

原文地址：[freeCodeCamp](#)（需梯子）

第一次翻译~ 如有纰漏，请多多指点~

JavaScript 完整手册

JavaScript 是世界上最流行的编程语言之一，现在也广泛用于浏览器之外的场景中。近几年，Node.js 的崛起解锁了长期以来被 Java, Ruby, Python, PHP 等传统服务端语言统治的后端开发领域。

这本 JavaScript 完整手册遵循二八定律（the 80/20 rule）：在 20% 的时间里了解 80% 的 JavaScript 知识。

快来了解所有你需要知道的 JavaScript 吧！

注意：你可以获取这篇 JavaScript 指南的 [PDF](#), [ePub](#), [Mobi](#) 版本以便在 Kindle 或平板上阅读。

介绍

JavaScript 是世界上最流行的编程语言之一。自从 20 年前诞生以来，已经走过很长的一段路。作为第一个而且是唯一一个被网页浏览器原生支持的脚本语言，它被下了死命令。

一开始，它并不像今天这样强大，主要被用于各种花哨的动画和当时被广泛称奇的动态 HTML（[DHTML](#)）中。伴随着网页平台发展的需要，JavaScript 也有责任变得更好，以满足世界上最广泛使用的生态系统之一的需求。平台本身引入了很多东西，包括浏览器 API，同时语言特性也成长了许多。

现在 JavaScript 也可以操作数据库和开发应用程序，甚至可以开发嵌入式程序，移动 APP，电视应用程序等等。曾经只是在浏览器中的小语言，现在也是世界上最流行的语言了~

JavaScript 基本定义

JavaScript 是这样的编程语言：

- 高级：它允许你更加注重本身的逻辑，忽略当前运行它的机器的详细信息。JavaScript 通过垃圾回收器自动管理内存，让你可以更专注代码而不是管理内存，它也提供了很多构造函数让你能够处理强大的变量和对象。
- 动态：和静态语言在编译时执行相反，动态语言在运行时才会执行。这有利有弊，JavaScript 给我们提供了强大的功能，比如：动态类型，延迟绑定，反射，函数式编程，对象运行时变更（object runtime alteration），闭包等等。
- 动态类型：变量不用定义类型。你可以为变量重新绑定任何类型，比如给一个已声明过的字符串变量绑定一个整型值。
- 弱类型：与强类型相反，弱类型语言不强制对象的类型。这使得操作更加灵活，但是我们也无法进行类型检查确保类型安全（TypeScript 和 Flow 旨在改善这个问题）。
- 解释型：JavaScript 通常被认为是一种解释型语言，这意味着在程序运行前不需要先编译，这恰恰与 C 语言，Java 或者是 Go 语言相反。事实上，出于性能考虑浏览器会在执行 JavaScript 之前进行编译，但是这一切都是自然而然发生的，不需要我们进行额外的操作。
- 多范型：JavaScript 不强制使用任何固定的编程范式，不像 Java 强制面向对象编程或者是 C

语言强制命令式编程。在 JavaScript 中，你可以使用原型和 ES6 中提供的 classes 语法面向对象编程，你也可以通过它的头等函数（first-class functions）编写函数式编程风格的代码，甚至以命令式编写程序（C-like）。

说明一下，JavaScript 和 Java 无关，这是一个不幸的命名选择，但是我们不得不接受现实。

JavaScript 版本

让我介绍一下 ECMAScript。我们有一个专门介绍 ECMAScript 的完整指南，你可以在那里深入了解它，但现在，你只需要知道 ECMAScript（也被称作 ES）是 JavaScript 标准的名字。JavaScript 是对 ECMAScript 标准的一种实现，这也是为什么你会听到 ES6, ES2015, ES2016, ES2017, ES2018 等等。

很久以前，所有浏览器中运行的 JavaScript 版本基于 ECMAScript 3。版本 4 因为特征蠕动（feature creep）被取消了（他们试图一次性添加很多特性）。虽然 ES5 是 JavaScript 的一个巨大版本，但是 ES2015（也被称作 ES6）也是 JavaScript 的重要更新。

从那时起，标准制定委员会决定每年更新一个版本，避免版本迭代间隔太久，也可以加快反馈速度。

现在，最新批准的 JavaScript 版本是 ES2017（译注：2018/11/10，最新版本是 ES2018）。

ECMAScript

无论何时你阅读关于 JavaScript 的内容时都无可避免的看到下面这些术语：

- ES3
- ES5
- ES6
- ES7
- ES8
- ES2015
- ES2016
- ES2017
- ECMAScript 2017
- ECMAScript 2016
- ECMAScript 2015

这些都是什么？他们都指 **JavaScript** 标准，被称作 ECMAScript。

ECMAScript 是 **JavaScript** 所基于的标准，经常被简称为 **ES**。

除了 JavaScript，实现了 ECMAScript 标准的还包括：

- *ActionScript*（Flash 脚本语言），自从官方决定从 2020 年起不再维护 Flash 便不再流行。
- *JScript*（微软），一开始只有 Netscape 支持 JavaScript，浏览器大战愈演愈烈，微软便实现了仅 IE 浏览器支持的版本。

但是毫无疑问，JavaScript 仍是最普遍的 ES 实现。

为什么是这个奇怪的名字？Ecma International是瑞士标准协会，负责制定国际标准。

当 JavaScript 被创建时，它被 Netscape 和 Sun Microsystems 提交到 Ecma，命名为 ECMA-262，也叫做 ECMAScript。根据维基百科上的内容，Netscape 和 Sun Microsystems（Java 制造商）联合发布的[新闻稿](#)可能会帮助了解如此命名的原因，其中可能包括微软在委员会里的法律和品牌问题。

IE9 之后，微软停止在浏览器中将 ES 实现称为 JScript，开始叫作 JavaScript（至少我不能任何引用）。

所以，从201x年起，唯一一个支持 ECMAScript 标准的流行语言就只有 JavaScript 了。

ECMAScript 最新版本

目前 ECMAScript 版本是 **ES2018**，即 **ES9**，2018年六月发布。

下个版本什么时候发布？

通常，JavaScript 会在每年夏天发布标准版本，所以我们可以 在 2019 年夏天见到 **ECMAScript 2019**（即 **ES2019** 或者 **ES10**），但这一切只是猜测。

TC39 是什么

TC39 是 JavaScript 发展委员会。

TC39 成员涉及 JavaScript 和浏览器供应商，包括火狐，谷歌，Facebook，Apple，微软，英特尔，PayPal，SalesForce 等等。

每一个标准版本的发布都必须通过[不同阶段](#)的提案。

ES 版本

我发现 ES 版本有时候通过版本号指代，有时候通过年份，这会让人困惑。

在 ES2015 之前，ECMAScript 标准通常按照版本号命名，所以 ES5 是2009年更新的 ECMAScript 标准官方命名。

为什么会这样？在 ES2015 发布时，名字从 ES6 变成了 ES2015，但是为时已晚，人们仍然习惯性地称为 ES6，社区也没有抛弃这个名字 - 世界人民仍然按照版本号的方式指代 ES 版本。

这个表格可以帮助理清思路：

版本	官方名称	发布日期
ES9	ES2018	2018年6月
ES8	ES2017	2017年6月

ES7	ES2016	2016年6月
ES6	ES2015	2015年6月
ES5.1	ES5.1	2011年6月
ES5	ES5	2009年12月
ES4	ES4	废弃
ES3	ES3	1999年12月
ES2	ES2	1998年6月
ES1	ES1	1997年6月

ES.Next 始终指 JavaScript 未来版本。在撰写本文时，ES9 已经发布，ES.Next 是 ES10。

ES6 改进

ECMAScript 2015，也即熟知的 ES6，是 ECMAScript 标准的基础版本，距离上个版本 ECMAScript 5.1 发布有四年之久，也是从这个版本开始将版本改为以年命名。所以它不应该叫做 ES6（尽管每个人都这么叫），而是 ES2015。

ES5 从 1999 年到2009年花费了十年时间完善，尽管对于这个语言来讲，它也是一个重要版本，但是太长时间过去了，已经不值得我们讨论 ES5 之前的代码是如何工作的。

从 ES5.1 到 ES6，JavaScript 语言有了重要的新特性以及在更好的实践中的关键更新。要了解 ES2015 的基本功能，请参阅规范文档的250页到600页。

ES2015 中的重要更新包括：

- 箭头函数
- Promises
- Generators
- let 和 const
- Classes
- 多行字符串
- 模板字符串
- 参数默认值
- 扩展运算符
- 解构赋值
- 增强的对象字面量
- for..of 循环
- Map 和 Set

