

## 读书笔记

笔记本: My Notebook

创建时间: 2021/1/18 21:06

更新时间: 2021/1/19 20:57

作者: L11\_1.15

标签: 0118

---

## 第 1 章 什么是JavaScript

JavaScript 与 Java 和 C++ 等传统面向对象

### DOM

**文档对象模型** (DOM, Document Object Model) 是一个应用编程 接口 (API), 用于在 HTML 中使用扩展的 XML。DOM 将整个页面抽象为一组分层节点。HTML 或 XML 页面的每个组成部分都是一种节点, 包含不同的数据

DOM 视图: 描述追踪文档不同视图 (如应用 CSS 样式前后的 文档) 的接口。

DOM 事件: 描述事件及事件处理的接口。

DOM 样式: 描述处理元素 CSS 样式的接口。

DOM 遍历和范围: 描述遍历和操作 DOM 树的接口。

### BOM

#### 浏览器对象模型

用于支持访问和操作浏览器的窗口。使用 BOM, 开发者可以操控浏览器显示页面之外的部分。而 BOM 真正独一无二的地方, 当然也是问题 最多的地方, 就是它是唯一一个没有相关标准的 JavaScript 实现。

### 小结

JavaScript 是一门用来与网页交互的脚本语言, 包含以下三个组成部分。ECMAScript: 由 ECMA-262 定义并提供核心功能。文档对象模型 (DOM): 提供与网页内容交互的方法和接口。浏览器对象模型 (BOM): 提供与浏览器交互的方法和接口。

## 第 2 章 HTML 中的 JavaScript

### 推迟执行脚本

HTML 4.01 为 <script> 元素定义了一个叫 defer 的属性 这个属性表示脚本在 执行的时候不会改变页面的结构

完全可以在整个页面解析完之后再运行。在 <script> 元素上设置 defer 属性, 会告诉浏览器应该立即开始下载, 但执行应该推迟

**defer 属性只对外部脚本文件才有效**

### 异步执行脚本

HTML5 为 <script> 元素定义了 async 属性。从改变脚本处理方式上看, async 属性与 defer 类似。当然, 它们两者也都只 适用于外部脚本, 都会告诉浏览器立即开始下载。不过, 与 defer 不同的是, 标记为 async 的脚本并不保证能按照它们出现的次序执行

### 2.2 行内代码与外部文件

虽然可以直接在 HTML 文件中嵌入 JavaScript 代码, 但通常认为最佳实践是尽可能将 JavaScript 代码放在外部文件中 不过这个最佳实践 并不是明确的强制性规则。推荐使用外部文件的理由如下

**可维护性。**JavaScript 代码如果分散到很多 HTML 页面, 会导致维护困难。而用一个目录保存所有 JavaScript 文件, 则更容易维护, 这样开发者就可以独立于使用它们的 HTML 页面来编辑代码。

**缓存。**浏览器会根据特定的设置缓存所有外部链接的JavaScript文件，这意味着如果两个页面都用到同一个文件，则该文件只需下载一次。这最终意味着页面加载更快。

**适应未来。**通过把JavaScript放到外部文件中，就不必考虑用XHTML或前面提到的注释黑科技。包含外部JavaScript文件的语法在HTML和XHTML中是一样的。在配置浏览器请求外部文件时，要重点考虑的一点是它们会占用多少带宽。在SPDY/HTTP2中，预请求的消耗已显著降低，以轻量、独立JavaScript组件形式向客户端送达脚本更具优势。

## 2.3 文档模式

即可以使用doctype切换文档模式。最初的文档模式有两种：混杂模式（quirks mode）和标准模式（standards mode）。前者让IE像IE5一样（支持一些非标准的特性），后者让IE具有兼容标准的行为。虽然这两种模式的主要区别只体现在通过CSS渲染的内容方面，但对JavaScript也有一些关联影响，或称为副作用

## 小结

要包含外部JavaScript文件，必须将src属性设置为要包含文件的URL。文件可以跟网页在同一台服务器上，也可以位于完全不同的域。

所有<script>元素在不使用defer和async属性的情况下，包含在<script>元素中的代码必须严格按次序解释

对不推迟执行的脚本，浏览器必须解释完位于<script>元素中的代码，然后才能继续渲染页面的剩余部分。为此，通常应该把<script>元素放到页面末尾，介于主内容之后及</body>标签之前

