_VALUE | 返回 JavaScript 中可能的最小数。 |
| NEGATIVE\_INFINITY | 表示负的无穷大（溢出返回）。 |
| NaN | 表示非数字值（"Not-a-Number"）。 |
| POSITIVE\_INFINITY | 表示无穷大（溢出返回）。 |

JavaScript MIN\_VALUE 和 MAX\_VALUE

MAX\_VALUE 返回 JavaScript 中可能的最大数字。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Number 对象属性</h1>

<p>MAX\_VALUE 返回 JavaScript 中可能的最大值。</p>

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

<script>

var x = Number.MAX\_VALUE;

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

</script>

</body>

</html>

MIN\_VALUE 返回 JavaScript 中可能的最小数字。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Number 对象属性</h1>

<p>MIN\_VALUE 返回 JavaScript 中可能的最小数字。</p>

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

<script>

var x = Number.MIN\_VALUE;

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

</script>

</body>

</html>

JavaScript POSITIVE\_INFINITY

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Number 对象属性</h1>

<p>POSITIVE\_INFINITY</p>

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

<script>

var x = Number.POSITIVE\_INFINITY;

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

</script>

</body>

</html>

溢出时返回 POSITIVE\_INFINITY：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Number 对象属性</h1>

<p>溢出时返回 POSITIVE\_INFINITY：</p>

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

<script>

var x = 1 / 0;

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

</script>

</body>

</html>

JavaScript NEGATIVE\_INFINITY

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Number 对象属性</h1>

<p>NEGATIVE\_INFINITY</p>

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

<script>

var x = Number.NEGATIVE\_INFINITY;

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

</script>

</body>

</html>

溢出时返回 NEGATIVE\_INFINITY：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Number 对象属性</h1>

<p>NEGATIVE\_INFINITY</p>

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

<script>

var x = -1 / 0;

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

</script>

</body>

</html>

JavaScript NaN - 非数字

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Number 对象属性</h1>

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

<script>

document.getElementById("demo").innerHTML = Number.NaN;

</script>

</body>

</html>

NaN 属于 JavaScript 保留字，指示数字并非合法的数字。

尝试使用非数字字符串进行算术运算将导致 NaN（非数字）：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字</h1>

<p>数字除以非数字字符串会成为 NaN (Not a Number)：</p>

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

<script>

document.getElementById("demo").innerHTML = 100 / "Apple";

</script>

</body>

</html>

数字属性不可用于变量

数字属性属于名为 number 的 JavaScript 数字对象包装器。

这些属性只能作为 Number.MAX\_VALUE 访问。

使用 myNumber.MAX\_VALUE，其中 myNumber 是变量、表达式或值，将返回 undefined：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Number 对象属性</h1>

<p>对变量、表达式或值使用数字属性，将返回 undefined：</p>

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

<script>

var x = 6;

document.getElementById("demo").innerHTML = x.MAX\_VALUE;

</script>

</body>

</html>

# 第24章 JavaScript 数组

JavaScript 数组用于在单一变量中存储多个值。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

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

<script>

var cars = ["Audi", "BMW", "porsche"];

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

</script>

</body>

</html>

什么是数组？

数组是一种特殊的变量，它能够一次存放一个以上的值。

如果您有一个项目清单（例如，汽车品牌列表），在单个变量中存储汽车品牌应该是这样的：

var car1 = "Saab";

var car2 = "Volvo";

var car3 = "BMW";

不过，假如您希望遍历所有汽车并找到一个特定的值？假如不是三个汽车品牌而是三百个呢？

解决方法就是数组！

数组可以用一个单一的名称存放很多值，并且还可以通过引用索引号来访问这些值。

创建数组

使用数组文本是创建 JavaScript 数组最简单的方法。

语法：

var array-name = [item1, item2, ...];

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

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

<script>

var cars = ["Audi", "BMW", "porsche"];

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

</script>

</body>

</html>

空格和折行并不重要。声明可横跨多行：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

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

<script>

var cars = [

"Audi",

"BMW",

"porsche"

];

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

</script>

</body>

</html>

请不要最后一个元素之后写逗号（比如 "BMW",）。

可能存在跨浏览器兼容性问题。

使用 JavaScript 关键词 new

下面的例子也会创建数组，并为其赋值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

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

<script>

var cars = new Array("Audi", "BMW", "porsche");

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

</script>

</body>

</html>

以上两个例子效果完全一样。无需使用 new Array()。

出于简洁、可读性和执行速度的考虑，请使用第一种方法（数组文本方法）。

访问数组元素

我们通过引用索引号（下标号）来引用某个数组元素。

这条语句访问 cars 中的首个元素的值：

var name = cars[0];

这条语句修改 cars 中的首个元素：

cars[0] = "Opel";

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>使用数字索引（从 0 开始）访问 JavaScript 数组元素。</p>

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

<script>

var cars = ["Audi", "BMW", "porsche"];

document.getElementById("demo").innerHTML = cars[0];

</script>

</body>

</html>

访问完整数组

通过 JavaScript，可通过引用数组名来访问完整数组：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

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

<script>

var cars = ["Audi", "BMW", "porsche"];

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

</script>

</body>

</html>

数组是对象

数组是一种特殊类型的对象。在 JavaScript 中对数组使用 typeof 运算符会返回 "object"。

但是，JavaScript 数组最好以数组来描述。

数组使用数字来访问其“元素”。在本例中，person[0] 返回 Bill：

数组：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>数组使用数字访问它的元素。</p>

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

<script>

var person = ["Bill", "Gates", 62];

document.getElementById("demo").innerHTML = person[0];

</script>

</body>

</html>

对象使用名称来访问其“成员”。在本例中，person.firstName 返回 Bill：

对象：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象</h1>

<p>JavaScript 使用名称来访问对象属性。</p>

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

<script>

var person = {firstName:"Bill", lastName:"Gates", age:19};

document.getElementById("demo").innerHTML = person["firstName"];

</script>

</body>

</html>

数组元素可以是对象

JavaScript 变量可以是对象。数组是特殊类型的对象。

正因如此，您可以在相同数组中存放不同类型的变量。

您可以在数组保存对象。您可以在数组中保存函数。你甚至可以在数组中保存数组：

myArray[0] = Date.now;

myArray[1] = myFunction;

myArray[2] = myCars;

数组属性和方法

JavaScript 数组的真实力量隐藏在数组的属性和方法中：

实例

var x = cars.length; // length 属性返回元素的数量

var y = cars.sort(); // sort() 方法对数组进行排序

我们将在下一章学习数组方法。

length 属性

length 属性返回数组的长度（数组元素的数目）。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>length 属性返回数组长度。</p>

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

<script>

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

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

</script>

</body>

</html>

length 属性始终大于最高数组索引（下标）。

访问第一个数组元素

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>使用数字索引（从 0 开始）访问 JavaScript 数组元素。</p>

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

<script>

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

var first = fruits[0];

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

</script>

</body>

</html>

访问最后一个数组元素

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>使用数字索引（从 0 开始）访问 JavaScript 数组元素。</p>

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

<script>

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

var last = fruits[fruits.length-1];

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

</script>

</body>

</html>

遍历数组元素

遍历数组的最安全方法是使用 "for" 循环：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>遍历数组的最佳方法是使用标准的 for 循环：</p>

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

<script>

var fruits, text, fLen, i;

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

fLen = fruits.length;

text = "<ul>";

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

text += "<li>" + fruits[i] + "</li>";

}

text += "</ul>";

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

</script>

</body>

</html>

您也可以使用 Array.foreach() 函数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>Array.forEach() 为每个数组元素调用函数。</p>

<p>Internet Explorer 8 以及更早的版本不支持 Array.forEach()。</p>

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

<script>

var fruits, text;

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

text = "<ul>";

fruits.forEach(myFunction);

text += "</ul>";

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

function myFunction(value) {

text += "<li>" + value + "</li>";

}

</script>

</body>

</html>

添加数组元素

向数组添加新元素的最佳方法是使用 push() 方法：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>push 方法向数组追加新元素。</p>

<button onclick="myFunction()">试一试</button>

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

<script>

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

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

function myFunction() {

fruits.push("Lemon");

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

}

</script>

</body>

</html>

也可以使用 length 属性向数组添加新元素：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>length 属性提供了一种在不使用 push() 方法的情况下将新元素附加到数组的简便方法。</p>

<button onclick="myFunction()">试一试</button>

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

<script>

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

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

function myFunction() {

fruits[fruits.length] = "Lemon";

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

}

</script>

</body>

</html>

警告！

添加最高索引的元素可在数组中创建未定义的“洞”：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>添加具有高索引的元素可以在数组中创建未定义的“孔”。</p>

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

<script>

var fruits, text, fLen, i;

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

fruits[6] = "Lemon";

fLen = fruits.length;

text = "";

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

text += fruits[i] + "<br>";

}

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

</script>

</body>

</html>

关联数组

很多编程元素支持命名索引的数组。

具有命名索引的数组被称为关联数组（或散列）。

JavaScript 不支持命名索引的数组。

在 JavaScript 中，数组只能使用数字索引。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

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

<script>

var person = [];

person[0] = "Bill";

person[1] = "Gates";

person[2] = 19;

document.getElementById("demo").innerHTML =

person[0] + " " + person.length;

</script>

</body>

</html>

警告！

假如您使用命名索引，JavaScript 会把数组重定义为标准对象。

之后，所有数组的方法和属性将产生非正确结果。

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>如果在访问数组时使用命名索引，JavaScript 会将数组重新定义为标准对象，并且某些数组方法和属性将产生未定义或不正确的结果。</p>

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

<script>

var person = [];

person["firstName"] = "Bill";

person["lastName"] = "Gates";

person["age"] = 19;

document.getElementById("demo").innerHTML =

person[0] + " " + person.length;

</script>

</body>

</html>

数组和对象的区别

在 JavaScript 中，数组使用数字索引。

在 JavaScript 中，对象使用命名索引。

数组是特殊类型的对象，具有数字索引。

何时使用数组，何时使用对象？

JavaScript 不支持关联数组

如果希望元素名为字符串（文本）则应该使用对象。

如果希望元素名为数字则应该使用数组。

避免 new Array()

没有必要使用 JavaScript 的内建数组构造器 new Array()。

请使用 [] 取而代之！

下面两条不同的语句创建了名为 points 的新的空数组：

var points = new Array(); // 差

var points = []; // 优

下面两条不同的语句创建包含六个数字的新数组：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>请避免使用 new Array()。请使用 [] 取而代之。</p>

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

<script>

//var points = new Array(40, 100, 1, 5, 25, 10);

var points = [40, 100, 1, 5, 25, 10];

document.getElementById("demo").innerHTML = points[0];

</script>

</body>

</html>

new 关键词只会使代码复杂化。它还会产生某些不可预期的结果：

var points = new Array(40, 100); // 创建包含两个元素的数组（40 和 100）

假如删除其中一个元素会怎么样？

var points = new Array(40); // 创建包含 40 个未定义元素的数组！！！

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>请避免使用 new Array()。</p>

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

<script>

var points = new Array(40);

document.getElementById("demo").innerHTML = points[0];

</script>

</body>

</html>

如何识别数组

常见的问题是：我如何知晓某个变量是否是数组？

问题在于 JavaScript 运算符 typeof 返回 "object"：

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

typeof fruits; // 返回 object

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>typeof 运算符在数组上使用时返回 object：</p>

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

<script>

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

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

</script>

</body>

</html>

typeof 运算符返回 "object"，因为 JavaScript 数组属于对象。

解决方案 1：

为了解决这个问题，ECMAScript 5 定义了新方法 Array.isArray()：

Array.isArray(fruits); // 返回 true

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>新的 ECMASCRIPT 5 方法 isArray 在数组上使用时返回 true：</p>

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

<script>

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

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

</script>

</body>

</html>

此方案的问题在于 ECMAScript 5 不支持老的浏览器。

解决方案 2：

创建您自己的 isArray() 函数以解决此问题：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>这个“自制的” isArray() 函数在数组上使用时返回 true：</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;

}

</script>

</body>

</html>

假如参数为数组，则上面的函数始终返回 true。

或者更准确的解释是：假如对象原型包含单词 "Array" 则返回 true。

解决方案 3：

假如对象由给定的构造器创建，则 instanceof 运算符返回 true：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>当在数组上使用时，instanceof 运算符返回true：</p>

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

<script>

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

document.getElementById("demo").innerHTML = fruits instanceof Array;

</script>

</body>

</html>

# 第25章 JavaScript 数组方法

JavaScript 数组的力量隐藏在数组方法中。

把数组转换为字符串

JavaScript 方法 toString() 把数组转换为数组值（逗号分隔）的字符串。

实例

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

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

结果

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>toString()</h2>

<p>toString() 方法以逗号分隔的字符串返回数组：</p>

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

<script>

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

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

</script>

</body>

</html>

join() 方法也可将所有数组元素结合为一个字符串。

它的行为类似 toString()，但是您还可以规定分隔符：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>join()</h2>

<p>join() 方法将数组元素连接成一个字符串。</p>

<p>在这个例子中我们使用“\*”作为元素之间的分隔符：</p>

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

<script>

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

document.getElementById("demo").innerHTML = fruits.join(" \* ");

</script>

</body>

</html>

Popping 和 Pushing

在处理数组时，删除元素和添加新元素是很简单的。

Popping 和 Pushing 指的是：

从数组弹出项目，或向数组推入项目。

Popping

pop() 方法从数组中删除最后一个元素：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>pop()</h2>

<p>pop() 方法从数组中删除最后一个元素。</p>

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

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

<script>

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

document.getElementById("demo1").innerHTML = fruits;

fruits.pop();

document.getElementById("demo2").innerHTML = fruits;

</script>

</body>

</html>

pop() 方法返回“被弹出”的值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>pop()</h2>

<p>pop() 方法从数组中删除最后一个元素。</p>

<p>pop() 方法的返回值是已删除的项目。</p>

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

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

<p id="demo3"></p>

<script>

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

document.getElementById("demo1").innerHTML = fruits;

document.getElementById("demo2").innerHTML = fruits.pop();

document.getElementById("demo3").innerHTML = fruits;

</script>

</body>

</html>

Pushing

push() 方法（在数组结尾处）向数组添加一个新的元素：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>push()</h2>

<p>push() 方法将新元素追加到数组中。</p>

<button onclick="myFunction()">试一试</button>

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

<script>

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

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

function myFunction() {

fruits.push("Kiwi");

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

}

</script>

</body>

</html>

push() 方法返回新数组的长度：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>push()</h2>

<p>push() 方法返回新的数组长度。</p>

<button onclick="myFunction()">试一试</button>

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

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

<script>

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

document.getElementById("demo1").innerHTML = fruits;

function myFunction() {

document.getElementById("demo2").innerHTML = fruits.push("Lemon");

document.getElementById("demo1").innerHTML = fruits;

}

</script>

</body>

</html>

位移元素

位移与弹出等同，但处理首个元素而不是最后一个。

shift() 方法会删除首个数组元素，并把所有其他元素“位移”到更低的索引。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>shift()</h2>

<p>shift() 方法删除数组的第一个元素（并将所有其他元素“移位”到左侧）：</p>

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

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

<script>

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

document.getElementById("demo1").innerHTML = fruits;

fruits.shift();

document.getElementById("demo2").innerHTML = fruits;

</script>

</body>

</html>

shift() 方法返回被“位移出”的字符串：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>shift()</h2>

<p>shift() 方法返回被移出的元素。</p>

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

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

<p id="demo3"></p>

<script>

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

document.getElementById("demo1").innerHTML = fruits;

document.getElementById("demo2").innerHTML = fruits.shift();

document.getElementById("demo3").innerHTML = fruits;

</script>

</body>

</html>

unshift() 方法（在开头）向数组添加新元素，并“反向位移”旧元素：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>unshift()</h2>

<p>unshift() 方法将新元素添加到数组的开头。</p>

<button onclick="myFunction()">试一试</button>

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

<script>

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

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

function myFunction() {

fruits.unshift("Lemon");

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

}

</script>

</body>

</html>

unshift() 方法返回新数组的长度。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>unshift()</h2>

<p>unshift() 方法返回新数组的长度：</p>

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

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

<p id="demo3"></p>

<script>

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

document.getElementById("demo1").innerHTML = fruits;

document.getElementById("demo2").innerHTML = fruits.unshift("Lemon");

document.getElementById("demo3").innerHTML = fruits;

</script>

<p><b>注释：</b>unshift() 方法在 Internet Explorer 8 及更早版本中无法正常工作，会插入值，但返回值将是 <em>undefined</em>。</p>

</body>

</html>

更改元素

通过使用它们的索引号来访问数组元素：

数组索引（下标）以 0 开始。[0] 是第一个数组元素，[1] 是第二个，[2] 是第三个 ...

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<p>使用索引号访问数组元素：</p>

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

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

<script>

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

document.getElementById("demo1").innerHTML = fruits;

fruits[0] = "Lemon";

document.getElementById("demo2").innerHTML = fruits;

</script>

</body>

</html>

length 属性提供了向数组追加新元素的简易方法：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<p>length 属性提供了一种在不使用 push() 方法的情况下将新元素附加到数组的简便方法。</p>

<button onclick="myFunction()">试一试</button>

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

<script>

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

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

function myFunction() {

fruits[fruits.length] = "Lemon";

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

}

</script>

</body>

</html>

删除元素

既然 JavaScript 数组属于对象，其中的元素就可以使用 JavaScript delete 运算符来删除：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<p>删除元素会在数组中留下未定义的孔。</p>

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

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

<script>

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

document.getElementById("demo1").innerHTML =

"第一种水果是：" + fruits[0];

delete fruits[0];

document.getElementById("demo2").innerHTML =

"第一种水果是：" + fruits[0];

</script>

</body>

</html>

使用 delete 会在数组留下未定义的空洞。请使用 pop() 或 shift() 取而代之。

拼接数组

splice() 方法可用于向数组添加新项：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>splice()</h2>

<p>splice() 方法将新元素添加到数组中。</p>

<button onclick="myFunction()">试一试</button>

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

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

<script>

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

document.getElementById("demo1").innerHTML = "原数组：<br>" + fruits;

function myFunction() {

fruits.splice(2, 0, "Lemon", "Kiwi");

document.getElementById("demo2").innerHTML = "新数组：<br>" + fruits;

}

</script>

</body>

</html>

第一个参数（2）定义了应添加新元素的位置（拼接）。

第二个参数（0）定义应删除多少元素。

其余参数（“Lemon”，“Kiwi”）定义要添加的新元素。

splice() 方法返回一个包含已删除项的数组：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>splice()</h2>

<p>splice() 方法向数组添加新元素，并返回包含已删除元素（如果有）的数组。</p>

<button onclick="myFunction()">试一试</button>

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

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

<p id="demo3"></p>

<script>

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

document.getElementById("demo1").innerHTML = "原数组：<br> " + fruits;

function myFunction() {

var removed = fruits.splice(2, 2, "Lemon", "Kiwi");

document.getElementById("demo2").innerHTML = "新数组：<br>" + fruits;

document.getElementById("demo3").innerHTML = "已删除项：<br> " + removed;

}

</script>

</body>

</html>

使用 splice() 来删除元素

通过聪明的参数设定，您能够使用 splice() 在数组中不留“空洞”的情况下移除元素：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>splice()</h2>

<p>splice() 方法可用于删除数组元素。</p>

<button onclick="myFunction()">试一试</button>

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

<script>

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

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

function myFunction() {

fruits.splice(0, 1);

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

}

</script>

</body>

</html>

第一个参数（0）定义新元素应该被添加（接入）的位置。

第二个参数（1）定义应该删除多个元素。

其余参数被省略。没有新元素将被添加。

合并（连接）数组

concat() 方法通过合并（连接）现有数组来创建一个新数组：

实例（合并两个数组）

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>concat()</h2>

<p>concat() 方法用于合并（连接）数组：</p>

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

<script>

var myGirls = ["Emma", "Isabella"];

var myBoys = ["Jacob", "Michael", "Ethan"];

var myChildren = myGirls.concat(myBoys);

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

</script>

</body>

</html>

concat() 方法不会更改现有数组。它总是返回一个新数组。

concat() 方法可以使用任意数量的数组参数：

实例（合并三个数组）

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>concat()</h2>

<p>concat() 方法用于合并（连接）数组：</p>

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

<script>

var arr1 = ["Emma", "Isabella"];

var arr2 = ["Jacob", "Michael", "Ethan"];

var arr3 = ["Joshua", "Daniel"];

var myChildren = arr1.concat(arr2, arr3);

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

</script>

</body>

</html>

concat() 方法也可以将值作为参数：

实例（将数组与值合并）

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>concat()</h2>

<p>concat() 方法也可以将值合并到数组：</p>

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

<script>

var arr1 = ["Emma", "Isabella"];

var myChildren = arr1.concat(["Jacob", "Michael", "Ethan"]);

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

</script>

</body>

</html>

裁剪数组

slice() 方法用数组的某个片段切出新数组。

本例从数组元素 1 （"Orange"）开始切出一段数组：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>slice()</h2>

<p>此例从数组元素 1（“Orange”）开始切出数组的一部分：</p>

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

<script>

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

var citrus = fruits.slice(1);

document.getElementById("demo").innerHTML = fruits + "<br><br>" + citrus;

</script>

</body>

</html>

slice() 方法创建新数组。它不会从源数组中删除任何元素。

本例从数组元素 3 （"Apple"）开始切出一段数组：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>slice()</h2>

<p>此例从数组元素 3（“Apple”）开始切出数组的一部分：</p>

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

<script>

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

var citrus = fruits.slice(3);

document.getElementById("demo").innerHTML = fruits + "<br><br>" + citrus;

</script>

</body>

</html>

slice() 可接受两个参数，比如 (1, 3)。

该方法会从开始参数选取元素，直到结束参数（不包括）为止。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>slice()</h2>

<p>当给 slice() 方法赋予两个参数时，它从 <i>start</i> 参数中选择数组元素，直到（但不包括）<i>end</i> 参数：</p>

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

<script>

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

var citrus = fruits.slice(1,3);

document.getElementById("demo").innerHTML = fruits + "<br><br>" + citrus;

</script>

</body>

</html>

如果结束参数被省略，比如第一个例子，则 slice() 会切出数组的剩余部分。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>slice()</h2>

<p>此例从数组元素 2（“Lemon”）开始切出数组的一部分：</p>

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

<script>

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

var citrus = fruits.slice(2);

document.getElementById("demo").innerHTML = fruits + "<br><br>" + citrus;

</script>

</body>

</html>

自动 toString()

如果需要原始值，则 JavaScript 会自动把数组转换为字符串。下面两个例子将产生相同的结果：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<h2>toString()</h2>

<p>toString() 方法以逗号分隔的字符串返回数组：</p>

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

<script>

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

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

</script>

</body>

</html>

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组方法</h1>

<p>当需要一个简单的值时，JavaScript 会自动将数组转换为逗号分隔的字符串：</p>

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

<script>

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

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

</script>

</body>

</html>

所有 JavaScript 对象都拥有 toString() 方法。

数组排序

我们将在下一章学习到数组排序的知识~。

查找数组中的最大和最小值

没有在 JavaScript 数组中查找最高和最低值的内建函数。

您将在本教程的下一章学习如何解决这个问题。

# 第26章 JavaScript 数组排序

sort() 方法是最强大的数组方法之一。

数组排序

sort() 方法以字母顺序对数组进行排序：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组排序</h1>

<p>sort() 方法按字母顺序对数组进行排序。</p>

<button onclick="myFunction()">试一试</button>

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

<script>

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

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

function myFunction() {

fruits.sort();

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

}

</script>

</body>

</html>

反转数组

reverse() 方法反转数组中的元素。

您可以使用它以降序对数组进行排序：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组排序反转</h1>

<p>reverse() 方法反转数组中的元素。</p>

<p>通过组合 sort() 和 reverse()，您可以按降序对数组进行排序。</p>

<button onclick="myFunction()">试一试</button>

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

<script>

// Create and display an array:

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

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

function myFunction() {

// First sort the array

fruits.sort();

// Then reverse it:

fruits.reverse();

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

}

</script>

</body>

</html>

数字排序

默认地，sort() 函数按照字符串顺序对值进行排序。

该函数很适合字符串（"Apple" 会排在 "Banana" 之前）。

不过，如果数字按照字符串来排序，则 "25" 大于 "100"，因为 "2" 大于 "1"。

正因如此，sort() 方法在对数值排序时会产生不正确的结果。

我们通过一个比值函数来修正此问题：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组排序</h1>

<p>单击按钮以升序对数组进行排序。</p>

<button onclick="myFunction()">试一试</button>

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

<script>

var points = [40, 100, 1, 5, 25, 10];

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

function myFunction() {

points.sort(function(a, b){return a - b});

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

}

</script>

</body>

</html>

使用相同的技巧对数组进行降序排序：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组排序</h1>

<p>单击按钮可按降序对数组进行排序。</p>

<button onclick="myFunction()">试一试</button>

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

<script>

var points = [40, 100, 1, 5, 25, 10];

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

function myFunction() {

points.sort(function(a, b){return b - a});

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

}

</script>

</body>

</html>

比值函数

比较函数的目的是定义另一种排序顺序。

比较函数应该返回一个负，零或正值，这取决于参数：

function(a, b){return a-b}

当 sort() 函数比较两个值时，会将值发送到比较函数，并根据所返回的值（负、零或正值）对这些值进行排序。

实例：

当比较 40 和 100 时，sort() 方法会调用比较函数 function(40,100)。

该函数计算 40-100，然后返回 -60（负值）。

排序函数将把 40 排序为比 100 更低的值。

您可以使用下面的代码片段来测试数值和字母排序：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组排序</h1>

<p>单击按钮可按字母顺序或数字顺序对数组进行排序。</p>

<button onclick="myFunction1()">按字母排序</button>

<button onclick="myFunction2()">按数字排序</button>

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

<script>

var points = [40, 100, 1, 5, 25, 10];

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

function myFunction1() {

points.sort();

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

}

function myFunction2() {

points.sort(function(a, b){return a - b});

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

}

</script>

</body>

</html>

以随机顺序排序数组

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组排序</h1>

<p>请反复点击按钮，对数组进行随机排序。</p>

<button onclick="myFunction()">试一试</button>

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

<script>

var points = [40, 100, 1, 5, 25, 10];

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

function myFunction() {

points.sort(function(a, b){return 0.5 - Math.random()});

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

}

</script>

</body>

</html>

查找最高（或最低）的数组值

JavaScript 不提供查找数组中最大或最小数组值的内建函数。

不过，在对数组进行排序之后，您能够使用索引来获得最高或最低值。

升序排序：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组排序</h1>

<p>最低值是：<span id="demo"></span></p>

<script>

var points = [40, 100, 1, 5, 25, 10];

points.sort(function(a, b){return a-b});

document.getElementById("demo").innerHTML = points[0];

</script>

</body>

</html>

降序排序：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组排序</h1>

<p>最高值是：<span id="demo"></span></p>

<script>

var points = [40, 100, 1, 5, 25, 10];

points.sort(function(a, b){return b-a});

document.getElementById("demo").innerHTML = points[0];

</script>

</body>

</html>

如果您仅仅需要找到最高或最低值，对整个数组进行排序是效率极低的方法。

对数组使用 Math.max()

您可以使用 Math.max.apply 来查找数组中的最高值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组排序</h1>

<p>最高值是：<span id="demo"></span></p>

<script>

var points = [40, 100, 1, 5, 25, 10];

document.getElementById("demo").innerHTML = myArrayMax(points);

function myArrayMax(arr) {

return Math.max.apply(null, arr);

}

</script>

</body>

</html>

Math.max.apply([1, 2, 3]) 等于 Math.max(1, 2, 3)。

对数组使用 Math.min()

您可以使用 Math.min.apply 来查找数组中的最低值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组排序</h1>

<p>最低值是：<span id="demo"></span></p>

<script>

var points = [40, 100, 1, 5, 25, 10];

document.getElementById("demo").innerHTML = myArrayMin(points);

function myArrayMin(arr) {

return Math.min.apply(null, arr);

}

</script>

</body>

</html>

Math.min.apply([1, 2, 3]) 等于 Math.min(1, 2, 3)。

我的 Min / Max JavaScript 方法

最快的解决方法是使用“自制”方法。

此函数遍历数组，用找到的最高值与每个值进行比较：

实例（查找 Max）

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组排序</h1>

<p>最高值是：<span id="demo"></span></p>

<script>

var points = [40, 100, 1, 5, 25, 10];

document.getElementById("demo").innerHTML = myArrayMax(points);

function myArrayMax(arr) {

var len = arr.length;

var max = -Infinity;

while (len--) {

if (arr[len] > max) {

max = arr[len];

}

}

return max;

}

</script>

</body>

</html>

此函数遍历数组，用找到的最低值与每个值进行比较：

实例（查找 Min）

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组排序</h1>

<p>最低值是：<span id="demo"></span></p>

<script>

var points = [40, 100, 1, 5, 25, 10];

document.getElementById("demo").innerHTML = myArrayMin(points);

function myArrayMin(arr) {

var len = arr.length;

var min = Infinity;

while (len--) {

if (arr[len] < min) {

min = arr[len];

}

}

return min;

}

</script>

</body>

</html>

排序对象数组

JavaScript 数组经常会包含对象：

实例

var cars = [

{type:"Volvo", year:2016},

{type:"Saab", year:2001},

{type:"BMW", year:2010}];

即使对象拥有不同数据类型的属性，sort() 方法仍可用于对数组进行排序。

解决方法是通过比较函数来对比属性值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组排序</h1>

<p>点击按钮按年份对汽车进行排序：</p>

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

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

<script>

var cars = [

{type:"BMW", year:2017},

{type:"Audi", year:2019},

{type:"porsche", year:2018}

];

displayCars();

function myFunction() {

cars.sort(function(a, b){return a.year - b.year});

displayCars();

}

function displayCars() {

document.getElementById("demo").innerHTML =

cars[0].type + " " + cars[0].year + "<br>" +

cars[1].type + " " + cars[1].year + "<br>" +

cars[2].type + " " + cars[2].year;

}

</script>

</body>

</html>

比较字符串属性会稍复杂：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组排序</h1>

<p>点击按钮按车型对汽车进行排序：</p>

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

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

<script>

var cars = [

{type:"BMW", year:2017},

{type:"Audi", year:2019},

{type:"porsche", year:2018}

];

displayCars();

function myFunction() {

cars.sort(function(a, b){

var x = a.type.toLowerCase();

var y = b.type.toLowerCase();

if (x < y) {return -1;}

if (x > y) {return 1;}

return 0;

});

displayCars();

}

function displayCars() {

document.getElementById("demo").innerHTML =

cars[0].type + " " + cars[0].year + "<br>" +

cars[1].type + " " + cars[1].year + "<br>" +

cars[2].type + " " + cars[2].year;

}

</script>

</body>

</html>

# 第27章 JavaScript 数组迭代

数组迭代方法对每个数组项进行操作。

Array.forEach()

forEach() 方法为每个数组元素调用一次函数（回调函数）。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.forEach()</h1>

<p>为每个元素调用一次函数。</p>

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

<script>

var txt = "";

var numbers = [45, 4, 9, 16, 25];

numbers.forEach(myFunction);

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

function myFunction(value, index, array) {

txt = txt + value + "<br>";

}

</script>

</body>

</html>

注释：该函数接受 3 个参数：

项目值

项目索引

数组本身

上面的例子只用了 value 参数。这个例子可以重新写为：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.forEach()</h1>

<p>为每个元素调用一次函数。</p>

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

<script>

var txt = "";

var numbers = [45, 4, 9, 16, 25];

numbers.forEach(myFunction);

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

function myFunction(value) {

txt = txt + value + "<br>";

}

</script>

</body>

</html>

Array.map()

map() 方法通过对每个数组元素执行函数来创建新数组。

map() 方法不会对没有值的数组元素执行函数。

map() 方法不会更改原始数组。

这个例子将每个数组值乘以2：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.map()</h1>

<p>通过对每个数组元素执行函数来创建新数组。</p>

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

<script>

var numbers1 = [45, 4, 9, 16, 25];

var numbers2 = numbers1.map(myFunction);

document.getElementById("demo").innerHTML = numbers2;

function myFunction(value, index, array) {

return value \* 2;

}

</script>

</body>

</html>

请注意，该函数有 3 个参数：

项目值

项目索引

数组本身

当回调函数仅使用 value 参数时，可以省略索引和数组参数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.map()</h1>

<p>通过对每个数组元素执行函数来创建新数组。</p>

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

<script>

var numbers1 = [45, 4, 9, 16, 25];

var numbers2 = numbers1.map(myFunction);

document.getElementById("demo").innerHTML = numbers2;

function myFunction(value) {

return value \* 2;

}

</script>

</body>

</html>

Array.filter()

filter() 方法创建一个包含通过测试的数组元素的新数组。

这个例子用值大于 18 的元素创建一个新数组：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.filter()</h1>

<p>使用通过测试的所有数组元素创建一个新数组。</p>

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

<script>

var numbers = [45, 4, 9, 16, 25];

var over18 = numbers.filter(myFunction);

document.getElementById("demo").innerHTML = over18;

function myFunction(value, index, array) {

return value > 18;

}

</script>

</body>

</html>

请注意此函数接受 3 个参数：

项目值

项目索引

数组本身

在上面的例子中，回调函数不使用 index 和 array 参数，因此可以省略它们：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.filter()</h1>

<p>使用通过测试的所有数组元素创建一个新数组。</p>

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

<script>

var numbers = [45, 4, 9, 16, 25];

var over18 = numbers.filter(myFunction);

document.getElementById("demo").innerHTML = over18;

function myFunction(value) {

return value > 18;

}

</script>

</body>

</html>

Array.reduce()

reduce() 方法在每个数组元素上运行函数，以生成（减少它）单个值。

reduce() 方法在数组中从左到右工作。另请参阅 reduceRight()。

reduce() 方法不会减少原始数组。

这个例子确定数组中所有数字的总和：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.reduce()</h1>

<p>此例确定数组中所有数字的总和：</p>

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

<script>

var numbers = [45, 4, 9, 16, 25];

var sum = numbers.reduce(myFunction);

document.getElementById("demo").innerHTML = "总和是：" + sum;

function myFunction(total, value, index, array) {

return total + value;

}

</script>

</body>

</html>

请注意此函数接受 4 个参数：

总数（初始值/先前返回的值）

项目值

项目索引

数组本身

上例并未使用 index 和 array 参数。可以将它改写为：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.reduce()</h1>

<p>此例确定数组中所有数字的总和：</p>

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

<script>

var numbers = [45, 4, 9, 16, 25];

var sum = numbers.reduce(myFunction);

document.getElementById("demo").innerHTML = "总和是：" + sum;

function myFunction(total, value) {

return total + value;

}

</script>

</body>

</html>

reduce() 方法能够接受一个初始值：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.reduce()</h1>

<p>此例确定数组中所有数字的总和：</p>

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

<script>

var numbers = [45, 4, 9, 16, 25];

var sum = numbers.reduce(myFunction, 100);

document.getElementById("demo").innerHTML = "总和是：" + sum;

function myFunction(total, value) {

return total + value;

}

</script>

</body>

</html>

Array.reduceRight()

reduceRight() 方法在每个数组元素上运行函数，以生成（减少它）单个值。

reduceRight() 方法在数组中从右到左工作。另请参阅 reduce()。

reduceRight() 方法不会减少原始数组。

这个例子确定数组中所有数字的总和：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.reduceRight()</h1>

<p>此例确定数组中所有数字的总和：</p>

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

<script>

var numbers = [45, 4, 9, 16, 25];

var sum = numbers.reduceRight(myFunction);

document.getElementById("demo").innerHTML = "总和是：" + sum;

function myFunction(total, value, index, array) {

return total + value;

}

</script>

</body>

</html>

请注意此函数接受 4 个参数：

总数（初始值/先前返回的值）

项目值

项目索引

数组本身

上例并未使用 index 和 array 参数。可以将它改写为：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.reduceRight()</h1>

<p>此例确定数组中所有数字的总和：</p>

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

<script>

var numbers = [45, 4, 9, 16, 25];

var sum = numbers.reduceRight(myFunction);

document.getElementById("demo").innerHTML = "总和是：" + sum;

function myFunction(total, value) {

return total + value;

}

</script>

</body>

</html>

Array.every()

every() 方法检查所有数组值是否通过测试。

这个例子检查所有数组值是否大于 18：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.every()</h1>

<p>every() 方法检查所有数组值是否通过测试。</p>

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

<script>

var numbers = [45, 4, 9, 16, 25];

var allOver18 = numbers.every(myFunction);

document.getElementById("demo").innerHTML = "所有大于 18 的是：" + allOver18;

function myFunction(value, index, array) {

return value > 18;

}

</script>

</body>

</html>

请注意此函数接受 3 个参数：

项目值

项目索引

数组本身

如果回调函数仅使用第一个参数（值）时，可以省略其他参数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.every()</h1>

<p>every() 方法检查所有数组值是否通过测试。</p>

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

<script>

var numbers = [45, 4, 9, 16, 25];

var allOver18 = numbers.every(myFunction);

document.getElementById("demo").innerHTML = "所有大于 18 的是：" + allOver18;

function myFunction(value) {

return value > 18;

}

</script>

</body>

</html>

Array.some()

some() 方法检查某些数组值是否通过了测试。

这个例子检查某些数组值是否大于 18：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.some()</h1>

<p>some() 方法检查某些数组值是否通过了测试。</p>

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

<script>

var numbers = [45, 4, 9, 16, 25];

var someOver18 = numbers.some(myFunction);

document.getElementById("demo").innerHTML = "某些值大于 18 是：" + someOver18;

function myFunction(value, index, array) {

return value > 18;

}

</script>

</body>

</html>

请注意此函数接受 3 个参数：

项目值

项目索引

数组本身

Array.indexOf()

indexOf() 方法在数组中搜索元素值并返回其位置。

注释：第一个项目的位置是 0，第二个项目的位置是 1，以此类推。

实例

检索数组中的项目 "Apple"：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.indexOf()</h1>

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

<script>

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

var a = fruits.indexOf("Apple");

document.getElementById("demo").innerHTML = "Apple 被找到的位置是：" + (a + 1);

</script>

<p>Internet Explorer 8 或更早的版本不支持 indexOf()。</p>

</body>

</html>

语法

array.indexOf(item, start)

item 必需。要检索的项目。

start 可选。从哪里开始搜索。负值将从结尾开始的给定位置开始，并搜索到结尾。

如果未找到项目，Array.indexOf() 返回 -1。

如果项目多次出现，则返回第一次出现的位置。

Array.lastIndexOf()

Array.lastIndexOf() 与 Array.indexOf() 类似，但是从数组结尾开始搜索。

实例

检索数组中的项目 "Apple"：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.lastIndexOf()</h1>

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

<script>

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

var a = fruits.lastIndexOf("Apple");

document.getElementById("demo").innerHTML = "Apple is found in position " + (a + 1);

</script>

<p>Internet Explorer 8 或更早的版本不支持 lastIndexOf()。</p>

</body>

</html>

语法

array.lastIndexOf(item, start)

item 必需。要检索的项目。

start 可选。从哪里开始搜索。负值将从结尾开始的给定位置开始，并搜索到开头。

Array.find()

find() 方法返回通过测试函数的第一个数组元素的值。

这个例子查找（返回）大于 18 的第一个元素的值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.find()</h1>