在这篇指南中，我将在每个章节中专门介绍它们。让我们开始吧！

箭头函数

箭头函数改变了大多数 JavaScript 代码的书写习惯和工作方式。

从视觉上来讲，它更简单了，也是受欢迎的改变，比如：

```
const foo = function foo() {  
  // ...  
}
```

变成：

```
const foo = () => {  
  // ...  
}
```

如果这个函数体只有一行，可以是这样：

```
const foo = () => doSomething()
```

如果只有一个参数，可以是这样：

```
const foo = param => doSomething(param)
```

和常规函数相比它没有带来任何不兼容变化，和以前一样工作。

新的 this 作用域

箭头函数中的 this 从执行上下文继承。

在以前的常规函数中，this 通常指最近的函数。然而在箭头函数中这个问题没有了，你不再需要重写一遍 var that = this。

Promises

Promises 帮我们解决了著名的“回调地狱”问题，虽然它引入了更复杂的问题（在 ES2017 中可以通过更高级的构造函数 async 解决）。

在 ES2015 之前，JavaScript 开发者就可以通过使用不同的库（jQuery, q, deferred.js, vow...）实现类似 Promises 的功能。该标准制定了更通用的方法。

通过使用 promises，你可以重构这个代码：

```
setTimeout(function() {  
  console.log('I promised to run after 1s')  
  setTimeout(function() {  
    console.log('I promised to run after 2s')  
  }, 1000)  
, 1000)
```

等同于：

```
const wait = () => new Promise((resolve, reject) => {  
  setTimeout(resolve, 1000)  
})  
wait().then(() => {  
  console.log('I promised to run after 1s')})
```

```
    return wait()
  })
  .then(() => console.log('I promised to run after 2s'))
```

Generators

生成器是一种特殊的函数，能够暂停输出，稍后恢复，而且允许其它代码在此期间运行。

代码本身决定了它必须等待，以便让其它代码“按照队列”顺序运行，而且保留了“当等待”完成时恢复操作的权利。所有的这些都通过一个简单的关键字 `yield` 来完成。当一个生成器包含该关键字，代码将暂停执行。生成器可以包含很多个 `yield` 关键字，因此可以暂停很多次，并且通过 `*function` 关键字标识，不要与 C 语言，C++ 或者 Go 等底层语言的指针反向引用操作符混淆。

在 JavaScript 中，生成器启用全新的编程范例，比如：

- 生成器运行中双向通信
- 持久的 while 循环，不会冻结程序

这里有个例子可以解释生成器是如何工作的：

```
function *calculator(input) {
  var doubleThat = 2 * (yield (input / 2))
  var another = yield (doubleThat)
  return (input * doubleThat * another)
}
```

初始化：

```
const calc = calculator(10)
```

然后开始迭代生成器：

```
calc.next()
```

第一次迭代，代码返回了 `this` 对象：

```
{
  done: false
  value: 5
}
```

发生了什么：函数开始运行时，`input = 10` 作为参数传入了生成器的构造函数中，直到遇到 `yield`，返回了 `yield` 的内容：`input / 2 = 5`。所以我们得到了一个值为 5，并且告诉我们迭代没有完成（仅仅是函数暂停了）。

在第二次迭代中，我们传入 7：

```
calc.next(7)
```

然后我们会得到：


```
{
  done: false
  value: 14
}
```

7 是 doubleThat 的值。

注意：你可能会认为 `input / 2` 是这个参数，但是它仅仅是第一次迭代的返回值，这次我们跳过了这一步，使用新的输入值 7 和 2 相乘。

我们继续第二次迭代，它返回了 doubleThat，值为 14。

下一个，最后一次迭代，我们传入 100：

```
calc.next(100)
```

返回：

```
{
  done: true
  value: 14000
}
```

整个迭代结束（没有 `yeild` 关键字了），我们得到了 `(input * doubleThat * another)` 的值：`10 * 14 * 100`。

let 和 const

`var` 是传统的函数作用域。

`let` 是新的声明变量的方法，拥有块级作用域。这意味着在 `for` 循环中，`if` 语句内或者 `plain` 块中使用 `let` 声明的变量不会“逃出”所在的块，而 `var` 变量则会被提升到函数定义。

`const` 和 `like` 相似，但是不可更改。

展望 JavaScript 的发展，`var` 声明会逐渐消失，只剩下 `let` 和 `const`。

更特别的是，由于不可变的特性，`const` 在今天已经出人意料的被广泛使用。

Classes

传统上，JavaScript 是唯一基于原型继承的语言。从基于类继承的语言转向使用 JavaScript 的程序员会觉得困惑，但是 ES2015 引入了 `classes`，作为 JavaScript 内部实现继承的语法糖，它改变了我们编写 JavaScript 程序的方式。

现在，继承变得非常简单，和其他面向对象的编程语言类似：

```
class Person {
  constructor(name) {
    this.name = name
  }
}
```

```

    }
    hello() {
        return 'Hello, I am ' + this.name + ' .'
    }
}
}
class Actor extends Person {
    hello() {
        return super.hello() + ' I am an actor.'
    }
}
}
var tomCruise = new Actor('Tom Cruise')
tomCruise.hello()

```

上面的代码会打印出：“*Hello, I am Tom Cruise. I am an actor.*”

Classes 没有显性的声明类变量，你必须在构造函数（constructor）中初始化变量。

Constructor

Classes 拥有一个特殊的方法 constructor，在使用 new 实例化类时被调用。

Getters 和 Setters

可以像这样声明一个 getter 属性：

```

class Person {
    get fullName() {
        return `${this.firstName} ${this.lastName}`
    }
}

```

用同样的方式声明 setter 属性：

```

class Person {
    set age(years) {
        this.theAge = years
    }
}

```

模块化

ES2015 之前，至少有三个主要的模块化标准，这分裂了整个社区：

- AMD
- RequireJS
- CommonJS

ES2015 制定了统一的模块化标准。

导入模块

通过 `import ... from ...` 导入模块：

```
import * from 'mymodule'  
import React from 'react'  
import { React, Component } from 'react'  
import React as MyLibrary from 'react'
```

导出模块

你可以使用关键字 `export` 将编写的模块内容导出到其它模块里：

```
export var foo = 2  
export function bar() { /* ... */ }
```

模板字符串

模板字符串是创建字符串的新方法：

```
const aString = `A string`
```

使用 `${a_variable}` 语法可以方便地将表达式的值插到字符串里：

```
const var = 'test'  
const string = `something ${var}` //something test
```

你还可以执行更复杂的表达式，像这样：

```
const string = `something ${1 + 2 + 3}`  
const string2 = `something ${foo() ? 'x' : 'y' }`
```

字符串可以是多行的：

```
const string3 = `Hey  
this  
string  
is awesome!`
```

对比一下 ES2015 之前的多行字符串的写法：

```
var str = 'One\n' +  
'Two\n' +  
'Three'
```

参数默认值

函数现在支持使用默认参数值：

```
const foo = function(index = 0, testing = true) { /* ... */ }  
foo()
```

扩展运算符

你可以通过扩展运算符 ... 扩展数组，对象或者是字符串。

让我们用数组举个例子：

```
const a = [1, 2, 3]
```

你可以这样创建一个新数组：

```
const b = [...a, 4, 5, 6]
```

你可以复制一个数组：

```
const c = [...a]
```

这对对象同样奏效，这样复制一个对象：

```
const newObj = { ...oldObj }
```

对于字符串，扩展运算符会生成一个对应每个字符的数组：

```
const hey = 'hey'  
const arrayized = [...hey] // ['h', 'e', 'y']
```

这个运算符非常有用。最重要的就是可以以一种十分简单的方式为一个函数传递数组形式的参数：

```
const f = (foo, bar) => {}  
const a = [1, 2]  
f(...a)
```

以前你可以使用 f.apply(null, a) 达到同样的效果，但是它不是很易读。

解构赋值

给你一个对象，你可以抽出一些值把他们赋值给别的变量：

```
const person = {  
  firstName: 'Tom',  
  lastName: 'Cruise',  
  actor: true,  
  age: 54, //made up  
}  
const {firstName: name, age} = person
```

name 和 age 包含这些值。

这个语法也可以用在数组中：

```
const a = [1, 2, 3, 4, 5]
```

```
[first, second, , , fifth] = a
```

加强的函数字面量

ES2015 中函数字面量更加强大。

声明变量的简单语法

以前：

```
const something = 'y'
const x = {
  something: something
}
```

现在你可以：

```
const something = 'y'
const x = {
  something
}
```

原型

可以像这样为变量指定原型：

```
const anObject = { y: 'y' }
const x = {
  __proto__: anObject
}
```

super()

```
const anObject = { y: 'y', test: () => 'zoo' }
const x = {
  __proto__: anObject,
  test() {
    return super.test() + 'x'
  }
}
x.test() //zoox
```