可以使用defer属性把脚本推迟到文档渲染完毕后再执行。推迟的脚本总是按照它们被列出的次序执行。

可以使用async属性表示脚本不需要等待其他脚本，同时也不阻塞文档渲染，即异步加载。异步脚本不能保证按照它们在页面中出现的次序执行。

通过使用<noscript>元素，可以指定在浏览器不支持脚本时显示的内容。如果浏览器支持并启用脚本，则<noscript>元素中的任何内容都不会被渲染。

## 第3章 语言基础

本章内容：

语法

数据类型

流控制语句

理解函数

## 暂时性死区

let与var的另一个重要的区别，就是let声明的变量不会在作用域中被提升。

## 全局声明

与var关键字不同，使用let在全局作用域中声明的变量不会成为window对象的属性（var声明的变量则会）

不过，let声明仍然是在全局作用域中发生的，相应变量会在页面的生命周期内存续。因此，为了避免SyntaxError，必须确保页面不会重复声明同一个变量。

## 条件声明

在使用var声明变量时，由于声明会被提升，JavaScript引擎会自动将多余的声明在作用域顶部合并为一个声明。因为let的作用域是块，所以不可能检查前面是否已经使用let声明过同名变量，同时也就不可能在没有声明的情况下声明它。

不能使用let进行条件式声明是件好事，因为条件声明是一种反模式，它让程序变得更难理解。

如果你发现自己在使用这个模式，那一定有更好的替代方式。

## const 声明

const的行为与let基本相同，唯一的一个重要的区别是用它声明变量时必须同时初始化变量，且尝试修改const声明的变量会导致运行时错误

## 3.4 数据类型

ECMAScript有6种简单数据类型（也称为原始类型）：Undefined、Null、Boolean、Number、String和Symbol。Symbol（符号）是ECMAScript 6新增的。还有一种复

杂数据类型叫 Object（对象）。Object 是一种无序名值对的集合。因为在 ECMAScript 中不能定义自己的数据类型，所有值都可以用上述 7 种数据类型之一来表示。只有 7 种数据类型似乎不足以表示全部数据。但 ECMAScript 的数据类型很灵活，一种数据类型可以当作多种数据类型来使用。

## typeof 操作符

typeof 操作符就是为此而生的。对一个值使用 typeof 操作符会返回下列字符串之一：

"undefined" 表示值未定义；

"boolean" 表示值为布尔值；

"string" 表示值为字符串；

"number" 表示值为数值；

"object" 表示值为对象（而不是函数）或 null；

"function" 表示值为函数；"symbol" 表示值为符号。

## Undefined 类型

Undefined 类型只有一个值，就是特殊值 undefined。当使用 var 或 let 声明了变量但没有初始化时，就相当于给变量赋予了 undefined 值。

## Null 类型

Null 类型同样只有一个值，即特殊值 null。逻辑上讲，null 值表示一个空对象指针。

## Boolean 类型

Boolean（布尔值）类型是 ECMAScript 中使用最频繁的类型之一，有两个字面值：true 和 false。这两个布尔值不同于数值，因此 true 不等于 1，false 不等于 0。

## Number 类型

ECMAScript 中最有意思的数据类型或许就是 Number 了。Number 类型使用 IEEE 754 格式表示整数和浮点值（在某些语言中也叫双精度值）。不同的数值类型相应地也有不同的数值字面量格式。

## String 类型

String（字符串）数据类型表示零或多个 16 位 Unicode 字符序列。字符串可以使用双引号（"）、单引号（'）或反引号（`）标示。

## 转换为字符串

有两种方式把一个值转换为字符串。首先是使用几乎所有值都有的 toString() 方法。这个方法唯一的用途就是返回当前值的字符串等价物。

## Symbol 类型

符号需要使用 Symbol() 函数初始化。因为符号本身是原始类型，所以 typeof 操作符对符号返回 symbol。

符号没有字面量语法，这也是它们发挥作用的关键。按照规范，你只要创建 Symbol() 实例并将其用作对象的新属性，就可以保证它不会覆盖已有的对象属性，无论是符号属性还是字符串属性。最重要的是，Symbol() 函数不能用作构造函数，与 new 关键字一起使用。这样做是为了避免创建符号包装对象，像使用 Boolean、String 或 Number 那样，它们都支持构造函数且可用于初始化包含原始值的包装对象。