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

<script>

var numbers = [4, 9, 16, 25, 29];

var first = numbers.find(myFunction);

document.getElementById("demo").innerHTML = "大于 18 的第一个值是：" + first;

function myFunction(value, index, array) {

return value > 18;

}

</script>

</body>

</html>

Array.findIndex()

findIndex() 方法返回通过测试函数的第一个数组元素的索引。

这个例子查找大于 18 的第一个元素的索引：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.findIndex()</h1>

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

<script>

var numbers = [4, 9, 16, 25, 29];

var first = numbers.findIndex(myFunction);

document.getElementById("demo").innerHTML = "大于 18 的第一个值的索引是：" + first;

function myFunction(value, index, array) {

return value > 18;

}

</script>

</body>

</html>

请注意此函数接受 3 个参数：

项目值

项目索引

数组本身

# 第28章 JavaScript 数组 Const

ECMAScript 2015 (ES6)

2015 年，JavaScript 引入了一个重要的新关键字：const。

使用 const 声明数组已成为一种常见做法：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript const</h2>

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

<script>

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

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

</script>

</body>

</html>

无法重新赋值

用 const 声明的数组不能重新赋值：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript const</h2>

<p>您不能重新赋值常量数组：</p>

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

<script>

try {

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

cars = ["Toyota", "Volvo", "Audi"];

}

catch (err) {

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

}

</script>

</body>

</html>

数组不是常量

关键字 const 有一定误导性。

它不定义常量数组。它定义的是对数组的常量引用。

因此，我们仍然可以更改常量数组的元素。

元素可以重新赋值

您可以更改常量数组的元素：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript const</h2>

<p>声明一个常量数组不会使元素不可更改：</p>

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

<script>

// 创建数组：

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;

</script>

</body>

</html>

声明时赋值

JavaScript const 变量在声明时必须赋值：

意思是：用 const 声明的数组必须在声明时进行初始化。

使用 const 而不初始化数组是一个语法错误：

实例

不起作用：

const cars;

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

用 var 声明的数组可以随时初始化。

您甚至可以在声明之前使用该数组：

实例

没问题：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 提升</h2>

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

<script>

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

var cars;

document.getElementById("demo").innerHTML = cars[0];

</script>

</body>

</html>

const 块作用域

用 const 声明的数组具有块作用域。

在块中声明的数组与在块外声明的数组不同：

实例

<!DOCTYPE html>

<html>

<body>

<h2>使用 const 声明数组</h2>

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

<script>

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

// 此处 cars[0] 是 "Tesla"

{

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

// 此处 cars[0] 是 "Toyota"

}

// 此处 cars[0] 是 "Saab"

document.getElementById("demo").innerHTML = cars[0];

</script>

</body>

</html>

用 var 声明的数组没有块作用域：

实例

<!DOCTYPE html>

<html>

<body>

<h2>使用 const 声明数组</h2>

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

<script>

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

// 此处 cars[0] 是 "Tesla"

{

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

// 此处 cars[0] 是 "Toyota"

}

// 此处 cars[0] 是 "Toyota"

document.getElementById("demo").innerHTML = cars[0];

</script>

</body>

</html>

重新声明数组

在程序中的任何位置都允许用 var 重新声明数组：

实例

var cars = ["Volvo", "BMW"]; // 允许

var cars = ["Toyota", "BMW"]; // 允许

cars = ["Volvo", "Saab"]; // 允许

不允许在同一作用域或同一块中将数组重新声明或重新赋值给 const：

实例

var cars = ["Volvo", "BMW"]; // 允许

const cars = ["Volvo", "BMW"]; // 不允许

{

var cars = ["Volvo", "BMW"]; // 允许

const cars = ["Volvo", "BMW"]; // 不允许

}

不允许在同一作用域或同一块中重新声明或重新赋值现有的 const 数组：

实例

const cars = ["Volvo", "BMW"]; // 允许

const cars = ["Volvo", "BMW"]; // 不允许

var cars = ["Volvo", "BMW"]; // 不允许

cars = ["Volvo", "BMW"]; // 不允许

{

const cars = ["Volvo", "BMW"]; // 允许

const cars = ["Volvo", "BMW"]; // 不允许

var cars = ["Volvo", "BMW"]; // 不允许

cars = ["Volvo", "BMW"]; // 不允许

}

允许在另一个作用域或另一个块中使用 const 重新声明数组：

实例

const cars = ["Volvo", "BMW"]; // 允许

{

const cars = ["Volvo", "BMW"]; // 允许

}

{

const cars = ["Volvo", "BMW"]; // 允许

}

# 第29章 JavaScript 日期

实例

var d = new Date();

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

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

<script>

var d = new Date();

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

</script>

</body>

</html>

JavaScript 日期输出

默认情况下，JavaScript 将使用浏览器的时区并将日期显示为全文本字符串：

Tue Apr 02 2019 09:01:19 GMT+0800 (中国标准时间)

稍后，您将在本教程学到更多关于如何显示日期的知识。

创建 Date 对象

Date 对象由新的 Date() 构造函数创建。

有 4 种方法创建新的日期对象：

new Date()

new Date(year, month, day, hours, minutes, seconds, milliseconds)

new Date(milliseconds)

new Date(date string)

new Date()

new Date() 用当前日期和时间创建新的日期对象：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

<p>使用 new Date()，使用当前日期和时间创建一个新的日期对象：</p>

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

<script>

var d = new Date();

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

</script>

</body>

</html>

日期对象是静态的。计算机时间正在滴答作响，但日期对象不会。

new Date(year, month, ...)

new Date(year, month, ...) 用指定日期和时间创建新的日期对象。

7个数字分别指定年、月、日、小时、分钟、秒和毫秒（按此顺序）：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

<p>使用 new Date(7 numbers)，使用指定的日期和时间创建一个新的日期对象：</p>

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

<script>

var d = new Date(2019, 12, 25, 10, 33, 30, 0);

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

</script>

</body>

</html>

注释：JavaScript 从 0 到 11 计算月份。

一月是 0。十二月是11。

6个数字指定年、月、日、小时、分钟、秒：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

<p>6 个数字指定年、月、日、时、分、秒：</p>

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

<script>

var d = new Date(2019, 12, 25, 10, 33, 30);

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

</script>

</body>

</html>

5个数字指定年、月、日、小时和分钟：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

<p>5 个数字指定年、月、日、时、分：</p>

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

<script>

var d = new Date(2018, 11, 24, 10, 33);

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

</script>

</body>

</html>

4个数字指定年、月、日和小时：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

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

<script>

var d = new Date(2018, 11, 24, 10);

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

</script>

</body>

</html>

3 个数字指定年、月和日：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

<p>3 个数字指定年、月、日：</p>

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

<script>

var d = new Date(2018, 11, 24);

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

</script>

</body>

</html>

2个数字指定年份和月份：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

<p>2 个数字指定年、月：</p>

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

<script>

var d = new Date(2018, 11);

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

</script>

</body>

</html>

您不能省略月份。如果只提供一个参数，则将其视为毫秒。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

<p>一个参数会被解释为：new Date(milliseconds)。</p>

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

<script>

var d = new Date(2018);

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

</script>

</body>

</html>

上个世纪

一位和两位数年份将被解释为 19xx 年：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

<p>两位年份将被解释为 19xx：</p>

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

<script>

var d = new Date(99, 12, 25);

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

</script>

</body>

</html>

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

<p>一位的年份会被解释为 19xx：</p>

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

<script>

var d = new Date(9, 12, 25);

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

</script>

</body>

</html>

new Date(dateString)

new Date(dateString) 从日期字符串创建一个新的日期对象：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

<p>可以使用指定的日期和时间创建 Date 对象：</p>

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

<script>

var d = new Date("December 25, 2019 12:46:00");

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

</script>

</body>

</html>

日期字符串将在下一章中介绍。

JavaScript 将日期存储为毫秒

JavaScript 将日期存储为自 1970 年 1 月 1 日 00:00:00 UTC（协调世界时）以来的毫秒数。

零时间是 1970 年 1 月 1 日 00:00:00 UTC。

现在的时间是：1970 年 1 月 1 日之后的 1554166879383 毫秒。

new Date(milliseconds)

new Date(milliseconds) 创建一个零时加毫秒的新日期对象：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

<p>使用 new Date(<i>milliseconds</i>)，创建一个零时（January 1, 1970, 00:00:00 Universal Time）加毫秒的新日期对象：</p>

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

<script>

var d = new Date(0);

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

</script>

</body>

</html>

1970年 1 月 1 日加上100 000 000 000毫秒，大约是 1973 年 3 月 3 日：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

<p>1970 年 1 月 1 日起的 100000000000 毫秒，大约是 1973 年 3 月 3 日：</p>

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

<script>

var d = new Date(100000000000);

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

</script>

</body>

</html>

1970 年 1 月 1 日减去 100 000 000 000 毫秒大约是 1966 年 10 月 31 日：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

<p>1970 年 1 月 1 日起的 100000000000 毫秒，大约是 1966 年 10 月 31 日：</p>

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

<script>

var d = new Date(-100000000000);

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

</script>

</body>

</html>

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

<p>使用 new Date(<i>milliseconds</i>)，创建一个零时（January 1, 1970, 00:00:00 Universal Time）加毫秒的新日期对象：:</p>

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

<script>

var d = new Date(86400000);

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

</script>

<p>一天（24小时）是 86,400,000 毫秒。</p>

</body>

</html>

一天（24 小时）是 86 400 000 毫秒。

日期方法

创建 Date 对象时，可以使用许多方法对其进行操作。

日期方法允许您使用本地时间或 UTC（通用或 GMT）时间来获取和设置日期对象的年、月、日、小时、分钟、秒和毫秒。

日期方法和时区将在下一章中介绍。

显示日期

JavaScript（默认情况下）将以全文本字符串格式输出日期：

Wed Mar 25 2015 08:00:00 GMT+0800 (中国标准时间)

在 HTML 中显示日期对象时，会使用 toString() 方法自动转换为字符串。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

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

<script>

var d = new Date();

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

</script>

</body>

</html>

等同于：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript toString()</h2>

<p>toString() 方法将日期转换为字符串：</p>

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

<script>

var d = new Date();

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

</script>

</body>

</html>

toUTCString() 方法将日期转换为 UTC 字符串（一种日期显示标准）。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Date()</h2>

<p>toUTCString() 方法将日期转换为 UTC 字符串（日期显示标准）：</p>

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

<script>

var d = new Date();

document.getElementById("demo").innerHTML = d.toUTCString();

</script>

</body>

</html>

toDateString() 方法将日期转换为更易读的格式：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript toDateString()</h2>

<p>toDateString() 方法将日期转换为日期字符串：</p>

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

<script>

var d = new Date();

document.getElementById("demo").innerHTML = d.toDateString();

</script>

</body>

</html>

# 第30章 JavaScript 日期格式

有四种 JavaScript 日期输入格式：

|  |  |
| --- | --- |
| 类型 | 实例 |
| ISO 日期 | "2018-02-19" （国际标准） |
| 短日期 | "02/19/2018" 或者 "2018/02/19" |
| 长日期 | "Feb 19 2018" 或者 "19 Feb 2019" |
| 完整日期 | "Monday February 25 2015" |

ISO 格式遵守 JavaScript 中的严格标准。

其他格式不太明确，可能是浏览器特定的。

JavaScript 日期输出

无论输入格式如何，JavaScript 默认将输出全文本字符串格式：

Mon Feb 19 2018 06:00:00 GMT+0800 (中国标准时间)

JavaScript ISO 日期

ISO 8601 是表现日期和时间的国际标准。

ISO 8601 语法 (YYYY-MM-DD) 也是首选的 JavaScript 日期格式：

实例（完整日期）

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript ISO 日期</h2>

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

<script>

var d = new Date("2019-05-02");

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

</script>

</body>

</html>

计算的日期相对于您的时区。

根据您的时区，上面的结果将在 2 月 18 日至 2 月 19 日之间变化。

ISO 日期（年和月）

写日期也可以不规定具体某日 (YYYY-MM)：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript ISO 日期</h2>

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

<script>

var d = new Date("2019-05");

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

</script>

</body>

</html>

时区会对结果在 2 月 28 日至 3 月 1 日之间产生变化。

ISO 日期（只有年）

写日期也可以不规定具体的月和日 (YYYY)：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript ISO 日期</h2>

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

<script>

var d = new Date("2019");

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

</script>

</body>

</html>

时区会对结果在 2017 年 12 月 31 日至 2018 年 1 月 1 日之间产生变化。

ISO 日期（完整的日期加时、分和秒）

写日期也可以添加时、分和秒 (YYYY-MM-DDTHH:MM:SS)：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript ISO 日期</h2>

<p>用大写字母 T 分隔日期和时间。</p>

<p>用大写字母 Z 标识 UTC 时间。</p>

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

<script>

var d = new Date("2019-05-02T12:00:00Z");

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

</script>

</body>

</html>

日期和时间通过大写字母 T 来分隔。

UTC 时间通过大写字母 Z 来定义。

如果您希望修改相对于 UTC 的时间，请删除 Z 并用 +HH:MM 或 -HH:MM 代替：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript ISO 日期</h2>

<p>通过加 +HH：MM或减 -HH：MM 时间来修改相对于 UTC 的时间。</p>

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

<script>

document.getElementById("demo").innerHTML =

new Date("2015-03-25T12:00:00-06:00");

</script>

</body>

</html>

UTC（Universal Time Coordinated）等同于 GMT（格林威治时间）。

注释：UTC，协调世界时，又称世界统一时间，世界标准时间，国际协调时间。

在日期-时间字符串中省略 T 或 Z，在不同浏览器中会产生不同结果。

时区

在设置日期时，如果不规定时区，则 JavaScript 会使用浏览器的时区。

当获取日期时，如果不规定时区，则结果会被转换为浏览器时区。

换句话说，假如日期/时间以 GMT（格林威治标准时间）创建，该日期/时间将被转换为 CST（中国标准时间），如果用户从中国进行浏览。

JavaScript 短日期

短日期通常使用 "MM/DD/YYYY" 这样的语法：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

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

<script>

var d = new Date("05/26/2019");

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

</script>

</body>

</html>

警告

在某些浏览器中，不带前导零的月或其会产生错误：

var d = new Date("2018-2-19");

“YYYY / MM / DD”的行为未定义。

有些浏览器会尝试猜测格式。有些会返回 NaN。

var d = new Date("2018/02/19");

“DD-MM-YYYY”的行为也是未定义的。

有些浏览器会尝试猜测格式。有些会返回 NaN。

var d = new Date("19-02-2018");

JavaScript 长日期

长日期通常以 "MMM DD YYYY" 这样的语法来写：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

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

<script>

var d = new Date("Mar 26 2019");

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

</script>

</body>

</html>

月和天能够以任意顺序出现：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

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

<script>

var d = new Date("26 Mar 2019");

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

</script>

</body>

</html>

并且，月能够以全称 (January) 或缩写 (Jan) 来写：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

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

<script>

var d = new Date("January 25 2019");

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

</script>

</body>

</html>

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

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

<script>

var d = new Date("Jan 25 2019");

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

</script>

</body>

</html>

逗号会被忽略，且对大小写不敏感：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript new Date()</h2>

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

<script>

var d = new Date("JANUARY, 25, 2019");

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

</script>

</body>

</html>

JavaScript 完整日期

JavaScript 接受“完整 JavaScript 格式”的日期字符串：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Date.parse()</h2>

<p>Date.parse() 返回日期和1970年1月1日之间的毫秒数：</p>

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

<script>

var msec = Date.parse("March 21, 2012");

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

</script>

</body>

</html>

JavaScript 会忽略日期名称和时间括号中的错误：

实例

<!DOCTYPE html>

<html>

<body>

<p>Date.parse(string) 返回毫秒。</p>

<p>您可以使用返回值将字符串转换为日期对象：</p>

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

<script>

var msec = Date.parse("March 22, 2019");

var d = new Date(msec);

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

</script>

</body>

</html>

# 第31章 JavaScript 日期获取方法

日期方法允许您获取并设置日期值（年、月、日、时、分、秒、毫秒）

日期获取方法

获取方法用于获取日期的某个部分（来自日期对象的信息）。下面是最常用的方法（以字母顺序排序）

|  |  |
| --- | --- |
| 方法 | 描述 |
| getDate() | 以数值返回天（1-31） |
| getDay() | 以数值获取周名（0-6） |
| getFullYear() | 获取四位的年（yyyy） |
| getHours() | 获取小时（0-23） |
| getMilliseconds() | 获取毫秒（0-999） |
| getMinutes() | 获取分（0-59） |
| getMonth() | 获取月（0-11） |
| getSeconds() | 获取秒（0-59） |
| getTime() | 获取时间（从 1970 年 1 月 1 日至今） |

getTime() 方法

getTime() 方法返回自 1970 年 1 月 1 日以来的毫秒数：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript getTime()</h2>

<p>JavaScript 中的内部时钟从 1970 年 1 月 1 日午夜开始计算。</p>

<p>getTime() 函数返回从那时起的毫秒数：</p>

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

<script>

var d = new Date();

document.getElementById("demo").innerHTML = d.getTime();

</script>

</body>

</html>

getFullYear() 方法

getFullYear() 方法以四位数字形式返回日期年份：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript getFullYear()</h2>

<p>getFullYear() 方法返回日期的完整年：</p>

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

<script>

var d = new Date();

document.getElementById("demo").innerHTML = d.getFullYear();

</script>

</body>

</html>

getMonth() 方法

getMonth() 以数字（0-11）返回日期的月份：

实例

var d = new Date();

document.getElementById("demo").innerHTML = d.getMonth();

在 JavaScript 中，第一个月（1 月）是月号 0，因此 12 月返回月号 11。

您可以使用名称数组，并使用 getMonth() 将月份作为名称返回：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript getMonth()</h2>

<p>getMonth（）方法以 0 到 11 之间的数字返回日期的月份。</p>

<p>要获得正确的月份，您必须添加 1：</p>

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

<script>

var d = new Date();

document.getElementById("demo").innerHTML = d.getMonth() + 1;

</script>

</body>

</html>

getDate() 方法

getDate() 方法以数字（1-31）返回日期的日：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript getDate()</h2>

<p>getDate() 方法以数字（1-31）返回日期的日：</p>

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

<script>

var d = new Date();

document.getElementById("demo").innerHTML = d.getDate();

</script>

</body>

</html>

getHours() 方法

getHours() 方法以数字（0-23）返回日期的小时数：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript getHours()</h2>

<p>getHours() 方法以数字（0-23）返回日期的小时：</p>

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

<script>

var d = new Date();

document.getElementById("demo").innerHTML = d.getHours();

</script>

</body>

</html>

getMinutes() 方法

getMinutes() 方法以数字（0-59）返回日期的分钟数：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript getMinutes()</h2>

<p>getMinutes() 方法以数字（0-59）返回日期的分钟：</p>

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

<script>

var d = new Date();

document.getElementById("demo").innerHTML = d.getMinutes();

</script>

</body>

</html>

getSeconds() 方法

getSeconds() 方法以数字（0-59）返回日期的秒数：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript getSeconds()</h2>

<p>getSeconds() 方法以数字（0-59）返回日期的秒：</p>

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

<script>

var d = new Date();

document.getElementById("demo").innerHTML = d.getSeconds();

</script>

</body>

</html>

getMilliseconds() 方法

getMilliseconds() 方法以数字（0-999）返回日期的毫秒数：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript getMilliseconds()</h2>

<p>getMilliseconds() 方法以数字（0-999）的形式返回日期的毫秒数：</p>

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

<script>

var d = new Date();

document.getElementById("demo").innerHTML = d.getMilliseconds();

</script>

</body>

</html>

getDay() 方法

getDay() 方法以数字（0-6）返回日期的星期名（weekday）：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript getDay()</h2>

<p>getDay() 方法将周名作为数字返回：</p>

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

<script>

var d = new Date();

document.getElementById("demo").innerHTML = d.getDay();

</script>

</body>

</html>

在 JavaScript 中，一周的第一天（0）表示“星期日”，即使世界上的一些国家认为周的第一天是“星期一”。

您可以使用名称数组，并使用 getDay() 将星期名作为名称返回：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript getDay()</h2>

<p>getDay() 方法将周名作为数字返回：</p>

<p>您可以使用数组来显示星期的名称：</p>

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

<script>

var d = new Date();

var days = ["Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"];

document.getElementById("demo").innerHTML = days[d.getDay()];

</script>

</body>

</html>

UTC 日期方法

UTC 日期方法用于处理 UTC 日期（通用时区日期，Univeral Time Zone dates）：

|  |  |
| --- | --- |
| 方法 | 描述 |
| getUTCDate() | 等于 getDate()，但返回 UTC 日期 |
| getUTCDay() | 等于 getDay()，但返回 UTC 日 |
| getUTCFullYear() | 等于 getFullYear()，但返回 UTC 年 |
| getUTCHours() | 等于 getHours()，但返回 UTC 小时 |
| getUTCMilliseconds() | 等于 getMilliseconds()，但返回 UTC 毫秒 |
| getUTCMinutes() | 等于 getMinutes()，但返回 UTC 分 |
| getUTCMonth() | 等于 getMonth()，但返回 UTC 月 |
| getUTCSeconds() | 等于 getSeconds()，但返回 UTC 秒 |

# 第32章 JavaScript 日期设置方法

使用“设置日期”方法可以设置日期对象的日期值（年、月、日、小时、分钟、秒、毫秒）。

日期设置方法

设置方法用于设置日期的某个部分。下面是最常用的方法（按照字母顺序排序）：

|  |  |
| --- | --- |
| 方法 | 描述 |
| setDate() | 以数值（1-31）设置日 |
| setFullYear() | 设置年（可选月和日） |
| setHours() | 设置小时（0-23） |
| setMilliseconds() | 设置毫秒（0-999） |
| setMinutes() | 设置分（0-59） |
| setMonth() | 设置月（0-11） |
| setSeconds() | 设置秒（0-59） |
| setTime() | 设置时间（从 1970 年 1 月 1 日至今的毫秒数） |

setFullYear() 方法

setFullYear() 方法设置日期对象的年份。这个例子设置为 2020 年：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript setFullYear()</h2>

<p>setFullYear() 方法设置日期对象的年份：</p>

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

<script>

var d = new Date();

d.setFullYear(2020);

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

</script>

</body>

</html>

setFullYear() 方法可以选择设置月和日：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript setFullYear()</h2>

<p>setFullYear() 方法可以选择设置月和日。</p>

<p>请注意，月份从 0 计数。十二月是 11：</p>

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

<script>

var d = new Date();

d.setFullYear(2020, 11, 3);

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

</script>

</body>

</html>

setMonth() 方法

setMonth() 方法设置日期对象的月份（0-11）：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript setMonth()</h2>

<p>setMonth() 方法设置日期对象的月份。</p>

<p>请注意，月份从 0 计数。十二月是 11：</p>

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

<script>

var d = new Date();

d.setMonth(11);

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

</script>

</body>

</html>

setDate() 方法

setDate() 方法设置日期对象的日（1-31）：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript setDate()</h2>

<p>setDate() 方法设置日期对象的日：</p>

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

<script>

var d = new Date();

d.setDate(15);

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

</script>

</body>

</html>

setDate() 方法也可用于将天数添加到日期：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript setDate()</h2>

<p>setDate() 方法可用于向日期添加天数。</p>

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

<script>

var d = new Date();

d.setDate(d.getDate() + 50);

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

</script>

</body>

</html>

如果添加天数，切换月份或年份，则更改将由 Date 对象自动处理。

setHours() 方法

setHours() 方法设置日期对象的小时（0-23）：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript setHours()</h2>

<p>setHours() 方法设置日期对象的小时数：</p>

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

<script>

var d = new Date();

d.setHours(22);

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

</script>

</body>

</html>

setMinutes() 方法

setMinutes() 方法设置日期对象的分钟（0-59）：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript setMinutes()</h2>

<p>setMinutes() 方法设置日期对象的分钟数（0-59）：</p>

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

<script>

var d = new Date();

d.setMinutes(30);

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

</script>

</body>

</html>

setSeconds() 方法

setSeconds() 方法设置日期对象的秒数（0-59）：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript setSeconds()</h2>

<p>setSeconds() 方法设置日期对象的秒数（0-59）：</p>

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

<script>

var d = new Date();

d.setSeconds(30);

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

</script>

</body>

</html>

比较日期

日期可以很容易地进行比较。

下面的例子把今日与 2049 年 1 月 16 日进行比较：

实例

<!DOCTYPE html>

<html>

<body>

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

<script>

var today, someday, text;

today = new Date();

someday = new Date();

someday.setFullYear(2100, 0, 15);

if (someday > today) {

text = "今天在 2100 年 1 月 15 日之前。";

} else {

text = "今天在 2100 年 1 月 15 日之后。";

}

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

</script>

</body>

</html>

JavaScript 从 0 到 11 计数月份。1 月是 0。12 月是 11。

# 第33章 JavaScript 数学

JavaScript Math 对象允许您对数字执行数学任务。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Math.PI</h1>

<p>Math.PI 返回圆周长与直径的比率：</p>

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

<script>

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

</script>

</body>

</html>

Math.round()

Math.round(x) 的返回值是 x 四舍五入为最接近的整数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Math.round()</h1>

<p>Math.round(x) 返回 x 四舍五入到最接近的整数的值：</p>

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

<script>

document.getElementById("demo").innerHTML = Math.round(4.4);

</script>

</body>

</html>

Math.pow()

Math.pow(x, y) 的返回值是 x 的 y 次幂：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Math.pow()</h1>

<p>Math.pow(x,y) 返回 x 的 y 次幂的值：</p>

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

<script>

document.getElementById("demo").innerHTML = Math.pow(8,2);

</script>

</body>

</html>

Math.sqrt()

Math.sqrt(x) 返回 x 的平方根：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Math.sqrt()</h1>

<p>Math.sqrt(x) 返回 x 的平方根：</p>

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

<script>

document.getElementById("demo").innerHTML = Math.sqrt(64);

</script>

</body>

</html>

Math.abs()

Math.abs(x) 返回 x 的绝对（正）值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Math.abs()</h1>

<p>Math.abs(x) 返回 x 的绝对值：</p>

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

<script>

document.getElementById("demo").innerHTML = Math.abs(-4.4);

</script>

</body>

</html>

Math.ceil()

Math.ceil(x) 的返回值是 x 上舍入最接近的整数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Math.ceil()</h1>

<p>Math.ceil() 把数字上舍入为最接近的整数：</p>

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

<script>

document.getElementById("demo").innerHTML = Math.ceil(4.4);

</script>

</body>

</html>

Math.floor()

Math.floor(x) 的返回值是 x 下舍入最接近的整数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Math.floor()</h1>

<p>Math.floor(x) 返回 x 被下舍入为最接近整数的值：</p>

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

<script>

document.getElementById("demo").innerHTML = Math.floor(4.7);

</script>

</body>

</html>

Math.sin()

Math.sin(x) 返回角 x（以弧度计）的正弦（介于 -1 与 1 之间的值）。

如果您希望使用角度替代弧度，则需要将角度转换为弧度：

Angle in radians = Angle in degrees x PI / 180.

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Math.sin()</h1>

<p>Math.sin(x) 返回 x（以弧度计）的正弦：</p>

<p>以弧度计的角度 = (以度数计的角度) \* PI / 180。</p>

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

<script>

document.getElementById("demo").innerHTML =

"The sine value of 90 degrees is " + Math.sin(90 \* Math.PI / 180);

</script>

</body>

</html>

Math.cos()

Math.cos(x) 返回角 x（以弧度计）的余弦（介于 -1 与 1 之间的值）。

如果您希望使用角度替代弧度，则需要将角度转换为弧度：

Angle in radians = Angle in degrees x PI / 180.

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Math.cos()</h1>

<p>Math.cos(x) 返回 x（以弧度计）的余弦：</p>

<p>以弧度计的角度 = (以度数计的角度) \* PI / 180。</p>

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

<script>

document.getElementById("demo").innerHTML =

"The cosine value of 0 degrees is " + Math.cos(0 \* Math.PI / 180);

</script>

</body>

</html>

Math.min() 和 Math.max()

Math.min() 和 Math.max() 可用于查找参数列表中的最低或最高值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Math.min()</h1>

<p>Math.min() 返回参数列表中的最低值：</p>

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

<script>

document.getElementById("demo").innerHTML =

Math.min(0, 150, 30, 20, -8, -200);

</script>

</body>

</html>

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Math.max()</h1>

<p>Math.max() 返回参数列表中的最高值：</p>

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

<script>

document.getElementById("demo").innerHTML =

Math.max(0, 150, 30, 20, -8, -200);

</script>

</body>

</html>

Math.random()

Math.random() 返回介于 0（包括） 与 1（不包括） 之间的随机数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Math.random()</h1>

<p>Math.random() 返回 0 与 1 之间的随机值：</p>

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

<script>

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

</script>

</body>

</html>

Math 属性（常量）

JavaScript 提供了可由 Math 对象访问的 8 个数学常量：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数学常量</h1>

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

<script>

document.getElementById("demo").innerHTML =

"<p><b>Math.E:</b> " + Math.E + "</p>" +

"<p><b>Math.PI:</b> " + Math.PI + "</p>" +

"<p><b>Math.SQRT2:</b> " + Math.SQRT2 + "</p>" +

"<p><b>Math.SQRT1\_2:</b> " + Math.SQRT1\_2 + "</p>" +

"<p><b>Math.LN2:</b> " + Math.LN2 + "</p>" +

"<p><b>Math.LN10:</b> " + Math.LN10 + "</p>" +

"<p><b>Math.LOG2E:</b> " + Math.LOG2E + "</p>" +

"<p><b>Math.Log10E:</b> " + Math.LOG10E + "</p>";

</script>

</body>

</html>

Math 构造器

与其他全局对象不同，Math对象没有构造函数。方法和属性是静态的。

可以在不首先创建Math对象的情况下使用所有方法和属性（常量）。

Math 对象方法

|  |  |
| --- | --- |
| 方法 | 描述 |
| abs(x) | 返回 x 的绝对值 |
| acos(x) | 返回 x 的反余弦值，以弧度计 |
| asin(x) | 返回 x 的反正弦值，以弧度计 |
| atan(x) | 以介于 -PI/2 与 PI/2 弧度之间的数值来返回 x 的反正切值。 |
| atan2(y,x) | 返回从 x 轴到点 (x,y) 的角度 |
| ceil(x) | 对 x 进行上舍入 |
| cos(x) | 返回 x 的余弦 |
| exp(x) | 返回 Ex 的值 |
| floor(x) | 对 x 进行下舍入 |
| log(x) | 返回 x 的自然对数（底为e） |
| max(x,y,z,...,n) | 返回最高值 |
| min(x,y,z,...,n) | 返回最低值 |
| pow(x,y) | 返回 x 的 y 次幂 |
| random() | 返回 0 ~ 1 之间的随机数 |
| round(x) | 把 x 四舍五入为最接近的整数 |
| sin(x) | 返回 x（x 以角度计）的正弦 |
| sqrt(x) | 返回 x 的平方根 |
| tan(x) | 返回角的正切 |

# 第34章 JavaScript 随机

Math.random()

Math.random() 返回 0（包括） 至 1（不包括） 之间的随机数：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Math.random()</h2>

<p>Math.random() 返回 0（包含）和 1（不包括）之间的随机数：</p>

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

<script>

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

</script>

</body>

</html>

Math.random() 总是返回小于 1 的数。

JavaScript 随机整数

Math.random() 与 Math.floor() 一起使用用于返回随机整数。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Math</h2>

<p>Math.floor(Math.random() \* 10) 返回 0 与 9 之间的随机整数（均包含）：</p>

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

<script>

document.getElementById("demo").innerHTML =

Math.floor(Math.random() \* 10);

</script>

</body>

</html>

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Math</h2>

<p>Math.floor(Math.random() \* 11) 返回 0 与 10 之间的随机整数（均包含）：</p>

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

<script>

document.getElementById("demo").innerHTML =

Math.floor(Math.random() \* 11);

</script>

</body>

</html>

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Math</h2>

<p>Math.floor(Math.random() \* 100)) 返回 0 与 99 之间的随机整数（均包含）：</p>

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

<script>

document.getElementById("demo").innerHTML =

Math.floor(Math.random() \* 100);

</script>

</body>

</html>

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Math</h2>

<p>Math.floor() 与 Math.random() \* 101 一起使用，返回 0 与 100 之间的随机整数（均包含）：</p>

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

<script>

document.getElementById("demo").innerHTML =

Math.floor(Math.random() \* 101);

</script>

</body>

</html>

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Math</h2>

<p>Math.floor(Math.random() \* 10) + 1) 返回 1 与 10 之间的随机整数（均包含）：</p>

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

<script>

document.getElementById("demo").innerHTML =

Math.floor(Math.random() \* 10) + 1;

</script>

</body>

</html>

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Math</h2>

<p>Math.floor(Math.random() \* 100) + 1) 返回 1 与 100 之间的随机整数（均包含）：</p>

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

<script>

document.getElementById("demo").innerHTML =

Math.floor(Math.random() \* 100) + 1;

</script>

</body>

</html>

一个适当的随机函数

正如你从上面的例子看到的，创建一个随机函数用于生成所有随机整数是一个好主意。

这个 JavaScript 函数始终返回介于 min（包括）和 max（不包括）之间的随机数：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Math.random()</h2>

<p>每当您点击按钮，getRndInteger(min, max) 就会返回 0 与 9（均包含）之间的随机数：</p>

<button onclick="document.getElementById('demo').innerHTML = getRndInteger(0,10)">点击我</button>

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

<script>

function getRndInteger(min, max) {

return Math.floor(Math.random() \* (max - min)) + min;

}

</script>

</body>

</html>

这个 JavaScript 函数始终返回介于 min 和 max（都包括）之间的随机数：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Math.random()</h2>

<p>每当您点击按钮，getRndInteger(min, max) 就会返回 1 与 10（均包含）之间的随机数：</p>

<button onclick="document.getElementById('demo').innerHTML = getRndInteger(1,10)">点击我</button>

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

<script>

function getRndInteger(min, max) {

return Math.floor(Math.random() \* (max - min + 1) ) + min;

}

</script>

</body>

</html>

# 第35章 JavaScript 逻辑

JavaScript 布尔（逻辑）代表两个值之一：true 或 false。

布尔值

通常，在编程中，您会需要只能有两个值之一的数据类型，比如

YES / NO

ON / OFF

TRUE / FALSE

鉴于此，JavaScript 提供一种布尔数据类型。它只接受值 true 或 false。

Boolean() 函数

您可以使用 Boolean() 函数来确定表达式（或变量）是否为真：

实例

<!DOCTYPE html>

<html>

<body>

<p>显示 Boolean(10 > 9) 的值：</p>

<button onclick="myFunction()">试一试</button>

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

<script>

function myFunction() {

document.getElementById("demo").innerHTML = Boolean(10 > 9);

}

</script>

</body>

</html>

或者甚至更简单：

实例

<!DOCTYPE html>

<html>

<body>

<p>显示 10 > 9 的值：</p>

<button onclick="myFunction()">试一试</button>

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

<script>

function myFunction() {

document.getElementById("demo").innerHTML = 10 > 9;

}

</script>

</body>

</html>

比较和条件

JS 比较这一章列出了完整的比较运算符。

JS 条件这一章列出了完整的条件语句。

下面是一些例子：

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 实例 |
| == | 等于 | if (day == "Monday") |
| > | 大于 | if (salary > 9000) |
| < | 小于 | if (age < 18) |

表达式的布尔值是 JavaScript 比较和条件的基础。

所有具有“真实”值的即为 True

实例

<!DOCTYPE html>

<html>

<body>

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

<script>

var b1 = Boolean(100);

var b2 = Boolean(3.14);

var b3 = Boolean(-15);

var b4 = Boolean("Hello");

var b5 = Boolean('false');

var b6 = Boolean(1 + 7 + 3.14);

document.getElementById("demo").innerHTML =

"100 是 " + b1 + "<br>" +

"3.14 是 " + b2 + "<br>" +

"-15 是 " + b3 + "<br>" +

"任何（非空）字符串是 " + b4 + "<br>" +

"即使字符串 'false' 也是 " + b5 + "<br>" +

"任何表达式（除了零）是 " + b6;

</script>

</body>

</html>

所有不具有“真实”值的即为 False

0（零）的布尔值为 false：

<!DOCTYPE html>

<html>

<body>

<p>显示 0 的布尔值：</p>

<button onclick="myFunction()">试一试</button>

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

<script>

function myFunction() {

var x = 0;

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

}

</script>

</body>

</html>

-0 （负零）的布尔值为 false：

<!DOCTYPE html>

<html>

<body>

<p>显示 -0 的布尔值：</p>

<button onclick="myFunction()">试一试</button>

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

<script>

function myFunction() {

var x = -0;

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

}

</script>

</body>

</html>

""（空值）的布尔值为 false：

<!DOCTYPE html>

<html>

<body>

<p>显示 "" 的布尔值：</p>

<button onclick="myFunction()">试一试</button>

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

<script>

function myFunction() {

var x = "";

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

}

</script>

</body>

</html>

undefined 的布尔值是 false：

<!DOCTYPE html>

<html>

<body>

<p>显示 undefined 的布尔值：</p>

<button onclick="myFunction()">试一试</button>

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

<script>

function myFunction() {

var x;

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

}

</script>

</body>

</html>

null 的布尔值是 false：

<!DOCTYPE html>

<html>

<body>

<p>显示 null 的布尔值：</p>

<button onclick="myFunction()">试一试</button>

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

<script>

function myFunction() {

var x = null;

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

}

</script>

</body>

</html>

false 的布尔值（正如您猜到的）是 false：

<!DOCTYPE html>

<html>

<body>

<p>显示 false 的布尔值：</p>

<button onclick="myFunction()">试一试</button>

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

<script>

function myFunction() {

var x = false;

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

}

</script>

</body>

</html>

NaN 的布尔值是 false：

var x = 10 / "H";

Boolean(x); // 返回 false

<!DOCTYPE html>

<html>

<body>

<p>显示 NaN 的布尔值：</p>

<button onclick="myFunction()">试一试</button>

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

<script>

function myFunction() {

var x = 10 / "H";

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

}

</script>

</body>

</html>

布尔可以是对象

通常 JavaScript 布尔是由字面量创建的原始值：

var x = false

但是布尔也可以通过关键词 new 作为对象来定义：

var y = new Boolean(false)

实例

<!DOCTYPE html>

<html>

<body>

<p>永远不要将布尔值创建为对象。</p>

<p>无法安全地比较布尔和对象。</p>

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

<script>

var x = false; // x 是布尔值

var y = new Boolean(false); // y 是对象

document.getElementById("demo").innerHTML = typeof x + "<br>" + typeof y;