动态属性

```
const x = {
  ['a' + '_' + 'b']: 'z'
}
x.a_b //z
```

for-of 循环

2009年的 ES5 引入了 `forEach()` 循环。虽然很好，但是不能像 `for` 那样中途跳出循环。

ES2015 引入了 `for-of` 循环，结合了 `forEach` 的简洁和跳出循环的能力。

```
//iterate over the value
for (const v of ['a', 'b', 'c']) {
  console.log(v);
}
//get the index as well, using `entries()`
for (const [i, v] of ['a', 'b', 'c'].entries()) {
  console.log(i, v);
}
```

Map 和 Set

Map 和 **Set**（以及各自的弱引用类型 **WeakMap** 和 **WeakSet**）是官方实现的两种非常流行的数据结构（稍后介绍）。

ES2016 改进

ES7，官方称作 ECMAScript 2016，2016年6月发布。

和 ES6 相比，ES7 是 JavaScript 的小版本更新，包括两个功能：

- `Array.prototype.includes`
- 指数运算符

`Array.prototype.includes()`

这个功能引入了更易读的语法来检查一个数组是否包括一个元素。

在 ES6 以及更低版本中，检查一个元素是否在数组中你不得不使用 `indexOf` 检查数组的索引，如果返回 `-1` 元素则不在数组中。

因为 `-1` 被认为是个真值，所以你无法判断下面这个例子：

```
if (![1,2].indexOf(3)) {
  console.log('Not found')
}
```

用 ES7 的新语法可以得到我们预期的结果：

```
if (![1,2].includes(3)) {
  console.log('Not found')
}
```

指数运算符

指数运算符 `**` 和 `Math.pow()` 一致，但是引入的是个语言特性，而不是一个库函数。

```
Math.pow(4, 2) === 4 ** 2
```

这个功能是 `Math` 密集型 JavaScript 应用很好的补充。`**` 运算符在很多语言中都已经标准化，比如 Python, Ruby, MATLAB, Lua, Perl 等等。

ES2017 改进

ECMAScript 2017, ECMA-262 标准的第八版（称作 **ES2017** 或者 **ES8**），2017年6月发布。

和 ES6 相比，ES8 仍然带来了很多有用的功能：

- String padding
- Object.values
- Object.entries
- Object.getOwnPropertyDescriptors()
- 函数参数尾逗号
- 异步函数
- Shared memory and atomics

String padding

String padding 的目的是为一个字符串添加字符，让它可以和声明的长度一致。

ES2017 引入了两个 String 上的方法：`padStart()` 和 `padEnd()`。

```
padStart(targetLength [, padString])
padEnd(targetLength [, padString])
```

例子：

padStart()	输出
<code>'test'.padStart(4)</code>	<code>'test'</code>
<code>'test'.padStart(5)</code>	<code>' test'</code>
<code>'test'.padStart(8)</code>	<code>' test'</code>
<code>'test'.padStart(8, 'abcd')</code>	<code>'abcdtest'</code>

padEnd()	输出
<code>'test'.padEnd(4)</code>	<code>'test'</code>
<code>'test'.padEnd(5)</code>	<code>'test '</code>
<code>'test'.padEnd(8)</code>	<code>'test '</code>
<code>'test'.padEnd(8, 'abcd')</code>	<code>'testabcd'</code>

Object.values()

这个方法返回一个包含对象自身属性值的数组。用法：

```
const person = { name: 'Fred', age: 87 }  
Object.values(person) // ['Fred', 87]
```

Object.values() 也可以用于数组：

```
const people = ['Fred', 'Tony']  
Object.values(people) // ['Fred', 'Tony']
```

Object.entries()

这个方法返回一个 [key, value] 形式的包含对象自身所有属性及值的数组。用法：

```
const person = { name: 'Fred', age: 87 }  
Object.entries(person) // [['name', 'Fred'], ['age', 87]]
```

Object.entries() 也可以用于数组：

```
const people = ['Fred', 'Tony']  
Object.entries(people) // [['0', 'Fred'], ['1', 'Tony']]
```

getOwnPropertyDescriptors()

这个方法返回对象自身的所有描述符（非继承）。

JavaScript 中的所有对象都有一个属性集合，每个属性都有一个描述符。

描述符是属性的 attribute 集合，它包括下面这些子集：

- **value**：属性的值
- **writable**：为 true 时属性可以重写
- **get**：属性的 getter 函数，当读取属性时被调用
- **set**：属性的 setter 函数，当设置属性值时被调用
- **configurable**：如果为 false，属性不能被删除，而且不能更改所有的 attribute 值，属性自身的值除外
- **enumerable**：为 true 时属性可枚举

Object.getOwnPropertyDescriptor(obj) 接受对象作为参数，返回一个包含描述符的对象。

这有什么用？

ES2015 给我们带来了 Object.assign()，方便我们复制一个或多个对象自身的可枚举属性，返回一个新对象。

然而这样操作有个问题，它无法正确复制没有默认 attributes 的属性。举个例子，如果一个对象只有一个 setter，就不能用 Object.assign() 正确复制它。

```
const person1 = {
```



```
set name(newName) {  
  console.log(newName)  
}  
}
```

这样不会工作：

```
const person2 = {}  
Object.assign(person2, person1)
```

但是这样可以：

```
const person3 = {}  
Object.defineProperties(person3,  
Object.getOwnPropertyDescriptors(person1))
```

你可以通过控制台简单测试一下：

```
person1.name = 'x'  
"x"  
person2.name = 'x'
```

```
person3.name = 'x'  
"x"
```

person2 没有 setter，它不能被正确复制。

使用 `Object.create()` 对对象浅克隆也有同样的限制。

尾后逗号

这个功能允许在函数声明和函数调用中使用尾后逗号：

```
const doSomething = (var1, var2,) => {  
  //...  
}  
doSomething('test2', 'test2',)
```

这个改变将鼓励开发者停止使用丑陋的“行首逗号”的习惯。

异步函数

ES2017 引入了异步函数的概念，它也是这个 ECMAScript 版本里最重要的变化。

异步函数结合了 promises 和 generators 以减少 promises 带来的样板风格和 promises 链的“不要打破调用链”限制。

为什么他们很有用

它是对 promises 函数的高级抽象。

当 ES2015 里引入 Promises 的时候，这个功能致力于解决异步代码的问题。但是在 ES2015 到 ES2017 发布的两年里，人们清楚的认识到这**不是解决问题的最终方法**。

引入 Promises 用来解决著名的回调地狱问题，但是也带来了自身的复杂性，加大了语法的复杂程度。他们是很好的开端，在这个基础上可以让开发者使用更好的语法：那就是异步函数。

一个简短的例子

使用异步函数的代码可以像这样：

```
function doSomethingAsync() {
  return new Promise((resolve) => {
    setTimeout(() => resolve('I did something'), 3000)
  })
}
async function doSomething() {
  console.log(await doSomethingAsync())
}
console.log('Before')
doSomething()
console.log('After')
```

上面的代码会在浏览器控制台里打印出：

```
Before
After
I did something //after 3s
```

串联多个异步函数

链式调用异步函数很简单，而且比原始的 promises 更易读：

```
function promiseToDoSomething() {
  return new Promise((resolve)=>{
    setTimeout(() => resolve('I did something'), 10000)
  })
}
async function watchOverSomeoneDoingSomething() {
  const something = await promiseToDoSomething()
  return something + ' and I watched'
}
async function watchOverSomeoneWatchingSomeoneDoingSomething() {
  const something = await watchOverSomeoneDoingSomething()
  return something + ' and I watched as well'
}
watchOverSomeoneWatchingSomeoneDoingSomething().then((res) => {
  console.log(res)
})
```

共享内存和原子

WebWorkers 用来在浏览器里构建多线程程序。

他们通过事件（events）提供一种通信协议。从 ES2017 开始，你可以通过 `SharedArrayBuffer` 在 web workers 和他们的构造者中间创建一个共享内存数组。

由于我们不知道写入一个共享内存的传递部分需要多长时间，因此原子（**Atomics**）是一种在读取值时执行操作的方法，并且完成了所有类型的写入操作。

更多内容可以在[这个提案](#)中找到，该提案已经被实现。

ES2018 改进

ES2018 是最新的 ECMAScript 标准。

它引入了什么新东西呢？

Rest/Spread 属性

ES6 引入了针对数组解构的剩余元素：

```
const numbers = [1, 2, 3, 4, 5]
[first, second, ...others] = numbers
```

和扩展元素：

```
const numbers = [1, 2, 3, 4, 5]
const sum = (a, b, c, d, e) => a + b + c + d + e
const sum = sum(...numbers) // 勘误：此处不能对 sum 重新赋值，感谢 [森蓝情\](https://juejin.im)
```