## Object 类型

ECMAScript 中的对象其实就是一组数据和功能的集合。对象通过 new 操作符后跟对象类型的名称来创建。开发者可以通过创建 Object 类型的实例来创建自己的对象，然后再给对象添加属性和方法。

## 布尔操作符

布尔操作符跟相等操作符几乎同样重要。如果没有能力测试两个值的关系，那么像 if-else 和循环这样的语句也没什么用了。布尔操作符一共有 3 个：逻辑非、逻辑与和逻辑或。

## 逻辑非

逻辑非操作符由一个叹号（!）表示。

如果操作数是对象，则返回 false。

如果操作数是空字符串，则返回 true。如果操作数是非空字符串，则返回 false。

如果操作数是数值0，则返回 true。  
如果操作数是非0数值（包括 Infinity），则返回 false。  
如果操作数是 null，则返回 true。  
如果操作数是 NaN，则返回 true。  
如果操作数是 undefined，则返回 true。

## 逻辑与

逻辑与操作符由两个和号（&&）表示  
逻辑与操作符可用于任何类型的操作数，不限于布尔值。  
如果有 操作数不是布尔值，则逻辑与并不一定会返回布尔值，而是遵循 如下规则。  
如果第一个操作数是对象，则返回第二个操作数。  
如果第二个操作数是对象，则只有第一个操作数求值为 true 才会返回该对象。  
如果两个操作数都是对象，则返回第二个操作数。  
如果有一个操作数是 null，则返回 null。  
如果有一个操作数是 NaN，则返回 NaN。  
如果有一个操作数是 undefined，则返回 undefined。

## 逻辑或

逻辑或操作符由两个管道符（||）表示，  
与逻辑与类似，如果有一个操作数不是布尔值，那么逻辑或操作符也不一定返回布尔值。它遵循如下规则。  
如果第一个操作数是对象，则返回第一个操作数。  
如果第一个操作数求值为 false，则返回第二个操作数。如果两个操作数都是对象，则返回第一个操作数。  
如果两个操作数都是 null，则返回 null。  
如果两个操作数都是 NaN，则返回 NaN。  
如果两个操作数都是 undefined，则返回 undefined。

## 关系操作符

关系操作符执行比较两个值的操作，包括小于（<）、大于（>）、小于等于（<=）和大于等于（>=），用法跟数学课上学的一样。这几个操作符都返回布尔值，  
如果操作数都是数值，则执行数值比较。  
如果操作数都是字符串，则逐个比较字符串中对应字符的编码。  
如果有任一操作数是数值，则将另一个操作数转换为数值，执行 数值比较。  
如果有任一操作数是对象，则调用其 valueOf() 方法，取得结果后再根据前面的规则执行比较。  
如果没有 valueOf() 操作符，则调用 toString() 方法，取得结果后再根据前面的规则 执行比较。  
如果有任一操作数是布尔值，则将其转换为数值再执行比较。

## if 语句

if 语句是使用最频繁的语句之一  
if (condition) statement1 else statement2  
这里的条件（condition）可以是任何表达式，并且求值结果 不一定是布尔值。ECMAScript会自动调用 Boolean() 函数将这个表达式的值转换为布尔值。如果条件求值为 true，则执行语句 statement1；如果条件求值为 false，则执行语句 statement2。这里的语句可能是一行代码，也可能是一个代码块（即包含在一对花括号中的多行代码）。

```
if (i > 25)
  console.log("Greater than 25."); // 只有一行代码 的语句
else {
  console.log("Less than or equal to 25."); // 一个语句块 }
```

## do-while 语句

do-while 语句是一种后测试循环语句，即循环体中的代码执行后才会对退出条件进行求值  
let i = 0;  
do {  
 i += 2;  
} while (i < 10);