</script>

</body>

</html>

不要创建布尔对象。它会拖慢执行速度。

new 关键词会使代码复杂化，并产生某些意想不到的结果：

当使用 == 运算符时，相等的布尔是相等的：

实例

<!DOCTYPE html>

<html>

<body>

<p>永远不要将布尔值创建为对象。</p>

<p>无法安全地比较布尔和对象。</p>

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

<script>

var x = false; // x 是布尔值

var y = new Boolean(false); // y 是对象

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

</script>

</body>

</html>

当使用 === 运算符时，相等的布尔是不相等的，因为 === 运算符需要在类型和值两方面同时相等。

实例

<!DOCTYPE html>

<html>

<body>

<p>永远不要将布尔值创建为对象。</p>

<p>无法安全地比较布尔和对象。</p>

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

<script>

var x = false; // x 是数字

var y = new Boolean(false); // y 是对象

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

</script>

</body>

</html>

或者甚至更糟。对象无法进行比较：

实例

<!DOCTYPE html>

<html>

<body>

<p>永远不要将布尔值创建为对象。</p>

<p>无法安全地比较布尔和对象。</p>

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

<script>

var x = new Boolean(false); // x 是对象

var y = new Boolean(false); // y 是对象

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

</script>

</body>

</html>

# 第36章 JavaScript 比较

比较和逻辑运算符用于测试 true 或 false。

比较运算符

比较运算符在逻辑语句中使用，以判定变量或值是否相等。

我们给定 x = 5，下表中解释了比较运算符：

|  |  |  |  |
| --- | --- | --- | --- |
| 运算符 | 描述 | 比较 | 返回 |
| == | 等于 | x == 8 | false |
| x == 5 | true | 试一试 |  |
| x == "5" | true | 试一试 |  |
| === | 值相等并且类型相等 | x === 5 | true |
| x === "5" | false | 试一试 |  |
| != | 不相等 | x != 8 | true |
| !== | 值不相等或类型不相等 | x !== 5 | false |
| x !== "5" | true | 试一试 |  |
| x !== 8 | true | 试一试 |  |
| > | 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 |

如何使用

比较运算符可用在条件语句中对值进行比较，并根据结果采取动作：

if (age < 18) text = "太年轻";

本教程接下来的章节中，您将在学到更多有关条件语句的知识。

逻辑运算符

逻辑运算符用于判定变量或值之间的逻辑。

我们给定 x = 6 且 y = 3，下表解释了逻辑运算符：

|  |  |  |
| --- | --- | --- |
| 运算符 | 描述 | 例子 |
| && | 与 | (x < 10 && y > 1) 为 true |
| || | 或 | (x == 5 || y == 5) 为 false |
| ! | 非 | !(x == y) 为 true |

条件（三元）运算符

JavaScript 也包含了可基于某些条件向变量赋值的条件运算符。

语法

variablename = (condition) ? value1:value2

实例

<!DOCTYPE html>

<html>

<body>

<p>输入您的年龄然后点击这个按钮：</p>

<input id="age" value="18" />

<button onclick="myFunction()">试一试</button>

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

<script>

function myFunction() {

var age, voteable;

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

voteable = (age < 18) ? "好年轻":"够成熟";

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

}

</script>

</body>

</html>

如果变量 age 的值小于 18，变量 voteable 的值将是 "太年轻"，否则变量 voteable 的值将是 "足够成熟"。

比较不同的类型

比较不同类型的数据也许会出现不可预料的结果。

如果将字符串与数字进行比较，那么在做比较时 JavaScript 会把字符串转换为数值。空字符串将被转换为 0。非数值字符串将被转换为始终为 false 的 NaN。

|  |  |
| --- | --- |
| 案例 | 值 |
| 2 < 12 | true |
| 2 < "12" | true |
| 2 < "Bill" | false |
| 2 > "Bill" | false |
| 2 == "Bill" | false |
| "2" < "12" | false |
| "2" > "12" | true |
| "2" == "12" | false |

当比较两个字符串时，"2" 大于 "12"，因为（按照字母排序）1 小于 2。

为了确保正确的结果，在比较值前应该把变量转换为合适的类型：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 比较</h1>

<p>请输入您的年龄然后点击按钮：</p>

<input id="age" value="18" />

<button onclick="myFunction()">试一试</button>

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

<script>

function myFunction() {

var age, voteable;

age = Number(document.getElementById("age").value);

if (isNaN(age)) {

voteable = "输入值不是数字";

} else {

voteable = (age < 18) ? "好年轻" : "够成熟";

}

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

}

</script>

</body>

</html>

# 第37章 JavaScript 条件

条件语句用于基于不同条件执行不同的动作。

条件语句

在您写代码时，经常会需要基于不同判断执行不同的动作。

您可以在代码中使用条件语句来实现这一点。

在 JavaScript 中，我们可使用如下条件语句：

使用 if 来规定要执行的代码块，如果指定条件为 true

使用 else 来规定要执行的代码块，如果相同的条件为 false

使用 else if 来规定要测试的新条件，如果第一个条件为 false

使用 switch 来规定多个被执行的备选代码块

if 语句

请使用 if 语句来规定假如条件为 true 时被执行的 JavaScript 代码块。

语法

if (条件) {

如果条件为 true 时执行的代码

}

注释：if 使用小写字母。大写字母（IF 或 If）会产生 JavaScript 错误。

实例

如果时间早于 18:00，则发出 "Good day" 的问候：

if (hour < 18) {

greeting = "Good day";

}

如果时间早于 18 点，则 greeting 的结果将是：

<!DOCTYPE html>

<html>

<body>

<p>如果小时小于18:00，显示“美好的一天！”：</p>

<p id="demo">晚安</p>

<script>

if (new Date().getHours() < 18) {

document.getElementById("demo").innerHTML = "美好的一天！";

}

</script>

</body>

</html>

else 语句

请使用 else 语句来规定假如条件为 false 时的代码块。

if (条件) {

条件为 true 时执行的代码块

} else {

条件为 false 时执行的代码块

}

实例

如果 hour 小于 18，创建 "Good day" 问候，否则 "Good evening"：

if (hour < 18) {

greeting = "Good day";

} else {

greeting = "Good evening";

}

greeting 的结果：

<!DOCTYPE html>

<html>

<body>

<p>单击按钮以显示基于时间的问候语：</p>

<button onclick="myFunction()">试一试</button>

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

<script>

function myFunction() {

var hour = new Date().getHours();

var greeting;

if (hour < 18) {

greeting = "日安";

} else {

greeting = "晚安";

}

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

}

</script>

</body>

</html>

else if 语句

请使用 else if 来规定当首个条件为 false 时的新条件。

语法

if (条件 1) {

条件 1 为 true 时执行的代码块

} else if (条件 2) {

条件 1 为 false 而条件 2 为 true 时执行的代码块

} else {

条件 1 和条件 2 同时为 false 时执行的代码块

}

实例

如果时间早于 10:00，则创建 "Good morning" 问候，如果不是，但时间早于 18:00，则创建 "Good day" 问候，否则创建 "Good evening"：

if (time < 10) {

greeting = "Good morning";

} else if (time < 18) {

greeting = "Good day";

} else {

greeting = "Good evening";

}

greeting 的结果：

<!DOCTYPE html>

<html>

<body>

<p>单击按钮以显示基于时间的问候语：</p>

<button onclick="myFunction()">试一试</button>

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

<script>

function myFunction() {

var greeting;

var time = new Date().getHours();

if (time < 10) {

greeting = "早安";

} else if (time < 20) {

greeting = "日安";

} else {

greeting = "晚安";

}

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

}

</script>

</body>

</html>

# 第38章 JavaScript Switch 语句

switch 语句用于基于不同条件执行不同动作。

JavaScript Switch 语句

请使用 switch 语句来选择多个需被执行的代码块之一。

语法

switch(表达式) {

case n:

代码块

break;

case n:

代码块

break;

default:

默认代码块

}

代码解释：

计算一次 switch 表达式

把表达式的值与每个 case 的值进行对比

如果存在匹配，则执行关联代码

实例

getDay() 方法返回 0 至 6 之间的周名数字（weekday number）。

(Sunday=0, Monday=1, Tuesday=2 ..)

本例使用周名数字来计算周的名称：

switch (new Date().getDay()) {

case 0:

day = "星期天";

break;

case 1:

day = "星期一";

break;

case 2:

day = "星期二";

break;

case 3:

day = "星期三";

break;

case 4:

day = "星期四";

break;

case 5:

day = "星期五";

break;

case 6:

day = "星期六";

}

结果将是：

<!DOCTYPE html>

<html>

<body>

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

<script>

var day;

switch (new Date().getDay()) {

case 0:

day = "周日";

break;

case 1:

day = "周一";

break;

case 2:

day = "周二";

break;

case 3:

day = "周三";

break;

case 4:

day = "周四";

break;

case 5:

day = "周五";

break;

case 6:

day = "周六";

}

document.getElementById("demo").innerHTML = "今天是" + day;

</script>

</body>

</html>

break 关键词

如果 JavaScript 遇到 break 关键词，它会跳出 switch 代码块。

此举将停止代码块中更多代码的执行以及 case 测试。

如果找到匹配，并完成任务，则随机中断执行（break）。无需更多测试。

break 能够节省大量执行时间，因为它会“忽略” switch 代码块中的其他代码的执行。

不必中断 switch 代码块中的最后一个 case。代码块在此处会自然结束。

default 关键词

default 关键词规定不存在 case 匹配时所运行的代码：

实例

getDay() 方法返回 0 至 6 的数字周名。

如果今日既不是周六（6）也不是周日（0），则输出一段默认消息：

switch (new Date().getDay()) {

case 6:

text = "今天是周六";

break;

case 0:

text = "今天是周日";

break;

default:

text = "期待周末~";

}

text 的结果是：

期待周末~

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript switch</h2>

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

<script>

var text;

switch (new Date().getDay()) {

case 6:

text = "今天是周六";

break;

case 0:

text = "今天是周日";

break;

default:

text = "期待周末";

}

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

</script>

</body>

</html>

默认的 case 不必是 switch 代码块中最后一个 case：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript switch</h2>

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

<script>

var text;

switch (new Date().getDay()) {

default:

text = "期待周末";

break;

case 6:

text = "今天是周六";

break;

case 0:

text = "今天是周日";

}

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

</script>

</body>

</html>

如果 default 不是 switch 代码块中最后一个 case，请记得用 break 结束默认 case。

常见的代码块

有时您会需要不同的 case 来使用相同的代码。

在本例中，case 4 和 5 分享相同的代码块，而 0 和 6 分享另一段代码块：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript switch</h2>

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

<script>

var text;

switch (new Date().getDay()) {

case 4:

case 5:

text = "周末马上就到了";

break;

case 0:

case 6:

text = "今天是周末";

break;

default:

text = "期待周末";

}

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

</script>

</body>

</html>

Switching 的细节

如果多种 case 匹配一个 case 值，则选择第一个 case。

如果未找到匹配的 case，程序将继续使用默认 label。

如果未找到默认 label，程序将继续 switch 后的语句。

严格的比较

Switch case 使用严格比较（===）。

值必须与要匹配的类型相同。

只有操作数属于同一类型时，严格比较才能为 true。

在这个例子中，x 将不匹配：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript switch</h2>

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

<script>

var x = "0";

switch (x) {

case 0:

text = "Off";

break;

case 1:

text = "On";

break;

default:

text = "未找到值";

}

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

</script>

</body>

</html>

# 第39章 JavaScript For 循环

循环可多次执行代码块。

JavaScript 循环

假如您需要运行代码多次，且每次使用不同的值，那么循环（loop）相当方便使用。

通常我们会遇到使用数组的例子：

不需要这样写：

text += cars[0] + "<br>";

text += cars[1] + "<br>";

text += cars[2] + "<br>";

text += cars[3] + "<br>";

text += cars[4] + "<br>";

text += cars[5] + "<br>";

您能够这样写：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 循环</h1>

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

<script>

var cars = ["BMW", "Volvo", "porsche", "Ford", "Fiat", "Audi"];

var text = "";

var i;

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

text += cars[i] + "<br>";

}

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

</script>

</body>

</html>

不同类型的循环

JavaScript 支持不同类型的循环：

for - 多次遍历代码块

for/in - 遍历对象属性

while - 当指定条件为 true 时循环一段代码块

do/while - 当指定条件为 true 时循环一段代码块

For 循环

for 循环是在您希望创建循环时经常使用的工具。

for 循环的语法如下：

for (语句 1; 语句 2; 语句 3) {

要执行的代码块

}

语句 1 在循环（代码块）开始之前执行。

语句 2 定义运行循环（代码块）的条件。

语句 3 会在循环（代码块）每次被执行后执行。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 循环</h1>

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

<script>

var text = "";

var i;

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

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

}

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

</script>

</body>

</html>

从上面的代码中，您可以了解到：

语句 1 在循环开始之前设置了一个变量（var i = 0）。

语句 2 定义运行循环的条件（i 必须小于 5）。

语句 3 会在代码块每次执行之后对值进行递增（i++）。

语句 1

通常，您会使用语句 1 来初始化循环中所使用的的变量（i = 0）。

但情况并不总是这样，JavaScript 不会在意。语句 1 是可选的。

您可以在语句 1 中初始化多个值（由逗号分隔）：

实例

<!DOCTYPE html>

<html>

<body>

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

<script>

var cars = ["BMW", "Volvo", "porsche", "Ford"];

var i, len, text;

for (i = 0, len = cars.length, text = ""; i < len; i++) {

text += cars[i] + "<br>";

}

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

</script>

</body>

</html>

而且您还可以省略语句 1（比如在循环开始前设置好值）：

实例

<!DOCTYPE html>

<html>

<body>

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

<script>

var cars = ["BMW", "Volvo", "porsche", "Ford"];

var i = 2;

var len = cars.length;

var text = "";

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

text += cars[i] + "<br>";

}

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

</script>

</body>

</html>

语句 2

通常语句 2 用于计算初始变量的条件。

但情况并不总是这样，JavaScript 不会在意。语句 2 也是可选的。

如果语句 2 返回 true，那么循环会重新开始，如果返回 false，则循环将结束。

如果省略语句 2，那么必须在循环中提供一个 break。否则循环永远不会结束。请在下一章阅读更多关于 break 的内容。

语句 3

通常语句 3 会递增初始变量的值。

但情况并不总是这样，JavaScript 不会在意。语句 3 也是可选的。

语句 3 可做任何事情，比如负递增（i--），正递增（i = i + 15），或者任何其他事情。

语句 3 也可被省略（比如当您在循环内递增值时）：

实例

<!DOCTYPE html>

<html>

<body>

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

<script>

var cars = ["BMW", "Volvo", "porsche", "Ford"];

var i = 0;

var len = cars.length;

var text = "";

for (; i < len; ) {

text += cars[i] + "<br>";

i++;

}

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

</script>

</body>

</html>

For/In 循环

JavaScript for/in 语句遍历对象的属性：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 循环</h1>

<p>for/in 语句循环遍历对象的属性。</p>

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

<script>

var txt = "";

var person = {fname:"Bill", lname:"Gates", age:62};

var x;

for (x in person) {

txt += person[x] + " ";

}

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

</script>

</body>

</html>

While 循环

我们将在下一章学习 while 循环和 do/while 循环。

# 第40章 JavaScript For In

For In 循环

JavaScript for in 语句循环遍历对象的属性：

语法

for (key in object) {

// code block to be executed

}

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript For In Loop</h2>

<p>for in 语句循环遍历对象的属性：</p>

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

<script>

const person = {fname:"Bill", lname:"Gates", age:19};

let txt = "";

for (let x in person) {

txt += person[x] + " ";

}

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

</script>

</body>

</html>

例子解释

for in 循环遍历 person 对象

每次迭代返回一个键 (x)

键用于访问键的值

键的值为 person[x]

For In 遍历数组

JavaScript for in 语句也可以遍历数组的属性：

语法

for (variable in array) {

code

}

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript For In</h2>

<p>for in 语句可以遍历数组值：</p>

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

<script>

const numbers = [45, 4, 9, 16, 25];

let txt = "";

for (let x in numbers) {

txt += numbers[x] + "<br>";

}

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

</script>

</body>

</html>

如果索引顺序很重要，请不要在数组上使用 for in。

索引顺序依赖于实现，可能不会按照您期望的顺序访问数组值。

当顺序很重要时，最好使用 for 循环、for of 循环或 Array.forEach()。

Array.forEach()

forEach() 方法为每个数组元素调用一次函数（回调函数）。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Array.forEach()</h2>

<p>为每个数组元素调用一次函数。</p>

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

<script>

const numbers = [45, 4, 9, 16, 25];

let txt = "";

numbers.forEach(myFunction);

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

function myFunction(value, index, array) {

txt += value + "<br>";

}

</script>

</body>

</html>

请注意，该函数采用 3 个参数：

项目值

项目索引

数组本身

上面的例子仅使用 value 参数。可以改写为：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Array.forEach()</h2>

<p>为每个数组元素调用一次函数。</p>

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

<script>

const numbers = [45, 4, 9, 16, 25];

let txt = "";

numbers.forEach(myFunction);

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

function myFunction(value) {

txt += value + "<br>";

}

</script>

</body>

</html>

# 第41章 JavaScript For Of

For Of 循环

JavaScript for of 语句循环遍历可迭代对象的值。

它允许您循环遍历可迭代的数据结构，例如数组、字符串、映射、节点列表等：

语法

for (variable of iterable) {

// code block to be executed

}

variable - 对于每次迭代，下一个属性的值都会分配给变量。变量可以用 const、let 或 var 声明。

iterable - 具有可迭代属性的对象。

遍历数组

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript For Of 循环</h2>

<p>for of 语句循环遍历任何可迭代对象的值：</p>

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

<script>

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

let text = "";

for (let x of cars) {

text += x + "<br>";

}

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

</script>

</body>

</html>

遍历字符串

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript For Of 循环</h2>

<p>for of 语句循环遍历可迭代对象的值。</p>

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

<script>

let language = "JavaScript";

let text = "";

for (let x of language) {

text += x + "<br>";

}

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

</script>

</body>

</html>

While 循环

我们将在下一章讲解 while 循环和 do/while 循环。

# 第42章 JavaScript While 循环

只要条件为 true，循环能够一直执行代码块。

While 循环

while 循环会一直循环代码块，只要指定的条件为 true。

语法

while (条件) {

要执行的代码块

}

实例

在下面的例子中，循环中的代码将运行，一遍又一遍，只要变量（i）小于 10：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript while</h2>

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

<script>

var text = "";

var i = 0;

while (i < 10) {

text += "<br>数字是 " + i;

i++;

}

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

</script>

</body>

</html>

如果您忘了对条件中使用的变量进行递增，那么循环永不会结束。这会导致浏览器崩溃。

Do/While 循环

do/while 循环是 while 循环的变体。在检查条件是否为真之前，这种循环会执行一次代码块，然后只要条件为真就会重复循环。

语法

do {

要执行的代码块

}

while (条件);

实例

下面的例子使用了 do/while 循环。该循环会执行至少一次，即使条件为 false，因为代码块会在条件测试之前执行：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript do ... while</h2>

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

<script>

var text = ""

var i = 0;

do {

text += "<br>数字是 " + i;

i++;

}

while (i < 10);

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

</script>

</body>

</html>

不要忘记对条件中所用变量进行递增，否则循环永不会结束！

比较 For 与 While

如果您已经阅读了之前关于循环的章节，您会发现 while 循环与 for 循环相当类似，其中的语句 1 和 语句 2 都可以省略。

本例中的循环使用 for 循环来提取 cars 数组中的汽车品牌：

实例

<!DOCTYPE html>

<html>

<body>

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

<script>

var cars = ["BMW", "Volvo", "porsche", "Ford"];

var i = 0;

var text = "";

for (;cars[i];) {

text += cars[i] + "<br>";

i++;

}

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

</script>

</body>

</html>

本例中的循环使用 while 循环来提取 cars 数组中的汽车品牌：

实例

<!DOCTYPE html>

<html>

<body>

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

<script>

var cars = ["BMW", "Volvo", "porsche", "Ford"];

var i = 0;

var text = "";

while (cars[i]) {

text += cars[i] + "<br>";

i++;

}

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

</script>

</body>

</html>

# 第43章 JavaScript Break 和 Continue

break 语句“跳出”循环。

continue 语句“跳过”循环中的一个迭代。

Break 语句

在本教程稍早的章节中，您已见到了 break 语句。它被用于“跳出” switch 语句。

break 语句也可用于跳出循环。

break 语句会中断循环，并继续执行循环之后的代码（如果有）：

实例

<!DOCTYPE html>

<html>

<body>

<p>带 break 的循环</p>

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

<script>

var text = "";

var i;

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

if (i === 3) { break; }

text += "数字是 " + i + "<br>";

}

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

</script>

</body>

</html>

Continue 语句

continue 语句中断（循环中）的一个迭代，如果发生指定的条件。然后继续循环中的下一个迭代。

本例跳过值 3 ：

实例

<!DOCTYPE html>

<html>

<body>

<p>这个循环在 i=3 时跳过这一步：</p>

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

<script>

var text = "";

var i;

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

if (i === 3) { continue; }

text += "数字是 " + i + "<br>";

}

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

</script>

</body>

</html>

JavaScript 标签

如需标记 JavaScript 语句，请将标签名和冒号置于语句之前：

label:

statements

break 和 continue 语句是仅有的可“跳出”代码块的 JavaScript 语句。

语法：

break labelname;

continue labelname;

continue 语句（不论有无标签引用）只能用于跳过一个迭代。

break 语句，如果没有标签引用，只能用于跳出一个循环或一个 switch。

如果有标签引用，则 break 语句可用于跳出任意代码块：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript break</h2>

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

<script>

var cars = ["BMW", "Volvo", "porsche", "Ford"];

var text = "";

list: {

text += cars[0] + "<br>";

text += cars[1] + "<br>";

break list;

text += cars[2] + "<br>";

text += cars[3] + "<br>";

}

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

</script>

</body>

</html>

代码块指的是 { 与 } 直接的代码片段。

# 第44章 JavaScript typeof

JavaScript typeof

在 JavaScript 中有 5 种不同的可以包含值的数据类型：

string

number

boolean

object

function

有 6 种类型的对象：

Object

Date

Array

String

Number

Boolean

以及 2 种不能包含值的数据类型：

null

undefined

typeof 运算符

您可以使用 typeof 运算符来确定 JavaScript 变量的数据类型。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript typeof 运算符</h1>

<p>typeof 运算符返回变量、对象、函数或表达式的类型。</p>

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

<script>

document.getElementById("demo").innerHTML =

typeof "john" + "<br>" +

typeof 3.14 + "<br>" +

typeof NaN + "<br>" +

typeof false + "<br>" +

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

typeof {name:'john', age:34} + "<br>" +

typeof new Date() + "<br>" +

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

typeof myCar + "<br>" +

typeof null;

</script>

</body>

</html>

请注意：

NaN 的数据类型是数字

数组的数据类型是对象

日期的数据类型是对象

null 的数据类型是 object

未定义变量的数据类型为 undefined \*

未赋值的变量的数据类型也是 undefined \*

您无法使用 typeof 来确定 JavaScript 对象是否是数组（或日期）。

原始日期

原始数据值指的是没有附加属性和方法的单个简单数据值。

typeof 运算符可以返回以下原始类型之一：

string

number

boolean

undefined

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript typeof</h1>

<p>typeof 运算符返回变量或表达式的类型。</p>

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

<script>

document.getElementById("demo").innerHTML =

typeof "john" + "<br>" +

typeof 3.14 + "<br>" +

typeof true + "<br>" +

typeof false + "<br>" +

typeof x;

</script>

</body>

</html>

复杂数据

typeof 运算符可以返回两种复杂类型之一：

function

object

typeof 运算符会为对象、数组和 null 返回 "object"。

typeof 运算符不会为函数返回 "object"。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript typeof</h1>

<p>typeof 运算符为对象、数组和 null 返回 object。</p>

<p>typeof 运算符不对函数返回 object。</p>

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

<script>

document.getElementById("demo").innerHTML =

typeof {name:'john', age:34} + "<br>" +

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

typeof null + "<br>" +

typeof function myFunc(){};

</script>

</body>

</html>

typeof 运算符会为数组返回 "object"，因为在 JavaScript 中数组是对象。

typeof 的数据类型

typeof 运算符并不是变量。它只是一个运算符。运算符 (+ - \* /) 没有任何数据类型。

但是，typeof 运算符总是返回字符串（包含操作数的类型）。

constructor 属性

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

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript constructor 属性</h1>

<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>" +

(3.14).constructor + "<br>" +

false.constructor + "<br>" +

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

{name:'john', age:34}.constructor + "<br>" +

new Date().constructor + "<br>" +

function () {}.constructor;

</script>

</body>

</html>

您可以检查 constructor 属性以确定对象是否为数组（包含 "Array" 一词）：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>这个“自制的” isArray () 函数在数组上使用时返回 true：</p>

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

<script>

const fruits = ["Banana", "Orange", "Apple"];

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

function isArray(myArray) {

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

}

</script>

</body>

</html>

或者更简单，您可以检查对象是否为数组函数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array 对象</h1>

<p>这个“自制的” isArray () 函数在数组上使用时返回 true：</p>

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

<script>

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

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

function isArray(myArray) {

return myArray.constructor === Array;

}

</script>

</body>

</html>

您可以检查 constructor 属性以确定对象是否为日期（包含 "Date" 一词）：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Date 对象</h1>

<p>这个“自制的” isDate () 函数在日期上使用时返回 true：</p>

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

<script>

const myDate = new Date();

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

function isDate(myDate) {

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

}

</script>

</body>

</html>

或者更简单，您可以检查对象是否为日期函数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Date 对象</h1>

<p>这个“自制的” isDate () 函数在日期上使用时返回 true：</p>

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

<script>

const myDate = new Date();

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

function isDate(myDate) {

return myDate.constructor === Date;

}

</script>

</body>

</html>

Undefined

在 JavaScript 中，没有值的变量的值是 undefined。类型也是 undefined。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript</h1>

<p>没有值的变量的值（和数据类型）是 <b>undefined</b>。</p>

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

<script>

let car;

document.getElementById("demo").innerHTML =

car + "<br>" + typeof car;

</script>

</body>

</html>

通过将其值设置为 undefined，可以清空任何变量。类型也将是 undefined。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript</h1>

<p>如果将值设置为 <b>undefined</b>，则可以清空变量。</p>

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

<script>

let car = "Volvo";

car = undefined;

document.getElementById("demo").innerHTML = car + "<br>" + typeof car;

</script>

</body>

</html>

空值

空值与 undefined 无关。

空字符串既有合法值又有类型。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript</h1>

<p>空字符串既有合法值又有类型：</p>

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

<script>

let car = "";

document.getElementById("demo").innerHTML =

"The value is: " +

car + "<br>" +

"The type is: " + typeof car;

</script>

</body>

</html>

Null

在 JavaScript 中 null 即是“无”。它应该是不存在的东西。

不幸的是，在 JavaScript 中，null 的数据类型是一个对象。

你可以认为它是 JavaScript 中的一个 bug，typeof null 是一个对象。类型应为 null。

您可以通过将对象设置为 null 来清空对象：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript</h1>

<p>可以通过将值设置为 <b>null</b> 来清空对象。</p>

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

<script>

let person = {firstName:"Bill", lastName:"Gates", age:19, eyeColor:"blue"};

person = null;

document.getElementById("demo").innerHTML = typeof person;

</script>

</body>

</html>

您还可以通过将对象设置为 undefined 来清空对象：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript</h1>

<p>可以通过将值设置为 <b>undefined</b> 来清空对象。</p>

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

<script>

let person = {firstName:"Bill", lastName:"Gates", age:19, eyeColor:"blue"};

person = undefined;

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

</script>

</body>

</html>

undefined 与 null 的区别

undefined 和 null 值相等但类型不同：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript</h1>

<p>undefined 和 null 值相等但类型不同：</p>

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

<script>

document.getElementById("demo").innerHTML =

typeof undefined + "<br>" +

typeof null + "<br><br>" +

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

(null == undefined);

</script>

</body>

</html>

# 第45章 JavaScript 类型转换

Number() 转换数值，String() 转换字符串，Boolean() 转换布尔值。

JavaScript 数据类型

JavaScript 中有五种可包含值的数据类型：

字符串（string）

数字（number）

布尔（boolean）

对象（object）

函数（function）

有三种对象类型：

对象（Object）

日期（Date）

数组（Array）

同时有两种不能包含值的数据类型：

null

undefined

typeof 运算符

您可以使用 typeof 运算符来确定 JavaScript 变量的数据类型。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript typeof 运算符</h2>

<p>typeof 运算符返回变量、对象、函数或表达式的类型。</p>

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

<script>

document.getElementById("demo").innerHTML =

typeof "john" + "<br>" +

typeof 3.14 + "<br>" +

typeof NaN + "<br>" +

typeof false + "<br>" +

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

typeof {name:'john', age:34} + "<br>" +

typeof new Date() + "<br>" +

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

typeof myCar + "<br>" +

typeof null;

</script>

</body>

</html>

请注意：

NaN 的数据类型是数值

数组的数据类型是对象

日期的数据类型是对象

null 的数据类型是对象

未定义变量的数据类型是 undefined

尚未赋值的变量的数据类型也是 undefined

您无法使用 typeof 去判断 JavaScript 对象是否是数组（或日期）。

typeof 的数据类型

typeof 运算符不是变量。它属于运算符。运算符（比如 + - \* /）没有数据类型。

但是，typeof 始终会返回字符串（包含运算数的类型）。

constructor 属性

constructor 属性返回所有 JavaScript 变量的构造器函数。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript constructor 属性</h2>

<p>constructor 属性返回变量或对象的构造器函数。</p>

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

<script>

document.getElementById("demo").innerHTML =

"john".constructor + "<br>" +

(3.14).constructor + "<br>" +

false.constructor + "<br>" +

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

{name:'john', age:34}.constructor + "<br>" +

new Date().constructor + "<br>" +

function () {}.constructor;

</script>

</body>

</html>

您可以通过检查 constructor 属性来确定某个对象是否为数组（包含单词 "Array"）：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数组</h2>

<p>这个“自制的” isArray() 函数在数组上使用时返回 true：</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;

}

</script>

</body>

</html>

或者更简单，您可以检查对象是否是数组函数：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数组对象</h2>

<p>这个“自制的” isArray() 函数在数组上使用时返回 true：</p>

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

<script>

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

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

function isArray(myArray) {

return myArray.constructor === Array;

}

</script>

</body>

</html>

您可以通过检查 constructor 属性来确定某个对象是否为日期（包含单词 "Date"）：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 日期对象</h2>

<p>这个“自制的” isDate() 函数在日期上使用时返回 true：</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;

}

</script>

</body>

</html>

或者更简单，您可以检查对象是否是日期函数：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 日期对象</h2>

<p>这个“自制的” isDate() 函数在日期上使用时返回 true：</p>

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

<script>

var myDate = new Date();

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

function isDate(myDate) {

return myDate.constructor === Date;

}

</script>

</body>

</html>

JavaScript 类型转换

JavaScript 变量能够被转换为新变量以及另一种数据类型：

通过使用 JavaScript 函数

通过 JavaScript 本身自动转换

把数值转换为字符串

全局方法 String() 能够把数字转换为字符串。

它可用于任意类型的数字、文字、变量或表达式：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript String() 方法</h2>

<p>String() 方法把数字转换为字符串。</p>

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

<script>

var x = 123;

document.getElementById("demo").innerHTML =

String(x) + "<br>" +

String(123) + "<br>" +

String(100 + 23);

</script>

</body>

</html>

数字方法 toString() 同理。

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 数字方法</h2>

<p>toString() 方法把数字转换为字符串。</p>

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

<script>

var x = 123;

document.getElementById("demo").innerHTML =

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

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

(100 + 23).toString();

</script>

</body>

</html>

|  |  |
| --- | --- |
| 方法 | 描述 |
| toExponential() | 返回字符串，对数字进行舍入，并使用指数计数法来写。 |
| toFixed() | 返回字符串，对数字进行舍入，并使用指定位数的小数来写。 |
| toPrecision() | 返回字符串，把数字写为指定的长度。 |

把布尔转换为字符串

全局方法 String() 能够将布尔转换为字符串。

String(false) // 返回 "false"

String(true) // 返回 "true"

布尔方法 toString() 同理。

false.toString() // 返回 "false"

true.toString() // 返回 "true"

把日期转换为字符串

全局方法 String() 可将日期转换为字符串。

String(Date()) "demo1">// 返回 "Thu Sep 01 2022 18:51:28 GMT+0800 (中国标准时间)"

日期方法 toString() 同理。

实例

Date().toString() "demo2">// 返回 "Thu Sep 01 2022 18:51:28 GMT+0800 (中国标准时间)"

在日期方法这一章，您能够找到更多可用于把日期转换为字符串的方法：

|  |  |
| --- | --- |
| 方法 | 描述 |
| getDate() | 获得以数值计（1-31）的日 |
| getDay() | 或者以数值计（0-6）的周 |
| getFullYear() | 获得四位的年（yyyy） |
| getHours() | 获得时（0-23） |
| getMilliseconds() | 获得毫秒（0-999） |
| getMinutes() | 获得分钟（0-59） |
| getMonth() | 获得月（0-11） |
| getSeconds() | 获得秒（0-59） |
| getTime() | 获得时间（1970 年 1 月 1 日以来的毫秒） |

把字符串转换为数值

全局方法 Number() 可把字符串转换为数字。

包含数字的字符串（比如 "3.14"）转换为数字（比如 3.14）。

空的字符串转换为 0。

其他字符串将转换为 NaN（Not a number，不是数字）。

Number("3.14") // 返回 3.14

Number(" ") // 返回 0

Number("") // 返回 0

Number("99 88") // 返回 NaN

在数字方法这一章中，您将找到更多可用于把字符串转换为数字的方法：

|  |  |
| --- | --- |
| 方法 | 描述 |
| parseFloat() | 解析字符串并返回浮点数。 |
| parseInt() | 解析字符串并返回整数。 |

一元 + 运算符

一元的 + 运算符可用于把变量转换为数字：

实例

var y = "5"; // y 是字符串

var x = + y; // x 是数字

如果无法转换变量，则仍会成为数字，但是值为 NaN（Not a number）：

实例

var y = "Bill"; // y 是字符串

var x = + y; // x 是数字 (NaN)

把布尔转换数值

全局方法 Number() 也可把布尔转换为数字。

Number(false) // 返回 0

Number(true) // 返回 1

把日期转换为数字

全局方法 Number() 可用于把日期转换为数字。

d = new Date();

Number(d) "demo3">// 返回 1662029489030

日期方法 getTime() 同理。

d = new Date();

d.getTime() "demo4">// 返回 1662029489031

自动类型转换

如果 JavaScript 尝试操作一种“错误”的数据类型，它会试图将该值转换为“正确”的类型。

结果并不总是你所期望的：

<!DOCTYPE html>

<html>

<body>

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

<script>

var x = [];

document.getElementById("demo").innerHTML =

(5 + null) + "<br>" +

("5" + null) + "<br>" +

("5" + 2) + "<br>" +

("5" - 2) + "<br>" +

("5" \* "2") + "<br>" +

("5" / "2") + "<br>"

</script>

</body>

</html>

自动字符串转换

JavaScript 自动调用变量的 toString() 函数，当您试图“输出”对象或变量时：

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

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

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

// 如果 myVar = new Date() // toString 转换为 "Thu Sep 01 2022 18:51:29 GMT+0800 (中国标准时间)"

数字和布尔也会被转换，但并不明显：

// 如果 myVar = 123 // toString 转换为 "123"

// 如果 myVar = true // toString 转换为 "true"

// 如果 myVar = false // toString 转换为 "false"

JavaScript 类型转换表

下表中列出了将不同 JavaScript 值转换为数字、字符串和布尔的结果：

|  |  |  |  |
| --- | --- | --- | --- |
| 原始值 | 转换为数字 | 转换为字符串 | 转换为逻辑 |
| 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 |
| "twenty" | NaN | "twenty" | true |
| [ ] | 0 | "" | true |
| [20] | 20 | "20" | true |
| [10,20] | NaN | "10,20" | true |
| ["twenty"] | NaN | "twenty" | true |
| ["ten","twenty"] | NaN | "ten,twenty" | true |
| function(){} | NaN | "function(){}" | true |
| { } | NaN | "[object Object]" | true |
| null | 0 | "null" | false |
| undefined | NaN | "undefined" | false |

引号中的值指示字符串值。

红色的值指示了（某些）程序员也许不希望的值。

# 第46章 JavaScript 位运算符

JavaScript 位运算符

|  |  |  |
| --- | --- | --- |
| 运算符 | 名称 | 描述 |
| & | AND | 如果两位都是 1 则设置每位为 1 |
| | | OR | 如果两位之一为 1 则设置每位为 1 |
| ^ | XOR | 如果两位只有一位为 1 则设置每位为 1 |
| ~ | NOT | 反转所有位 |
| << | 零填充左位移 | 通过从右推入零向左位移，并使最左边的位脱落。 |
| >> | 有符号右位移 | 通过从左推入最左位的拷贝来向右位移，并使最右边的位脱落。 |
| >>> | 零填充右位移 | 通过从左推入零来向右位移，并使最右边的位脱落。 |

实例

|  |  |  |  |
| --- | --- | --- | --- |
| 操作 | 结果 | 等同于 | 结果 |
| 5 & 1 | 1 | 0101 & 0001 | 0001 |
| 5 | 1 | 5 | 0101 | 0001 | 0101 |
| 5 ^ 1 | 4 | 0101 ^ 0001 | 0100 |
| ~ 5 | 10 | ~0101 | 1010 |
| 5 << 1 | 10 | 0101 << 1 | 1010 |
| 5 >> 1 | 2 | 0101 >> 1 | 0010 |
| 5 >>> 1 | 2 | 0101 >>> 1 | 0010 |

JavaScript 使用 32 位按位运算数

JavaScript 将数字存储为 64 位浮点数，但所有按位运算都以 32 位二进制数执行。

在执行位运算之前，JavaScript 将数字转换为 32 位有符号整数。

执行按位操作后，结果将转换回 64 位 JavaScript 数。

上面的例子使用 4 位无符号二进制数。所以 ~ 5 返回 10。

由于 JavaScript 使用 32 位有符号整数，JavaScript 将返回 -6。

00000000000000000000000000000101 (5)

11111111111111111111111111111010 (~5 = -6)

有符号整数使用最左边的位作为减号。

位运算 AND

当对一对数位执行位运算 AND 时，如果数位均为 1 则返回 1。

单位示例：

|  |  |
| --- | --- |
| 运算 | 结果 |
| 0 & 0 | 0 |
| 0 & 1 | 0 |
| 1 & 0 | 0 |
| 1 & 1 | 1 |

四位示例：

|  |  |  |
| --- | --- | --- |
| 运算 | 结果 |  |
| 1111 & 0000 | 0000 |  |
| 1111 & 0001 | 0001 |  |
| 1111 & 0010 | 0010 |  |
| 1111 & 0100 | 0100 |  |

位运算 OR

当对一对数位执行位运算 OR 时，如果其中一位是 1 则返回 1：

单位示例

|  |  |
| --- | --- |
| 运算 | 结果 |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

四位运算：

|  |  |
| --- | --- |
| 操作 | 结果 |
| 1111 | 0000 | 1111 |
| 1111 | 0001 | 1111 |
| 1111 | 0010 | 1111 |
| 1111 | 0100 | 1111 |

位运算 XOR

当对一对数位进行位运算 XOR 时，如果数位是不同的则返回 1：

单位示例：

|  |  |
| --- | --- |
| 运算 | 结果 |
| 0 ^ 0 | 0 |
| 0 ^ 1 | 1 |
| 1 ^ 0 | 1 |
| 1 ^ 1 | 0 |