ES2018 为对象带来了同样的功能。

剩余属性：

```
const { first, second, ...others } = { first: 1, second: 2, third: 3, fourth: 4, fifth: 5 }
first // 1
second // 2
others // { third: 3, fourth: 4, fifth: 5 }
```

扩展属性允许通过组合在 spread 运算符之后传递的对象的属性来创建新对象：

```
const items = { first, second, ...others }
items // { first: 1, second: 2, third: 3, fourth: 4, fifth: 5 }
```

异步迭代

新的构造函数 `for-await-of` 允许你使用异步可迭代对象作为循环迭代：

```
for await (const line of readLines(filePath)) {
```

```
    console.log(line)
}
```

因为这里用了 `await`，所以你只能在 `async` 函数内部使用它，就像一个普通的 `await`（参考 `async/await`）。

Promise.prototype.finally()

当一个 `promise` 完成（fulfilled），它会一个接一个的调用 `then()` 方法。如果在这个过程中出现了任何错误，`then()` 方法会被跳过，进而执行 `catch()` 方法。

无论 `promise` 执行成功还是失败，`finally()` 都会运行其中的代码：

```
fetch('file.json')
  .then(data => data.json())
  .catch(error => console.error(error))
  .finally(() => console.log('finished'))
```

正则表达式改进

RegExp 后行断言：根据前面的内容匹配字符串。

这是一个先行断言：使用 `?=` 匹配特定的子字符串：

```
/Roger(?=Waters)/
/Roger(?= Waters)/.test('Roger is my dog') //false
/Roger(?= Waters)/.test('Roger is my dog and Roger Waters is a famous musician') //true
```


`?!` 执行相反的操作，匹配字符串后面没有特定的子字符串：

```
/Roger(?!Waters)/
/Roger(?! Waters)/.test('Roger is my dog') //true
/Roger(?! Waters)/.test('Roger Waters is a famous musician') //false
```

先行断言使用 `?=` 标识符，它已经可用了。

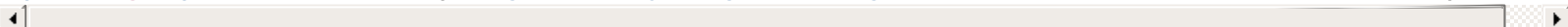
后行断言，新功能，使用 `?<=`。

```
/(?<=Roger) Waters/
/(?<=Roger) Waters/.test('Pink Waters is my dog') //false
/(?<=Roger) Waters/.test('Roger is my dog and Roger Waters is a famous musician') //true
```



使用 `?<!` 否定后行断言：

```
/(?<!Roger) Waters/
/(?<!Roger) Waters/.test('Pink Waters is my dog') //true
/(?<!Roger) Waters/.test('Roger is my dog and Roger Waters is a famous musician') //false
```



Unicode 属性转义 `\p{...}` 和 `\P{...}`

在一个正则表达式里，你可以使用 `\d` 匹配任意数字，`\s` 匹配不包括空格的任意字符，`\w` 匹配任意字母数字字符等等。

这个新功能通过引入 `\p{}` 和 `\P{}` 将此概念扩展到所有 Unicode 字符。

任何 unicode 字符都有一组特定的属性。举个例子，`Script` 决定了语言家族，`ASCII` 是 ASCII 字符串 `true` 的布尔值，等等。你可以把这个特性放到大括号中，正则表达式会检查是否为真：

```
/^\p{ASCII}+$/u.test('abc')    //
/^\p{ASCII}+$/u.test('ABC@')    //
/^\p{ASCII}+$/u.test('ABC' )    //
```

`ASCII_Hex_Digit` 是另一个布尔属性，它会检查字符串是否包含有效的十六进制数字：

```
/^\p{ASCII_Hex_Digit}+$/u.test('0123456789ABCDEF') //
/^\p{ASCII_Hex_Digit}+$/u.test('h')                  //
```

还有很多布尔属性，你只用把他们的名字放在大括号中就可以用了，比如：`Uppercase`，`Lowercase`，`White_Space`，`Alphabetic`，`Emoji` 等等：

```
/^\p{Lowercase}+$/u.test('h') //
/^\p{Uppercase}+$/u.test('H') //
/^\p{Emoji}+$/u.test('H')      //
/^\p{Emoji}+$/u.test('')       //
```

除了这些二进制属性之外，你还可以检查任意 unicode 字符是否匹配特定的值。在这个例子中，我检查了字符串是希腊语还是拉丁字母：

```
/^\p{Script=Greek}+$/u.test('ελληνικά') //
/^\p{Script=Latin}+$/u.test('hey')      //
```

了解更多内容可以阅读[提案](#)。

命名捕获组

在 ES2018 里匹配到的组可以绑定一个名字，而不是仅在结果数组中绑定一个插槽：

```
const re = /(<year>\d{4})-(<month>\d{2})-(<day>\d{2})/
const result = re.exec('2015-01-02')
// result.groups.year === '2015';
// result.groups.month === '01';
// result.groups.day === '02';
```

正则表达式 ‘s’

`s` 是单行的缩写，可以和 `.` 一起匹配新行，没有它，点标识符不会匹配新行：

```
/hi.welcome/.test('hi\nwelcome') // false
/hi.welcome/s.test('hi\nwelcome') // true
```

编程风格

JavaScript 编程风格是编写 JavaScript 时的一系列规范。

编程风格是你和你的团队达成的协议，用来保证项目的一致性。如果你没有团队，那么它就是你自己的协定，应该始终保证你的代码符合你的标准。

对代码编写格式有固定的规则有很大的好处，可以使代码更易读和更易管理。

流行的风格指南

在 JavaScript 里有很多风格指南，其中有两个最为常见的：

- [The Google JavaScript Style Guide](#)
- [The Airbnb JavaScript Style Guide](#)

你可以选择使用其中一种或者制定自己的代码风格。

和你的项目保持一致的风格

即使你有自己更喜欢的代码风格，在团队协作中你也要遵守团队项目风格。

Github 上的每个开源项目都可能会有一系列规则，你参与的另一个项目可能会有完全不同的规则。

[Prettier](#) 是格式代码的强大工具，你应该试试它。

本指南使用的规则

我们一直使用最新的 ES 版本，在旧版浏览器中使用 Babel。

- 缩进：用空格代替 tabs，缩进两个空格。
- 分号：不要使用分号。
- 每行长度：如果可能，尽力保证每行不超过 80 个字符。
- 行内注释：在代码里使用行内注释，只在文件里使用块级注释。
- 不要有无用代码：不要留下旧代码，“仅仅以防万一”它将来还有用。保证只有现在需要的代码，版本控制或者你的笔记应用就是为此而生的。
- 只在需要时注释：不要添加不能帮助理解的注释。如果代码自身有良好的变量、函数命名和 JSDoc 函数注释，不要添加注释。
- 变量声明：避免污染全局变量，永远不要使用 var。默认使用 const，只在需要重绑定变量时使用 let。

- **常量**：用大写字母声明所有常量，用 `_` 分割 `VARIABLE_NAME`。
- **函数**：使用箭头函数，除非你有使用常规函数的理由，比如在对象方法或者构造函数中，需要确定 `this` 的指向。用 `const` 声明函数，并在可能的情况使用隐式返回。使用嵌套函数隐藏助手函数的多余代码会让你觉得无拘无束。

```
const test = (a, b) => a + b
const another = a => a + 2
```

- **名字**：函数名，变量名和方法名以小写字母开始（除非他们是私有且只读的）的驼峰命名，只有构造函数和类名应该以大写字母开始。如果你使用一个有特定规范的框架，根据要求改变你的习惯。文件名应该全部小写，用 `-` 分割。
- **特定语句格式和规则**：

if

```
if (condition) {
  statements
}
if (condition) {
  statements
} else {
  statements
}
if (condition) {
  statements
} else if (condition) {
  statements
} else {
  statements
}
```

for：始终在初始化时缓存遍历对象的长度，不要把它插入到条件语句中。避免使用 `for in` 表达式，除了和 `.hasOwnProperty()` 配合使用，首选 `for of`：

```
for (initialization; condition; update) {
  statements
}
```

while

```
while (condition) {
  statements
}
```

do

```
do {
  statements
} while (condition);
```


switch

```
switch (expression) {  
  case expression:  
    statements  
  default:  
    statements  
}
```

try

```
try {  
  statements  
} catch (variable) {  
  statements  
}  
try {  
  statements  
} catch (variable) {  
  statements  
} finally {  
  statements  
}
```

- 空格：机智的通过空格改善代码可读性：在关键字和 (中插入一个空格；在二进制运算符 (+, -, /, *, && ...) 前面和后面插入一个空格；在语句内，每个 ; 后插入一个空格将每个语句分开；在每个 , 后插入一个空格。
- 新行：使用新行分隔执行逻辑相关操作的代码块。
- 引号：使用单引号 ' 代替双引号 "。双引号是 HTML 的标准属性，所以使用单引号避免在处理 HTML 字符串时可能遇到的问题。适当时使用模板字符串而不是变量插值。