在这个例子中，只要 i 小于10，循环就会重复执行。i 从0开始，每次循环递增2

## while 语句

while 语句是一种先测试循环语句，即先检测退出条件，再执行循环体内的代码。因此，while 循环体内的代码有可能不会执行。

```
let i = 0;
while (i < 10) {
  i += 2;
}
```

在这个例子中，变量 i 从0开始，每次循环递增2。只要 i 小于 10，循环就会继续。

## for 语句

for 语句也是先测试语句，只不过增加了进入循环之前的初始化代码，以及循环执行后要执行的表达式

```
let count = 10;
for (let i = 0; i < count; i++) {    console.log(i);
}
```

以上代码在循环开始前定义了变量 i 的初始值为0。然后求值条件表达式，如果求值结果为 true (i < count)，则执行循环体。因此循环体也可能不会被执行。如果循环体被执行了，则循环后表达式也会执行，以便递增变量 i。

## for-in 语句

for-in 语句是一种严格的迭代语句，用于枚举对象中的非符号键属性

```
for (const propName in window) {
  document.write(propName);
}
```

这个例子使用 for-in 循环显示了BOM对象 window 的所有属性。每次执行循环，都会给变量 propName 赋予一个 window 对象的属性作为值，直到 window 的所有属性都被枚举一遍。与 for 循环一样，这里控制语句中的 const 也不是必需的。但为了确保这个局部变量不被修改，推荐使用 const。

如果 for-in 循环要迭代的变量是 null 或 undefined，则不执行循环体

## for-of 语句

for-of 语句是一种严格的迭代语句，用于遍历可迭代对象的元素，

```
for (const el of [2,4,6,8]) {    document.write(el);
}
```

在这个例子中，我们使用 for-of 语句显示了一个包含4个元素的数组中的所有元素。循环会一直持续到将所有元素都迭代完。与 for 循环一样，这里控制语句中的 const 也不是必需的。但为了确保这个局部变量不被修改，推荐使用 const。

for-of 循环会按照可迭代对象的 next() 方法产生值的顺序迭代元素。关于可迭代对象，本书将在第7章详细介绍。

如果尝试迭代的变量不支持迭代，则 for-of 语句会抛出错误

## break 和 continue 语句

break 和 continue 语句为执行循环代码提供了更严格的控制手段。其中，break 语句用于立即退出循环，强制执行循环后的下一条语句。而 continue 语句也用于立即退出循环，但会再次从循环顶部开始执行。

```
let num = 0;
for (let i = 1; i < 10; i++) {
  if (i % 5 == 0) {
    break;
  } num++;
} console.log(num); // 4
```

在上面的代码中，for 循环会将变量 i 由1递增到10。而在循环体内，有一个 if 语句用于检查 i 能否被5整除（使用取模操作符）。如果是，则执行 break 语句，退出循环。变量 num 的初始值为0，表示循环在退出前执行了多少次。当 break 语句执行后，下一行执行的代码是 console.log(num)，显示4。之所以循环执行了4次，是因为当 i 等于5时，break 语句会导致循环退出，该次循环不会执行递增 num 的代码。如果将 break 换成 continue

## with 语句

with 语句的用途是将代码作用域设置为特定的对象

```
with(location) {  
    let qs = search.substring(1);  
    let hostName = hostname;  
    let url = href;  
}
```

这里，with 语句用于连接 location 对象。这意味着在这个 语句内部，每个变量首先会被认为是一个局部变量。如果没有找到该 局部变量，则会搜索 location 对象，看它是否有一个同名的属性。如果有，则该变量会被求值为 location 对象的属性。严格模式不允许使用 with 语句，否则会抛出错误。

## switch 语句

switch 语句是与 if 语句紧密相关的一种流控制语句，从其他 语言借鉴而来。ECMAScript 中 switch 语句跟 C 语言中 switch 语句的语法非常相似

```
switch (expression) {  
    case value1:  
        statement break;  
    case value2:  
        statement break;  
    case value3:  
        statement break;  
    case value4:  
        statement break;  
    default:  
        statement  
}
```

这里的每个 case（条件/分支）相当于：“如果表达式等于后面的值，则执行下面的语句。” break 关键字会导致代码执行跳出 switch 语句。如果没有 break，则代码会继续匹配下一个条件。default 关键字用于在任何条件都没有满足时指定默认执行的 语句（相当于 else 语句）