四位示例：

|  |  |
| --- | --- |
| 运算 | 结果 |
| 1111 ^ 0000 | 1111 |
| 1111 ^ 0001 | 1110 |
| 1111 ^ 0010 | 1101 |
| 1111 ^ 0100 | 1011 |

JavaScript 位运算 AND (&)

如果位数都是 1，则位运算 AND 返回 1：

|  |  |
| --- | --- |
| 十进制 | 二进制 |
| 5 | 00000000000000000000000000000101 |
| 1 | 00000000000000000000000000000001 |
| 5 & 1 | 00000000000000000000000000000001 (1) |

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 位运算 AND</h2>

<p id="demo">我的第一个段落。</p>

<script>

document.getElementById("demo").innerHTML = 5 & 1;

</script>

</body>

</html>

JavaScript 位运算 OR (|)

如果数位之一是 1，则位运算 OR 返回 1：

|  |  |
| --- | --- |
| 十进制 | 二进制 |
| 5 | 00000000000000000000000000000101 |
| 1 | 00000000000000000000000000000001 |
| 5 | 1 | 00000000000000000000000000000101 (5) |

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 位运算 OR</h2>

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

<script>

document.getElementById("demo").innerHTML = 5 | 1;

</script>

</body>

</html>

JavaScript 位运算 XOR (^)

如果数位是不同的，则 XOR 返回 1：

|  |  |
| --- | --- |
| 十进制 | 二进制 |
| 5 | 00000000000000000000000000000101 |
| 1 | 00000000000000000000000000000001 |
| 5 ^ 1 | 00000000000000000000000000000100 (4) |

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 位运算 XOR</h2>

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

<script>

document.getElementById("demo").innerHTML = 5 ^ 1;

</script>

</body>

</html>

JavaScript 位运算 NOT (~)

|  |  |
| --- | --- |
| 十进制 | 二进制 |
| 5 | 00000000000000000000000000000101 |
| ~5 | 11111111111111111111111111111010 (-6) |

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 位运算 NOT</h2>

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

<script>

document.getElementById("demo").innerHTML = ~ 5;

</script>

</body>

</html>

JavaScript（零填充）位运算左移（<<）

这是零填充的左移。一个或多个零数位从右被推入，最左侧的数位被移除：

|  |  |
| --- | --- |
| 十进制 | 二进制 |
| 5 | 00000000000000000000000000000101 |
| 5 << 1 | 00000000000000000000000000001010 (10) |

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 位运算左移</h2>

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

<script>

document.getElementById("demo").innerHTML = 5 << 1;

</script>

</body>

</html>

JavaScript（有符号）位运算右移（>>）

这是保留符号的右移。最左侧的数位被从左侧推入，最右侧的数位被移出：

|  |  |
| --- | --- |
| 十进制 | 二进制 |
| -5 | 11111111111111111111111111111011 |
| -5 >> 1 | 11111111111111111111111111111101 (-3) |

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript（有符号）位运算右移</h2>

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

<script>

document.getElementById("demo").innerHTML = -5 >> 1;

</script>

</body>

</html>

JavaScript（零填充）右移（>>>）

这是零填充的右移。一个或多个零数位从左侧被推入，最右侧的数位被移出：

|  |  |
| --- | --- |
| 十进制 | 二进制 |
| 5 | 00000000000000000000000000000101 |
| 5 >>> 1 | 00000000000000000000000000000010 (2) |

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 位运算右移</h2>

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

<script>

document.getElementById("demo").innerHTML = 5 >>> 1;

</script>

</body>

</html>

32-bit 有符号整数（二进制数）

仅设置一位的 32 位整数很好理解：

|  |  |
| --- | --- |
| 二进制表示 | 十进制值 |
| 00000000000000000000000000000001 | 1 |
| 00000000000000000000000000000010 | 2 |
| 00000000000000000000000000000100 | 4 |
| 00000000000000000000000000001000 | 8 |
| 00000000000000000000000000010000 | 16 |
| 00000000000000000000000000100000 | 32 |
| 00000000000000000000000001000000 | 64 |

多设置一些数位揭示了二进制的模式：

|  |  |
| --- | --- |
| 二进制表示 | 十进制值 |
| 00000000000000000000000000000101 | 5 (4 + 1) |
| 00000000000000000000000000101000 | 40 (32 + 8) |
| 00000000000000000000000000101101 | 45 (32 + 8 + 4 + 1) |

负数是正数的二进制补码加 1：

|  |  |
| --- | --- |
| 二进制表示 | 十进制值 |
| 00000000000000000000000000000101 | 5 |
| 11111111111111111111111111111011 | -5 |
| 00000000000000000000000000101000 | 40 |
| 11111111111111111111111111011000 | -40 |
| 11111111111111111111111111011001 | -41 |

把十进制转换为二进制

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 十进制转换二进制</h2>

<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML = dec2bin(-5);

function dec2bin(dec){

return (dec >>> 0).toString(2);

}

</script>

</body>

</html>

把二进制转换为十进制

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 二进制转换十进制</h2>

<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML = bin2dec(101);

function bin2dec(bin){

return parseInt(bin, 2).toString(10);

}

</script>

</body>

</html>

# 第47章 JavaScript 正则表达式

正则表达式是构成搜索模式的字符序列。

该搜索模式可用于文本搜索和文本替换操作。

什么是正则表达式？

正则表达式是构成搜索模式（search pattern）的字符序列。

当您搜索文本中的数据时，您可使用搜索模式来描述您搜索的内容。

正则表达式可以是单字符，或者更复杂的模式。

正则表达式可用于执行所有类型的文本搜索和文本替换操作。

语法

/pattern/modifiers;

实例

var patt = /w3school/i;

例子解释：

/w3school/i 是一个正则表达式。

w3school 是模式（pattern）（在搜索中使用）。

i 是修饰符（把搜索修改为大小写不敏感）。

使用字符串方法

在 JavaScript 中，正则表达式常用于两个字符串方法：search() 和 replace()。

search() 方法使用表达式来搜索匹配，然后返回匹配的位置。

replace() 方法返回模式被替换处修改后的字符串。

使用字符串方法 search() 来处理字符串

search() 方法也接受字符串作为搜索参数。字符串参数将被转换为正则表达式：

实例

使用字符串来执行对 "W3school" 的搜索：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>在字符串中搜索“W3School”，并显示匹配的位置：</p>

<p id="demo"></p>

<script>

var str = "Visit W3School!";

var n = str.search("W3School");

document.getElementById("demo").innerHTML = n;

</script>

</body>

</html>

在字符串方法 search() 中使用正则表达式

实例

使用正则表达式执行搜索字符串中 "w3school" 的大小写不敏感的搜索：

var str = "Visit W3School";

var n = str.search(/w3school/i);

n 中的结果将是：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 正则表达式</h1>

<p>在字符串中搜索“w3School”，并显示匹配的位置：</p>

<p id="demo"></p>

<script>

var str = "Visit W3School!";

var n = str.search(/w3School/i);

document.getElementById("demo").innerHTML = n;

</script>

</body>

</html>

使用字符串方法 replace() 处理字符串

replace() 也接受字符串作为搜索参数：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>请把下面的段落中的“Microsoft”替换为“W3School”：</p>

<button onclick="myFunction()">试一试</button>

<p id="demo">请访问 Microsoft！</p>

<script>

function myFunction() {

var str = document.getElementById("demo").innerHTML;

var txt = str.replace("Microsoft","W3School");

document.getElementById("demo").innerHTML = txt;

}

</script>

</body>

</html>

在字符串方法 replace() 中使用正则表达式

实例

使用大小写不明的正则表达式以 W3school 来替换字符串中的 Microsoft：

var str = "Visit Microsoft!";

var res = str.replace(/microsoft/i, "W3School");

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 正则表达式</h1>

<p>将“microsoft”替换为以下段落中的“W3School”：</p>

<button onclick="myFunction()">试一试</button>

<p id="demo">Please visit Microsoft and Microsoft!</p>

<script>

function myFunction() {

var str = document.getElementById("demo").innerHTML;

var txt = str.replace(/microsoft/i,"W3School");

document.getElementById("demo").innerHTML = txt;

}

</script>

</body>

</html>

您注意到了吗？

正则表达式参数（而不是字符串参数）可以在上面的方法中使用。

正则表达式可以使您的搜索更强大（例如，不区分大小写）。

正则表达式修饰符

修饰符可用于大小写不敏感的更全局的搜素：

|  |  |
| --- | --- |
| 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

括号用于查找一定范围的字符串：

|  |  |
| --- | --- |
| 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) | 查找由 | 分隔的任何选项。 |

元字符（Metacharacter）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。 |

Quantifiers 定义量词：

|  |  |
| --- | --- |
| 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 对象是带有预定义属性和方法的正则表达式对象。

使用 test()

test() 是一个正则表达式方法。

它通过模式来搜索字符串，然后根据结果返回 true 或 false。

下面的例子搜索字符串中的字符 "e"：

实例

var patt = /e/;

patt.test("The best things in life are free!");

由于字符串中有一个 "e"，以上代码的输出将是：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 正则表达式</h2>

<p>检索下面段落中的一个 "e"：</p>

<p id="p01">The best things in life are free!</p>

<p id="demo"></p>

<script>

text = document.getElementById("p01").innerHTML;

document.getElementById("demo").innerHTML = /e/.test(text);

</script>

</body>

</html>

您不必首先把正则表达式放入变量中。上面的两行可缩短为一行：

/e/.test("The best things in life are free!");

使用 exec()

exec() 方法是一个正则表达式方法。

它通过指定的模式（pattern）搜索字符串，并返回已找到的文本。

如果未找到匹配，则返回 null。

下面的例子搜索字符串中的字符 "e"：

实例

/e/.exec("The best things in life are free!");

由于字符串中有一个 "e"，以上代码的输出将是：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 正则表达式</h2>

<p id="demo"></p>

<script>

var obj = /e/.exec("The best things in life are free!");

document.getElementById("demo").innerHTML =

"Found " + obj[0] + " in position " + obj.index + " in the text: " + obj.input;

</script>

</body>

</html>

# 第48章 JavaScript 错误 - Throw 和 Try to Catch

try 语句使您能够测试代码块中的错误。

catch 语句允许您处理错误。

throw 语句允许您创建自定义错误。

finally 使您能够执行代码，在 try 和 catch 之后，无论结果如何。

错误总会发生！

当执行 JavaScript 代码时，会发生各种错误。

错误可能是程序员的编码错误，由错误输入引起的错误，或者由于其他不可预见的问题。

实例

在本例中，我们通过 adddlert 编写警告代码来故意制造了一个错误：

<!DOCTYPE html>

<html>

<body>

<p id="demo"></p>

<script>

try {

adddlert("欢迎您，亲爱的用户！");

}

catch(err) {

document.getElementById("demo").innerHTML = err.message;

}

</script>

</body>

</html>

JavaScript 将 adddlert 捕获为一个错误，然后执行代码来处理该错误。

JavaScript try 和 catch

try 语句允许您定义一个代码块，以便在执行时检测错误。

catch 语句允许你定义一个要执行的代码块，如果 try 代码块中发生错误。

JavaScript 语句 try 和 catch 成对出现：

try {

供测试的代码块

}

catch(err) {

处理错误的代码块

}

JavaScript 抛出错误

当发生错误时，JavaScript 通常会停止并产生错误消息。

技术术语是这样描述的：JavaScript 将抛出异常（抛出错误）。

JavaScript 实际上会创建带有两个属性的 Error 对象：name 和 message。

throw 语句

throw 语句允许您创建自定义错误。

从技术上讲您能够抛出异常（抛出错误）。

异常可以是 JavaScript 字符串、数字、布尔或对象：

throw "Too big"; // 抛出文本

throw 500; //抛出数字

如果把 throw 与 try 和 catch 一同使用，就可以控制程序流并生成自定义错误消息。

输入验证案例

本例会检查输入。如果值是错误的，将抛出异常（err）。

该异常（err）被 catch 语句捕获并显示一条自定义的错误消息：

<!DOCTYPE html>

<html>

<body>

<p>请输入 5 - 10 之间的数字：</p>

<input id="demo" type="text">

<button type="button" onclick="myFunction()">测试输入</button>

<p id="message"></p>

<script>

function myFunction() {

var message, x;

message = document.getElementById("message");

message.innerHTML = "";

x = document.getElementById("demo").value;

try {

if(x == "") throw "空的";

if(isNaN(x)) throw "不是数字";

x = Number(x);

if(x < 5) throw "太小";

if(x > 10) throw "太大";

}

catch(err) {

message.innerHTML = "输入是 " + err;

}

}

</script>

</body>

</html>

HTML 验证

以上代码仅仅是一个例子。

现代浏览器通常会结合 JavaScript 与内置的 HTML 验证，通过使用在 HTML 属性中定义的预定义的验证规则：

<input id="demo" type="number" min="5" max="10" step="1">

您将在本教程稍后的章节学到更多有关表单验证的知识。

finally 语句

finally 语句允许您在 try 和 catch 之后执行代码，无论结果：

try {

// 供测试的代码块

}

catch(err) {

// 处理错误的代码块

}

finally {

// 无论结果如何都执行的代码块

}

实例

function myFunction() {

var message, x;

message = document.getElementById("message");

message.innerHTML = "";

x = document.getElementById("demo").value;

try {

if(x == "") throw "是空的";

if(isNaN(x)) throw "不是数字";

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 = "";

}

}

<!DOCTYPE html>

<html>

<body>

<p>请输入 5 到 10 之间的数字：</p>

<input id="demo" type="text">

<button type="button" onclick="myFunction()">检测输入</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 "是空的";

if(isNaN(x)) throw "不是数字";

x = Number(x);

if(x > 10) throw "太大";

if(x < 5) throw "太小";

}

catch(err) {

message.innerHTML = "输入：" + err;

}

finally {

document.getElementById("demo").value = "";

}

}

</script>

</body>

</html>

Error 对象

JavaScript 拥有当错误发生时提供错误信息的内置 error 对象。

error 对象提供两个有用的属性：name 和 message。

Error 对象属性

|  |  |
| --- | --- |
| 属性 | 描述 |
| name | 设置或返回错误名 |
| message | 设置或返回错误消息（一条字符串） |

Error Name Values

error 的 name 属性可返回六个不同的值：

|  |  |
| --- | --- |
| 错误名 | 描述 |
| EvalError | 已在 eval() 函数中发生的错误 |
| RangeError | 已发生超出数字范围的错误 |
| ReferenceError | 已发生非法引用 |
| SyntaxError | 已发生语法错误 |
| TypeError | 已发生类型错误 |
| URIError | 在 encodeURI() 中已发生的错误 |

下面为您详解这六个不同的值。

Eval 错误

EvalError 指示 eval() 函数中的错误。

更新版本的 JavaScript 不会抛出任何 EvalError。请使用 SyntaxError 代替。

范围错误

RangeError 会在您使用了合法值的范围之外的数字时抛出。

例如：您不能将数字的有效位数设置为 500。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 异常</h1>

<p>您不能将数字的有效位数设置为500：</p>

<p id="demo">

<script>

var num = 1;

try {

num.toPrecision(500);

}

catch(err) {

document.getElementById("demo").innerHTML = err.name;

}

</script>

</body>

</html>

引用错误

假如您使用（引用）了尚未声明的变量，则 ReferenceError 会被抛出：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 异常</h1>

<p>您不能使用不存在的变量的值：</p>

<p id="demo"></p>

<script>

var x;

try {

x = y + 1;

}

catch(err) {

document.getElementById("demo").innerHTML = err.name;

}

</script>

</body>

</html>

语法错误

假如您计算带语法错误的代码，会 SyntaxError 被抛出：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 异常</h1>

<p>您无法评估包含语法错误的代码：</p>

<p id="demo"></p>

<script>

try {

eval("alert('Hello)");

}

catch(err) {

document.getElementById("demo").innerHTML = err.name;

}

</script>

</body>

</html>

类型错误

假如您使用的值不在期望值的范围之内，则 TypeError 被抛出：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 异常</h1>

<p>您无法将数字转换为大写：</p>

<p id="demo"></p>

<script>

var num = 1;

try {

num.toUpperCase();

}

catch(err) {

document.getElementById("demo").innerHTML = err.name;

}

</script>

</body>

</html>

URI 错误

假如您在 URI 函数中使用非法字符，则 URIError 被抛出：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 异常</h1>

<p>使用 decodeURI() 无法解码某些字符：</p>

<p id="demo"></p>

<script>

try {

decodeURI("%%%");

}

catch(err) {

document.getElementById("demo").innerHTML = err.name;

}

</script>

</body>

</html>

非标准的 Error 对象属性

Mozilla 和 Microsoft 定义了非标准的 error 对象属性：

fileName (Mozilla)

lineNumber (Mozilla)

columnNumber (Mozilla)

stack (Mozilla)

description (Microsoft)

number (Microsoft)

请勿在公共网站使用这些属性。它们并不会在所有浏览器中工作。

# 第49章 JavaScript 作用域

作用域指的是您有权访问的变量集合。

JavaScript 函数作用域

在 JavaScript 中有两种作用域类型：

局部作用域

全局作用域

JavaScript 拥有函数作用域：每个函数创建一个新的作用域。

作用域决定了这些变量的可访问性（可见性）。

函数内部定义的变量从函数外部是不可访问的（不可见的）。

局部 JavaScript 变量

在 JavaScript 函数中声明的变量，会成为函数的局部变量。

局部变量的作用域是局部的：只能在函数内部访问它们。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 局部作用域</h1>

<p>myFunction() 之外的 carName 未定义。</p>

<p id="demo1"></p>

<p id="demo2"></p>

<script>

myFunction();

function myFunction() {

var carName = "porsche";

document.getElementById("demo1").innerHTML =

typeof carName + " " + carName;

}

document.getElementById("demo2").innerHTML =

typeof carName;

</script>

</body>

</html>

由于只能在函数内部识别局部变量，因此能够在不同函数中使用同名变量。

在函数开始时会创建局部变量，在函数完成时会删除它们。

全局 JavaScript 变量

函数之外声明的变量，会成为全局变量。

全局变量的作用域是全局的：网页的所有脚本和函数都能够访问它。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 全局作用域</h1>

<p>可以从任何脚本或函数中访问全局变量。</p>

<p id="demo"></p>

<script>

var carName = "porsche";

myFunction();

function myFunction() {

document.getElementById("demo").innerHTML =

"I can display " + carName;

}

</script>

</body>

</html>

JavaScript 变量

在 JavaScript 中，对象和函数也是变量。

作用域决定了从代码不同部分对变量、对象和函数的可访问性。

自动全局

如果您为尚未声明的变量赋值，此变量会自动成为全局变量。

这段代码将声明一个全局变量 carName，即使在函数内进行了赋值。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 自动全局</h1>

<p>如果您向未声明的变量赋值，它会自动成为全局变量：</p>

<p id="demo"></p>

<script>

myFunction();

// 此处的代码可以把 carName 作为全局变量使用。

document.getElementById("demo").innerHTML = "我可以显示 " + carName;

function myFunction() {

carName = "porsche";

}

</script>

</body>

</html>

严格模式

所有现代浏览器都支持以“严格模式”运行 JavaScript。

您将在本教程稍后的章节学习更多如何使用严格模式的知识。

在“严格模式”中不会自动创建全局变量。

HTML 中的全局变量

通过 JavaScript，全局作用域形成了完整的 JavaScript 环境。

在 HTML 中，全局作用域是 window。所有全局变量均属于 window 对象。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 作用域</h1>

<p>在 HTML 中，通过 <b>var</b> 定义的全局变量，会成为 window 变量。</p>

<p id="demo"></p>

<script>

var carName = "porsche";

// code here can use window.carName

document.getElementById("demo").innerHTML = "我可以显示 " + window.carName;

</script>

</body>

</html>

警告

除非有意为之，否则请勿创建全局变量。

您的全局变量（或函数）能够覆盖 window 变量（或函数）。

任何函数，包括 window 对象，能够覆盖您的全局变量和函数。

JavaScript 变量的有效期

JavaScript 变量的有效期始于其被创建时。

局部变量会在函数完成时被删除。

全局变量会在您关闭页面是被删除。

函数参数

函数参数也是函数内的局部变量。

# 第50章 JavaScript Hoisting

提升（Hoisting）是 JavaScript 将声明移至顶部的默认行为。

JavaScript 声明会被提升

在 JavaScript 中，可以在使用变量之后对其进行声明。

换句话说，可以在声明变量之前使用它。

例子 1 与例子 2 的结果相同：

例子 1

<!DOCTYPE html>

<html>

<body>

<p id="demo"></p>

<script>

x = 5; // 把 5 赋值给 x

elem = document.getElementById("demo"); // 查找元素

elem.innerHTML = x; // 在这个元素中显示 x

var x; // Declare x

</script>

</body>

</html>

例子 2

<!DOCTYPE html>

<html>

<body>

<p id="demo"></p>

<script>

var x; // 声明 x

x = 5; // 把 5 赋值给 x

elem = document.getElementById("demo"); // 查找元素

elem.innerHTML = x; // 在这个元素中显示 x

</script>

</body>

</html>

为了理解这一点，您必须理解术语 "hoisting"。

Hoisting 是 JavaScript 将所有声明提升到当前作用域顶部的默认行为（提升到当前脚本或当前函数的顶部）。

let 和 const 关键字

用 let 或 const 声明的变量和常量不会被提升！

请在 JS Let / Const 中阅读更多关于 let 和 const 的内容。

JavaScript 初始化不会被提升

JavaScript 只提升声明，而非初始化。

例子 1 与例子 2 的结果不相同：

例子 1

<!DOCTYPE html>

<html>

<body>

<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>

</body>

</html>

例子 2

<!DOCTYPE html>

<html>

<body>

<p id="demo"></p>

<script>

var x = 5; // 初始化 x

elem = document.getElementById("demo"); // 查找元素

elem.innerHTML = "x is " + x + " and y is " + y; // 显示 x 和 y

var y = 7; // Initialize y

</script>

</body>

</html>

最后一个例子中的 y 仍然是未定义能说得通吗？

这是因为只有声明（var y）而不是初始化（=7）被提升到顶部。

由于 hoisting，y 在其被使用前已经被声明，但是由于未对初始化进行提升，y 的值仍是未定义。

例子 2 也一样：

实例

<!DOCTYPE html>

<html>

<body>

<p id="demo"></p>

<script>

var x = 5; // 初始化 x

var y; // 声明 y

elem = document.getElementById("demo"); // 查找元素

elem.innerHTML = x + " " + y; // 显示 x 和 y

y = 7; // Assign 7 to y

</script>

</body>

</html>

在顶部声明您的变量！

Hoisting（对很多开发者来说）是 JavaScript 的一种未知的或被忽视的行为。

如果开发者不理解 hoisting，程序也许会包含 bug（错误）。

为了避免 bug，请始终在每个作用域的开头声明所有变量。

由于这就是 JavaScript 解释代码的方式，请保持这个好习惯。

严格模式中的 JavaScript 不允许在未被声明的情况下使用变量。

请在下一章学习“use strict”。

# 第51章 JavaScript 严格模式

"use strict"; 定义 JavaScript 代码应该以“严格模式”执行。

"use strict" 指令

"use strict" 是 JavaScript 1.8.5 中的新指令（ECMAScript version 5）。

它不算一条语句，而是一段文字表达式，更早版本的 JavaScript 会忽略它。

"use strict"; 的作用是指示 JavaScript 代码应该以“严格模式”执行。

在严格模式中，您无法，例如，使用未声明的变量。

以下版本的浏览器支持严格模式：

版本 10 以后的 IE

版本 4 以后的 Firefox

版本 13 以后的 Chrome

版本 5.1 以后的 Safari

版本 12 以后的 Opera

声明严格模式

通过在脚本或函数的开头添加 "use strict"; 来声明严格模式。

在脚本开头进行声明，拥有全局作用域（脚本中的所有代码均以严格模式来执行）：

实例

<!DOCTYPE html>

<html>

<body>

<h1>使用 "use strict":</h1>

<h3>不允许使用变量而不声明它。</h3>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

x = 3.14; // 会引发错误（x 未定义）

</script>

</body>

</html>

实例

<!DOCTYPE html>

<html>

<body>

<h1>全局 "use strict" 声明</h1>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

myFunction();

function myFunction() {

y = 3.14; // 会引发错误（y 未定义）

}

</script>

</body>

</html>

在函数中声明严格模式，拥有局部作用域（只有函数中的代码以严格模式执行）：

<!DOCTYPE html>

<html>

<body>

<p>函数中的“use strict”只会导致该函数出错。</p>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

x = 3.14; // 不会引发错误

myFunction();

function myFunction() {

"use strict";

y = 3.14; // 会引发错误（y 未定义）

}

</script>

</body>

</html>

"use strict"; 语法

声明严格模式的语法被设计为兼容更早版本的 JavaScript。

编译 JavaScript 程序中的数值文字（比如 4+5）或字符串文字（"Bill Gates"）不会有负面效果。它只会编译为不存在的变量然后消失。

所有 "use strict"; 只会对“理解”其含义的新编译器产生影响。

为什么使用严格模式？

严格模式使我们更容易编写“安全的” JavaScript。

严格模式把之前可接受的“坏语法”转变为真实的错误。

举例来说，在普通的 JavaScript 中，错打变量名会创建新的全局变量。在严格模式中，此举将抛出错误，这样就不可能意外创建全局变量。

在普通 JavaScript 中，如果向不可写属性赋值，开发者不会得到任何错误反馈。

在严格模式中，向不可写的、只能读取的、不存在的属性赋值，或者向不存在的变量或对象赋值，将抛出错误。

严格模式中不允许的事项

在不声明变量的情况下使用变量，是不允许的：

<!DOCTYPE html>

<html>

<body>

<h1>使用 "use strict":</h1>

<h3>不允许使用变量而不声明它。</h3>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

x = 3.14; // 会引发错误（x 未定义）

</script>

</body>

</html>

对象也是变量

在不声明对象的情况下使用对象也是不允许的：

<!DOCTYPE html>

<html>

<body>

<h1>使用 "use strict"：</h1>

<h3>不允许使用对象而不声明它。</h3>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

x = {p1:10, p2:20}; // 会引发错误（x 未定义）

</script>

</body>

</html>

删除变量（或对象）是不允许的：

<!DOCTYPE html>

<html>

<body>

<h1>使用 "use strict"：</h1>

<h3>不允许删除变量（或对象）。</h3>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

var x = 3.14;

delete x; // 会引发错误

</script>

</body>

</html>

删除函数是不允许的：

<!DOCTYPE html>

<html>

<body>

<h1>使用 "use strict"：</h1>

<h3>不允许删除函数。</h3>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

function x(p1, p2) {};

delete x; // 会引发错误

</script>

</body>

</html>

重复参数名是不允许的：

<!DOCTYPE html>

<html>

<body>

<h1>使用 "use strict"：</h1>

<h3>不允许复制参数名称。</h3>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

function x(p1, p1) {}; // 会引发错误

</script>

</body>

</html>

八进制数值文本是不允许的：

<!DOCTYPE html>

<html>

<body>

<h1>使用 "use strict"：</h1>

<h3>不允许使用八进制数字字面量。</h3>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

var x = 010; // 会引发错误

</script>

</body>

</html>

转义字符是不允许的：

<!DOCTYPE html>

<html>

<body>

<h1>使用 "use strict"：</h1>

<h3>不允许使用八进制转义字符。</h3>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

var x = "\010"; // 会引发错误

</script>

</body>

</html>

写入只读属性是不允许的：

<!DOCTYPE html>

<html>

<body>

<h1>使用 "use strict"：</h1>

<h2>不允许写入只读属性。</h2>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

var obj = {};

Object.defineProperty(obj, "x", {value:0, writable:false});

obj.x = 3.14; // 会引发错误

</script>

</body>

</html>

写入只能获取的属性是不允许的：

<!DOCTYPE html>

<html>

<body>

<h1>使用 "use strict"：</h1>

<h2>不允许写入 get-only 属性。</h2>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

var obj = {get x() {return 0} };

obj.x = 3.14; // 会引发错误

</script>

</body>

</html>

删除不可删除的属性是不允许的：

<!DOCTYPE html>

<html>

<body>

<h1>使用 "use strict"：</h1>

<h3>不允许删除 undeletable 属性。</h3>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

delete Object.prototype; // 会引发错误

</script>

</body>

</html>

字符串 "eval" 不可用作变量：

<!DOCTYPE html>

<html>

<body>

<h1>使用 "use strict"：</h1>

<h3>字符串“eval”不能用作变量。</h3>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

var eval = 3.14; // 会引发错误

</script>

</body>

</html>

字符串 "arguments" 不可用作变量：

<!DOCTYPE html>

<html>

<body>

<h1>使用 "use strict"：</h1>

<h3>字符串“arguments”不能用作变量。</h3>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

var arguments = 3.14; // 会引发错误

</script>

</body>

</html>

with 语句是不允许的：

<!DOCTYPE html>

<html>

<body>

<h1>使用 "use strict"：</h1>

<h3>不允许使用 with 语句。</h3>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

with (Math){x = cos(2)}; // 会引发错误

</script>

</body>

</html>

处于安全考虑，不允许 eval() 在其被调用的作用域中创建变量：

<!DOCTYPE html>

<html>

<body>

<h1>使用 "use strict"：</h1>

<h3>出于安全原因，不允许 eval() 在调用它的作用域内创建变量。</h3>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

eval ("var x = 2");

alert (x); // This will cause an error

</script>

</body>

</html>

在类似 f() 的函数调用中，this 的值是全局对象。在严格模式中，现在它成为了 undefined。

对未来的保障

严格模式中不允许使用为未来预留的关键词。它们是：

implements

interface

let

package

private

protected

public

static

Yield

<!DOCTYPE html>

<html>

<body>

<h1>使用 "use strict"：</h1>

<h3>严格模式中不允许使用为未来预留的关键词：</h3>

<p>在浏览器中激活调试（F12）以查看错误报告。</p>

<script>

"use strict";

var public = 1500; // 会引发错误

</script>

</body>

</html>

警告

"use strict" 指令只能在脚本或函数的开头被识别。

# 第52章 JavaScript this 关键词

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript <b>this</b> 关键词</h1>

<p>在本例中，<b>this</b> 代表 <b>person</b> 对象。</p>

<p>因为 person 对象“拥有” fullName 方法。</p>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

id : 678,

fullName : function() {

return this.firstName + " " + this.lastName;

}

};

// 显示来自对象的数据：

document.getElementById("demo").innerHTML = person.fullName();

</script>

</body>

</html>

this 是什么？

JavaScript this 关键词指的是它所属的对象。

它拥有不同的值，具体取决于它的使用位置：

在方法中，this 指的是所有者对象。

单独的情况下，this 指的是全局对象。

在函数中，this 指的是全局对象。

在函数中，严格模式下，this 是 undefined。

在事件中，this 指的是接收事件的元素。

像 call() 和 apply() 这样的方法可以将 this 引用到任何对象。

方法中的 this

在对象方法中，this 指的是此方法的“拥有者”。

在本页最上面的例子中，this 指的是 person 对象。

person 对象是 fullName 方法的拥有者。

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript <b>this</b> 关键词</h1>

<p>在本例中，<b>this</b> 代表 <b>person</b> 对象。</p>

<p>因为 person 对象“拥有” fullName 方法。</p>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

id : 678,

fullName : function() {

return this.firstName + " " + this.lastName;

}

};

// 显示来自对象的数据：

document.getElementById("demo").innerHTML = person.fullName();

</script>

</body>

</html>

单独的 this

在单独使用时，拥有者是全局对象，因此 this 指的是全局对象。

在浏览器窗口中，全局对象是 [object Window]：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript <b>this</b> 关键词</h1>

<p>在本例中，<b>this</b> 引用 window 对象：</p>

<p id="demo"></p>

<script>

var x = this;

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

在严格模式中，如果单独使用，那么 this 指的是全局对象 [object Window]：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript <b>this</b> 关键词</h1>

<p>在本例中，<b>this</b> 引用 window 对象：</p>

<p id="demo"></p>

<script>

"use strict";

var x = this;

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

函数中的 this（默认）

在 JavaScript 函数中，函数的拥有者默认绑定 this。

因此，在函数中，this 指的是全局对象 [object Window]。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript <b>this</b> 关键词</h1>

<p>在本例中，<b>this</b> 代表“拥有” myFunction 的对象：</p>

<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML = myFunction();

function myFunction() {

return this;

}

</script>

</body>

</html>

函数中的 this（严格模式）

JavaScript 严格模式不允许默认绑定。

因此，在函数中使用时，在严格模式下，this 是未定义的（undefined）。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript <b>this</b> 关键词</h1>

<p>在函数中，默认地，<b>this</b> 引用全局对象。</p>

<p>在严格模式中，<b>this</b> 是 <b>undefined</b>，因为严格模式不允许默认绑定：</p>

<p id="demo"></p>

<script>

"use strict";

document.getElementById("demo").innerHTML = myFunction();

function myFunction() {

return this;

}

</script>

</body>

</html>

事件处理程序中的 this

在 HTML 事件处理程序中，this 指的是接收此事件的 HTML 元素：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript <b>this</b> 关键词</h1>

<button onclick="this.style.display='none'">单击来删除我！</button>

</body>

</html>

对象方法绑定

在此例中，this 是 person 对象（person 对象是该函数的“拥有者”）：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript <b>this</b> 关键词</h1>

<p>在本例中，<b>this</b> 代表“拥有” fullName 方法的 person 对象。</p>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName : "Bill",

lastName : "Gates",

id : 678,

myFunction : function() {

return this;

}

};

// 显示来自对象的数据：

document.getElementById("demo").innerHTML = person.myFunction();

</script>

</body>

</html>

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript <b>this</b> 关键词</h1>

<p>在本例中，<b>this</b> 代表 <b>person</b> 对象。</p>

<p>因为 person 对象“拥有” fullName 方法。</p>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

id : 678,

fullName : function() {

return this.firstName + " " + this.lastName;

}

};

// 显示来自对象的数据：

document.getElementById("demo").innerHTML = person.fullName();

</script>

</body>

</html>

换句话说，this.firstName 意味着 this（person）对象的 firstName 属性。

显式函数绑定

call() 和 apply() 方法是预定义的 JavaScript 方法。

它们都可以用于将另一个对象作为参数调用对象方法。

您可以在本教程后面阅读有关 call() 和 apply() 的更多内容。

在下面的例子中，当使用 person2 作为参数调用 person1.fullName 时，this 将引用 person2，即使它是 person1 的方法：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript <b>this</b> 关键词</h1>

<p>在本例中，<strong>this</strong> 引用 person2，即使它是 person1 的方法：</p>

<p id="demo"></p>

<script>

var person1 = {

fullName: function() {

return this.firstName + " " + this.lastName;

}

}

var person2 = {

firstName:"Bill",

lastName: "Gates",

}

var x = person1.fullName.call(person2);

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

# 第53章 JavaScript 箭头函数

ES6 中引入了箭头函数。

箭头函数允许我们编写更短的函数

语法

：

之前：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 函数</h2>

<p>这个例子展示了函数的语法，没有使用箭头函数的语法。</p>

<p id="demo"></p>

<script>

var hello;

hello = function() {

return "Hello World!";

}

document.getElementById("demo").innerHTML = hello();

</script>

</body>

</html>

用了箭头函数之后：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 箭头函数</h2>

<p>此例展示了箭头函数的语法以及如何使用它。</p>

<p id="demo"></p>

<script>

var hello;

hello = () => {

return "Hello World!";

}

document.getElementById("demo").innerHTML = hello();

</script>

</body>

</html>

确实变短了！如果函数只有一个语句，并且该语句返回一个值，则可以去掉括号和 return 关键字：

箭头函数默认返回值：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 箭头函数</h2>

<p>此例展示了没有括号或 return 关键字的箭头函数。</p>

<p id="demo"></p>

<script>

var hello;

hello = () => "Hello World!";

document.getElementById("demo").innerHTML = hello();

</script>

</body>

</html>

注释：这仅在函数只有一条语句时才有效。

如果您有参数，则将它们传递到括号内：

带参数的箭头函数：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 箭头函数</h2>

<p>此例展示了带参数的箭头函数。</p>

<p id="demo"></p>

<script>

var hello;

hello = (val) => "Hello " + val;

document.getElementById("demo").innerHTML = hello("Universe!");

</script>

</body>

</html>

事实上，如果只有一个参数，您也可以略过括号：

不带括号的箭头函数：

hello = val => "Hello " + val;

this 怎么办？

与常规函数相比，箭头函数对 this 的处理也有所不同。

简而言之，使用箭头函数没有对 this 的绑定。

在常规函数中，关键字 this 表示调用该函数的对象，可以是窗口、文档、按钮或其他任何东西。

对于箭头函数，this 关键字始终表示定义箭头函数的对象。

让我们看两个例子来理解其中的差异。

这两个例子都调用了两次方法，第一次是在页面加载时，第二次是在用户单击按钮时。

第一个例子使用常规函数，第二个例子使用箭头函数。

结果显示第一个例子返回两个不同的对象（window 和 button），第二个例子返回两次 window 对象，因为 window 对象是函数的“所有者”。

实例

对于常规函数，this 表示调用该函数的对象：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript "this"</h2>

<p>此例演示在常规函数中，“this”关键字代表不同的对象，具体取决于函数的调用方式。</p>

<p>点击按钮再次执行“hello”函数，你会看到这次“this”代表的是 button 对象。</p>

<button id="btn">点击我！</button>

<p id="demo"></p>

<script>

var hello;

hello = function() {

document.getElementById("demo").innerHTML += this;

}

//window 对象调用函数：

window.addEventListener("load", hello);

//button 对象调用函数：

document.getElementById("btn").addEventListener("click", hello);

</script>

</body>

</html>

实例

用了箭头函数，则 this 表示函数的拥有者：

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript "this"</h2>

<p>此例演示在箭头函数中，“this”关键字表示拥有该函数的对象，无论是谁调用该函数。</p>

<p>点击按钮再次执行“hello”函数，你会看到“this”仍然代表 window 对象。</p>

<button id="btn">点击我！</button>

<p id="demo"></p>

<script>

var hello;

hello = () => {

document.getElementById("demo").innerHTML += this;

}

//window 对象调用函数：

window.addEventListener("load", hello);

//button 对象调用函数：

document.getElementById("btn").addEventListener("click", hello);

</script>

</body>

</html>

使用函数时请记住这些差异。有时，常规函数的行为正是您想要的，如果不是，请使用箭头函数。

# 第54章 JavaScript 类

ECMAScript 2015，也称 ES6，引入了 JavaScript 类。

JavaScript 类是 JavaScript 对象的模板。

JavaScript 类的

语法

请使用关键字 class 创建类。

请始终添加名为 constructor() 的方法：

语法

class ClassName {

constructor() { ... }

}

实例

class Car {

constructor(name, year) {

this.name = name;

this.year = year;

}

}

上面的例子创建了一个名为 "Car" 的类。

该类有两个初始属性："name" 和 "year"。

JavaScript 类不是对象。

它只是 JavaScript 对象的模板。

使用类

如果您有一个类，那么您可以使用该类来创建对象：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 类</h1>

<p>如何使用 JavaScript 类。</p>

<p id="demo"></p>

<script>

class Car {

constructor(name, year) {

this.name = name;

this.year = year;

}

}

const myCar = new Car("Ford", 2014);

document.getElementById("demo").innerHTML =

myCar.name + " " + myCar.year;