词汇结构

现在我们将深入探讨JavaScript的构建模块：unicode，分号，空格，大小写敏感，注释，字面量，标识符和保留字。

Unicode

JavaScript 用 Unicode 编写。这意味着你可以用 Emojis 作为变量名。更重要的是，你可以用任何语言书写标识符，比如日文或者中文，[相关规则](#)。

分号

JavaScript 语法和 C 语言类似，你可能会在很多示例代码里看到每行代码末尾都有分号。

分号不是强制性的，JavaScript 不使用分号没有任何问题。现在很多开发者，特别是从不需要分号的语言转过来的那部分开始避免使用分号。

你只需要避免一些奇怪的做法，比如将语句分割成多行：

```
return
variable
```

或者是在一行前以 `[` 或者 `(` 开头。这样你在 99.9% 的时间里都是安全的（你的 linter 也会警告你）。

这取决于个人喜好，最近我决定再也不加无用的分号，所以在这篇文章里你看不到任何分号。

空格

JavaScript 不认为空格有意义。空格和断行可以凭你的喜好添加，即使理论上是可行的。

在实践中，你很可能遵守良好的风格和人们习以为常的规则，强制使用 linter 或者 *Prettier* 这样的风格工具。

比如，我喜欢缩进两个字符。

大小写敏感

JavaScript 是大小写敏感的。变量 `something` 和 `Somethin` 是不同的。这在任何标识符里都是一致的。

注释

在 JavaScript 里，有两种注释方式：

```
/* */
//
```

第一个可以进行多行注释并且需要闭合。

第二个会注释掉当前行位于其右侧的所有内容。

字面量和标识符

我们将源码里的值定义为字面量，例如数字，字符串，布尔值或者更高级的构造，像对象字面量或者数组字面量：

```
5
'Test'
true
['a', 'b']
{color: 'red', shape: 'Rectangle'}
```

一个标识符可以用来识别一个变量，一个函数，一个对象。它可以用一个美元符号 `$` 或者一个下划线 `_` 开头，它也可以包含数字。使用 Unicode，字母可以是任何被允许的字符，比如 emoji `:smile:`。

Test
test
TEST
_test
Test1
\$test

美元符号通常被用来区分 DOM 元素。

保留字

你不能使用下列标识符，因为他们是语言保留字。

break
do
instanceof
typeof
case
else
new
var
catch
finally
return
void
continue
for
switch
while
debugger
function
this
with
default
if
throw
delete
in
try
class
enum
extends
super
const
export
import
implements
let
private
public
interface
package
protected

```
static
yield
```

变量

变量是一个标识符绑定了一个字面量，所以你可以在后续的代码里引用和使用它。我们将会学习如果在 JavaScript 里声明一个变量。

介绍 JavaScript 变量

变量是一个标识符绑定了一个字面量，所以你可以在后续的代码里引用和使用它。在 JavaScript 里，变量没有绑定任何类型。即使你为一个变量绑定了一个特定的类型，你也可以在后面重新绑定其它任何类型，这不会造成类型错误或者其它问题。

这也是为什么有时候提到 JavaScript 会被认为是“无类型的”。

变量必须在你用到之前声明。有三种声明方法：使用 `var`, `let` 或者 `const`。这三种方式的不同在于后续你如何和这个变量进行交互。

使用 var

直到 ES2015, `var` 是定义变量的唯一方法。

```
var a = 0
```

如果你忘了加 `var`，你将给未声明的变量绑定值，这个结果可能会有不同：在现代环境里，开启严格模式，这样会报错。在旧环境里（或者关闭严格模式），这样会初始化一个变量并把它绑定到全局对象。

如果你声明变量时没有初始化，它会获得一个 `undefined` 值直到你为它绑定一个值。

```
var a //typeof a === 'undefined'
```

你可以重复声明同一个变量，重写它的值：

```
var a = 1
var a = 2
```

你可以一次性声明多个变量：

```
var a = 1, b = 2
```

代码的作用域是变量可见的范围。

在任何函数外部通过 `var` 初始化的变量会绑定到全局对象，拥有全局作用域，在任何位置都可用。在函数内部通过 `var` 初始化的变量会绑定在这个函数内，仅在函数内部可用，就和函数的参数一样。

重要的是要了解一个块（用一对花括号区分）没有定义一个新的作用域。新作用域只会随着函

数的创建被创建，因为 `var` 没有块级作用域，只有函数作用域。

在函数内部，其中定义的任何变量在函数代码里都是可见的，即使是定义在尾部的变量也会在函数头部被引入，这是因为 JavaScript 会自动将所有变量移动到顶部（即变量提升）。为了避免造成困扰，始终在函数头部声明变量。

使用 `let`

`let` 是 ES2015 里引入的新特性，本质上是拥有块级作用域的 `var`。他的作用域被限制在定义它的块，语句或者表达式，以及所有包含的内部块。

现代 JavaScript 开发者可能会只用 `let` 而完全放弃使用 `var`。

如果 `let` 看起来是个模糊的术语，就看看 `let color = 'red'`，让颜色变成红色，这样容易明白得多。

在函数外部使用 `let`，和 `var` 相反，没有创建一个全局变量。

使用 `const`

用 `var` 或者 `let` 声明的变量可以在后面进行更改和重绑定。一旦 `const` 被初始化，它的值将不能被修改，也不能绑定其它的值。

```
const a = 'test'
```

我们不能为 `a` 绑定不同的值。然而，如果 `a` 是一个提供改变其内容的方法的对象，我们仍然可以改变它。

`const` 不提供不变性，只能确保不会更改引用。

`const` 和 `let` 一样，提供块级作用域。

现在 JavaScript 开发者可能会选择 `const` 作为声明将来不需要重新绑定的变量的标识符。

为什么？因为我们应该始终使用最简单的构造变量避免发生错误。

类型

有时你会读到 JS 是无类型的，但是这是不正确的。你可以为一个变量绑定不同的类型是确实存在的，但是 JavaScript 是有类型的。它提供了基本类型和对象类型。

基本类型

原始类型有：

- Numbers
- Strings

- Booleans

还有两个特别类型：

- null
- undefined

让我们在下个章节深入了解他们。

Numbers

在语言内部，JavaScript 只有一种数字类型：所有数字都是浮点型。

一个数字字面量是源码里表示的数字，amd 取决于它的编写方式，它可以是整型字面量或者是浮点型字面量。

整型：

```
10
5354576767321
0xCC //hex
```

浮点型：

```
3.14
.1234
5.2e4 //5.2 * 10^4
```

strings

字符串类型是一系列字符。在源码里，它被定义成字符串字面量，用单引号或者双引号包裹。

```
'A string'
"Another string"
```

字符串可以通过反斜杠跨越多行：

```
"A \
string"
```

字符串可以包含转义序列，在打印字符串时解释，例如 用来换行。当你需要防止字符被误解为闭合引号时，反斜杠也很有用：

```
'I\'m a developer'
```

字符串可以用 + 操作符拼接：

```
"A " + "string"
```

模板字符串

在 ES2015 引入，模板字符串允许用更强大的方法定义一个字符串字面量。

```
`a string`
```

你可以嵌入任何 JavaScript 表达式并替换执行后的结果：

```
`a string with ${something}`  
`a string with ${something+somethingElse}`  
`a string with ${obj.something()}`
```

你可以很容易的获得多行字符串：

```
`a string  
with  
${something}`
```

Booleans

对于布尔型，JavaScript 有两个保留字：true 和 false。大多数比较运算符 == === > < 等等都返回其中的一个。

if，while 语句和其它控制结构使用布尔值决定程序的流程。

他们不仅接受 true 和 false，也会接受 **truthy** 和 **falsy** 的值。

Falsy，值被解释为 **false**：

```
0  
-0  
NaN  
undefined  
null  
'' //empty string
```

其余的都属于 **truthy**。

null

null 是一个特殊值，表示没有值。这在其它语言里也是普遍的观念，比如 nil 或者 Python 里的 None。

undefined

undefined 表示变量还没有初始化，值为空。

函数里没有 return 值时就会返回 undefined。当函数参数没有被调用者赋值时，也是 undefined。

检测一个值是否为 undefined，你可以用这个方法：

```
typeof variable === 'undefined'
```

对象类型

所有不是基本类型的都是对象类型。

函数，数组和我们叫做对象的都是对象类型。它们本身是特殊的，但是它们继承了 objects 的很多特性，比如具有特性和使特性生效的方法。

表达式

表达式是可以执行并解析为值的代码单元。JS 中的表达式可以分为几类。

算术表达式

这个分类下所有表达式对数字进行求值：

```
1 / 2
i++
i -= 2
i * 2
```

字符串表达式

对字符串求值：

```
'A ' + 'string'
'A ' += 'string'
```

原始表达式

这个分类下，是变量引用，字面量和常量：

```
2
0.02
'something'
true
false
this //the current object
undefined
i //where i is a variable or a constant
```