## 3.7 函数

函数对任何语言来说都是核心组件，因为它们可以封装语句，然后 在任何地方、任何时间执行。

function 关键字声明，后跟一组参数，然后是函数体。

## 小结

ECMAScript 中的基本数据类型包括 Undefined、Null、Boolean、Number、String 和 Symbol

与其他语言不同，ECMAScript 不区分整数和浮点值，只有 Number 一种数值数据类型。

Object 是一种复杂数据类型，它是这门语言中所有对象的基 类。

严格模式为这门语言中某些容易出错的部分施加了限制。

ECMAScript 提供了 C 语言和类 C 语言中常见的很多基本操作符，包括数学操作符、布尔操作符、关系操作符、相等操作符和赋值 操作符等。

这门语言中的流控制语句大多是从其他语言中借鉴而来的，比如 if 语句、for 语句和 switch 语句等。ECMAScript 中的函数与其他语言中的函数不一样。

不需要指定函数的返回值，因为任何函数可以在任何时候返回任 何值。

不指定返回值的函数实际上会返回特殊值 undefined。

## 第 4 章 变量、作用域与内存

### 本章内容

通过变量使用原始值与引用值

理解执行上下文

理解垃圾回收

### 原始值与引用值



ECMAScript变量可以包含两种不同类型的数据：原始值和引用值。原始值（primitive value）就是最简单的数据，引用值（reference value）则是由多个值构成的对象。

## 动态属性

原始值和引用值的定义方式很类似，都是创建一个变量，然后给它赋一个值。不过，在变量保存了这个值之后，可以对这个值做什么，则大有不同。对于引用值而言，可以随时添加、修改和删除其属性和方法。

首先创建了一个对象，并把它保存在变量 person 中。然后，给这个对象添加了一个名为 name 的属性，并给这个属性赋值了一个字符串 "Nicholas"。在此之后，就可以访问这个新属性，直到对象被销毁或属性被显式地删除。原始值不能有属性，尽管尝试给原始值添加属性不会报错。

```
let name = "Nicholas";
name.age = 27;
console.log(name.age); // undefined
```

## 作用域链增强

虽然执行上下文主要有全局上下文和函数上下文两种（eval() 调用内部存在第三种上下文），但有其他方式来增强作用域链。某些语句会导致在作用域链前端临时添加一个上下文，这个上下文在代码执行后会被删除。通常在两种情况下会出现这个现象，即代码执行到下面任意一种情况时

try / catch 语句的 catch 块 with 语句 这两种情况下，都会在作用域链前端添加一个变量对象。对 with 语句来说，会向作用域链前端添加指定的对象；对 catch 语句而言，则会创建一个新的变量对象，这个变量对象会包含要抛出的错误对象的声明。

```
function buildUrl() {
  let qs = "?debug=true";
  with(location){
    let url = href + qs;
  }
  return url;
}
```

## 变量声明

ES6之后，JavaScript的变量声明经历了翻天覆地的变化。直到 ECMAScript 5.1，var 都是声明变量的唯一关键字。ES6不仅增加了 let 和 const 两个关键字，而且还让这两个关键字压倒性地超越 var 成为首选。

### 使用 var 的函数作用域声明

在使用 var 声明变量时，变量会被自动添加到最接近的上下文。在函数中，最接近的上下文就是函数的局部上下文。在 with 语句中，最接近的上下文也是函数上下文。如果变量未经声明就被初始化了，那么它就会自动被添加到全局上下文，如下面的例子所示

```
function add(num1, num2) {
  var sum = num1 + num2;
  return sum;
}
let result = add(10, 20); // 30
console.log(sum); // 报错: sum在这里不是 有效变量
```