</script>

</body>

</html>

上面的例子使用 Car 类创建了两个 Car 对象。

在创建新对象时会自动调用 constructor 方法。

Constructor 方法

构造方法是一种特殊的方法：

它必须拥有确切名称的“构造函数”

创建新对象时自动执行

用于初始化对象属性

如果未定义构造函数方法，JavaScript 会添加空的构造函数方法。

Class 方法

类方法的创建

语法

与对象方法相同。

请使用关键字 class 创建类。

请始终添加 constructor() 方法。

然后添加任意数量的方法。

语法

class ClassName {

constructor() { ... }

method\_1() { ... }

method\_2() { ... }

method\_3() { ... }

}

创建名为 "age" 的类方法，它返回车龄：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 类方法</h1>

<p>如何定义和使用类方法。</p>

<p id="demo"></p>

<script>

class Car {

constructor(name, year) {

this.name = name;

this.year = year;

}

age() {

let date = new Date();

return date.getFullYear() - this.year;

}

}

let myCar = new Car("Ford", 2014);

document.getElementById("demo").innerHTML =

"My car is " + myCar.age() + " years old.";

</script>

</body>

</html>

您可以向类方法发送参数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 类方法</h1>

<p>将参数传递给 "age()" 方法。</p>

<p id="demo"></p>

<script>

class Car {

constructor(name, year) {

this.name = name;

this.year = year;

}

age(x) {

return x - this.year;

}

}

let date = new Date();

let year = date.getFullYear();

let myCar = new Car("Ford", 2014);

document.getElementById("demo").innerHTML=

"My car is " + myCar.age(year) + " years old.";

</script>

</body>

</html>

# 第55章 JavaScript JSON

SON 是存储和传输数据的格式。

JSON 经常在数据从服务器发送到网页时使用。

什么是 JSON？

JSON 指的是 JavaScript Object Notation

JSON 是轻量级的数据交换格式

JSON 独立于语言 \*

JSON 是“自描述的”且易于理解

\* JSON 的语法是来自 JavaScript 对象符号的语法，但 JSON 格式是纯文本。读取和生成 JSON 数据的代码可以在任何编程语言编写的。

JSON 实例

JSON 语法定义了一个雇员对象：包含三条员工记录的数组（对象）：

JSON 实例

{

"employees":[

{"firstName":"Bill", "lastName":"Gates"},

{"firstName":"Steve", "lastName":"Jobs"},

{"firstName":"Alan", "lastName":"Turing"}

]

}

JSON 格式评估为 JavaScript 对象

JSON 格式在语法上与创建 JavaScript 对象的代码相同。

由于这种相似性，JavaScript 程序可以很容易地将 JSON 数据转换成本地的 JavaScript 对象。

JSON 语法规则

数据是名称/值对

数据由逗号分隔

花括号保存对象

方括号保存数组

JSON 数据 - 名称和值

JSON 数据的书写方式是名称/值对，类似 JavaScript 对象属性。

名称/值对由（双引号中的）字段名构成，其后是冒号，再其后是值：

"firstName":"Bill"

JSON 名称需要双引号。JavaScript 名称不需要。

JSON 对象

JSON 对象是在花括号内书写的。

类似 JavaScript，对象能够包含多个名称/值对：

{"firstName":"Bill", "lastName":"Gates"}

JSON 数组

JSON 数组在方括号中书写。

类似 JavaScript，数组能够包含对象：

"employees":[

{"firstName":"Bill", "lastName":"Gates"},

{"firstName":"Steve", "lastName":"Jobs"},

{"firstName":"Alan", "lastName":"Turing"}

]

在上面的例子中，对象 "employees" 是一个数组。它包含了三个对象。

每个对象代表一个人的一条记录（带有名和姓）。

把 JSON 文本转换为 JavaScript 对象

JSON 的通常用法是从 web 服务器读取数据，然后在网页中显示数据。

为了简单起见，可以使用字符串作为输入演示。

首先，创建包含 JSON 语法的 JavaScript 字符串：

var text = '{ "employees" : [' +

'{ "firstName":"Bill" , "lastName":"Gates" },' +

'{ "firstName":"Steve" , "lastName":"Jobs" },' +

'{ "firstName":"Alan" , "lastName":"Turing" } ]}';

然后，使用 JavaScript 的内建函数 JSON.parse() 来把这个字符串转换为 JavaScript 对象：

var obj = JSON.parse(text);

最后，请在您的页面中使用这个新的 JavaScript 对象：

实例

<!DOCTYPE html>

<html>

<body>

<h1>用 JSON 字符串创建对象</h1>

<p id="demo"></p>

<script>

var text = '{"employees":[' +

'{"firstName":"Bill","lastName":"Gates" },' +

'{"firstName":"Steve","lastName":"Jobs" },' +

'{"firstName":"Elon","lastName":"Musk" }]}';

obj = JSON.parse(text);

document.getElementById("demo").innerHTML =

obj.employees[1].firstName + " " + obj.employees[1].lastName;

</script>

</body>

</html>

# 第56章 JavaScript 调试

错误总会发生，每当您写一些新的计算机代码时。

JavaScript 调试

在没有调试器的情况下写 JavaScript 是有难度的。

您的代码中也许包含了语法错误，或者逻辑错误，这些都难以诊断。

通常，如果 JavaScript 代码包含错误，也不会发生任何事情。不会有错误消息，并且不会有任何可供查找错误的指示信息。

通常，每当你尝试编写新的 JavaScript 代码，就可能发生错误。

JavaScript 调试器

查找编程代码中的错误被称为代码调试。

调试并不简单。但幸运地是，所有现代浏览器都有内置的调试器。

内置的调试器可打开或关闭，强制将错误报告给用户。

通过调试器，您也可以设置断点（代码执行被中断的位置），并在代码执行时检查变量。

通常通过 F12 键启动浏览器中的调试器，然后在调试器菜单中选择“控制台”。

console.log() 方法

如果您的浏览器支持调试，那么您可以使用 console.log() 在调试窗口中显示 JavaScript 的值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>我的第一张网页</h1>

<p>使用F12在浏览器（Chrome、IE、Firefox）中激活调试，然后在调试器菜单中选择“控制台”。</p>

<script>

a = 5;

b = 6;

c = a + b;

console.log(c);

</script>

</body>

</html>

提示：请访问我们的 JavaScript Console 参考手册，获取更多有关 console.log() 方法的信息。

设置断点

在调试窗口中，您可在 JavaScript 代码中设置断点。

在每个断点中，JavaScript 将停止执行，以使您能够检查 JavaScript 的值。

在检查值之后，您可以恢复代码执行。

debugger 关键词

debugger 关键词会停止 JavaScript 的执行，并调用（如果有）调试函数。

这与在调试器中设置断点的功能是一样的。

如果调试器不可用，debugger 语句没有效果。

如果调试器已打开，此代码会在执行第三行之前停止运行。

实例

<!DOCTYPE html>

<html>

<head>

</head>

<body>

<p id="demo"></p>

<p>打开调试器后，下面的代码应该在执行第三行之前停止执行。</p>

<script>

var x = 15 \* 5;

debugger;

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

主流浏览器的调试工具

通常，您在浏览器中通过 F12 键启用调试，并在调试器菜单中选择“控制台”。

否则请按照以下步骤：

Chrome

打开浏览器

从菜单中选择工具

从工具中选择开发者工具

最后，选择控制台

Firefox Firebug

打开浏览器

前往网页：http://www.getfirebug.com

根据如下指令：如何安装 Firebug

Internet Explorer

打开浏览器

从菜单选择工具

从工具选择开发者工具

最后，选择控制台

Opera

打开浏览器

请前往网页：http://dev.opera.com

根据如下指令：如何安装 Firebug Lite

Safari Develop Menu

点击 Safari 菜单，偏好设置，高级

选中“在菜单栏中启用开发菜单”

当菜单中出现新选项“开发”时，选择“显示错误控制台”

您知道吗？

调试是测试、找到并减少计算机程序中 bug（错误）的过程。

历史上首个已知的计算机 bug 是一只卡在电子器件中的真实的虫子（一种昆虫）。

# 第57章 JavaScript 样式指南

请始终为您所有的 JavaScript 项目使用相同的代码约定。

JavaScript 代码约定

代码约定(Coding conventions)指的是编程的样式指导方针。这些原则大体上包括：

变量和函数的命名和声明规则

使用空格、缩进和注释的规则

编程习惯和准则

代码约定确保质量：

改善代码可读性

提升代码可维护性

代码约定可以是团队遵守的成文规则，也可以是您个人的编码习惯。

本页介绍 W3School 使用的通用 JavaScript 代码约定。

您应该继续阅读下一章“最佳实践”，学习如何避免编码陷阱。

变量名

在 W3School，我们对标识符名称（变量和函数）使用了驼峰式大小写。

所有名称以字母开头。

在本页的底部，我们会更广泛地讨论命名规则。

firstName = "Bill";

lastName = "Gates";

price = 19.90;

tax = 0.20;

fullPrice = price + (price \* tax);

运算符周围的空格

请始终在运算符（ = + - \* / ）周围以及逗号之后添加空格：

实例

var x = y + z;

var values = ["Volvo", "Saab", "Fiat"];

代码缩进

请始终使用对代码块缩进使用 4 个空格：

函数

function toCelsius(fahrenheit) {

return (5 / 9) \* (fahrenheit - 32);

}

请不要对缩进使用制表符。不同的编辑器对 tab 的解释也不尽相同。

语句规则

简单语句的通用规则：

请始终以分号结束单条语句：

实例

var values = ["Volvo", "Saab", "Fiat"];

var person = {

firstName: "Bill",

lastName: "Gates",

age: 50,

eyeColor: "blue"

};

针对复杂语句（compound）的通用规则：

请在第一行的结尾处写开括号

请在开括号前使用一个空格

请在新行上写闭括号，不带前导空格

请不要以分号来结束复杂语句

函数：

function toCelsius(fahrenheit) {

return (5 / 9) \* (fahrenheit - 32);

}

循环：

for (i = 0; i < 5; i++) {

x += i;

}

条件：

if (time < 20) {

greeting = "Good day";

} else {

greeting = "Good evening";

}

对象规则

针对对象定义的通用规则：

把开括号与对象名放在同一行

在每个属性与其值之间使用冒号加一个空格

不要在最后一个属性值对后面写逗号

请在新行上写闭括号，不带前导空格

请始终以分号结束对象定义

实例

var person = {

firstName: "Bill",

lastName: "Gates",

age: 19,

eyeColor: "blue"

};

可以对短对象在一行中进行压缩，只在属性之间使用空格，就像这样：

var person = {firstName:"Bill", lastName:"Gates", age:50, eyeColor:"blue"};

行长度小于 80

为了提高可读性，请避免每行的长度超过 80 个字符。

如果 JavaScript 语句超过一行的长度，换行的最佳位置是运算符或逗号之后。

实例

<!DOCTYPE html>

<html>

<body>

<h2>我的网页</h2>

<p>代码行换行的最佳位置是在运算符或逗号之后。</p>

<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML =

"Hello Dolly.";

</script>

</body>

</html>

命名约定

请始终对您所有的代码使用相同的命名约定。例如：

变量和函数名以驼峰大小写来写

全局变量使用大写（我们不这样做，但是相当普遍）

常量（比如 PI）使用大写

我们是否应在变量名中使用连字符、驼峰大小写或下划线吗？

这是程序员们经常讨论的问题。答案取决于这个问题是谁回答的：

HTML 和 CSS 中的连字符：

HTML5 属性能够以 data- 开头（data-quantity, data-price）。

CSS 在 property-names 中使用连字符（font-size）。

Hyphens 可被错误地视为减法运算符。JavaScript 命名不允许使用连字符。

下划线：

许多程序员喜欢使用下划线（date\_of\_birth），特别是在 SQL 数据库中。

下划线经常被用在 PHP 参考资料中。

帕斯卡命名法（PascalCase）：

C 语言程序员经常使用帕斯卡命名法。

驼峰大小写（camelCase）：

JavaScript 本身、jQuery 以及其他 JavaScript 库使用驼峰大小写。

JavaScript 命名请不要以 $ 符号开头。此举会引起 JavaScript 库名称冲突。

在 HTML 中加载 JavaScript

使用简单的语法来加载外部脚本（type 属性不是必需的）：

<script src="myscript.js"></script>

访问 HTML 元素

使用“不整洁的” HTML 样式的后果，也许是导致 JavaScript 错误。

这两条 JavaScript 语句会产生不同的结果：

var obj = getElementById("Demo")

var obj = getElementById("demo")

如果可能，请在 HTML 中使用相同的命名约定（就像 JavaScript 那样）。

访问 HTML 样式指南。

文件扩展名

HTML 文件应该使用 .html 扩展名（而非 .htm）。

CSS 文件应该使用 .css 扩展名。

JavaScript 文件应该使用 .js 扩展名。

使用小写文件名

大多数 web 服务器（Apache、Unix）对文件名的大小写敏感：

london.jpg 无法视作 London.jpg 进行访问。

其他 web 服务器（微软的 IIS）对大小写不敏感：

london.jpg 能够以 London.jpg 或 london.jpg 来访问。

如果您混合使用大小写，则必须严格保持连续和一致。

如果您将站点从大小写不敏感的服务器转移至对大小写敏感的服务器，即使这种小错误也可能破坏您的网站。

为了避免这些问题，请始终使用小写文件名（如果可能）。

性能

计算机不会使用代码约定。大部分规则对程序的执行影响很小。

缩进和额外的空格对小段脚本并不重要。

对于开发中的脚本，应该优先考虑可读性。应该缩小更大型的生产脚本。

# 第58章 JavaScript 最佳实践

请避免全局变量、new、===、eval()

避免全局变量

请尽量少地使用全局变量。

它包括所有的数据类型、对象和函数。

全局变量和函数可被其他脚本覆盖。

请使用局部变量替代，并学习如何使用闭包。

始终声明局部变量

所有在函数中使用的变量应该被声明为局部变量。

局部变量必须通过 var 关键词来声明，否则它们将变成全局变量。

严格模式不允许未声明的变量。

在顶部声明

一项好的编码习惯是把所有声明放在每段脚本或函数的顶部。

这么做的好处是：

获得更整洁的代码

提供了查找局部变量的好位置

更容易避免不需要的全局变量

减少不需要的重新声明的可能性

// 在顶部声明

var firstName, lastName, price, discount, fullPrice;

// 稍后使用

firstName = "Bill";

lastName = "Gates";

price = 19.90;

discount = 0.10;

fullPrice = price \* 100 / discount;

也可以用于循环变量：

// 在顶部声明

var i;

// 稍后使用

for (i = 0; i < 5; i++) {

默认地，JavaScript 会将所有声明移至顶部（JavaScript hoisting）。

初始化变量

在您声明变量时对其进行初始化是个好习惯。

这么做的好处是：

更整洁的代码

在单独的位置来初始化变量

避免未定义值

// 在开头进行声明和初始化

var firstName = "",

lastName = "",

price = 0,

discount = 0,

fullPrice = 0,

myArray = [],

myObject = {};

变量初始化使我们能够了解预期用途和预期的数据类型。

请不要声明数值、字符串或布尔对象

请始终将数值、字符串或布尔值视作原始值。而非对象。

如果把这些类型声明为对象，会拖慢执行速度，并产生讨厌的副作用：

实例

<!DOCTYPE html>

<html>

<body>

<p>切勿将字符串创建为对象。</p>

<p>字符串和对象无法安全地进行比较。</p>

<p id="demo"></p>

<script>

var x = "Bill"; // x 是字符串

var y = new String("Bill"); // y 是对象

document.getElementById("demo").innerHTML = x===y;

</script>

</body>

</html>

或者甚至更糟：

实例

<!DOCTYPE html>

<html>

<body>

<p>切勿将字符串创建为对象。</p>

<p>JavaScript 无法比较对象。</p>

<p id="demo"></p>

<script>

var x = new String("Bill");

var y = new String("Bill");

document.getElementById("demo").innerHTML = x==y;

</script>

</body>

</html>

请勿使用 new Object()

请使用 {} 来代替 new Object()

请使用 "" 来代替 new String()

请使用 0 来代替 new Number()

请使用 false 来代替 new Boolean()

请使用 [] 来代替 new Array()

请使用 /()/ 来代替 new RegExp()

请使用 function (){}来代替 new Function()

实例

var x1 = {}; // 新对象

var x2 = ""; // 新的原始字符串值

var x3 = 0; // 新的原始数值

var x4 = false; // 新的原始布尔值

var x5 = []; // 新的数组对象

var x6 = /()/; // 新的正则表达式

var x7 = function(){}; // 新的函数对象

意识到自动类型转换

请意识到数值会被意外转换为字符串或 NaN（Not a Number）。

JavaScript 属于松散类型。变量可包含不同的数据类型，并且变量能够改变其数据类型：

实例

var x = "Hello"; // typeof x 为字符串

x = 5; // 把 typeof x 更改为数值

如果进行数学运算，JavaScript 能够将数值转换为字符串：

实例

var x = 5 + 7; // x.valueOf() 是 12, typeof x 是数值

var x = 5 + "7"; // x.valueOf() 是 57, typeof x 是字符串

var x = "5" + 7; // x.valueOf() 是 57, typeof x 是字符串

var x = 5 - 7; // x.valueOf() 是 -2, typeof x 是数值

var x = 5 - "7"; // x.valueOf() 是 -2, typeof x 是数值

var x = "5" - 7; // x.valueOf() 是 -2, typeof x 是数值

var x = 5 - "x"; // x.valueOf() 是 NaN, typeof x 是数值

用字符串减去字符串，不会产生错误而是返回 NaN（Not a Number）：

实例

"Hello" - "Dolly" // 返回 NaN

使用 === 比较

== 比较运算符总是在比较之前进行类型转换（以匹配类型）。

=== 运算符会强制对值和类型进行比较：

实例

0 == ""; // true

1 == "1"; // true

1 == true; // true

0 === ""; // false

1 === "1"; // false

1 === true; // false

使用 Parameter Defaults

如果调用函数时缺少一个参数，那么这个缺失参数的值会被设置为 undefined。

undefined 值会破坏您的代码。为参数设置默认值是一个好习惯。

实例

function myFunction(x, y) {

if (y === undefined) {

y = 0;

}

}

请在函数参数这一章阅读更多有关函数参数的内容。

用 default 来结束 switch

请使用 default 来结束您的 switch 语句。即使您认为没有这个必要。

实例

switch (new Date().getDay()) {

case 0:

day = "Sunday";

break;

case 1:

day = "Monday";

break;

case 2:

day = "Tuesday";

break;

case 3:

day = "Wednesday";

break;

case 4:

day = "Thursday";

break;

case 5:

day = "Friday";

break;

case 6:

day = "Saturday";

break;

default:

day = "Unknown";

}

避免使用 eval()

eval() 函数用于将文本作为代码来允许。在几乎所有情况下，都没有必要使用它。

因为允许任意代码运行，它同时也意味着安全问题。

# 第59章 JavaScript 常见错误

本章指出一些常见的 JavaScript 错误。

意外使用赋值运算符

如果程序员在 if 语句中意外使用赋值运算符（=）而不是比较运算符（===），JavaScript 程序可能会产生一些无法预料的结果。

这条 if 语句返回 false（正如预期），因为 x 不等于 10：

var x = 0;

if (x == 10)

亲自试一试

这条 if 语句返回 true（也许不像预期），因为 10 为 true：

var x = 0;

if (x = 10)

亲自试一试

这条 if 语句返回 false（也许不像预期），因为 0 为 false：

var x = 0;

if (x = 0)

亲自试一试

赋值总是返回赋值的值。

期望松散的比较

在常规比较中，数据类型不重要。这条 if 语句返回 true：

var x = 10;

var y = "10";

if (x == y)

亲自试一试

在严格比较中，数据类型确实重要。这条 if 语句返回 false：

var x = 10;

var y = "10";

if (x === y)

亲自试一试

有一个常见的错误是忘记在 switch 语句中使用严格比较：

这条 switch 语句会显示提示框：

var x = 10;

switch(x) {

case 10: alert("Hello");

}

亲自试一试

这条 switch 语句不会显示提示框：

var x = 10;

switch(x) {

case "10": alert("Hello");

}

亲自试一试

令人困惑的加法和级联

加法用于加数值。

级联（Concatenation）用于加字符串。

在 JavaScript 中，这两种运算均使用相同的 + 运算符。

正因如此，将数字作为数值相加，与将数字作为字符串相加，将产生不同的结果：

var x = 10 + 5; // x 中的结果是 15

var x = 10 + "5"; // x 中的结果是　"105"

亲自试一试

如果是两个变量相加，很难预测结果：

var x = 10;

var y = 5;

var z = x + y; // z 中的结果是 15

var x = 10;

var y = "5";

var z = x + y; // z 中的结果是 "105"

亲自试一试

令人误解的浮点

JavaScript 中的数字均保存为 64 位的浮点数（Floats）。

所有编程语言，包括 JavaScript，都存在处理浮点值的困难：

var x = 0.1;

var y = 0.2;

var z = x + y // z 中的结果并不是 0.3

亲自试一试

为了解决上面的问题，请使用乘除运算：

实例

var z = (x \* 10 + y \* 10) / 10; // z 中的结果将是 0.3

亲自试一试

对 JavaScript 字符串换行

JavaScript 允许您把一条语句换行为两行：

例子 1

var x =

"Hello World!";

亲自试一试

但是，在字符串中间来换行是不对的：

例子 2

var x = "Hello

World!";

亲自试一试

如果必须在字符串中换行，则必须使用反斜杠：

例子 3

var x = "Hello \

World!";

亲自试一试

错位的分号

因为一个错误的分号，此代码块无论 x 的值如何都会执行：

if (x == 19);

{

// code block

}

亲自试一试

对 return 语句进行换行

在一行的结尾自动关闭语句是默认的 JavaScript 行为。

正因如此，下面两个例子返回相同的结果：

例子 1

function myFunction(a) {

var power = 10

return a \* power

}

亲自试一试

例子 2

function myFunction(a) {

var power = 10;

return a \* power;

}

亲自试一试

JavaScript 也允许您将一条语句换行为两行。

正因如此，例子 3 也将返回相同的结果：

例子 3

function myFunction(a) {

var

power = 10;

return a \* power;

}

亲自试一试

但是，如果把 return 语句换行为两行会发生什么呢：

例子 4

function myFunction(a) {

var

power = 10;

return

a \* power;

}

亲自试一试

此函数将返回 undefined！

为什么呢？因为 JavaScript 认为你的意思是：

例子 5

function myFunction(a) {

var

power = 10;

return;

a \* power;

}

亲自试一试

解释

如果某条语句是不完整的：

var

JavaScript 将通过读取下一行来完成这条语句：

power = 10;

但是由于这条语句是完整的：

return

JavaScript 会自动关闭该语句：

return;

发生这种情况是因为，在 JavaScript 中，用分号来关闭（结束）语句是可选的。

JavaScript 会在行末关闭 return 语句，因为它本身就是一条完整的语句。

所以，绝不要对 return 语句进行换行。

通过命名索引来访问数组

很多编程语言支持带有命名索引的数组。

带有命名索引的数组被称为关联数组（或散列）。

JavaScript 不支持带有命名索引的数组。

在 JavaScript 中，数组使用数字索引：

实例

var person = [];

person[0] = "Bill";

person[1] = "Gates";

person[2] = 46;

var x = person.length; // person.length 将返回 3

var y = person[0]; // person[0] 将返回 "Bill"

亲自试一试

在 JavaScript 中，对象使用命名索引。

如果您使用命名索引，那么在访问数组时，JavaScript 会将数组重新定义为标准对象。

在自动重定义之后，数组方法或属性将产生未定义或非正确的结果：

实例

var person = [];

person["firstName"] = "Bill";

person["lastName"] = "Gates";

person["age"] = 46;

var x = person.length; // person.length 将返回 0

var y = person[0]; // person[0] 将返回 undefined

亲自试一试

用逗号来结束定义

对象和数组定义中的尾随逗号在 ECMAScript 5 中是合法的。

对象实例：

person = {firstName:"Bill", lastName:"Gates", age:62,}

数组实例：

points = [35, 450, 2, 7, 30, 16,];

警告！！

Internet Explorer 8 会崩溃。

JSON 不允许尾随逗号。

JSON:

person = {firstName:"Bill", lastName:"Gates", age:62}

JSON:

points = [35, 450, 2, 7, 30, 16];

Undefined 不是 Null

JavaScript 对象、变量、属性和方法可以是未定义的。

此外，空的 JavaScript 对象的值可以为 null。

这可能会使测试对象是否为空变得有点困难。

您可以通过测试类型是否为 undefined，来测试对象是否存在：

实例

if (typeof myObj === "undefined")

亲自试一试

但是您无法测试对象是否为 null，因为如果对象未定义，将抛出错误：

不正确的：

if (myObj === null)

要解决此问题，必须测试对象是否为 null，而不是未定义。

但这仍然会引发错误：

不正确的：

if (myObj !== null && typeof myObj !== "undefined")

因此，在测试非 null 之前，必须先测试未定义：

正确的：

if (typeof myObj !== "undefined" && myObj !== null)

亲自试一试

期望块级范围

JavaScript 不会为每个代码块创建新的作用域。

很多编程语言都是如此，但是 JavaScript 并非如此。

认为这段代码会返回 undefined，是新的 JavaScript 开发者的常见错误：

实例

for (var i = 0; i < 10; i++) {

// 代码块

}

return i;

# 第60章 JavaScript 性能

如何加速您的 JavaScript 代码。

减少循环中的活动

编程经常会用到循环。

循环每迭代一次，循环中的每条语句，包括 for 语句，都会被执行。

能够放在循环之外的语句或赋值会使循环运行得更快。

差的代码：

var i;

for (i = 0; i < arr.length; i++) {

更好的代码：

var i;

var l = arr.length;

for (i = 0; i < l; i++) {

循环每次迭代时，坏代码就会访问数组的 length 属性。

好代码在循环之外访问 length 属性，使循环更快。

减少 DOM 访问

与其他 JavaScript 相比，访问 HTML DOM 非常缓慢。

假如您期望访问某个 DOM 元素若干次，那么只访问一次，并把它作为本地变量来使用：

实例

var obj;

obj = document.getElementById("demo");

obj.innerHTML = "Hello";

亲自试一试

缩减 DOM 规模

请尽量保持 HTML DOM 中较少的元素数量。

这么做总是会提高页面加载，并加快渲染（页面显示），尤其是在较小的设备上。

每一次试图搜索 DOM（比如 getElementsByTagName）都将受益于一个较小的 DOM。

避免不必要的变量

请不要创建不打算存储值的新变量。

通常您可以替换代码：

var fullName = firstName + " " + lastName;

document.getElementById("demo").innerHTML = fullName;

用这段代码：

document.getElementById("demo").innerHTML = firstName + " " + lastName

延迟 JavaScript 加载

请把脚本放在页面底部，使浏览器首先加载页面。

脚本在下载时，浏览器不会启动任何其他的下载。此外所有解析和渲染活动都可能会被阻塞。

HTTP 规范定义浏览器不应该并行下载超过两种要素。

一个选项是在 script 标签中使用 defer="true"。defer 属性规定了脚本应该在页面完成解析后执行，但它只适用于外部脚本。

如果可能，您可以在页面完成加载后，通过代码向页面添加脚本：

实例

<script>

window.onload = downScripts;

function downScripts() {

var element = document.createElement("script");

element.src = "myScript.js";

document.body.appendChild(element);

}

</script>

避免使用 with

请避免使用 with 关键词。它对速度有负面影响。它也将混淆 JavaScript 作用域。

严格模式中不允许 with 关键词。

# 第61章 JavaScript 保留词

JavaScript 保留词

在 JavaScript 中，您不能把这些保留词作为变量、标记或函数名来使用：

|  |  |  |  |
| --- | --- | --- | --- |
| abstract | arguments | await\* | 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 |

用星号标记的关键词是 ECMAScript 5 和 6 中的新词。

您可以在 JS 版本中阅读更多有关不同 JavaScript 版本的内容。

被删除的保留词

以下保留词已被从 ECMAScript 5/6 标准中删除：

|  |  |  |  |
| --- | --- | --- | --- |
| abstract | boolean | byte | char |
| double | final | float | goto |
| int | long | native | short |
| synchronized | throws | transient | volatile |

请不要将这些保留词用作变量。并非所有浏览器都完全支持 ECMAScript 5/6。

JavaScript 对象、属性和方法

您还应该避免使用 JavaScript 内建对象的名称、属性和方法：

|  |  |  |  |
| --- | --- | --- | --- |
| Array | Date | eval | function |
| hasOwnProperty | Infinity | isFinite | isNaN |
| isPrototypeOf | length | Math | NaN |
| name | Number | Object | prototype |
| String | toString | undefined | valueOf |

Java 保留词

JavaScript 常与 Java 一起使用。您应该避免把某些 Java 对象和属性用作 JavaScript 标识符：

|  |  |  |  |
| --- | --- | --- | --- |
| getClass | java | JavaArray | javaClass |
| JavaObject | JavaPackage |  |  |

其他保留词

JavaScript 能够在很多应用程序中被用作编程语言。

您还应该避免使用 HTML 和 Window 对象和属性的名称：

|  |  |  |  |
| --- | --- | --- | --- |
| 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 事件处理程序

此外您应该避免使用所有 HTML 事件处理程序的名称。

例如：

|  |  |  |  |
| --- | --- | --- | --- |
| onblur | onclick | onerror | onfocus |
| onkeydown | onkeypress | onkeyup | onmouseover |
| onload | onmouseup | onmousedown | onsubmit |

# 第62章 JavaScript 版本

JavaScript 版本

JavaScript 由 Brendan Eich 于 1995 年发明，并于 1997 年成为 ECMA 标准。

ECMAScript 是该语言的官方名称。

从 2015 年起，ECMAScript 按年命名（ECMAScript 2015）。

ECMAScript 版本

|  |  |  |
| --- | --- | --- |
| 版本 | 官方名称 | 描述 |
| 1 | ECMAScript 1 (1997) | 第一版。 |
| 2 | ECMAScript 2 (1998) | 只改变编辑方式。 |
| 3 | ECMAScript 3 (1999) | 添加了正则表达式。  添加了 try/catch。 |
| 4 | ECMAScript 4 | 从未发布过。 |
| 5 | ECMAScript 5 (2009)  阅读更多：JS ES5 | 添加了“严格模式”。  添加了 JSON 支持。  添加了 String.trim()。  添加了 Array.isArray()。  添加了数组迭代方法。 |
| 5.1 | ECMAScript 5.1 (2011) | 编辑改变。 |
| 6 | ECMAScript 2015  阅读更多：JS ES6 | 添加了 let 和 const  添加了默认参数值  添加了 Array.find()  添加了 Array.findIndex() |
| 7 | ECMAScript 2016 | 添加了指数运算符（\*\*）。  添加了 Array.prototype.includes。 |
| 8 | ECMAScript 2017 | 添加了字符串填充。  添加了新的 Object 属性。  添加了异步功能。  添加了共享内存。 |
| 9 | ECMAScript 2018 | 添加了 rest / spread 属性。  添加了异步迭代。  添加了 Promise.finally()。  增加 RegExp |

ECMAScript 通常缩写为 ES。

浏览器支持

所有浏览器都完全支持 ECMAScript 3。

所有现代浏览器都完全支持 ECMAScript 5。

对 ES5 的浏览器支持（2009）

|  |  |  |
| --- | --- | --- |
| 浏览器 | 版本 | 起始日期 |
| Chrome | 23 | 2012 年 9 月 |
| Firefox | 21 | 2013 年 4 月 |
| IE | 9\* | 2011 年 3 月 |
| IE / Edge | 10 | 2012 年 9 月 |
| Safari | 6 | 2012 年 7 月 |
| Opera | 15 | 2013 年 7 月 |

对 ES6 的浏览器支持（ECMAScript 2015）

|  |  |  |
| --- | --- | --- |
| 浏览器 | 版本 | 日期 |
| Chrome | 58 | 2017 年 4 月 |
| Firefox | 54 | 2017 年 6 月 |
| Edge | 14 | 2016 年 8 月 |
| Safari | 10 | 2016 年 9 月 |
| Opera | 55 | 2017 年 8 月 |

对 ES7 的浏览器支持（ECMAScript 2016）

|  |  |  |
| --- | --- | --- |
| 浏览器 | 版本 | 日期 |
| Chrome | 68 | 2018 年 5 月 |
| Opera | 47 | 2018 年 7 月 |

JavaScript / ECMAScript

JavaScript 是为 Netscape 开发的。 Netscape 2 是第一个运行 JavaScript 的浏览器。

在 Netscape 之后，Mozilla 基金会继续为 Firefox 浏览器开发 JavaScript。

最新的 JavaScript 版本是 1.8.5。 （与 ECMAScript 5 相同）。

ECMAScript 由 ECMA International 在组织采用 JavaScript 之后开发的。

第一版 ECMAScript 于 1997 年发布。

此列表比较了不同产品的版本号：

|  |  |  |  |
| --- | --- | --- | --- |
| 年份 | JavaScript | ECMA | 浏览器 |
| 1996 | 1.0 |  | Netscape 2 |
| 1997 |  | ECMAScript 1 | IE 4 |
| 1998 | 1.3 |  | Netscape 4 |
| 1999 |  | ECMAScript 2 | IE 5 |
| 2000 |  | ECMAScript 3 | IE 5.5 |
| 2000 | 1.5 |  | Netscape 6 |
| 2000 | 1.5 |  | Firefox 1 |
| 2011 |  | ECMAScript 5 | IE 9（除了 "use strict"） |
| 2011 | 1.8.5 |  | Firefox 4（除了 parseInt 中的前导零） |
| 2012 |  |  | IE 10 |
| 2012 |  |  | Chrome 23 |
| 2012 |  |  | Safari 6 |
| 2013 |  |  | Firefox 21 |
| 2013 |  |  | Opera 15 |
| 2015 | ECMAScript 2015 | 在所有浏览器中部分支持 |  |

IE 4 是第一个支持 ECMAScript 1（1997）的浏览器。

IE 5 是第一个支持 ECMAScript 2（1999）的浏览器。

IE 5.5 是第一个支持 ECMAScript 3（2000）的浏览器。

IE 9 \* 是第一个支持 ECMAScript 5（2011）的浏览器。

Internet Explorer 9 不支持 ECMAScript 5“use strict”。

Chrome 23、IE 10 和Safari 6 是第一批完全支持 ECMAScript 5 的浏览器：

# 第63章 JavaScript ES5

ECMAScript 5 是什么？

ECMAScript 5 也称为 ES5 和 ECMAScript 2009。

本章介绍 ES5 的一些最重要的特性。

ECMAScript 5 特性

这些是 2009 年发布的新特性：

"use strict" 指令

String.trim()

Array.isArray()

Array.forEach()

Array.map()

Array.filter()

Array.reduce()

Array.reduceRight()

Array.every()

Array.some()

Array.indexOf()

Array.lastIndexOf()

JSON.parse()

JSON.stringify()

Date.now()

属性 Getter 和 Setter

新的对象属性和方法

ECMAScript 5 语法更改

对字符串的属性访问 [ ]

数组和对象字面量中的尾随逗号

多行字符串字面量

作为属性名称的保留字

"use strict" 指令

“use strict” 定义 JavaScript 代码应该以“严格模式”执行。

例如，使用严格模式，不能使用未声明的变量。

您可以在所有程序中使用严格模式。它可以帮助您编写更清晰的代码，例如阻止您使用未声明的变量。

“use strict” 只是一个字符串表达式。旧浏览器如果不理解它们就不会抛出错误。

请阅读 JS 严格模式 中的更多内容。

String.trim()

String.trim() 删除字符串两端的空白字符。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript String.trim()</h1>

<p>点击这个按钮来输出已删除空白字符的字符串。</p>

<button onclick="myFunction()">试一试</button>

<p><strong>注释：</strong>Internet Explorer 8 或更低版本不支持 trim() 方法。</p>

<script>

function myFunction() {

var str = " Hello World! ";

alert(str.trim());

}

</script>

</body>

</html>

Array.isArray()

isArray() 方法检查对象是否为数组。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.isArray()</h1>

<p>单击按钮以检查“fruits”是否为数组。</p>

<button onclick="myFunction()">试一试</button>

<p id="demo"></p>

<script>

function myFunction() {

var fruits = ["Banana", "Orange", "Apple", "Mango"];

var x = document.getElementById("demo");

x.innerHTML = Array.isArray(fruits);

}

</script>

</body>

</html>

Array.forEach()

forEach() 方法为每个数组元素调用一次函数。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.forEach()</h1>

<p>为每个数组元素调用一次函数。</p>

<p id="demo"></p>

<script>

var txt = "";

var numbers = [45, 4, 9, 16, 25];

numbers.forEach(myFunction);

document.getElementById("demo").innerHTML = txt;

function myFunction(value) {

txt = txt + value + "<br>";

}

</script>

</body>

</html>

Array.map()

这个例子给每个数组值乘以 2：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.map()</h1>

<p>通过对每个数组元素执行函数来创建新数组。</p>

<p id="demo"></p>

<script>

var numbers1 = [45, 4, 9, 16, 25];

var numbers2 = numbers1.map(myFunction);

document.getElementById("demo").innerHTML = numbers2;

function myFunction(value, index, array) {

return value \* 2;

}

</script>

</body>

</html>

Array.filter()

此例用值大于 18 的元素创建一个新数组：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.filter()</h1>

<p>使用通过测试的所有数组元素创建一个新数组。</p>

<p id="demo"></p>

<script>

var numbers = [45, 4, 9, 16, 25];

var over18 = numbers.filter(myFunction);

document.getElementById("demo").innerHTML = over18;

function myFunction(value, index, array) {

return value > 18;

}

</script>

</body>

</html>

Array.reduce()

这个例子确定数组中所有数的总和：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.reduce()</h1>

<p>此例确定数组中所有数字的总和：</p>

<p id="demo"></p>

<script>

var numbers = [45, 4, 9, 16, 25];

var sum = numbers.reduce(myFunction);

document.getElementById("demo").innerHTML = "The sum is " + sum;

function myFunction(total, value, index, array) {

return total + value;

}

</script>

</body>

</html>

Array.reduceRight()

这个例子同样是确定数组中所有数的总和：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.reduceRight()</h1>

<p>此例确定数组中所有数字的总和：</p>

<p id="demo"></p>

<script>

var numbers = [45, 4, 9, 16, 25];

var sum = numbers.reduceRight(myFunction);

document.getElementById("demo").innerHTML = "The sum is " + sum;

function myFunction(total, value) {

return total + value;

}

</script>

</body>

</html>

Array.every()

这个例子检查是否所有值都超过 18：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.every()</h1>

<p>every() 方法检查所有数组值是否通过测试。</p>

<p id="demo"></p>

<script>

var numbers = [45, 4, 9, 16, 25];

var allOver18 = numbers.every(myFunction);

document.getElementById("demo").innerHTML = "All over 18 is " + allOver18;

function myFunction(value, index, array) {

return value > 18;

}

</script>

</body>

</html>

Array.some()

这个例子检查某些值是否超过 18：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.some()</h1>

<p>some() 方法检查某些数组值是否通过了测试。</p>

<p id="demo"></p>

<script>

var numbers = [45, 4, 9, 16, 25];

var allOver18 = numbers.some(myFunction);

document.getElementById("demo").innerHTML = "Some over 18 is " + allOver18;

function myFunction(value) {

return value > 18;