也有一些语言关键字：

```
function
class
function* //the generator function
yield //the generator pauser/resumer
yield* //delegate to another generator or iterator
```

```
async function* //async function expression
await //async function pause/resume/wait for completion
/pattern/i //regex
() // grouping
```

数组和对象初始表达式

```
[] //array literal
{} //object literal
[1,2,3]
{a: 1, b: 2}
{a: {b: 1}}
```

逻辑表达式

逻辑表达式使用逻辑运算符，获得一个布尔值：

```
a && b
a || b
!a
```

Left-hand-side 表达式

```
new //create an instance of a constructor
super //calls the parent constructor
...obj //expression using the spread operator
```

属性访问表达式

```
object.property //reference a property (or method) of an object
object[property]
object['property']
```

对象构造表达式

```
new object()
new a(1)
new MyRectangle('name', 2, {a: 4})
```

函数定义表达式

```
function() {}
function(a, b) { return a * b }
(a, b) => a * b
a => a * 2
() => { return 2 }
```

调用表达式

```
a.x(2)
```



```
window.resize()
```

原型继承

JavaScript 在流行编程语言领域是一个非常独特的存在，原因在于对原型继承的使用。让我们看看这是什么意思。

大多数面向对象语言都是基于类继承模式，JavaScript 基于原型继承。

这是什么意思？

每一个 JavaScript 对象都有一个特性，称作 prototype，指向不同的对象。

这个不同的对象就是原型对象。

我们的对象会继承原型对象的特性和方法。

假设你通过对象字面量语法创建了一个对象：

```
const car = {}
```

或者通过 new Object 创建：

```
const car = new Object()
```

无论哪一种方法，car 的原型都是 Object。

初始化一个数组，也是一个对象：

```
const list = []  
//or  
const list = new Array()
```

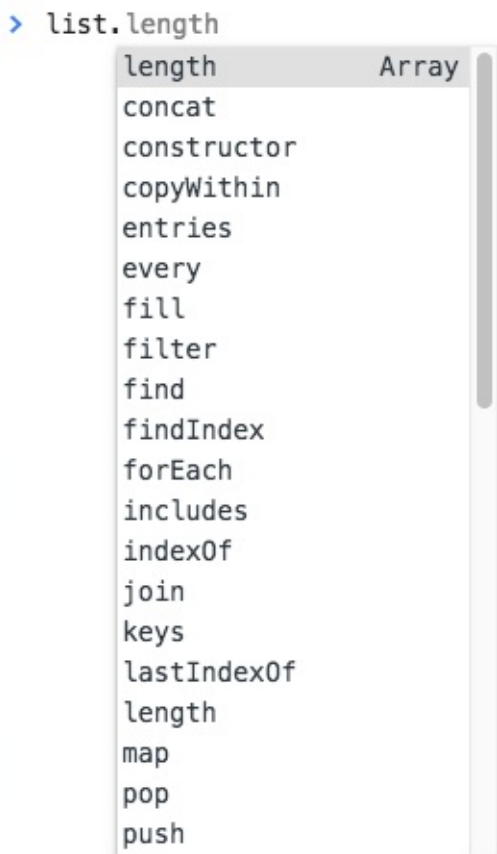
这个的原型是 Array。

你可以通过 __proto__ 属性验证：

```
car.__proto__ == Object.prototype //true  
car.__proto__ == new Object().__proto__ //true  
list.__proto__ == Object.prototype //false  
list.__proto__ == Array.prototype //true  
list.__proto__ == new Array().__proto__ //true
```

这里的 __proto__ 属性不是标准的但是在浏览器里被广泛实现。一个更可靠的获得原型的方法是 Object.getPrototypeOf(new Object())。

原型上所有的特性和方法在具有该原型的对象上都是可用的：



`Object.prototype` 是所有对象的原型。

```
Array.prototype.__proto__ == Object.prototype
```

如果你想知道 `Object.prototype` 的原型是什么，它没有原型。这是一片特殊的雪花。❄

上面的例子是原型链的示例。

我可以创建一个扩展 `Array` 和任何实例化对象的对象，这个对象的原型链上会有 `Array` 和 `Object`，它会继承所有祖先的特性和方法。

除了用 `new` 创建一个对象，或者用对象和数组的字面量语法，你还可以用 `Object.create()` 实例化一个对象。

第一个参数作为对象的原型：

```
const car = Object.create({})
const list = Object.create(Array)
```

你可以用 `isPrototypeOf()` 方法检查这个对象的原型：

```
Array.isPrototypeOf(list) //true
```

注意，因为你可以这样实例化数组：

```
const list = Object.create(Array.prototype)
```

因此，`Array.isPrototypeOf(list)` 等于 `false`，而 `Array.prototype.isPrototypeOf(list)` 等于 `true`。

类

在 2015 年发布的 ECMAScript 6 (ES6) 标准引入了类。

在那之前，JavaScript 只能通过一个十分奇怪的方法实现继承。那就是原型继承，我认为这很神奇，和其它的流行语言都不同。

从 Java 或者 Python 或者其它语言来的人们很难理解原型继承的复杂性，因此，ECMAScript 委员会决定基于此引入语法糖，类似于其它语言的实现方法。

这很重要：JavaScript 底层仍然和之前一样，你可以以一种普适的方法访问原型对象。

定义类

类看起来是这样：

```
class Person {
  constructor(name) {
    this.name = name
  }
  hello() {
    return 'Hello, I am ' + this.name + ' .'
  }
}
```

类有一个标识符，可以让我们通过 `new ClassIdentifier()` 创建一个新对象。

当这个对象被实例化，会调用 `constructor` 方法，可以传递任何参数。

一个类可以有很多方法，在这个例子中 `hello` 是一个方法，所有从这个类派生的对象都可以调用它：

```
const flavio = new Person('Flavio')
flavio.hello()
```

类继承

类可以扩展另一个类，使用该类初始化的对象继承这两个类的所有方法。

如果继承的类和更高一级的类有同名的方法，则按最近优先的原则调用：

```
class Programmer extends Person {
  hello() {
    return super.hello() + ' I am a programmer.'
  }
}
```

```
const flavio = new Programmer('Flavio')
flavio.hello()
```

上面的程序会打印出：“*Hello, I am Flavio. I am a programmer.*”。

类没有显式地声明变量，但是必须要在构造函数中初始化变量。

在类中，你可以通过调用 `super()` 引入父类。

静态方法

通常来讲，方法定义在实例上，而不是类上。

而静态方法能在类上执行：

```
class Person {
  static genericHello() {
    return 'Hello'
  }
}
Person.genericHello() //Hello
```

私有方法

JavaScript 没有内置定义私有方法或者保护方法的手段。有解决方法，但是这里将不再赘述。

Getters 和 Setters

你可以添加 `get` 或者 `set` 前缀创建一个 `getter` 和 `setter`，用哪一种方法取决于你想做什么：访问变量或者修改变量的值。

```
class Person {
  constructor(name) {
    this.name = name
  }
  set name(value) {
    this.name = value
  }
  get name() {
    return this.name
  }
}
```

如果属性只有一个 `getter`，这个属性就不能被设置，所有的修改都会被忽略：

```
class Person {
  constructor(name) {
    this.name = name
  }
  get name() {
    return this.name
  }
}
```

```
}  
}
```

如果属性只有一个 setter，你可以随意修改它的值，但是不能从外部访问它：

```
class Person {  
  constructor(name) {  
    this.name = name  
  }  
  set name(value) {  
    this.name = value  
  }  
}
```

异常

当代码发生意外错误时，JavaScript 习惯通过 exceptions 处理错误。

创建异常

通过关键字 throw 创建异常：

```
throw value
```

value 可以是任意的 JavaScript 值，包括字符串，数字或者一个对象。

当 JavaScript 代码执行到这一行，就会暂停正常的程序流程，控制会跳转到最近的异常处理器。

处理异常

异常处理器是 try/catch 语句。

try 代码块内抛出的异常都会在相应的 catch 中处理：

```
try {  
  //lines of code  
} catch (e) {  
  
}
```

在这个例子中，e 代表异常值。

你可以添加多个处理器，它们可以捕捉不同的错误。

finally

为了完成整个代码语句，JavaScript 有另一个语句，finally，不管程序流程如何，异常是否被处理，是否产生异常，其中的代码都会执行：

```
try {  
    //lines of code  
} catch (e) {  
  
}  
  
finally {  
  
}
```