**注意** 未经声明而初始化变量是JavaScript编程中一个非常常见的错误，会导致很多问题。为此，读者在初始化变量之前一定要先声明变量。在严格模式下，未经声明就初始化变量会报错。

### 使用 let 的块级作用域声明

ES6新增的 let 关键字跟 var 很相似，但它的作用域是块级的，这也是JavaScript中的新概念。块级作用域由最近的一对包含花括号 {} 界定。换句话说，if 块、while 块、function 块，甚至连单独的块也是 let 声明变量的作用域

```
if (true) {
  let a;
} console.log(a); // ReferenceError: a没有定义
while (true) {
  let b;
} console.log(b); // ReferenceError: b没有定义
function foo() {
  let c;
```

```
} console.log(c); // ReferenceError: c没有定义
// 这没什么可奇怪的
// var声明也会导致报错
// 这不是对象字面量，而是一个独立的块
// JavaScript解释器会根据其中内容识别出它来
{
  let d;
} console.log(d); // ReferenceError: d没有定义
```

let 与 var 的另一个不同之处是在同一作用域内不能声明两次。重复的 var 声明会被忽略，而重复的 let 声明会抛出 SyntaxError

## 使用 const 的常量声明

除了 let，ES6同时还增加了 const 关键字。使用 const 声明的变量必须同时初始化为某个值。一经声明，在其生命周期的任何时候都不能再重新赋予新值

```
const a; // SyntaxError: 常量声明时没有初始化
const b = 3;
console.log(b); // 3
b = 4; // TypeError: 给常量赋值
```

const 除了要遵循以上规则，其他方面与 let 声明是一样的

const 声明只应用到顶级原语或者对象。换句话说，赋值为对象的 const 变量不能再被重新赋值为其他引用值，但对象的键则不受限制。

## 标识符查找

当在特定上下文中为读取或写入而引用一个标识符时，必须通过搜索确定这个标识符表示什么。搜索开始于作用域链前端，以给定的名称搜索对应的标识符。如果在局部上下文中找到该标识符，则搜索停止，变量确定；如果没有找到变量名，则继续沿作用域链搜索。（注意，作用域链中的对象也有一个原型链，因此搜索可能涉及每个对象的原型链。）这个过程一直持续到搜索至全局上下文的变量对象。如果仍然没有找到标识符，则说明其未声明。

```
var color = 'blue';
function getColor() {
  return color;
}
console.log(getColor()); // 'blue'
```

标识符查找并非没有代价。访问局部变量比访问全局变量要快，因为不用切换作用域。不过，JavaScript引擎在优化标识符查找上做了很多工作，将来这个差异可能就微不足道了。

## 通过 const 和 let 声明提升性能

ES6增加这两个关键字不仅有助于改善代码风格，而且同样有助于改进垃圾回收的过程。因为 const 和 let 都以块（而非函数）为作用域，所以相比于使用 var，使用这两个新关键字可能会更早地让垃圾回收程序介入，尽早回收应该回收的内存。在块作用域比函数作用域更早终止的情况下，这就有可能发生。

## 小结

JavaScript变量可以保存两种类型的值：原始值和引用值。原始值可能是以下6种原始数据类型之一：Undefined、Null、Boolean、Number、String 和 Symbol。原始值和引用值有以下特点。

原始值大小固定，因此保存在栈内存上。

从一个变量到另一个变量复制原始值会创建该值的第二个副本。

引用值是对象，存储在堆内存上。

包含引用值的变量实际上只包含指向相应对象的一个指针，而不是对象本身。

从一个变量到另一个变量复制引用值只会复制指针，因此结果是两个变量都指向同一个对象。

typeof 操作符可以确定值的原始类型，而 instanceof 操作符用于确保值的引用类型。

任何变量（不管包含的是原始值还是引用值）都存在于某个执行上下文中（也称为作用域）。这个上下文（作用域）决定了变量的生命周期，以及它们可以访问代码的哪些部分。

执行上下文可以总结如下。执行上下文分全局上下文、函数上下文和块级上下文。

代码执行流每进入一个新上下文，都会创建一个作用域链，用于搜索变量和函数。

函数或块的局部上下文不仅可以访问自己作用域内的变量，而且也可以访问任何包含上下文乃至全局上下文中的变量。全局上下文只能访问全局上下文中的变量和函数，不能直接访问局部上下文中的任何数据。变量的执行上下文用于确定什么时候释放内存。JavaScript是使用垃圾回收的编程语言，开发者不需要操心内存分配和回收。JavaScript的垃圾回收程序可以总结如下。