}

</script>

</body>

</html>

Array.indexOf()

检索数组中的某个元素值并返回其位置：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.indexOf()</h1>

<p id="demo"></p>

<script>

var fruits = ["Apple", "Orange", "Apple", "Mango"];

var a = fruits.indexOf("Apple");

document.getElementById("demo").innerHTML = "Apple 被找到的位置是：" + (a + 1);

</script>

<p>Internet Explorer 8 或更早的版本不支持 indexOf()。</p>

</body>

</html>

Array.lastIndexOf()

Array.lastIndexOf() 与 Array.indexOf() 类似，但是从数组结尾处开始检索。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.lastIndexOf()</h1>

<p id="demo"></p>

<script>

var fruits = ["Apple", "Orange", "Apple", "Mango"];

var a = fruits.lastIndexOf("Apple");

document.getElementById("demo").innerHTML = "Apple is found in position " + (a + 1);

</script>

<p>Internet Explorer 8 或更早的版本不支持 lastIndexOf()。</p>

</body>

</html>

JSON.parse()

JSON 的一个常见用途是从 Web 服务器接收数据。

想象一下，您从Web服务器收到这条文本字符串：

'{"name":"Bill", "age":62, "city":"Seatle"}'

JavaScript 函数 JSON.parse() 用于将文本转换为 JavaScript 对象：

var obj = JSON.parse('{"name":"Bill", "age":62, "city":"Seatle"}');

<!DOCTYPE html>

<html>

<body>

<h1>用 JSON 字符串创建对象</h1>

<p id="demo"></p>

<script>

var txt = '{"name":"Bill Gates", "age":62, "city":"Seattle"}'

var obj = JSON.parse(txt);

document.getElementById("demo").innerHTML = obj.name + ", " + obj.age;

</script>

</body>

</html>

JSON.stringify()

JSON 的一个常见用途是将数据发送到Web服务器。

将数据发送到 Web 服务器时，数据必须是字符串。

想象一下，我们在 JavaScript 中有这个对象：

var obj = {"name":"Bill", "age":62, "city":"Seatle"};

请使用 JavaScript 函数 JSON.stringify() 将其转换为字符串。

var myJSON = JSON.stringify(obj);

结果将是遵循 JSON 表示法的字符串。

myJSON 现在是一个字符串，准备好发送到服务器：

实例

var obj = {"name":"Bill", "age":62, "city":"Seatle"};

var myJSON = JSON.stringify(obj);

document.getElementById("demo").innerHTML = myJSON;

Date.now()

Date.now() 返回自零日期（1970 年 1 月 1 日 00:00:00:00）以来的毫秒数。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Date.now()</h1>

<p>Date.now() 是ECMAScript 5（2009）中的新功能：</p>

<p id="demo1"></p>

<p id="demo2"></p>

<script>

document.getElementById("demo1").innerHTML = Date.now();

var d = new Date();

document.getElementById("demo2").innerHTML = d.getTime();

</script>

</body>

</html>

Date.now() 的返回与在 Date 对象上执行 getTime() 的结果相同。

请在 JS 日期 中学习更多。

属性 Getter 和 Setter

ES5 允许您使用类似于获取或设置属性的语法来定义对象方法。

这个例子为名为 fullName 的属性创建一个 getter：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Getter 和 Setter</h1>

<p>Getter 和 Setter 允许您通过方法获取和设置属性。</p>

<p>此例为名为 fullName 的属性创建一个 getter。</p>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

get fullName() {

return this.firstName + " " + this.lastName;

}

};

// 使用 getter 来显示来自对象的数据：

document.getElementById("demo").innerHTML = person.fullName;

</script>

</body>

</html>

这个例子为语言属性创建一个 setter 和一个 getter：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Getter 和 Setter</h1>

<p>Getter 和 Setter 允许您通过方法获取和设置属性。</p>

<p>此例为 language 属性创建 setter 和 getter。</p>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

language : "NO",

get lang() {

return this.language;

},

set lang(value) {

this.language = value;

}

};

// 使用 setter 设置对象属性：

person.lang = "en";

// 使用 getter 显示来自对象的数据：

document.getElementById("demo").innerHTML = person.lang;

</script>

</body>

</html>

这个例子使用 setter 来确保语言的大写更新：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Getter 和 Setter</h1>

<p>Getter 和 Setter 允许您通过方法获取和设置属性。</p>

<p>此例具有修改的 setter 以保护语言的大写更新。</p>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

language : "",

set lang(value) {

this.language = value.toUpperCase();

}

};

// 使用 setter 设置对象属性：

person.lang = "en";

// 显示对象中的数据：

document.getElementById("demo").innerHTML = person.language;

</script>

</body>

</html>

新的对象属性和方法

Object.defineProperty() 是 ES5 中的新对象方法。

它允许您定义对象属性和/或更改属性的值和/或元数据。

实例

<!DOCTYPE html>

<html>

<head>

<meta content="text/html; charset=iso-8859-2" http-equiv="Content-Type">

</head>

<body>

<h1>JavaScript defineProperty()</h1>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

?????? language : "NO",

};

// 更改属性：

Object.defineProperty(person, "language", {

value: "EN",

writable : true,

enumerable : true,

configurable : true

});

// 枚举属性

txt = "";

for (var x in person) {

txt += person[x] + "<br>";

}

document.getElementById("demo").innerHTML = txt;

</script>

</body>

</html>

下一个例子是相同的代码，但它隐藏了枚举中的语言属性：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript defineProperty()</h1>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

?? language : "NO",

};

// 更改属性：

Object.defineProperty(person, "language", {

value: "EN",

writable : true,

enumerable : false,

configurable : true

});

// 枚举属性：

txt = "";

for (var x in person) {

txt += person[x] + "<br>";

}

document.getElementById("demo").innerHTML = txt;

</script>

</body>

</html>

此例创建一个 setter 和 getter 来确保语言的大写更新：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript defineProperty()</h1>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

language : "NO"

};

// 更改属性：

Object.defineProperty(person, "language", {

get : function() { return language },

set : function(value) { language = value.toUpperCase()}

});

// 修改语言：

person.language = "en";

// 显示语言：

document.getElementById("demo").innerHTML = person.language;

</script>

</body>

</html>

ES5 新的对象方法

ECMAScript 5 为 JavaScript 添加了许多新的对象方法：

// 添加或更改对象属性

Object.defineProperty(object, property, descriptor)

// 添加或更改多个对象属性

Object.defineProperties(object, descriptors)

// 访问属性

Object.getOwnPropertyDescriptor(object, property)

// 将所有属性作为数组返回

Object.getOwnPropertyNames(object)

// 将可枚举属性作为数组返回

Object.keys(object)

// 访问原型

Object.getPrototypeOf(object)

// 防止向对象添加属性

Object.preventExtensions(object)

// 如果可以将属性添加到对象，则返回 true

Object.isExtensible(object)

// 防止更改对象属性（而不是值）

Object.seal(object)

// 如果对象被密封，则返回 true

Object.isSealed(object)

// 防止对对象进行任何更改

Object.freeze(object)

// 如果对象被冻结，则返回 true

Object.isFrozen(object)

请在 对象 ECMAScript5 中学习更多内容。

对字符串的属性访问

charAt() 方法返回字符串中指定索引（位置）的字符：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>charAt() 方法返回字符串中给定位置的字符：</p>

<p id="demo"></p>

<script>

var str = "HELLO WORLD";

document.getElementById("demo").innerHTML = str.charAt(0);

</script>

</body>

</html>

ECMAScript 5 允许对字符串进行属性访问：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>ECMAScript 5 字符串的属性访问：</p>

<p id="demo"></p>

<script>

var str = "HELLO WORLD";

document.getElementById("demo").innerHTML = str[0];

</script>

</body>

</html>

尾随逗号(Trailing Commas)

ECMAScript 5 允许在对象和数组定义中使用尾随逗号：

Object 实例

person = {

firstName: "Bill",

lastName: " Gates",

age: 62,

}

Array 实例

points = [

1,

5,

10,

25,

40,

100,

];

警告！！！

Internet Explorer 8 将崩溃。

JSON 不允许使用尾随逗号。

JSON 对象：

// 允许：

var person = '{"firstName":"Bill", "lastName":"Gates", "age":62}'

JSON.parse(person)

// 不允许：

var person = '{"firstName":"Bill", "lastName":"Gates", "age":62,}'

JSON.parse(person)

JSON 数组：

// 允许：

points = [40, 100, 1, 5, 25, 10]

// 不允许：

points = [40, 100, 1, 5, 25, 10,]

多行字符串

如果使用反斜杠转义，ECMAScript 5 允许多行的字符串文字（字面量）：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串</h1>

<p>您可以使用反斜杠在文本字符串中断开代码行。</p>

<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML = "Hello \

Dolly!";

</script>

</body>

</html>

\ 方法可能没有得到普遍的支持。

较旧的浏览器可能会以不同的方式处理反斜杠周围的空格。

一些旧的浏览器不允许 \ 字符后面的空格。

分解字符串文字的一种更安全的方法是使用字符串添加：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串</h1>

<p>打破字符串中代码行的最安全方法是使用字符串添加。</p>

<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML = "Hello " +

"Dolly!";

</script>

</body>

</html>

保留字作为属性名称

ECMAScript 5 允许保留字作为属性名称：

对象实例

<!DOCTYPE html>

<html>

<body>

<h1>ECMAScript 5</h1>

<p>ECMAScript 5 允许保留字作为属性名称：</p>

<p id="demo"></p>

<script>

var obj = {name: "Bill", new: "yes"};

document.getElementById("demo").innerHTML = obj.new;

</script>

</body>

</html>

# 第64章 JavaScript ES6

ECMAScript 6 是什么？

ECMAScript 6 也称为 ES6 和 ECMAScript 2015。

一些人把它称作 JavaScript 6。

本章介绍 ES6 中的一些新特性。

JavaScript let

JavaScript const

幂 (\*\*)

默认参数值

Array.find()

Array.findIndex()

JavaScript let

let 语句允许您使用块作用域声明变量。

实例

<!DOCTYPE html>

<html>

<body>

<h1>使用 let 声明变量</h1>

<p id="demo"></p>

<script>

var x = 10;

// Here x is 10

{

let x = 2;

// Here x is 2

}

// Here x is 10

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

JavaScript const

const 语句允许您声明常量（具有常量值的 JavaScript 变量）。

常量类似于 let 变量，但不能更改值。

实例

<!DOCTYPE html>

<html>

<body>

<h1>使用 const 声明变量</h1>

<p id="demo"></p>

<script>

var x = 10;

// Here x is 10

{

const x = 2;

// Here x is 2

}

// Here x is 10

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

指数运算符

取幂运算符（\*\*）将第一个操作数提升到第二个操作数的幂。

实例

<!DOCTYPE html>

<html>

<body>

<h1>\*\* 运算符</h1>

<p id="demo"></p>

<script>

var x = 5;

document.getElementById("demo").innerHTML = x \*\* 2;

</script>

</body>

</html>

x \*\* y 的结果与 Math.pow(x,y) 相同：

实例

<!DOCTYPE html>

<html>

<body>

<h1>Math.pow()</h1>

<p id="demo"></p>

<script>

var x = 5;

document.getElementById("demo").innerHTML = Math.pow(x,2);

</script>

</body>

</html>

默认参数值

ES6 允许函数参数具有默认值。

实例

<!DOCTYPE html>

<html>

<body>

<h1>默认参数值</h1>

<p id="demo"></p>

<script>

function myFunction(x, y = 10) {

// 如果未通过或未定义，则 y 为 10

return x + y;

}

document.getElementById("demo").innerHTML = myFunction(5);

</script>

</body>

</html>

Array.find()

find() 方法返回通过测试函数的第一个数组元素的值。

此例查找（返回）第一个大于 18 的元素（的值）：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.find()</h1>

<p id="demo"></p>

<script>

var numbers = [4, 9, 16, 25, 29];

var first = numbers.find(myFunction);

document.getElementById("demo").innerHTML = "大于 18 的第一个值是：" + first;

function myFunction(value, index, array) {

return value > 18;

}

</script>

</body>

</html>

请注意此函数接受 3 个参数：

项目值

项目索引

数组本身

Array.findIndex()

findIndex() 方法返回通过测试函数的第一个数组元素的索引。

此例确定大于 18 的第一个元素的索引：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Array.findIndex()</h1>

<p id="demo"></p>

<script>

var numbers = [4, 9, 16, 25, 29];

var first = numbers.findIndex(myFunction);

document.getElementById("demo").innerHTML = "大于 18 的第一个值的索引是：" + first;

function myFunction(value, index, array) {

return value > 18;

}

</script>

</body>

</html>

请注意此函数接受 3 个参数：

项目值

项目索引

数组本身

新的数字属性

ES6 将以下属性添加到 Number 对象：

EPSILON

MIN\_SAFE\_INTEGER

MAX\_SAFE\_INTEGER

实例

<!DOCTYPE html>

<html>

<body>

<h1>数字对象属性</h1>

<p>EPSILON</p>

<p id="demo"></p>

<script>

var x = Number.EPSILON;

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

实例

<!DOCTYPE html>

<html>

<body>

<h1>数字对象属性</h1>

<p>MIN\_SAFE\_INTEGER</p>

<p id="demo"></p>

<script>

var x = Number.MIN\_SAFE\_INTEGER;

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

实例

<!DOCTYPE html>

<html>

<body>

<h1>数字对象属性</h1>

<p>MAX\_SAFE\_INTEGER</p>

<p id="demo"></p>

<script>

var x = Number.MAX\_SAFE\_INTEGER;

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

新的数字方法

ES6 为 Number 对象添加了 2 个新方法：

Number.isInteger()

Number.isSafeInteger()

Number.isInteger() 方法

如果参数是整数，则 Number.isInteger() 方法返回 true。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字方法</h1>

<p>如果参数是整数，则 isInteger() 方法返回 true。</p>

<p>否则返回 false。</p>

<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML =

Number.isInteger(10) + "<br>" + Number.isInteger(10.5);

</script>

</body>

</html>

Number.isSafeInteger() 方法

安全整数是可以精确表示为双精度数的整数。

如果参数是安全整数，则 Number.isSafeInteger() 方法返回 true。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字方法</h1>

<p>如果参数是安全整数，则 isSafeInteger() 方法返回 true。</p>

<p>否则返回 false。</p>

<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML =

Number.isSafeInteger(10) + "<br>" + Number.isSafeInteger(12345678901234567890);

</script>

</body>

</html>

安全整数指的是从 -(253 - 1) 到 +(253 - 1) 的所有整数。

这是安全的：9007199254740991。这是不安全的：9007199254740992。

新的全局方法

ES6 还增加了 2 个新的全局数字方法：

isFinite()

isNaN()

isFinite() 方法

如果参数为 Infinity 或 NaN，则全局 isFinite() 方法返回 false。

否则返回 true：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字方法</h1>

<p>如果参数为 Infinity 或 NaN，则 isFinite() 方法返回 false。</p>

<p>否则返回 true。</p>

<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML =

isFinite(10 / 0) + "<br>" + isFinite(10 / 1);

</script>

</body>

</html>

isNaN() 方法

如果参数是 NaN，则全局 isNaN() 方法返回 true。否则返回 false：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数字方法</h1>

<p>如果参数是 NaN，则 isNan() 方法返回 true。否则返回 false。</p>

<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML =

isNaN("Hello") + "<br>" + isNaN("10");

</script>

</body>

</html>

箭头函数（Arrow Function）

箭头函数允许使用简短的语法来编写函数表达式。

您不需要 function 关键字、return 关键字以及花括号。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 箭头函数</h1>

<p>使用箭头函数，您不必键入 function 关键字、return 关键字和花括号。</p>

<p>IE11 或更早版本不支持箭头功能。</p>

<p id="demo"></p>

<script>

const x = (x, y) => x \* y;

document.getElementById("demo").innerHTML = x(5, 5);

</script>

</body>

</html>

箭头功能没有自己的 this。它们不适合定义对象方法。

箭头功能未被提升。它们必须在使用前进行定义。

使用 const 比使用 var 更安全，因为函数表达式始终是常量值。

如果函数是单个语句，则只能省略 return 关键字和花括号。因此，保留它们可能是一个好习惯：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 箭头函数</h1>

<p>IE11 或更早版本不支持箭头功能。</p>

<p id="demo"></p>

<script>

const x = (x, y) => { return x \* y };

document.getElementById("demo").innerHTML = x(5, 5);

</script>

</body>

</html>

# 第65章 ECMAScript 2016

JavaScript 命名约定始于 ES1、ES2、ES3、ES5 和 ES6。

但是，ECMAScript 2016 和 2017 未被称为 ES7 和 ES8。

自 2016 年以来，新版本按年份命名（ECMAScript 2016/2017/2018）。

ECMAScript 2016 中的新特性

本章介绍 ECMAScript 2016 的新特性：

JavaScript 求幂 (\*\*)

JavaScript 求幂赋值 (\*\*=)

JavaScript Array.prototype.includes

求幂运算符

幂运算符 (\*\*) 将第一个操作数提高到第二个操作数的幂。

实例

<!DOCTYPE html>

<html>

<body>

<h1>\*\* 运算符</h1>

<p id="demo"></p>

<script>

let x = 5;

let z = x \*\* 2;

document.getElementById("demo").innerHTML = z;

</script>

</body>

</html>

x \*\* y 产生与 Math.pow(x, y) 相同的结果：

实例

<!DOCTYPE html>

<html>

<body>

<h1>Math.pow()</h1>

<p id="demo"></p>

<script>

let x = 5;

let z = Math.pow(x, 2)

document.getElementById("demo").innerHTML = z;

</script>

</body>

</html>

求幂赋值

幂赋值运算符 (\*\*=) 将变量的值提高到右操作数的幂。

实例

<!DOCTYPE html>

<html>

<body>

<h1>求幂赋值 (\*\*=)</h1>

<p id="demo"></p>

<script>

let x = 5;

x \*\*= 2;

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

JavaScript Array.includes()

ECMAScript 2016 将 Array.prototype.includes 引入数组。这允许我们检查元素是否存在于数组中：

实例

<!DOCTYPE html>

<html>

<body>

<h1>Array includes()</h1>

<p>检查 fruit 数组是否包含 "Mango"：</p>

<p id="demo"></p>

<p><b>注释：</b>Edge 13（及更早版本）不支持 includes 方法。</p>

<script>

const fruits = ["Banana", "Orange", "Apple", "Mango"];

document.getElementById("demo").innerHTML = fruits.includes("Mango");

</script>

</body>

</html>

# ECMAScript 2017

JavaScript 命名约定始于 ES1、ES2、ES3、ES5 和 ES6。

但是，ECMAScript 2016 和 2017 未被称为 ES7 和 ES8。

自 2016 年以来，新版本按年份命名（ECMAScript 2016/2017/2018）。

ECMAScript 2017 中的新特性

本章介绍 ECMAScript 2017 的新特性：

JavaScript 字符串填充

JavaScript Object.entries

JavaScript Object.values

JavaScript 异步函数

JavaScript 共享内存

JavaScript 字符串填充

ECMAScript 2017 添加了两个 String 方法：padStart 和 padEnd，以支持在字符串的开头和结尾进行填充。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>padStart() 方法用另一段字符串来填充字符串：</p>

<p id="demo"></p>

<script>

let text = "5";

document.getElementById("demo").innerHTML = text.padStart(4,0);

</script>

</body>

</html>

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 字符串方法</h1>

<p>padEnd() 方法用另一段字符串填充字符串：</p>

<p id="demo"></p>

<script>

let text = "5";

document.getElementById("demo").innerHTML = text.padEnd(4,0);

</script>

</body>

</html>

JavaScript 对象条目

ECMAScript 2017 向对象添加了新的 Object.entries 方法。

Object.entries() 方法返回对象中键/值对的数组：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象方法</h1>

<p>Object.entries() 方法返回对象中键/值对的数组：</p>

<p id="demo"></p>

<script>

const person = {

firstName : "Bill",

lastName : "Gates",

age : 19,

eyeColor : "blue"

};

document.getElementById("demo").innerHTML = Object.entries(person);

</script>

</body>

</html>

Object.entries() 使循环中使用对象变简单了：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象方法</h1>

<p>Object.entries() 使得在循环中使用对象变得简单：</p>

<p id="demo"></p>

<script>

const fruits = {Bananas:300, Oranges:200, Apples:500};

let text = "";

for (let [fruit, amount] of Object.entries(fruits)) {

text += fruit + ": " + amount + "<br>";

}

document.getElementById("demo").innerHTML = text;

</script>

</body>

</html>

Object.entries() 也使得将对象转换为映射变得简单：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象方法</h1>

<p>Object.entries() 使将对象转换为 Map 变得简单：</p>

<p id="demo"></p>

<script>

const fruits = {Bananas:300, Oranges:200, Apples:500};

const myMap = new Map(Object.entries(fruits));

document.getElementById("demo").innerHTML = myMap;

</script>

</body>

</html>

JavaScript 对象值

Object.values 类似 Object.entries，但返回对象值的单维数组：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象方法</h1>

<p>Object.values() 方法从一个对象返回一个值数组：</p>

<p id="demo"></p>

<script>

const person = {

firstName : "Bill",

lastName : "Gates",

age : 19,

eyeColor : "blue"

};

document.getElementById("demo").innerHTML = Object.values(person);

</script>

</body>

</html>

JavaScript Async 函数

等待超时

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript async / await</h1>

<p>请等待 3 秒（3000 毫秒）让此页面发生变化。</p>

<h1 id="demo"></h1>

<script>

async function myDisplay() {

let myPromise = new Promise(function(myResolve, myReject) {

setTimeout(function() { myResolve("I love You !!"); }, 3000);

});

document.getElementById("demo").innerHTML = await myPromise;

}

myDisplay();

</script>

</body>

</html>

# 第67章 ECMAScript 2018

JavaScript 命名约定始于 ES1、ES2、ES3、ES5 和 ES6。

但是，ECMAScript 2016 和 2017 未被称为 ES7 和 ES8。

自 2016 年以来，新版本按年份命名（ECMAScript 2016/2017/2018）。

ECMAScript 2018 中的新特性

本章介绍 ECMAScript 2018 的新特性：

异步迭代

Promise Finally

对象 Rest 属性

新的 RegExp 特性

JavaScript 异步迭代

ECMAScript 2018 添加了异步迭代器和可迭代对象。

通过异步迭代，我们可以在 for/of 循环中使用 await 关键字。

实例

for await () {}

JavaScript Promise.finally

ECMAScript 2018 使用 Promise.finally 完成了 Promise 对象的完整实现：

实例

let myPromise = new Promise();

myPromise.then();

myPromise.catch();

myPromise.finally();

JavaScript 对象 Rest 属性

ECMAScript 2018 添加了 Rest 属性。

这允许我们破坏一个对象并将剩余物收集到一个新对象上：

实例

let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };

x; // 1

y; // 2

z; // { a: 3, b: 4 }

新的 JavaScript RegExp 特性

ECMAScript 2018 添加了 4 个新的 RegExp 特性：

Unicode 属性转义 (\p{...})

后行断言（Lookbehind Assertions） (?<= ) 和 (?<! )

命名捕获组（Named Capture Groups）

s (dotAll) 标志

# 第68章 Internet Explorer / Edge

Internet Explorer 退役

Windows 11 删除 Internet Explorer

2021 年 6 月 24 日报道：

Internet Explorer 在 Windows 11 中被禁用

来源：Windows 11 规范

Windows 10 取代 Internet Explorer

2020 年 6 月 15 日报道：

Internet Explorer 在 Windows 10 中被 Edge “取代”

来源 ...

Internet Explorer 退休

2020 年 6 月 15 日报道：

IE 11 将于 2022 年 6 月 15 日停止支持

来源 ...

退休日期：

30-11-2020 MS Teams 终止对 IE 11 的支持

31-12-2020 Azure DevOps Services 终止对 IE 11 的支持

31-03-2021 Azure 门户终止对 IE 11 的支持

17-08-2021 Microsoft 终止了对跨 Web 服务的 IE 11 的支持：

Office 365 - OneDrive Outlook

15-06-2022 微软终止对 IE 11 的所有支持

来源 ...

Microsoft Edge Legacy

Edge 是 Windows 10 的默认浏览器。

它是用微软的浏览器引擎 EdgeHTML 和他们的 Chakra JavaScript 引擎构建的。

Edge 的第一个版本 (12-18) 现在被称为 "Edge Legacy"。

Microsoft 对 Edge Legacy 的支持已于 2021 年 3 月 9 日结束。

未来的 Windows 更新将用 The New Edge 取代 Edge Legacy。

2020 年 8 月 17 日公布的退役日期：

2020 年 12 月 31 日 Azure DevOps 服务停止支持 Edge Legacy

2021 年 3 月 9 日 Edge Legacy 的 Windows 更新结束

2021 年 4 月 13 日 未来的 Windows 更新将删除 Edge Legacy

来源 ...

新的 Edge

新的 Microsoft Edge 基于 Chromium，带有 Blink 和 V8 引擎。

它于 2020 年 1 月发布，适用于 Windows 7、8 和 10。

也可以为 macOS、iOS 和 Android 下载它。

新 Edge 遵循现代生命周期策略（大约每六周更新一次）。

开发者支持？

IE 11 仍然是许多 Windows 操作系统的一个组件，并遵循安装它的产品的生命周期策略：

系统 默认浏览器

Windows Embedded Compact 7 Internet Explorer 7

Windows Embedded Compact 2013 Internet Explorer 7

Windows 7 Internet Explorer 11

Windows 8.X Internet Explorer 11

Windows 10 Internet Explorer 11

Windows Server 2012 Internet Explorer 11

Windows Server 2012 R2 Internet Explorer 11

Windows Server 2016 Internet Explorer 11

Windows Embedded Standard 7 Internet Explorer 11

Windows Embedded POSReady 7 Internet Explorer 11

Windows Thin PC Internet Explorer 11

Windows Embedded 8 Standard Internet Explorer 11

Windows 8.1 Industry Update Internet Explorer 11

较旧的应用程序

许多旧电脑（如公共图书馆）仍在使用 Internet Explorer。

许多旧版 Web 应用程序都在使用 Internet Explorer。

许多旧版 Web 应用程序都在使用 Internet Explorer runtime DLL。

许多基于 AJAX 的应用程序都使用 Microsoft ActiveX 组件。

为了简化从 Internet Explorer 的迁移，Microsoft Edge 提供 Internet Explorer 模式，提供向后兼容性并使客户能够继续运行旧的 Web 应用程序。

www.gov.uk //// designing-for-different-browsers-and-devices

# 第69章 JavaScript 历史

JavaScript / ECMAScript

JavaScript 是 Brendan Eich 在 1995 年发明的。

它是为 Netscape 2 开发的，并于 1997 年成为 ECMA-262 标准。

Netscape 将 JavaScript 交给 ECMA 后，Mozilla 基金会继续为 Firefox 浏览器开发 JavaScript。 Mozilla 的最新版本是 1.8.5.（与 ES5 相同）。

Internet Explorer (IE4) 是第一个支持 ECMA-262 Edition 1 (ES1) 的浏览器。

|  |  |  |
| --- | --- | --- |
| 年份 | ECMA | 浏览器 |
| 1995 年 |  | JavaScript 是由 Brendan Eich 发明 |
| 1996 年 |  | Netscape 2 与 JavaScript 1.0 一起发布 |
| 1997 年 |  | JavaScript 成为 ECMA 标准 (ECMA-262) |
| 1997 年 | ES1 | ECMAScript 1 发布 |
| 1997 年 | ES1 | IE 4 是第一个支持 ES1 的浏览器 |
| 1998 年 | ES2 | ECMAScript 2 发布 |
| 1998 年 |  | Netscape 42 与 JavaScript 1.3 一起发布 |
| 1999 年 | ES2 | IE 5 是第一个支持 ES2 的浏览器 |
| 1999 年 | ES3 | ECMAScript 3 发布 |
| 2000 年 | ES3 | IE 5.5 是第一个支持 ES3 的浏览器 |
| 2000 年 |  | Netscape 62 与 JavaScript 1.5 一起发布 |
| 2000 年 |  | Firefox 1 与 JavaScript 1.5 一起发布 |
| 2008 年 | ES4 | ECMAScript 4 被放弃 |
| 2009 年 | ES5 | ECMAScript 5 发布 |
| 2011 年 | ES5 | IE 9 是第一个支持 ES5 的浏览器 \* |
| 2011 年 |  | ES5 Firefox 4 与 JavaScript 1.8.5 一起发布 |
| 2012 年 | ES5 | Safari 6 完全支持 ES5 |
| 2012 年 | ES5 | IE 10 完全支持 ES5 |
| 2012 年 | ES5 | Chrome 23 完全支持 ES5 |
| 2013 年 | ES5 | Firefox 21 完全支持 ES5 |
| 2013 年 | ES5 | Opera 15 完全支持 ES5 |
| 2014年 | ES5 | 在所有浏览器中完全支持 ES5 |
| 2015 年 | ES6 | ECMAScript 6 发布 |
| 2016 年 | ES6 | Chrome 51 完全支持 ES6 |
| 2016 年 | ES6 | Opera 38 完全支持 ES6 |
| 2016 年 | ES6 | Edge 14 完全支持 ES6 |
| 2016 年 | ES6 | Safari 10 完全支持 ES6 |
| 2015 年 | ES6 | 在 Firefox 52 中完全支持 ES6 |
| 2018 年 | ES6 | 在浏览器中完全支持 ES6 |

ECMA 技术委员会 39

1996 年，Netscape 和 Brendan Eich 将 JavaScript 带到 ECMA 国际标准组织，并成立了一个技术委员会（TC39）来开发该语言。

ECMA-262 第 1 版于 1997 年 6 月发布。

从 ES4 到 ES6

当 TC39 委员会于 2008 年在奥斯陆聚集在一起就 ECMAScript 4 达成一致时，他们分成了两个截然不同的阵营：

ECMAScript 3.1 Camp：

微软和雅虎希望从 ES3 进行增量升级。

ECMAScript 4 Camp：

Adobe、Mozilla、Opera 和 Google，他们想要大规模的 ES4 升级。

2008 年 8 月 13 日，Brendan Eich 写了一封电子邮件：

JavaScript 标准机构 Ecma 技术委员会 39 已经分裂了一年多，这已经不是什么秘密了，一些成员支持 ES4，这是 ECMA-262 的主要第四版，而另一些成员则支持基于现有 ECMA-262 的 ES3.1 第 3 版 (ES3) 规范。现在，我很高兴地报告，分裂结束了。

解决方案是一起工作：

ECMAScript 4 重命名为 ES5

ES5 应该是 ECMAScript 3 的增量升级。

ECMAScript 4 的特性应该在以后的版本中被采用。

TC39 应该开发一个新的主要版本，范围比 ES5 更大。

计划中的新版本 (ES6) 代号为 "Harmony"（因为它造成了分裂？）。

ES5 取得了巨大的成功。它于 2009 年发布，到 2013 年 7 月，所有主要浏览器（包括 Internet Explorer）都完全兼容：

# 第70章 JavaScript 对象定义

在 JavaScript 中，对象是王。如果您理解了对象，就理解了 JavaScript。

在 JavaScript 中，几乎“所有事物”都是对象。

布尔是对象（如果用 new 关键词定义）

数字是对象（如果用 new 关键词定义）

字符串是对象（如果用 new 关键词定义）

日期永远都是对象

算术永远都是对象

正则表达式永远都是对象

数组永远都是对象

函数永远都是对象

对象永远都是对象

所有 JavaScript 值，除了原始值，都是对象。

JavaScript 原始值

原始值指的是没有属性或方法的值。

原始数据类型指的是拥有原始值的数据。

JavaScript 定义了 5 种原始数据类型：

string

number

boolean

null

undefined

原始值是一成不变的（它们是硬编码的，因此不能改变）。

假设 x = 3.14，您能够改变 x 的值。但是您无法改变 3.14 的值。

|  |  |  |
| --- | --- | --- |
| 值 | 类型 | 注释 |
| "Hello" | string | "Hello" 始终是 "Hello" |
| 3.14 | number | 3.14 始终是 3.14 |
| true | boolean | true 始终是 true |
| false | boolean | false 始终是 false |
| null | null | (object) null 始终是 null |
| undefined | undefined | undefined 始终是 undefined |

对象是包含变量的变量

JavaScript 变量能够包含单个的值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 变量</h1>

<p id="demo"></p>

<script>

// 创建并显示变量：

var person = "Bill Gates";

document.getElementById("demo").innerHTML = person;

</script>

</body>

</html>

对象也是变量。但是对象能够包含很多值。

值按照名称 : 值对的形式编写（名称和值以冒号分隔）。

实例

<!DOCTYPE html>

<html>

<body>

<h1>创建 JavaScript 对象</h1>

<p id="demo"></p>

<script>

var person = {

firstName : "Bill",

lastName : "Gates",

age : 62,

eyeColor : "blue"

};

document.getElementById("demo").innerHTML = person.firstName + " " + person.lastName;

</script>

</body>

</html>

JavaScript 对象是命名值的集合。

对象属性

JavaScript 对象中的命名值，被称为属性。

|  |  |
| --- | --- |
| 属性 | 值 |
| firstName | Bill |
| lastName | Gates |
| age | 62 |
| eyeColor | blue |

以名称值对书写的对象类似于：

PHP 中的关联数组

Python 中的字典

C 中的哈希表

Java 中的哈希映射

Ruby 和 Perl 中的散列

对象方法

方法是可以在对象上执行的动作。

对象属性可以是原始值、其他对象以及函数。

对象方法是包含函数定义的对象属性。

|  |  |
| --- | --- |
| 属性 | 值 |
| firstName | Bill |
| lastName | Gates |
| age | 62 |
| eyeColor | blue |
| fullName | function() {return this.firstName + " " + this.lastName;} |

JavaScript 对象是被称为属性和方法的命名值的容器。

您将在下一章中学到更多有关方法的知识。

创建 JavaScript 对象

通过 JavaScript，您能够定义和创建自己的对象。

有不同的方法来创建对象：

定义和创建单个对象，使用对象文字。

定义和创建单个对象，通过关键词 new。

定义对象构造器，然后创建构造类型的对象。

在 ECMAScript 5 中，也可以通过函数 Object.create() 来创建对象。

使用对象字面量

这是创建对象最简答的方法。

使用对象文字，您可以在一条语句中定义和创建对象。

对象文字指的是花括号 {} 中的名称:值对（比如 age:62）。

下面的例子创建带有四个属性的新的 JavaScript 对象：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象创建</h1>

<p id="demo"></p>

<script>

// 创建对象：

var person = {firstName:"Bill", lastName:"Gates", age:62, eyeColor:"blue"};

// 显示对象中的数据：

document.getElementById("demo").innerHTML =

person.firstName + " 已经 " + person.age + " 岁了。";

</script>

</body>

</html>

空格和折行不重要。对象定义可横跨多行：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象创建</h1>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName : "Bill",

lastName : "Gates",

age : 62,

eyeColor : "blue"

};

// 显示对象中的数据：

document.getElementById("demo").innerHTML =

person.firstName + " 已经 " + person.age + " 岁了。";

</script>

</body>

</html>

使用 JavaScript 关键词 new

下面的例子也创建了带有四个属性的新的 JavaScript 对象：

实例

<!DOCTYPE html>

<html>

<body>

<p id="demo"></p>

<script>

var person = new Object();

person.firstName = "Bill";

person.lastName = "Gates";

person.age = 50;

person.eyeColor = "blue";

document.getElementById("demo").innerHTML =

person.firstName + " is " + person.age + " years old.";

</script>

</body>

</html>

上面的两个例子结果是一样的。无需使用 new Object()。

出于简易性、可读性和执行速度的考虑，请使用第一种创建方法（对象文字方法）。

JavaScript 对象是易变的

对象是易变的：它们通过引用来寻址，而非值。

如果 person 是一个对象，下面的语句不会创建 person 的副本：

var x = person; // 这不会创建 person 的副本。

对象 x 并非 person 的副本。它就是 person。x 和 person 是同一个对象。

对 x 的任何改变都将改变 person，因为 x 和 person 是相同的对象。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象是易变的</h1>

<p>对对象副本的任何更改也将更改原始对象。</p>

<p id="demo"></p>

<script>

var person = {firstName:"Bill", lastName:"Gates", age:62, eyeColor:"blue"}

var x = person;

x.age = 10;

document.getElementById("demo").innerHTML =

person.firstName + " is " + person.age + " years old.";

</script>

</body>

</html>

注释：JavaScript 变量不是易变的。只有 JavaScript 对象如此。

# 第71章 JavaScript 对象属性

属性是任何 JavaScript 对象最重要的部分。

JavaScript 属性

属性指的是与 JavaScript 对象相关的值。

JavaScript 对象是无序属性的集合。

属性通常可以被修改、添加和删除，但是某些属性是只读的。

访问 JavaScript 属性

访问对象属性的语法是：

objectName.property // person.age

或者：

objectName["property"] // person["age"]

或者：

objectName[expression] // x = "age"; person[x]

表达式必须计算为属性名。

例子 1

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象属性</h1>

<p>访问对象属性有两种不同的方法：</p>

<p>您可以使用 .property 或 ["property"]。</p>

<p id="demo"></p>

<script>

var person = {

firstname:"Bill",

lastname:"Gates",

age:62,

eyecolor:"blue"

};

document.getElementById("demo").innerHTML = person.firstname + " is " + person.age + " years old.";

</script>

</body>

</html>

例子 2

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象属性</h1>

<p>访问对象属性有两种不同的方法：</p>

<p>您可以使用 .property 或 ["property"]。</p>

<p id="demo"></p>

<script>

var person = {

firstname:"Bill",

lastname:"Gates",

age:62,

eyecolor:"blue"

};

document.getElementById("demo").innerHTML = person["firstname"] + " is " + person["age"] + " years old.";

</script>

</body>

</html>

JavaScript for...in 循环

JavaScript for...in 语句遍历对象的属性。

语法

for (variable in object) {

要执行的代码

}

for...in 循环中的代码块会为每个属性执行一次。

循环对象的属性：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象属性</h1>

<p id="demo"></p>

<script>

var txt = "";

var person = {fname:"Bill", lname:"Gates", age:62};

var x;

for (x in person) {

txt += person[x] + " ";

}

document.getElementById("demo").innerHTML = txt;

</script>

</body>

</html>

添加新属性

您可以通过简单的赋值，向已存在的对象添加新属性。

假设 person 对象已存在 - 那么您可以为其添加新属性：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象属性</h1>

<p>您可以向现有对象添加新属性。</p>

<p id="demo"></p>

<script>

var person = {

firstname:"Bill",

lastname:"Gates",

age:62,

eyecolor:"blue"

};

person.nationality = "English";

document.getElementById("demo").innerHTML =

person.firstname + " is " + person.nationality + ".";

</script>

</body>

</html>

您不能使用预留词作为属性名（或方法名）。请使用 JavaScript 命名规则。

删除属性

delete 关键词从对象中删除属性：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象属性</h1>

<p>您可以删除对象属性。</p>

<p id="demo"></p>

<script>

var person = {

firstname:"Bill",

lastname:"Gates",

age:62,

eyecolor:"blue"

};

delete person.age;

document.getElementById("demo").innerHTML =

person.firstname + " is " + person.age + " years old.";

</script>

</body>

</html>

delete 关键词会同时删除属性的值和属性本身。

删除完成后，属性在被添加回来之前是无法使用的。

delete 操作符被设计用于对象属性。它对变量或函数没有影响。

delete 操作符不应被用于预定义的 JavaScript 对象属性。这样做会使应用程序崩溃。

属性值

所有属性都有名称。此外它们还有值。

值是属性的特性之一。

其他特性包括：可列举、可配置、可写。

这些特性定义了属性被访问的方式（是可读的还是可写的？）

在 JavaScript 中，所有属性都是可读的，但是只有值是可修改的（只有当属性为可写时）。

（ECMAScript 5 拥有获取和设置所有属性特性的方法）

原型属性

JavaScript 对象继承了它们的原型的属性。

delete 关键词不会删除被继承的属性，但是如果您删除了某个原型属性，则将影响到所有从原型继承的对象。

# 第72章 JavaScript 对象方法

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript <b>this</b> 关键词</h2>

<p>在本例中，<b>this</b> 代表 <b>person</b> 对象。</p>

<p>因为 person 对象“拥有” fullName 方法。</p>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

id : 678,

fullName : function() {

return this.firstName + " " + this.lastName;

}

};

// 显示来自对象的数据：

document.getElementById("demo").innerHTML = person.fullName();

</script>

</body>

</html>

JavaScript 方法

JavaScript 方法是能够在对象上执行的动作。

JavaScript 方法是包含函数定义的属性。

|  |  |
| --- | --- |
| 属性 | 值 |
| firstName | Bill |
| lastName | Gates |
| age | 62 |
| eyeColor | blue |
| fullName | function() {return this.firstName + " " + this.lastName;} |

方法是存储为对象属性的函数。

this 关键词

在 JavaScript 中，被称为 this 的事物，指的是拥有该 JavaScript 代码的对象。

this 的值，在函数中使用时，是“拥有”该函数的对象。

请注意 this 并非变量。它是关键词。您无法改变 this 的值。

访问对象方法

请使用如下语法创建对象方法：

methodName : function() { 代码行 }

请通过如下语法来访问对象方法：

objectName.methodName()

您通常会把 fullName() 描述为 person 对象的方法，把 fullName 描述为属性。

fullName 属性在被通过 () 调用后会以函数形式执行。

此例访问 person 对象的 fullName() 方法：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象方法</h1>

<p>对象方法是一种函数定义，存储为属性值。</p>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

id : 12345,

fullName : function() {

return this.firstName + " " + this.lastName;

}

};

// 显示对象中的数据：

document.getElementById("demo").innerHTML = person.fullName();

</script>

</body>

</html>

如果您访问 fullName 属性时没有使用 ()，则将返回函数定义：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象方法</h1>

<p>如果您不使用 () 访问对象，则返回函数定义：</p>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

id : 12345,

fullName : function() {

return this.firstName + " " + this.lastName;

}

};

// 显示对象中的数据：

document.getElementById("demo").innerHTML = person.fullName;

</script>

</body>

</html>

使用内建方法

此例使用 String 对象的 toUpperCase() 方法，把文本转换为大写：

var message = "Hello world!";

var x = message.toUpperCase();

x 的值，在以上代码执行后将是：

HELLO WORLD!