你也可以在没有 `catch` 的情况下使用 `finally`，以清除可能在 `try` 中打开的任何资源，比如文件或者网络请求：

```
try {  
    //lines of code  
} finally {  
}
```

嵌套 `try` 代码块

`try` 可以被嵌套使用，异常始终在最近的 `catch` 块中被处理：

```
try {  
    //lines of code  
    try {  
        //other lines of code  
    } finally {  
        //other lines of code  
    }  
} catch (e) {  
  
}
```

如果在内部的 `try` 中发现异常，它会在外部的 `catch` 中被处理。

分号

JavaScript 分号是可选的，我个人倾向于不写分号，但是很多人喜欢带上分号。

分号在 JavaScript 社区中产生了很大的分歧。一些人不论在什么情况都很喜欢用它，而另一部分则相反。

在用了几年分号之后，2017 年秋天我决定尝试避免使用分号，我通过 Prettier 自动移除代码中的分号，除非有特定的代码构造需要它。

现在我很自然就会避免分号，我认为代码看起来更优秀，也更易读了。

因为 JavaScript 不强制要求使用分号，所以万事皆有可能。当一个位置需要分号，它会自动添加。

执行此操作的过程成为自动插入分号。

了解使用分号的规则非常重要，可以避免编写与预期行为不一致的错误代码。

JavaScript 自动插入分号规则

在解释源码期间，发现一下特定情况，JavaScript 解释器会自动添加分号：

- 当下一行的起始代码打断了当前行（代码可能是多行的）
- 当下一行代码以 `}` 开头，闭合了当前代码块
- 当执行到源代码末尾
- 当前行有 `return` 语句
- 当前行有 `break` 语句
- 当前行有 `throw` 语句
- 当前行有 `continue` 语句

代码行为不一致的例子

基于上面的规则，这有一些其它例子，比如：

```
const hey = 'hey'
const you = 'hey'
const heyYou = hey + ' ' + you
['h', 'e', 'y'].forEach((letter) => console.log(letter))
```

你会看到错误 `Uncaught TypeError: Cannot read property 'forEach' of undefined`，因为规则 1 尝试把代码解释为：

```
const hey = 'hey';
const you = 'hey';
const heyYou = hey + ' ' + you['h', 'e', 'y'].forEach((letter) => console.log(letter))
```

还有：

```
(1 + 2).toString() // 3
```

```
const a = 1
const b = 2
const c = a + b
(a + b).toString()
```

相反，上面的代码抛出 `TypeError: b is not a function` 异常，因为 JavaScript 尝试把它解释为：

```
const a = 1
const b = 2
const c = a + b(a + b).toString()
```

另一个关于规则 4 的例子：

```
(( ) => {
  return
```

```
{
  color: 'white'
}
})();
```

你希望这个立即执行函数可以返回一个包含 `color` 属性的对象，但是没有。它返回的是 `undefined`，因为 JavaScript 会在 `return` 后插入一个分号。

正确的做法是把花括号放在 `return` 的后面：

```
(() => {
  return {
    color: 'white'
  }
})();
```

你认为这个代码会弹出 0：

```
1 + 1
-1 + 1 === 0 ? alert(0) : alert(2)
```

但是它显示的是 2，因为规则 1 把它解释成：

```
1 + 1 -1 + 1 === 0 ? alert(0) : alert(2)
```

小心点。有些人对待分号很偏执，我是无所谓，这个工具既然给了我们不用分号的选项，我们就应该避免使用分号。我不是在推荐什么，你自己选择。

我们只需要稍微注意一些特殊情况，即使这些很少会出现在你的代码中。

记住这些规则：

- 小心使用 `return` 语句。如果你想返回一些东西，应该和 `return` 放在同一行（`break`, `throw`, `continue` 同理）
- 永远不要用圆括号开头，这可能会和上一行连接形成函数调用或者数组元素引用

最后，始终测试你的代码，确保它是你想要的。

引号

现在我们将谈谈 JavaScript 里的引号和特殊的功能。

JavaScript 允许三种形式的引号：

- 单引号
- 双引号
- 反引号

前两种差不多：


```
const test = 'test'
const bike = "bike"
```

使用其中一种没有什么大的差别。唯一的不同在于必须转移用于分隔字符串的引号字符：

```
const test = 'test'
const test = 'te\'st'
const test = 'te"st'
const test = "te\'st"
const test = "te"st"
```

有很多不同的风格指南，建议始终使用其中一种。

我个人一直都用单引号，只在 HTML 中使用双引号。

反引号是另一个选项，在 2015 年的 ES6 中引入。

它们都有一个特殊的功能 - 允许多行字符串。

多行字符串可能是常规字符串加上转义字符：

```
const multilineString = 'A string\nnon multiple lines'
```

使用反引号（键盘左上角的数字 1 键），你可以不用转义字符：

```
const multilineString = `A string
on multiple lines`
```

不仅如此。你可以用 `${}` 解析变量：

```
const multilineString = `A string
on ${1+1} lines`
```

这也叫做模板字符串。

模板字符串

ES2015，即ES6引入，模板字符串提供了一个新的方式展示字符串，一些新的有趣的构造已经被广泛使用。

与 ES5 及更低版本相比，模板字符串作为 ES2015/ES6 的功能允许你用更新颖的方式处理字符串。

乍一看，反引号语法和单引号，双引号相比很简单：

```
const a_string = `something`
```

它们和普通的字符串相比提供了更多的功能，特别是：

- 提供了优秀的语法定义多行字符串

- 提供了简单的方式解释字符串中的变量和表达式
- 允许通过模板标签创建 DSLs

让我们仔细看看这些功能。

多行字符串

在 ES6 之前，创建多行字符串需要在每行行尾添加 \ 字符：

```
const string = 'first part \
second part'
```

这样看是两行字符串，但是渲染时会变成一行：

```
"first part second part"
```

为了能够渲染多行字符串，你需要在每行末尾添加 \n：

```
const string = 'first line\n \
second line'
```

或者

```
const string = 'first line\n' +
  'second line'
```

模板字符串使创建多行字符串变得很简单：

使用反引号开始多行字符串，你只用按下回车键就能创建一个新行，不用什么特别的字符，像这样：

```
const string = `Hey
this
```

```
string
is awesome!`
```

注意：空格也是有意义的，这样做：

```
const string = `First
  Second`
```

将会创建一个这样的字符串：

```
First
    Second
```

解决这个问题有一个简单的办法：让第一行是空的，在闭合反引号后面加上 `trim()` 方法，这样会忽略掉所有第一个字符前的空格：

```
const string = `
```

```
First
Second`.trim()
```

插值

模板字符串提供了一个在字符串中插入变量和表达式的简单方法。

你可以使用 `${...}` 语法：

```
const var = 'test'
const string = `something ${var}` //something test
```

在 `${}` 里你可以添加任何东西，甚至是表达式：

```
const string = `something ${1 + 2 + 3}`
const string2 = `something ${foo() ? 'x' : 'y' }`
```

模板标签

模板标签是一个最初听起来可能不那么有用的功能，但实际上，它被很多流行库使用，比如 [Styled Components](#)、[Apollo](#)、[GraphQL](#) 客户端/服务端库，所以理解它的原理也很重要。

在 `Styled Components` 中，模板标签被用来定义 CSS：

```
const Button = styled.button`
  font-size: 1.5em;
  background-color: black;
  color: white;
`;
```

在 `Apollo` 中，模板标签被用来定义 GraphQL 查询表：

```
const query = gql`
  query {
    ...
  }
`;
```

这些例子中高亮的 `styled button` 和 `gql` 模板标签都是函数：

```
function gql(literals, ...expressions) {
}
```

这个函数返回一个字符串，该字符串可以是任何计算结果。

`literals` 是一个包含被表达式插值标记的模板标签数组。

`expressions` 包含所有的插值。

比如上面的例子：

```
const string = `something ${1 + 2 + 3}`
```

`literals` 数组包含两项。第一个从 `something` 开始直到遇到第一个插值字符串，第二个是空字符串，是第一个插值末尾（我们只有一个）和整个字符串末尾之间的空格。

一个更复杂的例子是：

```
const string = `something
another ${'x'}
new line ${1 + 2 + 3}
test`
```

在这个例子中，`literals` 数组中第一项是：

```
`something
another `
```

第二项是：

```
`
new line `
```

第三项是：

```
`
test`
```

`expressions` 是包含 `x` 和 `6` 的数组。

传递这些值的函数可以对它们进行任何操作，这也是这个功能的强大之处。

最简单的例子就是赋值插值字符串的功能，通过加入 `literals` 和 `expressions`：

```
const interpolated = interpolate`I paid ${10}€`
```

`interpolate` 是这样：

```
function interpolate(literals, ...expressions) {
  let string = ``
  for (const [i, val] of expressions) {
    string += literals[i] + val
  }
  string += literals[literals.length - 1]
  return string
}
```