添加新的方法

向对象添加方法是在构造器函数内部完成的：

实例

function person(firstName, lastName, age, eyeColor) {

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

this.eyeColor = eyeColor;

this.changeName = function (name) {

this.lastName = name;

};

}

changeName() 函数 name 的值赋给了 person 的 lastName 属性。

现在您可以尝试：

<!DOCTYPE html>

<html>

<body>

<p id="demo"></p>

<script>

var person = {

firstName: "Bill",

lastName : "Gates",

id : 678,

};

person.name = function() {

return this.firstName + " " + this.lastName;

};

document.getElementById("demo").innerHTML =

"My friend is " + person.name();

</script>

</body>

</html>

通过用 myMother “替代” this，JavaScript 清楚您指的是哪个 person

# 第73章 JavaScript 显示对象

如何显示 JavaScript 对象？

显示 JavaScript 对象将输出 [object Object]。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象</h1>

<p>显示 JavaScript 对象将输出 [object Object]：</p>

<p id="demo"></p>

<script>

const person = {

name: "Bill",

age: 19,

city: "Seattle"

};

document.getElementById("demo").innerHTML = person;

</script>

</body>

</html>

显示 JavaScript 对象的一些常见解决方案是：

按名称显示对象属性

循环显示对象属性

使用 Object.values() 显示对象

使用 JSON.stringify() 显示对象

显示对象属性

对象的属性可显示为字符串：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象</h1>

<p>显示对象属性：</p>

<p id="demo"></p>

<script>

const person = {

name: "Bill",

age: 19,

city: "Seattle"

};

document.getElementById("demo").innerHTML = person.name + ", " + person.age + ", " + person.city;

</script>

</body>

</html>

在循环中显示对象

可以在循环中收集对象的属性：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象</h1>

<p>显示对象属性：</p>

<p id="demo"></p>

<script>

const person = {

name: "Bill",

age: 19,

city: "Seattle"

};

let txt = "";

for (let x in person) {

txt += person[x] + " ";

};

document.getElementById("demo").innerHTML = txt;

</script>

</body>

</html>

您必须在循环中使用 person[x]。

person.x 将不起作用（因为 x 是一个变量）。

使用 Object.values()

通过使用 Object.values()，任何 JavaScript 对象都可以被转换为数组：

const person = {

name: "Bill",

age: 19,

city: "Seattle"

};

const myArray = Object.values(person);

myArray 现在是 JavaScript 数组，可以显示了：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象</h1>

<p>Object.values() 将对象转换为数组。</p>

<p id="demo"></p>

<script>

const person = {

name: "Bill",

age: 19,

city: "Seattle"

};

document.getElementById("demo").innerHTML = Object.values(person);

</script>

</body>

</html>

使用 JSON.stringify()

任何 JavaScript 对象都可以使用 JavaScript 函数 JSON.stringify() 进行字符串化（转换为字符串）：

const person = {

name: "Bill",

age: 19,

city: "Seattle"

};

let myString = JSON.stringify(person);

myString 现在是 JavaScript 字符串，可以显示了：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象</h1>

<p>以 JSON 格式显示属性：</p>

<p id="demo"></p>

<script>

const person = {

name: "Bill",

age: 19,

city: "Seattle"

};

document.getElementById("demo").innerHTML = JSON.stringify(person);

</script>

</body>

</html>

结果将是一个遵循 JSON 标记法的字符串：

{"name":"Bill","age":19,"city":"Seattle"}

JSON.stringify() 包含在 JavaScript 中，所有主流浏览器都支持。

日期字符串化

JSON.stringify 将日期转换为字符串：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象</h1>

<p>JSON.stringify 将日期转换为字符串：</p>

<p id="demo"></p>

<script>

var person = {

name: "Bill",

today: new Date()

};

document.getElementById("demo").innerHTML = JSON.stringify(person);

</script>

</body>

</html>

函数字符串化

JSON.stringify 不会对函数进行字符串化：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象</h1>

<p>JSON.stringify 不会对函数进行字符串化：</p>

<p id="demo"></p>

<script>

const person = {

name: "Bill",

age: function () {return 19;}

};

document.getElementById("demo").innerHTML = JSON.stringify(person);

</script>

</body>

</html>

如果在字符串化之前将函数转换为字符串，这可以是“固定的”。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 显示对象</h1>

<p>JSON.stringify will not stringify functions.</p>

<p>You have to convert functions to strings first:</p>

<p id="demo"></p>

<script>

const person = {

name: "Bill",

age: function () {return 19;}

};

person.age = person.age.toString();

document.getElementById("demo").innerHTML = JSON.stringify(person);

</script>

</body>

</html>

数组字符串化

也可以对 JavaScript 数组进行字符串化：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 数组</h1>

<p>JSON.stringify 可以对数组进行字符串化：</p>

<p id="demo"></p>

<script>

const arr = ["Bill", "Steve", "Elon", "David"];

document.getElementById("demo").innerHTML = JSON.stringify(arr);

</script>

</body>

</html>

# JavaScript 对象访问器

JavaScript 访问器（Getter 和 Setter）

ECMAScript 5 (2009) 引入了 Getter 和 Setter。

Getter 和 Setter 允许您定义对象访问器（被计算的属性）。

JavaScript Getter（get 关键词）

本例使用 lang 属性来获取 language 属性的值。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Getter 和 Setter</h1>

<p>Getter 和 Setter 允许您通过方法获取和设置属性。</p>

<p>此示例使用 lang 属性来获取语言属性的值。</p>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

language : "en",

get lang() {

return this.language;

}

};

// 使用 getter 显示来自对象的数据：

document.getElementById("demo").innerHTML = person.lang;

</script>

</body>

</html>

JavaScript Setter（set 关键词）

本例使用 lang 属性来设置 language 属性的值。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Getter 和 Setter</h1>

<p>Getter 和 Setter 允许您通过方法获取和设置属性。</p>

<p>此示例使用 lang 属性设置语言属性的值。</p>

<p id="demo"></p>

<script>

// Create an object:

var person = {

firstName: "Bill",

lastName : "Gates",

language : "en",

set lang(value) {

this.language = value;

}

};

// 使用 set 设置属性：

person.lang = "zh";

// 显示对象的数据：

document.getElementById("demo").innerHTML = person.language;

</script>

</body>

</html>

JavaScript 函数还是 Getter？

下面两个例子的区别在哪里？

例子 1

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象方法</h1>

<p>可以使用存储为函数的属性来访问对象数据。</p>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

fullName : function() {

return this.firstName + " " + this.lastName;

}

};

// 使用方法显示对象的数据：

document.getElementById("demo").innerHTML = person.fullName();

</script>

</body>

</html>

例子 2

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象方法</h1>

<p>可以使用 getter 访问对象数据。</p>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

get fullName() {

return this.firstName + " " + this.lastName;

}

};

// 使用 getter 显示对象的数据：

document.getElementById("demo").innerHTML = person.fullName;

</script>

</body>

</html>

例子 1 以函数形式访问 fullName：person.fullName()。

例子 2 以属性形式访问 fullName：person.fullName。

第二个例子提供了更简洁的语法。

数据质量

使用 getter 和 setter 时，JavaScript 可以确保更好的数据质量。

在本例中，使用 lang 属性以大写形式返回 language 属性的值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Getter 和 Setter</h1>

<p>Getter 和 Setter 允许您通过方法获取和设置属性。</p>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

language : "en",

get lang() {

return this.language.toUpperCase();

}

};

// 使用 getter 显示对象的数据：

document.getElementById("demo").innerHTML = person.lang;

</script>

</body>

</html>

在本例中，使用 lang 属性将大写值存储在 language 属性中：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Getter 和 Setter</h1>

<p>Getter 和 Setter 允许您通过方法获取和设置属性。</p>

<p id="demo"></p>

<script>

// 创建对象：

var person = {

firstName: "Bill",

lastName : "Gates",

language : "",

set lang(lang) {

this.language = lang.toUpperCase();

}

};

// 使用 set 设置属性：

person.lang = "en";

// 显示对象的数据：

document.getElementById("demo").innerHTML = person.language;

</script>

</body>

</html>

为什么使用 Getter 和 Setter？

它提供了更简洁的语法

它允许属性和方法的语法相同

它可以确保更好的数据质量

有利于后台工作

一个计数器实例

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Getter 和 Setter</h1>

<p>创建完美的计数器对象：</p>

<p id="demo"></p>

<script>

var obj = {

?? counter : 0,

?? get reset() {

?????? this.counter = 0;

?? },

?? get increment() {

?????? this.counter++;

?? },

?? get decrement() {

?????? this.counter--;

?? },

?? set add(value) {

?????? this.counter += value;

?? },

?? set subtract(value) {

?????? this.counter -= value;

?? }

};

// 操作计数器：

obj.reset;

obj.add = 5;

obj.subtract = 1;

obj.increment;

obj.decrement;

// 显示计数器：

document.getElementById("demo").innerHTML = obj.counter;

</script>

</body>

</html>

Object.defineProperty() 方法也可用于添加 Getter 和 Setter：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Getter 和 Setter</h1>

<p>创建完美的计数器：</p>

<p id="demo"></p>

<script>

// 定义对象

var obj = {counter : 0};

// 定义 Setter 和 Getter

Object.defineProperty(obj, "reset", {

????get : function () {this.counter = 0;}

});

Object.defineProperty(obj, "increment", {

????get : function () {this.counter++;}

});

Object.defineProperty(obj, "decrement", {

????get : function () {this.counter--;}

});

Object.defineProperty(obj, "add", {

????set : function (value) {this.counter += value;}

});

Object.defineProperty(obj, "subtract", {

????set : function (value) {this.counter -= value;}

});

// 操作计数器：

obj.reset;

obj.add = 5;

obj.subtract = 1;

obj.increment;

obj.decrement;

document.getElementById("demo").innerHTML = obj.counter;

</script>

</body>

</html>

# 第75章 JavaScript 对象构造器

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象构造器</h1>

<p id="demo"></p>

<script>

// Person 对象的构造器函数

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

// 创建 Person 对象

var myFriend = new Person("Bill", "Gates", 62, "blue");

// 显示年龄

document.getElementById("demo").innerHTML =

"My friend is " + myFriend.age + ".";

</script>

</body>

</html>

用大写首字母对构造器函数命名是个好习惯。

对象类型（蓝图）（类）

前一章的实例是有限制的。它们只创建单一对象。

有时我们需要创建相同“类型”的许多对象的“蓝图”。

创建一种“对象类型”的方法，是使用对象构造器函数。

在上面的例子中，函数 Person() 就是对象构造器函数。

通过 new 关键词调用构造器函数可以创建相同类型的对象：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象构造器</h1>

<p id="demo"></p>

<script>

// Person 对象的构造器函数

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

// 创建 Person 对象

var myFriend = new Person("Bill", "Gates", 62, "blue");

var myBrother = new Person("Steve", "Jobs", 56, "green");

// 显示年龄

document.getElementById("demo").innerHTML =

"My friend is " + myFriend.age + ". My brother is " + myBrother.age + ".";

</script>

</body>

</html>

this 关键词

在 JavaScript 中，被称为 this 的事物是代码的“拥有者”。

this 的值，在对象中使用时，就是对象本身。

在构造器函数中，this 是没有值的。它是新对象的替代物。 当一个新对象被创建时，this 的值会成为这个新对象。

请注意 this 并不是变量。它是关键词。您无法改变 this 的值。

为对象添加属性

为已有的对象添加新属性很简单：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象构造器</h1>

<p id="demo"></p>

<script>

// Person 对象的构造器函数

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

// 创建两个 Person 对象

var myFriend = new Person("Bill", "Gates", 62, "blue");

var myBrother = new Person("Steve", "Jobs", 56, "green");

// 为第一个对象添加国籍

myFriend.nationality = "English";

// 显示国籍

document.getElementById("demo").innerHTML =

"My friend is " + myFriend.nationality;

</script>

</body>

</html>

新属性被添加到 myFather。不是 myMother，也不是任何其他 person 对象。

为对象添加方法

为已有的对象添加新方法很简单：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象构造器</h1>

<p id="demo"></p>

<script>

// Person 对象的构造器函数

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

// 创建两个 Person 对象

var myFriend = new Person("Bill", "Gates", 62, "blue");

var myBrother = new Person("Steve", "Jobs", 56, "green");

// 向第一个对象添加 name 方法

myFriend.name = function() {

return this.firstName + " " + this.lastName;

};

// 显示全名

document.getElementById("demo").innerHTML =

"My friend is " + myFriend.name();

</script>

</body>

</html>

新方法被添加到 myFather。不是 myMother，也不是任何其他 person 对象。

为构造器添加属性

与向已有对象添加新属性不同，您无法为对象构造器添加新属性：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象构造器</h1>

<p>您不能将新属性添加到构造函数。</p>

<p id="demo"></p>

<script>

// Person 对象的构造器函数

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

// 您不能将新属性添加到构造函数

Person.nationality = "English";

// 创建两个 Person 对象

var myFriend = new Person("Bill", "Gates", 62, "blue");

var myBrother = new Person("Steve", "Jobs", 56, "green");

// 显示国籍

document.getElementById("demo").innerHTML =

"The nationality of my friend is " + myFriend.nationality;

</script>

</body>

</html>

如需向构造器添加一个新属性，您必须添加到构造器函数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象构造器</h1>

<p id="demo"></p>

<script>

// Person 对象的构造器函数

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

this.nationality = "English";

}

// 创建两个 Person 对象

var myFriend = new Person("Bill", "Gates", 62, "blue");

var myBrother = new Person("Steve", "Jobs", 56, "green");

// 显示国籍

document.getElementById("demo").innerHTML =

"My friend is " + myFriend.nationality + ". My brother is " + myBrother.nationality;

</script>

</body>

</html>

这样对象属性就可以拥有默认值。

为构造器添加方法

您的构造器函数也可以定义方法：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象构造器</h1>

<p id="demo"></p>

<script>

// Person 对象的构造器函数

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

this.name = function() {

return this.firstName + " " + this.lastName

};

}

// 创建 Person 对象

var myFriend = new Person("Bill", "Gates", 62, "blue");

// 显示全名

document.getElementById("demo").innerHTML =

"My friend is " + myFriend.name();

</script>

</body>

</html>

与向已有对象添加新方法不同，您无法为对象构造器添加新方法。

必须在构造器函数内部向一个对象添加方法：

实例

function Person(firstName, lastName, age, eyeColor) {

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

this.eyeColor = eyeColor;

this.changeName = function (name) {

this.lastName = name;

};

}

changeName() 函数把 name 赋值给 person 的 lastName 属性。

现在您可以试一试：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象构造器</h1>

<p id="demo"></p>

<script>

// Person 对象的构造器函数

function Person(firstName,lastName,age,eyeColor) {

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

this.eyeColor = eyeColor;

this.changeName = function (name) {

this.lastName = name;

}

}

// 创建 Person 对象

var myFriend = new Person("Bill","Gates",62,"green");

// 修改姓氏

myFriend.changeName("Jobs");

// 显示姓氏

document.getElementById("demo").innerHTML =

"My friend's last name is " + myFriend.lastName;

</script>

</body>

</html>

通过用 myMother 替代 this，JavaScript 可以获知目前处理的哪个 person。

内建 JavaScript 构造器

JavaScript 提供用于原始对象的构造器：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象构造器</h1>

<p id="demo"></p>

<script>

var x1 = new Object(); // 新的 Object 对象

var x2 = new String(); // 新的 String 对象

var x3 = new Number(); // 新的 Number 对象

var x4 = new Boolean(); // 新的 Boolean 对象

var x5 = new Array(); // 新的 Array 对象

var x6 = new RegExp(); // 新的 RegExp 对象

var x7 = new Function(); // 新的 Function 对象

var x8 = new Date(); // 新的 Date 对象

// 显示所有对象的类型

document.getElementById("demo").innerHTML =

"x1: " + typeof x1 + "<br>" +

"x2: " + typeof x2 + "<br>" +

"x3: " + typeof x3 + "<br>" +

"x4: " + typeof x4 + "<br>" +

"x5: " + typeof x5 + "<br>" +

"x6: " + typeof x6 + "<br>" +

"x7: " + typeof x7 + "<br>" +

"x8: " + typeof x8 + "<br>";

</script>

<p>没有必要使用 String()、Number()、Boolean()、Array() 以及 RegExp()。</p>

</body>

</html>

Math() 对象不再此列。Math 是全局对象。new 关键词不可用于 Math。

您知道吗？

正如以上所见，JavaScript 提供原始数据类型字符串、数字和布尔的对象版本。但是并无理由创建复杂的对象。原始值快得多！

请使用对象字面量 {} 代替 new Object()。

请使用字符串字面量 "" 代替 new String()。

请使用数值字面量代替 Number()。

请使用布尔字面量代替 new Boolean()。

请使用数组字面量 [] 代替 new Array()。

请使用模式字面量代替 new RexExp()。

请使用函数表达式 () {} 代替 new Function()。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象构造器</h1>

<p id="demo"></p>

<script>

var x1 = {}; // 对象

var x2 = ""; // 字符串

var x3 = 0; // 数字

var x4 = false; // 布尔

var x5 = []; // 对象（非数组）

var x6 = /()/; // 对象

var x7 = function(){}; // 函数

// 显示所有类型

document.getElementById("demo").innerHTML =

"x1: " + typeof x1 + "<br>" +

"x2: " + typeof x2 + "<br>" +

"x3: " + typeof x3 + "<br>" +

"x4: " + typeof x4 + "<br>" +

"x5: " + typeof x5 + "<br>" +

"x6: " + typeof x6 + "<br>" +

"x7: " + typeof x7 + "<br>";

</script>

</body>

</html>

字符串对象

通常，字符串被创建为原始值：var firstName = "Bill"

但是也可以使用 new 关键词创建字符串对象：var firstName = new String("Bill")

请在 JS 字符串这一章中学习为何不应该把字符串创建为对象。

数字对象

通常，数值被创建为原始值：var x = 456

但是也可以使用 new 关键词创建数字对象：var x = new Number(456)

请在 JS 数字这一章中学习为何不应该把数值创建为对象。

布尔对象

通常，逻辑值被创建为原始值：var x = false

但是也可以使用 new 关键词创建逻辑对象：var x = new Boolean(false)

请在 JS 逻辑这一章中学习为何不应该把逻辑值创建为对象。

# 第76章 JavaScript 对象原型

所有 JavaScript 对象都从原型继承属性和方法。

在前一章里，我们学到了如何使用对象构造器：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象</h1>

<p id="demo"></p>

<script>

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

var myFriend = new Person("Bill", "Gates", 62, "blue");

var myBrother = new Person("Steve", "Jobs", 56, "green");

document.getElementById("demo").innerHTML =

"My friend is " + myFriend.age + ". My brother is " + myBrother.age;

</script>

</body>

</html>

我们已经认识到，您无法为已有的对象构造器添加新属性：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象</h1>

<p>您不能将新属性添加到构造函数。</p>

<p id="demo"></p>

<script>

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

Person.nationality = "English";

var myFriend = new Person("Bill", "Gates", 62, "blue");

var myBrother = new Person("Steve", "Jobs", 48, "green");

document.getElementById("demo").innerHTML =

"The nationality of my friend is " + myFriend.nationality;

</script>

</body>

</html>

如需向构造器添加一个新属性，则必须把它添加到构造器函数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象</h1>

<p id="demo"></p>

<script>

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

this.nationality = "English";

}

var myFriend = new Person("Bill", "Gates", 62, "blue");

var myBrother = new Person("Steve", "Jobs", 56, "green");

document.getElementById("demo").innerHTML =

"The nationality of my friend is " + myFriend.nationality + ". The nationality of my brother is " + myBrother.nationality;

</script>

</body>

</html>

原型继承

所有 JavaScript 对象都从原型继承属性和方法。

日期对象继承自 Date.prototype。数组对象继承自 Array.prototype。Person 对象继承自 Person.prototype。

Object.prototype 位于原型继承链的顶端：

日期对象、数组对象和 Person 对象都继承自 Object.prototype。

向对象添加属性和方法

有时，您希望向所有给定类型的已有对象添加新属性（或方法）。

有时，您希望向对象构造器添加新属性（或方法）。

使用 prototype 属性

JavaScript prototype 属性允许您为对象构造器添加新属性：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象</h1>

<p id="demo"></p>

<script>

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

Person.prototype.nationality = "English";

var myFriend = new Person("Bill", "Gates", 62, "blue");

document.getElementById("demo").innerHTML =

"The nationality of my friend is " + myFriend.nationality;

</script>

</body>

</html>

JavaScript prototype 属性也允许您为对象构造器添加新方法：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 对象</h1>

<p id="demo"></p>

<script>

function Person(first, last, age, eye) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eye;

}

Person.prototype.name = function() {

return this.firstName + " " + this.lastName

};

var myFriend = new Person("Bill", "Gates", 62, "blue");

document.getElementById("demo").innerHTML =

"My friend is " + myFriend.name();

</script>

</body>

</html>

# 第77章 JavaScript ES5 对象方法

ECMAScript 5 (2009) 向 JavaScript 添加了许多新的对象方法。

管理对象

// 以现有对象为原型创建对象

Object.create()

// 添加或更改对象属性

Object.defineProperty(object, property, descriptor)

// 添加或更改对象属性

Object.defineProperties(object, descriptors)

// 访问属性

Object.getOwnPropertyDescriptor(object, property)

// 以数组返回所有属性

Object.getOwnPropertyNames(object)

// 访问原型

Object.getPrototypeOf(object)

// 以数组返回可枚举属性

Object.keys(object)

保护对象

// 防止向对象添加属性

Object.preventExtensions(object)

// 如果属性可以添加到对象，则返回 true

Object.isExtensible(object)

// 防止更改对象属性（不是值）

Object.seal(object)

// 如果对象被密封，则返回 true

Object.isSealed(object)

// 防止对对象进行任何更改

Object.freeze(object)

// 如果对象被冻结，则返回 true

Object.isFrozen(object)

更改属性值

语法

Object.defineProperty(object, property, {value : value})

实例

此例更改属性值：

const person = {

firstName: "Bill",

lastName : "Gates",

language : "EN"

};

// 修改属性

Object.defineProperty(person, "language", {value : "NO"});

更改元数据

ES5 允许更改以下属性元数据：

writable : true // 属性值可更改

enumerable : true // 属性可枚举

configurable : true // 属性可重新配置

writable : false // 属性值不可更改

enumerable : false // 属性不可枚举

configurable : false // 属性不可重新配置

ES5 允许更改 getter 和 setter：

// 定义 getter

get: function() { return language }

// 定义 setter

set: function(value) { language = value }

此例设置 language 为只读：

Object.defineProperty(person, "language", {writable:false});

这个例子使 language 不可枚举：

Object.defineProperty(person, "language", {enumerable:false});

列出所有属性

这个例子列出了一个对象的所有属性：

实例

const person = {

firstName: "Bill",

lastName : "Gates",

language : "EN"

};

Object.defineProperty(person, "language", {enumerable:false});

Object.getOwnPropertyNames(person); // 返回数组的数组

列出可枚举的属性

此例仅列出对象的可枚举属性：

实例

const person = {

firstName: "Bill",

lastName : "Gates",

language : "EN"

};

Object.defineProperty(person, "language", {enumerable:false});

Object.keys(person); // 返回可枚举属性的数组

添加属性

此例向对象添加新属性：

实例

// 创建对象

const person = {

firstName: "Bill",

lastName : "Gates",

language : "EN"

};

// 添加属性

Object.defineProperty(person, "year", {value:"2008"});

添加 Getter 和 Setter

Object.defineProperty() 方法也可以用来添加 Getter 和 Setter：

实例

// 创建对象

const person = {firstName:"Bill", lastName:"Gates"};

// 定义 getter

Object.defineProperty(person, "fullName", {

get: function () {return this.firstName + " " + this.lastName;}

});

一个计数器实例

// 定义对象

const obj = {counter:0};

// 定义 setter

Object.defineProperty(obj, "reset", {

get : function () {this.counter = 0;}

});

Object.defineProperty(obj, "increment", {

get : function () {this.counter++;}

});

Object.defineProperty(obj, "decrement", {

get : function () {this.counter--;}

});

Object.defineProperty(obj, "add", {

set : function (value) {this.counter += value;}

});

Object.defineProperty(obj, "subtract", {

set : function (i) {this.counter -= i;}

});

// 操作计数器：

obj.reset;

obj.add = 5;

obj.subtract = 1;

obj.increment;

obj.decrement;

# 第78章 JavaScript Map 对象

Map 对象存有键值对，其中的键可以是任何数据类型。

Map 对象记得键的原始插入顺序。

Map 对象具有表示映射大小的属性。

基本的 Map() 方法

|  |  |
| --- | --- |
| Method | Description |
| new Map() | 创建新的 Map 对象。 |
| set() | 为 Map 对象中的键设置值。 |
| get() | 获取 Map 对象中键的值。 |
| entries() | 返回 Map 对象中键/值对的数组。 |
| keys() | 返回 Map 对象中键的数组。 |
| values() | 返回 Map 对象中值的数组。 |

Map() 属性

|  |  |
| --- | --- |
| Property | Description |
| size | 获取 Map 对象中某键的值。 |

创建 Map 对象

能够使用对象作为键是 Map 的一个重要特性。

实例

// 创建对象

const apples = {name: 'Apples'};

const bananas = {name: 'Bananas'};

const oranges = {name: 'Oranges'};

// 创建新的 Map

const fruits = new Map();

// Add new Elements to the Map

fruits.set(apples, 500);

fruits.set(bananas, 300);

fruits.set(oranges, 200);

您可以将 Array 传递给 new Map() 构造函数：

实例

// 返回

const apples = {name: 'Apples'};

const bananas = {name: 'Bananas'};

const oranges = {name: 'Oranges'};

// 创建新的 Map

const fruits = new Map([;

[apples, 500],

[bananas, 300],

[oranges, 200]

]);

获取键的值

get() 方法获取 Map 中键的值：

实例

fruits.get(apples); // 返回 500

Remember: The key is an object (apples), not a string ("apples"):

实例

fruits.get("apples"); // 返回 undefined

其他 Map() 方法

|  |  |  |
| --- | --- | --- |
| 方法 | 描述 |  |
| clear() | 删除 Map 中的所有元素。 |  |
| delete() | 删除由键指定的元素。 |  |
| has() | 如果键存在，则返回 true。 |  |
| forEach() | 为每个键/值对调用回调。 |  |

Map() 属性

|  |  |
| --- | --- |
| 属性 | 描述 |
| size | 获取 Map 中键的值。 |

Map.size

Map.size 返回 Map 中元素的数量：

实例

fruits.size;

Map.delete()

Map.delete() 删除 Map 元素：

实例

fruits.delete(apples);

Map.clear()

Map.clear() 从 Map 中移除所有元素：

实例

fruits.clear();

Map.has()

如果 Map 中存在键，则 Map.has() 返回 true：

实例

fruits.has(apples);

JavaScript 对象 vs Map

JavaScript 对象和 Map 之间的差异：

|  |  |  |
| --- | --- | --- |
|  | 对象 | Map |
| Size | 对象没有 size 属性 | Maps 有 size 属性 |
| 键类型 | 对象键必须是字符串（或符号） | Map 键可以是任何数据类型 |
| 键顺序 | 对象键没有很好地排序 | Map 键按插入排序 |
| 默认 | 对象有默认键 | Map 没有默认键 |

# 第79章 JavaScript Set 对象

Set 是唯一值的集合。

每个值在 Set 中只能出现一次。

一个 Set 可以容纳任何数据类型的任何值。

如何创建 Set

创建一个 Set 并添加现有变量：

实例

// 创建新的变量

const a = "a";

const b = "b";

const c = "c";

// 创建 Set

const letters = new Set();

// Add the values to the Set

letters.add(a);

letters.add(b);

letters.add(c);

创建 Set 并添加文字值：

实例

// 创建 Set

const letters = new Set();

// 向 Set 添加一些值

letters.add("a");

letters.add("b");

letters.add("c");

将 Array 传递给 new Set() 构造函数：

实例

// 创建新的 Set

const letters = new Set(["a","b","c"]);

对于 Set，typeof 返回对象：

typeof letters; // 返回 object。

对于 Set，instanceof Set 返回 true：

letters instanceof Set; // 返回 true

向 Set 添加元素

实例

letters.add("d");

letters.add("e");

如果您添加相等的元素，则只会保存第一个元素：

实例

letters.add("a");

letters.add("b");

letters.add("c");

letters.add("c");

letters.add("c");

letters.add("c");

letters.add("c");

letters.add("c");

Set 对象的方法和属性

|  |  |
| --- | --- |
| new Set() | 创建新的 Set 对象。 |
| add() | 向 Set 添加新元素。 |
| clear() | 从 Set 中删除所有元素。 |
| delete() | 删除由其值指定的元素。 |
| entries() | 返回 Set 对象中值的数组。 |
| has() | 如果值存在则返回 true。 |
| forEach() | 为每个元素调用回调。 |
| keys() | 返回 Set 对象中值的数组。 |
| values() | 与 keys() 相同。 |
| size | 返回元素计数。 |

# 第80章 JavaScript 函数定义

JavaScript 函数是通过 function 关键词定义的。

您可以使用函数声明或函数表达式。

函数声明

在本教程中稍早之前，您学到了通过如下语法声明函数：

function functionName(parameters) {

要执行的代码

}

被声明的函数不会直接执行。它们被“保存供稍后使用”，将在稍后执行，当它们被调用时。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>此例调用一个执行计算并返回结果的函数：</p>

<p id="demo"></p>

<script>

var x = myFunction(4, 3);

document.getElementById("demo").innerHTML = x;

function myFunction(a, b) {

return a \* b;

}

</script>

</body>

</html>

分号用于分隔可执行的 JavaScript 语句。

由于函数声明不是可执行的语句，以分号结尾并不常见。

函数表达式

JavaScript 函数也可以使用表达式来定义。

函数表达式可以在变量中存储：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>函数可以存储在变量中：</p>

<p id="demo"></p>

<script>

var x = function (a, b) {return a \* b};

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

在变量中保存函数表达式之后，此变量可用作函数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>函数存储在变量后，变量可以用作函数：</p>

<p id="demo"></p>

<script>

var x = function (a, b) {return a \* b};

document.getElementById("demo").innerHTML = x(4, 3);

</script>

</body>

</html>

上面的函数实际上是一个匿名函数（没有名称的函数）。

存放在变量中的函数不需要函数名。他们总是使用变量名调用。

上面的函数使用分号结尾，因为它是可执行语句的一部分。

Function() 构造器

正如您在之前的例子中看到的，JavaScript 函数是通过 function 关键词定义的。

函数也可以通过名为 Function() 的内建 JavaScript 函数构造器来定义。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>JavaScript 有一个内置的函数构造函数。</p>

<p id="demo"></p>

<script>

var myFunction = new Function("a", "b", "return a \* b");

document.getElementById("demo").innerHTML = myFunction(4, 3);

</script>

</body>

</html>

您实际上无需使用函数构造器。上面的例子这么写也是一样的：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p id="demo"></p>

<script>

var myFunction = function (a, b) {return a \* b}

document.getElementById("demo").innerHTML = myFunction(4, 3);

</script>

</body>

</html>

大多数情况下，您可以避免在 JavaScript 中使用 new 关键词。

函数提升

在本教程中稍早前，您已经学到了“提升”（hoisting）。

Hoisting 是 JavaScript 将声明移动到当前作用域顶端的默认行为。

Hoisting 应用于变量声明和函数声明。

正因如此，JavaScript 函数能够在声明之前被调用：

myFunction(5);

function myFunction(y) {

return y \* y;

}

使用表达式定义的函数不会被提升。

自调用函数

函数表达式可以作为“自调用”。

自调用表达式是自动被调用（开始）的，在不进行调用的情况下。

函数表达式会自动执行，假如表达式后面跟着 ()。

您无法对函数声明进行自调用。

您需要在函数周围添加括号，以指示它是一个函数表达式：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>可以在不调用的情况下自动调用函数：</p>

<p id="demo"></p>

<script>

(function () {

document.getElementById("demo").innerHTML = "Hello! I called myself";

})();

</script>

</body>

</html>

上面的函数实际上是一个匿名的自调用函数（没有名称的函数）。

函数可用作值

JavaScript 函数可被用作值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>函数可以视为值：</p>

<p>x = myFunction(4,3) or x = 12</p>

<p>在这两种情况下，x 都变为值为 12 的数字。</p>

<p id="demo"></p>

<script>

function myFunction(a, b) {

return a \* b;

}

var x = myFunction(4, 3);

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

JavaScript 函数可用在表达式中：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>函数可以在表达式中使用。</p>

<p id="demo"></p>

<script>

function myFunction(a, b) {

return a \* b;

}

var x = myFunction(4, 3) \* 2;

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

函数是对象

JavaScript 中的 typeof 运算符会为函数返回 "function"。

但是最好是把 JavaScript 函数描述为对象。

JavaScript 函数都有属性和方法。

arguments.length 会返回函数被调用时收到的参数数目：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>arguments.length 属性返回函数接收的参数数量：</p>

<p id="demo"></p>

<script>

function myFunction(a, b) {

return arguments.length;

}

document.getElementById("demo").innerHTML = myFunction(4, 3);

</script>

</body>

</html>

toString() 方法以字符串返回函数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>toString() 方法将函数作为字符串返回：</p>

<p id="demo"></p>

<script>

function myFunction(a, b) {

return a \* b;

}

document.getElementById("demo").innerHTML = myFunction.toString();

</script>

</body>

</html>

定义为对象属性的函数，被称为对象的方法。

为创建新对象而设计的函数，被称为对象构造函数（对象构造器）。

箭头函数

箭头函数允许使用简短的语法来编写函数表达式。

您不需要 function 关键字、return 关键字和花括号。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 箭头函数</h1>

<p>使用箭头函数，您不必键入 function 关键字、return 关键字和花括号。</p>

<p>IE11 或更早版本不支持箭头功能。</p>

<p id="demo"></p>

<script>

const x = (x, y) => x \* y;

document.getElementById("demo").innerHTML = x(5, 5);

</script>

</body>

</html>

箭头函数没有自己的 this。它们不适合定义对象方法。

箭头函数未被提升。它们必须在使用前进行定义。

使用 const 比使用 var 更安全，因为函数表达式始终是常量值。

如果函数是单个语句，则只能省略 return 关键字和大括号。因此，保留它们可能是一个好习惯：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 箭头函数</h1>

<p>IE11 或更早版本不支持箭头功能。</p>

<p id="demo"></p>

<script>

const x = (x, y) => { return x \* y };

document.getElementById("demo").innerHTML = x(5, 5);

</script>

</body>

</html>

# 第81章 JavaScript 函数参数

JavaScript 函数不会对参数值进行任何检查。

函数参数

在本教程中稍早的时间，您已经学到了函数可以拥有参数：

functionName(parameter1, parameter2, parameter3) {

要执行的代码

}

函数参数（parameter）指的是在函数定义中列出的名称。

函数参数（argument）指的是传递到函数或由函数接收到的真实值。

参数规则

JavaScript 函数定义不会为参数（parameter）规定数据类型。

JavaScript 函数不会对所传递的参数（argument）实行类型检查。

JavaScript 函数不会检查所接收参数（argument）的数量。

参数默认

如果调用参数时省略了参数（少于被声明的数量），则丢失的值被设置为：undefined。

有时这是可以接受的，但是有时最好给参数指定默认值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>将默认值设置为函数参数</h1>

<p id="demo"></p>

<script>

function myFunction(x, y) {

if (y === undefined) {

y = 0;

}

return x \* y;

}

document.getElementById("demo").innerHTML = myFunction(4);

</script>

</body>

</html>

如果函数调用的参数太多（超过声明），则可以使用 arguments 对象来达到这些参数。

arguments 对象

JavaScript 函数有一个名为 arguments 对象的内置对象。

arguments 对象包含函数调用时使用的参数数组。

这样，您就可以简单地使用函数来查找（例如）数字列表中的最高值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>找到最大的数字</h1>

<p id="demo"></p>

<script>

function findMax() {

var i;

var max = -Infinity;

for(i = 0; i < arguments.length; i++) {

if (arguments[i] > max) {

max = arguments[i];

}

}

return max;

}

document.getElementById("demo").innerHTML = findMax(4, 5, 6);

</script>

</body>

</html>

或创建一个函数来总和所有输入值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>所有参数的总和：</h1>

<p id="demo"></p>

<script>

function sumAll() {

var i;

var 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);

</script>

</body>

</html>

参数通过值传递

函数调用中的参数（parameter）是函数的参数（argument）。

JavaScript 参数通过值传递：函数只知道值，而不是参数的位置。

如果函数改变了参数的值，它不会改变参数的原始值。

参数的改变在函数之外是不可见的。

对象是由引用传递的

在 JavaScript 中，对象引用是值。

正因如此，对象的行为就像它们通过引用来传递：

如果函数改变了对象属性，它也改变了原始值。

对象属性的改变在函数之外是可见的。

# 第82章 JavaScript 函数调用

JavaScript 函数内部的代码会在“某物”调用它时执行。

调用 JavaScript 函数

在函数被定义时，函数内部的代码不会执行。

在函数被调用时，函数内部的代码会被执行。

调用函数通常也可以说“启动函数”或“执行函数”。

在本教程中，我们使用“调用”。

以函数形式调用函数

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>全局函数（myFunction）返回参数（a，b）的乘积：</p>

<p id="demo"></p>

<script>

function myFunction(a, b) {

return a \* b;

}

document.getElementById("demo").innerHTML = myFunction(10, 2);

</script>

</body>

</html>

以上函数不属于任何对象。但是在 JavaScript 中，始终存在一种默认的全局对象。

在 HTML 中，默认全局对象是 HTML 页面本身，所有上面的函数“属于”HTML 页面。

在浏览器中，这个页面对象就是浏览器窗口。上面的函数自动成为一个窗口函数。

myFunction() 和 window.myFunction() 是同一个函数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>全局函数自动成为 window 方法。调用 myFunction() 与调用 window.myFunction() 相同。</p>

<p id="demo"></p>

<script>

function myFunction(a, b) {

return a \* b;

}

document.getElementById("demo").innerHTML = window.myFunction(10, 2);

</script>

</body>

</html>

这是调用函数的常见方法，但并不是一个好习惯。

全局变量、方法或函数很容易在全局对象中产生命名冲突和漏洞。

this 关键词

在 JavaScript 中，被称为 this 的事物，指的是“拥有”当前代码的对象。

this 的值，在函数中使用时，是“拥有”该函数的对象。

请注意 this 并不是变量。它属于关键词。您无法改变 this 的值。

全局对象

当不带拥有者对象调用对象时，this 的值成为全局对象。

在 web 浏览器中，全局对象就是浏览器对象。

本例以 this 的值返回这个 window 对象：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>在 HTML 中，全局函数中 <b>this </b> 的值是 window 对象。</p>

<p id="demo"></p>

<script>

var x = myFunction();

function myFunction() {

return this;

}

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

调用一个函数作为一个全局函数，会导致 this 的值成为全局对象。

作为变量来使用 window 对象很容易使程序崩溃。

作为方法来调用函数

在 JavaScript 中，您可以把函数定义为对象方法。

下面的例子创建了一个对象（myObject），带有两个属性（firstName 和 lastName），以及一个方法（fullName）：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>myObject.fullName() 将返回 Bill Gates：</p>

<p id="demo"></p>

<script>

var myObject = {

firstName:"Bill",

lastName: "Gates",

fullName: function() {

return this.firstName + " " + this.lastName;

}

}

document.getElementById("demo").innerHTML = myObject.fullName();

</script>

</body>

</html>

fullName 方法是一个函数。该函数属于对象。myObject 是函数的拥有者。

被称为 this 的事物，是“拥有”这段 JavaScript 代码的对象。在此例中，this 的值是 myObject。

测试一下！修改这个 fullName 方法来返回 this 的值：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>在对象方法中，<b>this</b> 的值是所有者对象。</p>

<p id="demo"></p>

<script>

var myObject = {

firstName:"Bill",

lastName: "Gates",

fullName: function() {

return this;

}

}

document.getElementById("demo").innerHTML = myObject.fullName();

</script>

</body>

</html>

以对象方法来调用函数，会导致 this 的值成为对象本身。

通过函数构造器来调用函数

如果函数调用的前面是 new 关键字，那么这是一个构造函数调用。

它看起来像你创建一个新的函数，但由于 JavaScript 函数是对象，你实际上创建一个新对象：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>在这个例子中，myFunction 是一个函数构造函数：</p>

<p id="demo"></p>

<script>

function myFunction(arg1, arg2) {

this.firstName = arg1;

this.lastName = arg2;

}

var x = new myFunction("Bill","Gates")

document.getElementById("demo").innerHTML = x.firstName;

</script>

</body>

</html>

构造器调用会创建新对象。新对象会从其构造器继承属性和方法。

构造器内的 this 关键词没有值。

this 的值会成为调用函数时创建的新对象。

# 第83章 JavaScript 函数 Call

方法重用

使用 call() 方法，您可以编写能够在不同对象上使用的方法。

函数是对象方法

在 JavaScript 中，函数是对象的方法。

如果一个函数不是 JavaScript 对象的方法，那么它就是全局对象的函数（参见前一章）。

下面的例子创建了带有三个属性的对象（firstName、lastName、fullName）。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>此例创建一个具有三个属性（firstName，lastName，fullName）的对象。</p>

<p>fullName 属性是一个方法：</p>

<p id="demo"></p>

<script>

var myObject = {

firstName:"Bill",

lastName: "Gates",

fullName: function() {

return this.firstName + " " + this.lastName;

}

}

x = myObject.fullName();

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

fullName 属性是一个方法。person 对象是该方法的拥有者。

fullName 属性属于 person 对象的方法。

JavaScript call() 方法

call() 方法是预定义的 JavaScript 方法。

它可以用来调用所有者对象作为参数的方法。

通过 call()，您能够使用属于另一个对象的方法。

本例调用 person 的 fullName 方法，并用于 person1：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>此例调用 person 的 fullName 方法，在 person1 上使用它：</p>

<p id="demo"></p>

<script>

var person = {

fullName: function() {

return this.firstName + " " + this.lastName;

}

}

var person1 = {

firstName:"Bill",

lastName: "Gates"

}

var person2 = {

firstName:"Steve",

lastName: "Jobs"

}

var x = person.fullName.call(person1);

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

本例调用 person 的 fullName 方法，并用于 person2：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>此示例调用 person 的 fullName 方法，在 person2 上使用它：</p>

<p id="demo"></p>

<script>

var person = {

fullName: function() {

return this.firstName + " " + this.lastName;

}

}

var person1 = {

firstName:"Bill",

lastName: "Gates"

}

var person2 = {

firstName:"Steve",

lastName: "Jobs"

}

x = person.fullName.call(person2);

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

带参数的 call() 方法

call() 方法可接受参数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>此例调用 person 的 fullName 方法，在 person1 上使用它：</p>

<p id="demo"></p>

<script>

var person = {

fullName: function(city, country) {

return this.firstName + " " + this.lastName + "," + city + "," + country;

}

}

var person1 = {

firstName:"Bill",

lastName: "Gates"

}

var person2 = {

firstName:"Steve",

lastName: "Jobs"

}

var x = person.fullName.call(person1, "Seatle", "USA");

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

# 第84章 JavaScript 函数 Apply

方法重用

通过 apply() 方法，您能够编写用于不同对象的方法。

JavaScript apply() 方法

apply() 方法与 call() 方法非常相似：

在本例中，person 的 fullName 方法被应用到 person1：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>在此示例中，person 的 fulllName 方法在 person1 上<b>应用</b>：</p>

<p id="demo"></p>

<script>

var person = {

fullName: function() {

return this.firstName + " " + this.lastName;

}

}

var person1 = {

firstName:"Bill",

lastName: "Gates"

}

var x = person.fullName.apply(person1);

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

call() 和 apply() 之间的区别

不同之处是：

call() 方法分别接受参数。

apply() 方法接受数组形式的参数。

如果要使用数组而不是参数列表，则 apply() 方法非常方便。

带参数的 apply() 方法

apply() 方法接受数组中的参数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>在此示例中，person 的 fulllName 方法在 person1 上<b>应用</b>：</p>

<p id="demo"></p>

<script>

var person = {

fullName: function(city, country) {

return this.firstName + " " + this.lastName + "," + city + "," + country;

}

}

var person1 = {

firstName:"Bill",

lastName: "Gates"

}

var x = person.fullName.apply(person1, ["Seatle", "USA"]);

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

与 call() 方法对比：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数</h1>

<p>此例调用 person 的 fullName 方法，在 person1 上使用它：</p>

<p id="demo"></p>

<script>

var person = {

fullName: function(city, country) {

return this.firstName + " " + this.lastName + "," + city + "," + country;

}

}

var person1 = {

firstName:"Bill",

lastName: "Gates"

}

var person2 = {

firstName:"Steve",

lastName: "Jobs"

}

var x = person.fullName.call(person1, "Seatle", "USA");

document.getElementById("demo").innerHTML = x;

</script>

</body>

</html>

在数组上模拟 max 方法

您可以使用 Math.max() 方法找到（数字列表中的）最大数字：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Math.max()</h1>

<p>此例返回数字参数列表中的最大数字：</p>

<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML = Math.max(1,2,3);

</script>

</body>

</html>

由于 JavaScript 数组没有 max() 方法，因此您可以应用 Math.max() 方法。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript apply()</h1>

<p>此例返回数字数组中的最大数：</p>

<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML = Math.max.apply(null, [1,2,3]);

</script>

</body>

</html>

第一个参数（null）无关紧要。在本例中未使用它。

这些例子会给出相同的结果：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript apply()</h1>

<p>此例返回数字数组中的最大数：</p>

<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML = Math.max.apply(Math, [1,2,3]);

</script>

</body>

</html>

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript apply()</h1>

<p>此例返回数字数组中的最大数：</p>

<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML = Math.max.apply(" ", [1,2,3]);

</script>

</body>

</html>

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript apply()</h1>

<p>此例返回数字数组中的最大数：</p>

<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML = Math.max.apply(0, [1,2,3]);

</script>

</body>

</html>

JavaScript 严格模式

在 JavaScript 严格模式下，如果 apply() 方法的第一个参数不是对象，则它将成为被调用函数的所有者（对象）。在“非严格”模式下，它成为全局对象。

# 第85章 JavaScript 闭包

JavaScript 变量属于本地或全局作用域。

全局变量能够通过闭包实现局部（私有）。

全局变量

函数能够访问函数内部定义的所有变量，比如：

实例

<!DOCTYPE html>

<html>

<body>

<p>函数可以访问函数内定义的变量：</p>

<button type="button" onclick="myFunction()">单击这里！</button>

<p id="demo"></p>

<script>

function myFunction() {

var a = 4;

document.getElementById("demo").innerHTML = a \* a;

}

</script>

</body>

</html>

但是函数也能访问函数外部定义的变量，比如：

实例

<!DOCTYPE html>

<html>

<body>

<p>函数可以访问函数外部定义的变量：</p>

<button type="button" onclick="myFunction()">单击这里！</button>

<p id="demo"></p>

<script>

var a = 4;

function myFunction() {

document.getElementById("demo").innerHTML = a \* a;

}

</script>

</body>

</html>

在最后这个例子中，a 是全局变量。

在网页中，全局变量属于 window 对象。

全局变量能够被页面中（以及窗口中）的所有脚本使用和修改。

在第一个例子中，a 是局部变量。

局部变量只能用于其被定义的函数内部。对于其他函数和脚本代码来说它是不可见的。

拥有相同名称的全局变量和局部变量是不同的变量。修改一个，不会改变其他。

不通过关键词 var 创建的变量总是全局的，即使它们在函数中创建。

变量的生命周期

全局变量活得和您的应用程序（窗口、网页）一样久。

局部变量活得不长。它们在函数调用时创建，在函数完成后被删除。

一个计数器的困境

假设您想使用变量来计数，并且您希望此计数器可用于所有函数。

您可以使用全局变量和函数来递增计数器：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数闭包</h1>

<p>使用全局变量计数。</p>

<p id="demo"></p>

<script>

// 初始化计数器

var counter = 0;

// 递增计数器的函数

function add() {

counter += 1;

}

// 调用三次 add()

add();

add();

add();

// 计数器目前应该是 3

document.getElementById("demo").innerHTML = "计数器是：" + counter;

</script>

</body>

</html>

上述解决方案有一个问题：页面上的任何代码都可以更改计数器，而无需调用 add()。

对于 add() 函数，计数器应该是局部的，以防止其他代码更改它：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数闭包</h1>

<p>使用局部变量计数。</p>

<p id="demo"></p>

<script>

// 初始化计数器

var counter = 0;

// 递增计数器的函数

function add() {

var counter = 0;

counter += 1;

}

// 调用三次 add()

add();

add();

add();

// 结果不是 3，因为您混用了全局本地计数器

document.getElementById("demo").innerHTML = "计数器是：" + counter;

</script>

</body>

</html>

它没有用，因为我们显示全局计数器而不是本地计数器。

通过让函数返回它，我们可以删除全局计数器并访问本地计数器：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 闭包</h1>

<p>使用局部变量计数。</p>

<button type="button" onclick="myFunction()">计数！</button>

<p id="demo">0</p>

<script>

// 递增计数器的函数

function add() {

var counter = 0;

counter += 1;

return counter;

}

// 试图递增计数器

function myFunction(){

document.getElementById("demo").innerHTML = add();

}

</script>

</body>

</html>

它没有用，因为我们每次调用函数时都会重置本地计数器。

JavaScript 内部函数可以解决这个问题。

JavaScript 嵌套函数

所有函数都有权访问全局作用域。

事实上，在 JavaScript 中，所有函数都有权访问它们“上面”的作用域。

JavaScript 支持嵌套函数。嵌套函数可以访问其上的作用域。

在本例中，内部函数 plus() 可以访问父函数中的 counter 计数器变量：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 函数闭包</h1>

<p>使用局部变量计数。</p>

<p id="demo">0</p>

<script>

document.getElementById("demo").innerHTML = add();

function add() {

var counter = 0;

function plus() {counter += 1;}

plus();

return counter;

}

</script>

</body>

</html>

这样即可解决计数器困境，如果我们能够从外面访问 plus() 函数。

我们还需要找到只执行一次 counter = 0 的方法。

我们需要闭包（closure）。

JavaScript 闭包

记得自调用函数吗？这种函数会做什么呢？

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 闭包</h1>

<p>使用局部变量计数。</p>

<button type="button" onclick="myFunction()">计数！</button>

<p id="demo">0</p>

<script>

var add = (function () {

var counter = 0;

return function () {counter += 1; return counter;}

})();

function myFunction(){

document.getElementById("demo").innerHTML = add();

}

</script>

</body>

</html>

例子解释

变量 add 的赋值是自调用函数的返回值。

这个自调用函数只运行一次。它设置计数器为零（0），并返回函数表达式。

这样 add 成为了函数。最“精彩的”部分是它能够访问父作用域中的计数器。

这被称为 JavaScript 闭包。它使函数拥有“私有”变量成为可能。

计数器被这个匿名函数的作用域保护，并且只能使用 add 函数来修改。

闭包指的是有权访问父作用域的函数，即使在父函数关闭之后。

# 第86章 JavaScript 类

ECMAScript 2015，也称为 ES6，引入了 JavaScript 类。

JavaScript 类是 JavaScript 对象的模板。

JavaScript 类的语法

请使用关键字 class 创建一个类。

请始终添加一个名为 constructor() 的方法：

语法

class ClassName {

constructor() { ... }

}

实例

class Car {

constructor(name, year) {

this.name = name;

this.year = year;

}

}

上面的例子创建了一个名为 "Car" 的类。

该类有两个初始属性："name" 和 "year"。

JavaScript 类不是对象。

它是 JavaScript 对象的模板。

使用类

当您有了一个类时，可以使用该类来创建对象：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 类</h1>

<p>如何使用 JavaScript 类。</p>

<p id="demo"></p>

<script>

class Car {

constructor(name, year) {

this.name = name;

this.year = year;

}

}

const myCar = new Car("Ford", 2014);

document.getElementById("demo").innerHTML =

myCar.name + " " + myCar.year;

</script>

</body>

</html>

上面的例子使用 Car 类来创建两个 Car 对象。

在创建新对象时会自动调用构造方法（constructor method）。

构造方法

构造方法是一种特殊的方法：

它必须有确切的名称的 “constructor”

创建新对象时自动执行

用于初始化对象属性

如果您没有定义构造方法，JavaScript 会添加一个空的构造方法。

类方法

创建类方法的语法与对象方法相同。

请使用关键字 class 创建一个类。

请始终添加 constructor() 方法。

然后添加任意数量的方法。

语法

class ClassName {

constructor() { ... }

method\_1() { ... }

method\_2() { ... }

method\_3() { ... }

}

创建一个名为 "age" 的类方法，它返回车年：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 类方法</h1>

<p>如何定义和使用类方法。</p>

<p id="demo"></p>

<script>

class Car {

constructor(name, year) {

this.name = name;

this.year = year;

}

age() {

let date = new Date();

return date.getFullYear() - this.year;

}

}

let myCar = new Car("Ford", 2014);

document.getElementById("demo").innerHTML =

"My car is " + myCar.age() + " years old.";

</script>

</body>

</html>

您可以向类方法发送参数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 类方法</h1>

<p>将参数传递给 "age()" 方法。</p>

<p id="demo"></p>

<script>

class Car {

constructor(name, year) {

this.name = name;

this.year = year;

}

age(x) {

return x - this.year;

}

}

let date = new Date();

let year = date.getFullYear();

let myCar = new Car("Ford", 2014);

document.getElementById("demo").innerHTML=

"My car is " + myCar.age(year) + " years old.";

</script>

</body>

</html>

"use strict"

类中的语法必须以“严格模式”编写。

如果您不遵循“严格模式”规则，将收到错误消息。

实例

在“严格模式”下，如果您使用变量而不声明它，会得到错误：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 类是以“严格模式”编写的</h1>

<p>在 JavaScript 类中，您不能在未声明的情况下使用变量。</p>

<p id="demo"></p>

<script>

class Car {

constructor(name, year) {

this.name = name;

this.year = year;

}

age() {

// date = new Date(); // This will not work

let date = new Date(); // This will work

return date.getFullYear() - this.year;

}

}

myCar = new Car("Ford", 2014);

document.getElementById("demo").innerHTML =

"My car is " + myCar.age() + " years old.";

</script>

</body>

</html>

# 第87章 JavaScript 类继承

类继承

如需创建类继承，请使用 extends 关键字。

使用类继承创建的类继承了另一个类的所有方法：

实例

创建一个名为 "Model" 的类，它将继承 "Car" 类的方法：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Class 继承</h1>

<p>请使用 "extends" 关键字从另一个类继承所有方法。</p>

<p>使用 "super" 方法调用父级的构造函数。</p>

<p id="demo"></p>

<script>

class Car {

constructor(brand) {

this.carname = brand;

}

present() {

return 'I have a ' + this.carname;

}

}

class Model extends Car {

constructor(brand, mod) {

super(brand);

this.model = mod;

}

show() {

return this.present() + ', it is a ' + this.model;

}

}

let myCar = new Model("Ford", "Mustang");

document.getElementById("demo").innerHTML = myCar.show();

</script>

</body>

</html>

super() 方法引用父类。

通过在 constructor 方法中调用 super() 方法，我们调用了父级的 constructor 方法，获得了父级的属性和方法的访问权限。

继承对于代码可重用性很有用：在创建新类时重用现有类的属性和方法。

Getter 和 Setter

类还允许您使用 getter 和 setter。

为您的属性使用 getter 和 setter 很聪明，特别是如果您想在返回它们之前或在设置它们之前对值做一些特殊的事情。

如需在类中添加 getter 和 setter，请使用 get 和 set 关键字。

实例

为 "carname" 属性创建 getter 和 setter：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Class Gettter/Setter</h1>

<p>演示如何在类中添加 getter 和 setter，以及如何使用 getter 获取属性值。</p>

<p id="demo"></p>

<script>

class Car {

constructor(brand) {

this.carname = brand;

}

get cnam() {

return this.carname;

}

set cnam(x) {

this.carname = x;

}

}

let myCar = new Car("Ford");

document.getElementById("demo").innerHTML = myCar.cnam;

</script>

</body>

</html>

注释：即使 getter 是一个方法，当你想要获取属性值时也不要使用括号。

getter/setter 方法的名称不能与属性名称相同，在本例中为 carname。

许多程序员在属性名称前使用下划线字符 \_ 将 getter/setter 与实际属性分开：

实例

您可以使用下划线字符将 getter/setter 与实际属性分开：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Class Gettter/Setter</h1>

<p>在 JavaScript 中使用 getter/setter 时，使用下划线字符是常见做法，但不是强制性的，您可以随意命名它们，但不能与属性名称相同。</p>

<p id="demo"></p>

<script>

class Car {

constructor(brand) {

this.\_carname = brand;

}

get carname() {

return this.\_carname;

}

set carname(x) {

this.\_carname = x;

}

}

let myCar = new Car("Ford");

document.getElementById("demo").innerHTML = myCar.carname;

</script>

</body>

</html>

如需使用 setter，请使用与设置属性值相同的语法，不带括号：

实例

使用 setter 将汽车名称更改为 "Volvo"：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Class Setter</h1>

<p>使用 setter 设置属性值时，请不要使用括号。</p>

<p id="demo"></p>

<script>

class Car {

constructor(brand) {

this.\_carname = brand;

}

set carname(x) {

this.\_carname = x;

}

get carname() {

return this.\_carname;

}

}

let myCar = new Car("Ford");

myCar.carname = "Volvo";

document.getElementById("demo").innerHTML = myCar.carname;

</script>

</body>

</html>

Hoisting

与函数和其他 JavaScript 声明不同，类声明不会被提升。

这意味着您必须先声明类，然后才能使用它：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 类没有被提升</h1>

<p>如果在声明类之前尝试使用类，则会出现错误。</p>

<p id="demo"></p>

<script>

//You cannot use the class yet.

//myCar = new Car("Ford")

//This would raise an error.

class Car {

constructor(brand) {

this.carname = brand;

}

}

//Now you can use the class:

let myCar = new Car("Ford")

</script>

</body>

</html>

注释：对于其他声明，如函数，在声明之前尝试使用它时不会出错，因为 JavaScript 声明的默认行为是提升（将声明移到顶部）。

# 第88章 JavaScript Static 方法

static 类方法是在类本身上定义的。

您不能在对象上调用 static 方法，只能在对象类上调用。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Class Static 方法</h1>

<p>static 方法是使用 "static" 关键字创建的，您只能在类本身上调用该方法。</p>

<p id="demo"></p>

<script>

class Car {

constructor(name) {

this.name = name;

}

static hello() {

return "Hello!!";

}

}

let myCar = new Car("Ford");

//You can call 'hello()' on the Car Class:

document.getElementById("demo").innerHTML = Car.hello();

// But NOT on a Car Object:

// document.getElementById("demo").innerHTML = myCar.hello();

// this will raise an error.

</script>

</body>

</html>

如果要在 static 方法中使用 myCar 对象，可以将其作为参数发送：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Class Static 方法</h1>

<p>如需在静态方法中使用 "myCar" 对象，您可以将其作为参数发送。</p>

<p id="demo"></p>

<script>

class Car {

constructor(name) {

this.name = name;

}

static hello(x) {

return "Hello " + x.name;

}

}

let myCar = new Car("Ford");

document.getElementById("demo").innerHTML = Car.hello(myCar);

</script>

</body>

</html>

# 第89章 JavaScript 回调

函数序列

JavaScript 函数按照它们被调用的顺序执行。而不是以它们被定义的顺序。

此例最终将显示 "Goodbye"：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 函数序列</h2>

<p>JavaScript 函数按照它们被调用的顺序执行。</p>

<p id="demo"></p>

<script>

function myDisplayer(some) {

document.getElementById("demo").innerHTML = some;

}

function myFirst() {

myDisplayer("Hello");

}

function mySecond() {

myDisplayer("Goodbye");

}

myFirst();

mySecond();

</script>

</body>

</html>

本例子最终会显示 "Hello"：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 函数序列</h2>

<p>JavaScript 函数按照它们被调用的顺序执行。</p>

<p id="demo"></p>

<script>

function myDisplayer(some) {

document.getElementById("demo").innerHTML = some;

}

function myFirst() {

myDisplayer("Hello");

}

function mySecond() {

myDisplayer("Goodbye");

}

mySecond();

myFirst();

</script>

</body>

</html>

顺序控制

有时您希望更好地控制何时执行函数。

假设您要进行计算，然后显示结果。

您可以调用计算器函数 (myCalculator)，保存结果，然后调用另一个函数 (myDisplayer) 来显示结果：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 函数</h2>

<p>进行计算，然后显示结果。</p>

<p id="demo"></p>

<script>

function myDisplayer(some) {

document.getElementById("demo").innerHTML = some;

}

function myCalculator(num1, num2) {

let sum = num1 + num2;

return sum;

}

let result = myCalculator(5, 5);

myDisplayer(result);

</script>

</body>

</html>

或者，您可以调用计算器函数 (myCalculator)，并让计算器函数调用显示函数 (myDisplayer)：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript 回调</h2>

<p>进行计算，然后显示结果。</p>

<p id="demo"></p>

<script>

function myDisplayer(some) {

document.getElementById("demo").innerHTML = some;

}

function myCalculator(num1, num2) {

let sum = num1 + num2;

myDisplayer(sum);

}

myCalculator(5, 5);

</script>

</body>

</html>

上面第一个例子的问题是你必须调用两个函数来显示结果。

第二个例子的问题是您无法阻止计算器函数显示结果。

现在是引入回调的时候了。

JavaScript 回调

回调是作为参数传递给另一个函数的函数。

使用回调，您可以通过回调调用计算器函数（myCalculator），并在计算完成后让计算器函数运行回调：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 回调</h1>

<p>进行计算，然后显示结果。</p>

<p id="demo"></p>

<script>

function myDisplayer(some) {

document.getElementById("demo").innerHTML = some;

}

function myCalculator(num1, num2, myCallback) {

let sum = num1 + num2;

myCallback(sum);

}

myCalculator(5, 5, myDisplayer);

</script>

</body>

</html>

在上面的例子中， myDisplayer 是函数的名称。

它作为参数传递给 myCalculator()。

当您将函数作为参数传递时，请记住不要使用括号。

正确：myCalculator(5, 5, myDisplayer);

错误：myCalculator(5, 5, myDisplayer());

何时使用回调？

上面的例子并不是很令人兴奋。

它们经过简化了，为的是向您讲解回调的语法。

回调真正闪光之处是异步函数，其中一个函数必须等待另一个函数（例如等待文件加载）。

下一章将介绍异步函数。

# 第90章 异步的 JavaScript

异步 JavaScript

上一章中使用的例子做了很大地简化。

其目的是演示回调函数的语法：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 回调</h1>

<p>进行计算，然后显示结果。</p>

<p id="demo"></p>

<script>

function myDisplayer(some) {

document.getElementById("demo").innerHTML = some;

}

function myCalculator(num1, num2, myCallback) {

let sum = num1 + num2;

myCallback(sum);

}

myCalculator(5, 5, myDisplayer);

</script>

</body>

</html>

在上面的例子中，myDisplayer 是函数的名称。

它作为参数传递给 myCalculator()。

在现实世界中，回调最常与异步函数一起使用。

一个典型的例子是 JavaScript setTimeout()。

等待超时

在使用 JavaScript 函数 setTimeout() 时，可以指定超时时执行的回调函数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 回调</h1>

<p>等待 3 秒（3000 毫秒）让此页面发生变化。</p>

<h1 id="demo"></h1>

<script>

setTimeout(myFunction, 3000);

function myFunction() {

document.getElementById("demo").innerHTML = "I love You !!";

}

</script>

</body>

</html>

在上面的示例中，myFunction 被用作回调。

函数（函数名）作为参数传递给 setTimeout()。

3000 是超时前的毫秒数，所以 3 秒后会调用 myFunction()。

当您将函数作为参数传递时，请记住不要使用括号。

正确：setTimeout(myFunction, 3000);

错误：setTimeout(myFunction(), 3000);

若不将函数的名称作为参数传递给另一个函数，您始终可以传递整个函数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript SetTimeout()</h1>

<p>等待 3 秒（3000 毫秒）让此页面发生变化。</p>

<h1 id="demo"></h1>

<script>

setTimeout(function() { myFunction("I love You !!!"); }, 3000);

function myFunction(value) {

document.getElementById("demo").innerHTML = value;

}

</script>

</body>

</html>

在上面的例子中，function(){ myFunction("I love You !!!"); } 用作回调。它是一个完整的函数。完整的函数作为参数被传递给 setTimeout()。

3000 是超时前的毫秒数，所以 3 秒后会调用 myFunction()。

等待间隔：

在使用 JavaScript 函数 setInterval() 时，可以指定每个间隔执行的回调函数：

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript setInterval()</h1>

<p>使用 setInterval() 显示时间，间隔为一秒（1000 毫秒）。</p>

<h1 id="demo"></h1>

<script>

setInterval(myFunction, 1000);

function myFunction() {

let d = new Date();

document.getElementById("demo").innerHTML=

d.getHours() + ":" +

d.getMinutes() + ":" +

d.getSeconds();

}

</script>

</body>

</html>

在上面的例子中，myFunction 用作回调。

函数（函数名）作为参数传递给 setInterval()。

1000 是间隔之间的毫秒数，因此 myFunction() 将每秒调用一次。

等待文件

如果您创建函数来加载外部资源（如脚本或文件），则在内容完全加载之前无法使用这些内容。

这是使用回调的最佳时机。

此例加载一个 HTML 文件 (mycar.html)，并在文件完全加载后在网页中显示该 HTML 文件：

等待文件：

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 回调</h1>

<p id="demo"></p>

<script>

function myDisplayer(some) {

document.getElementById("demo").innerHTML = some;

}

function getFile(myCallback) {

let req = new XMLHttpRequest();

req.open('GET', "/demo/js/mycar.html");

req.onload = function() {

if (req.status == 200) {

myCallback(this.responseText);

} else {

myCallback("Error: " + req.status);

}

}

req.send();

}

getFile(myDisplayer);

</script>

</body>

</html>

在上面的示例中，myDisplayer 用作回调。

函数（函数名）作为参数传递给 getFile()。

以下是 mycar.html 的副本：

mycar.html

<img src="img\_car.jpg" alt="Nice car" style="width:100%">

<p>A car is a wheeled, self-powered motor vehicle used for transportation.

Most definitions of the term specify that cars are designed to run primarily on roads, to have seating for one to eight people, to typically have four wheels.</p>

<p>(Wikipedia)</p>

# 第91章 JavaScript Promise

JavaScript Promise 对象

JavaScript Promise 对象包含生产代码和对消费代码的调用：

Promise 语法

let myPromise = new Promise(function(myResolve, myReject) {

// "Producing Code"（可能需要一些时间）

myResolve(); // 成功时

myReject(); // 出错时

});

// "Consuming Code" （必须等待一个兑现的承诺）

myPromise.then(

function(value) { /\* 成功时的代码 \*/ },

function(error) { /\* 出错时的代码 \*/ }

);

当执行代码获得结果时，它应该调用两个回调之一：

|  |  |
| --- | --- |
| 结果 | 调用 |
| 成功 | myResolve(result value) |
| 出错 | myReject(error object) |

Promise 对象属性

JavaScript Promise 对象可以是：

Pending

Fulfilled

Rejected

Promise 对象支持两个属性：state 和 result。

当 Promise 对象 "pending"（工作）时，结果是 undefined。

当 Promise 对象 "fulfilled" 时，结果是一个值。

当一个 Promise 对象是 "rejected" 时，结果是一个错误对象。

|  |  |
| --- | --- |
| myPromise.state | myPromise.result |
| "pending" | undefined |
| "fulfilled" | 结果值 |
| "rejected" | error 对象 |

您无法访问 Promise 属性 state 和 result。

您必须使用 Promise 方法来处理 Promise。

如何使用 Promise

以下是使用 Promise 的方法：

myPromise.then(

function(value) { /\* code if successful \*/ },

function(error) { /\* code if some error \*/ }

);

Promise.then() 有两个参数，一个是成功时的回调，另一个是失败时的回调。

两者都是可选的，因此您可以为成功或失败添加回调。

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Promise</h1>

<p id="demo"></p>

<script>

function myDisplayer(some) {

document.getElementById("demo").innerHTML = some;

}

let myPromise = new Promise(function(myResolve, myReject) {

let x = 0;

// 一些代码（请尝试将 x 改为 5）

if (x == 0) {

myResolve("OK");

} else {

myReject("Error");

}

});

myPromise.then(

function(value) {myDisplayer(value);},

function(error) {myDisplayer(error);}

);

</script>

</body>

</html>

JavaScript Promise 实例

为了演示 Promise 的使用，我们将使用上一章中的回调实例：

等待超时

等待文件

等待超时

使用回调的例子

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript SetTimeout()</h1>

<p>等待 3 秒（3000 毫秒）让此页面发生变化。</p>

<h1 id="demo"></h1>

<script>

setTimeout(function() { myFunction("I love You !!!"); }, 3000);

function myFunction(value) {

document.getElementById("demo").innerHTML = value;

}

</script>

</body>

</html>

使用 Promise 的例子

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Promise</h1>

<p>等待 3 秒（3000 毫秒）让此页面发生变化。</p>

<h1 id="demo"></h1>

<script>

const myPromise = new Promise(function(myResolve, myReject) {

setTimeout(function(){ myResolve("I love You !!"); }, 3000);

});

myPromise.then(function(value) {

document.getElementById("demo").innerHTML = value;

});

</script>

</body>

</html>

等待文件

使用回调的例子

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript 回调</h1>

<p id="demo"></p>

<script>

function myDisplayer(some) {

document.getElementById("demo").innerHTML = some;

}

function getFile(myCallback) {

let req = new XMLHttpRequest();

req.open('GET', "/demo/js/mycar.html");

req.onload = function() {

if (req.status == 200) {

myCallback(this.responseText);

} else {

myCallback("Error: " + req.status);

}

}

req.send();

}

getFile(myDisplayer);

</script>

</body>

</html>

使用 Promise 的例子

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript Promise</h1>

<p id="demo"></p>

<script>

function myDisplayer(some) {

document.getElementById("demo").innerHTML = some;

}

let myPromise = new Promise(function(myResolve, myReject) {

let req = new XMLHttpRequest();

req.open('GET', "/demo/js/mycar.html");

req.onload = function() {

if (req.status == 200) {

myResolve(req.response);

} else {

myReject("File not Found");

}

};

req.send();

});

myPromise.then(

function(value) {myDisplayer(value);},

function(error) {myDisplayer(error);}

);

</script>

</body>

</html>

# 第92章 JavaScript Async

Async 语法

函数前的关键字 async 使函数返回 promise：

实例

async function myFunction() {

return "Hello";

}

等同于：

async function myFunction() {

return Promise.resolve("Hello");

}

以下是使用 Promise 的方法：

myFunction().then(

function(value) { /\* 成功时的代码 \*/ },

function(error) { /\* 出错时的代码 \*/ }

);

实例

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript async / await</h1>

<p id="demo"></p>

<script>

function myDisplayer(some) {

document.getElementById("demo").innerHTML = some;

}

async function myFunction() {return "Hello";}

myFunction().then(

function(value) {myDisplayer(value);},

function(error) {myDisplayer(error);}

);</script>

</body>

</html>

或者更简单，因为您期望正常值（正常响应，而不是错误）：

实例

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript async / await</h2>

<p id="demo"></p>

<script>

function myDisplayer(some) {

document.getElementById("demo").innerHTML = some;

}

async function myFunction() {return "Hello";}

myFunction().then(

function(value) {myDisplayer(value);}

);

</script>

</body>

</html>

Await 语法

函数前的关键字 await 使函数等待 promise：

let value = await promise;

await 关键字只能在 async 函数中使用。

实例

让我们慢慢来学习如何使用它。

基础语法

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript async / await</h2>

<h1 id="demo"></h1>

<script>

async function myDisplay() {

let myPromise = new Promise(function(myResolve, myReject) {

myResolve("I love You !!");

});

document.getElementById("demo").innerHTML = await myPromise;

}

myDisplay();

</script>

</body>

</html>

等待超时

<!DOCTYPE html>

<html>

<body>

<h1>JavaScript async / await</h1>

<p>请等待 3 秒（3000 毫秒）让此页面发生变化。</p>

<h1 id="demo"></h1>

<script>

async function myDisplay() {

let myPromise = new Promise(function(myResolve, myReject) {

setTimeout(function() { myResolve("I love You !!"); }, 3000);

});

document.getElementById("demo").innerHTML = await myPromise;

}

myDisplay();

</script>

</body>

</html>

等待文件

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript async / await</h2>

<p id="demo"></p>

<script>

async function getFile() {

let myPromise = new Promise(function(myResolve, myReject) {

let req = new XMLHttpRequest();

req.open('GET', "/demo/js/mycar.html");

req.onload = function() {

if (req.status == 200) {

myResolve(req.response);

} else {

myResolve("File not Found");

}

};

req.send();

});

document.getElementById("demo").innerHTML = await myPromise;

}

getFile();

</script>

</body>

</html>

# 第93章 JavaScript HTML DOM

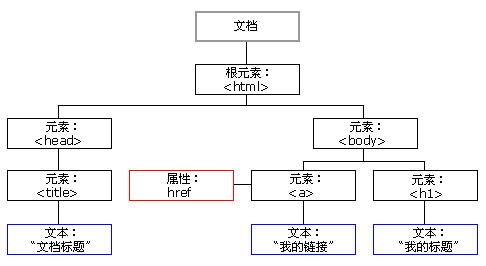
通过 HTML DOM，JavaScript 能够访问和改变 HTML 文档的所有元素。

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 模型被结构化为对象树：

对象的 HTML DOM 树



通过这个对象模型，JavaScript 获得创建动态 HTML 的所有力量：

JavaScript 能改变页面中的所有 HTML 元素

JavaScript 能改变页面中的所有 HTML 属性

JavaScript 能改变页面中的所有 CSS 样式

JavaScript 能删除已有的 HTML 元素和属性

JavaScript 能添加新的 HTML 元素和属性

JavaScript 能对页面中所有已有的 HTML 事件作出反应

JavaScript 能在页面中创建新的 HTML 事件

您将学到什么

在本教程下面的章节中，您将学到：

如何改变 HTML 元素的内容

如何改变 HTML 元素的样式（CSS）

如何对 HTML DOM 事件作出反应

如何添加和删除 HTML 元素

什么是 DOM？

DOM 是一项 W3C (World Wide Web Consortium) 标准。

DOM 定义了访问文档的标准：

“W3C 文档对象模型（DOM）是中立于平台和语言的接口，它允许程序和脚本动态地访问、更新文档的内容、结构和样式。”

W3C DOM 标准被分为 3 个不同的部分：

Core DOM - 所有文档类型的标准模型

XML DOM - XML 文档的标准模型

HTML DOM - HTML 文档的标准模型

什么是 HTML DOM？

HTML DOM 是 HTML 的标准对象模型和编程接口。它定义了：

作为对象的 HTML 元素

所有 HTML 元素的属性

访问所有 HTML 元素的方法

所有 HTML 元素的事件

换言之：HTML DOM 是关于如何获取、更改、添加或删除 HTML 元素的标准。

第94章