JavaScript 函数

现在我们由面及点地了解所有函数功能，帮助你使用它们。

JavaScript 里的一切都是函数。

函数是一个自包含的代码块，定义一次，可以使用无数次。

函数参数是可选的，只能返回一个值。

JavaScript 里的函数也是对象，一个特殊的对象：函数对象。它们的超能力在于它们可以被调用。

另外，函数被称为头等函数（**first class functions**），因为它们可以绑定值，可以传递参数，可以返回一个值。

让我们先从“旧”的 ES6 之前的语法开始。这是一个函数声明式：

```
function dosomething(foo) {  
  // do something  
}
```

现在，在 ES6/ES2015 里，属于常规函数。

函数可以绑定给一个变量（这叫做函数表达式）：

```
const dosomething = function(foo) {  
  // do something  
}
```

命名函数表达式很简单，但在堆栈调用跟踪中很好用，当错误发生时，它会显示这个函数的名字：

```
const dosomething = function dosomething(foo) {  
  // do something  
}
```

ES6/ES2015 引入了箭头函数，在作为参数或者回调的行内函数时很好用：

```
const dosomething = foo => {  
  //do something  
}
```

箭头函数和上面其它函数有一个巨大的差异，我们会在后面的话题中深入它。

参数

一个函数可以有一个或多个参数。

```
const dosomething = () => {  
  //do something  
}
```

```
const dosomethingElse = foo => {  
  //do something  
}
```

```
const dosomethingElseAgain = (foo, bar) => {  
  //do something  
}
```

ES6/ES2015 开始，函数参数可以有默认值：

```
const dosomething = (foo = 1, bar = 'hey') => {  
  //do something  
}
```

这让你不用填满参数也可以调用调用函数：

```
dosomething(3)  
dosomething()
```

ES2018 可以为参数添加尾逗号，帮助减少移动参数时漏掉逗号的 bug（比如把最后一个参数移到中间）：

```
const dosomething = (foo = 1, bar = 'hey') => {  
  //do something  
}  
dosomething(2, 'ho!')
```

你可以用数组包裹所有参数，然后在调用时用扩展运算符展开：

```
const dosomething = (foo = 1, bar = 'hey') => {  
  //do something  
}  
const args = [2, 'ho!']  
dosomething(...args)
```

记住：函数参数是有序的。使用对象作为参数，可以获得参数的名字：

```
const dosomething = ({ foo = 1, bar = 'hey' }) => {  
  //do something  
  console.log(foo) // 2  
  console.log(bar) // 'ho!'  
}  
const args = { foo: 2, bar: 'ho!' }  
dosomething(args)
```

返回值

每个函数都会返回一个值，默认是 undefined。

```
> const dosomething = (foo = 1, bar = 'hey') => {  
  //do something  
}  
< undefined  
  
> dosomething()  
< undefined
```

所有函数都会最后一行代码执行完后终止，或者执行到 `return` 关键字。当 JavaScript 遇到这个关键字时会自动终止函数执行，并把控制权交给调用者。

如果你传递一个值，这个值就会作为函数的返回值。

```
const dosomething = () => {  
  return 'test'  
}  
const result = dosomething() // result === 'test'
```

你只能返回一个值：

为了模拟返回多个值，你可以返回一个对象字面量或一个数组，调用函数时用解构绑定单一值。

使用数组：

```
> const dosomething = () => {  
  return ['Roger', 6]  
}  
const [ name, age ] = dosomething()  
< undefined  
  
> name  
< "Roger"  
  
> age  
< 6
```

使用对象：

```
> const dosomething = () => {  
    return { name: 'Roger', age: 6 }  
}  
const { name, age } = dosomething()  
  
< undefined  
  
> name  
  
< "Roger"  
  
> age  
  
< 6
```

嵌套函数

函数可以定义在另一个函数内部：

```
const dosomething = () => {  
    const dosomethingelse = () => {}  
    dosomethingelse()  
    return 'test'  
}
```

嵌套的函数在外部函数的作用域里，不能在除此之外的位置调用。

对象方法

作为对象特性时，函数作为方法调用：

```
const car = {  
    brand: 'Ford',  
    model: 'Fiesta',  
    start: function() {  
        console.log(`Started`)  
    }  
}  
car.start()
```

箭头函数中的“this”

作为对象方法，箭头函数和常规函数“this”的指向很重要。看看这个例子：

```
const car = {  
    brand: 'Ford',  
    model: 'Fiesta',
```



```
start: function() {  
  console.log(`Started ${this.brand} ${this.model}`)  
},  
stop: () => {  
  console.log(`Stopped ${this.brand} ${this.model}`)  
}  
}
```

这个 `stop()` 方法不会像你预期的工作。

```
> const car = {  
  brand: 'Ford',  
  model: 'Fiesta',  
  start: function() {  
    console.log(`Started ${this.brand} ${this.model}`)  
  },  
  stop: () => {  
    console.log(`Stopped ${this.brand} ${this.model}`)  
  }  
}
```

```
car.start()  
car.stop()
```

Started Ford Fiesta

Stopped undefined undefined

这是因为两种函数风格中的 `this` 不一样。箭头函数中的 `this` 指向封闭的上下文，在这个例子中是 `window` 对象：

```
const car = {
  brand: 'Ford',
  model: 'Fiesta',
  start: function() {
    console.log(this)
    console.log(`Started ${this.brand} ${this.model}`)
  },
  stop: () => {
    console.log(this)
    console.log(`Stopped ${this.brand} ${this.model}`)
  }
}
```

```
car.start()
car.stop()
```

```
▶ {brand: "Ford", model: "Fiesta", start: f, stop: f}
```

```
Started Ford Fiesta
```

```
▶ Window {postMessage: f, blur: f, focus: f, close: f,
, ...}
```

```
Stopped undefined undefined
```

使用 `function()`，`this` 指向宿主对象。

这意味着箭头函数不适合用于对象方法和构造函数（箭头构造函数会在调用中跑出 `TypeError` 错误）。

IIFE, 立即执行函数表达式

IIFE 在声明之后会立即执行：

```
;(function dosomething() {
  console.log('executed')
})()
```

你可以将结果赋值给变量：

```
const something = (function dosomething() {
  return 'something'
})()
```

它们非常方便，因为你无需在定义后单独调用该函数。

函数提升

在执行代码之前，JavaScript 会根据规则将其重新排序。

比如将函数移动到作用域的顶部。这就是这样写合法的原因：

```
> dosomething()  
function dosomething() {  
  console.log('did something')  
}  
  
did something
```

在底层，JavaScript 把这个函数移动到调用语句之前，和其它函数处于同一作用域中：

```
function dosomething() {  
  console.log('did something')  
}  
dosomething()
```

现在，如果你使用命名函数表达式，因为你用了 [variables](#)，事情会有所不同。变量被提升了，但是值没有，也就不是一个函数了。

```
dosomething()  
const dosomething = function dosomething() {  
  console.log('did something')  
}
```

不会工作：

```
dosomething()  
const dosomething = function dosomething() {  
  console.log('did something')  
}
```

```
► Uncaught ReferenceError: dosomething is not defined  
   at <anonymous>:1:1
```

这是因为内部变成了：

```
const dosomething  
dosomething()  
dosomething = function dosomething() {  
  console.log('did something')
```

```
}
```

这在 `let` 声明，`var` 声明也一样不会工作，但是抛出的错误不同：

```
> dosomething()
const dosomething = function dosomething() {
  console.log('did something')
}
```

```
✖ ▶ Uncaught ReferenceError: dosomething is not defined
   at <anonymous>:1:1
```

```
> dosomething2()
var dosomething2 = function dosomething() {
  console.log('did something')
}
```

```
✖ ▶ Uncaught TypeError: dosomething2 is not a function
   at <anonymous>:1:1
```

这是因为 `var` 声明被提升并初始化值为 `undefined`，而 `const` 和 `let` 仅仅只会被提升。

箭头函数

箭头函数是 ES6/ES2015 中最重要的改变，现在被广泛使用。它们和常规函数不同，下面我们看看为什么。

在上面我已经介绍了箭头函数，但是它们很重要，所以单独介绍一下它们。

箭头函数在 ES6/ES2015 中引入，它们的存在永远地改变了 JavaScript 代码的写法（和工作）。

在我的观点中，这个改变很受欢迎，以致于现代代码很少用 `function` 关键字。

在视觉上，它是个受欢迎和简单的变化，让你可以用更简单的语法写一个函数，从：

```
const myFunction = function foo() {
  //...
}
```

变成：

```
const myFunction = () => {
  //...
}
```

如果函数体只有一个语句，你可以忽略花括号，然后把所有内容写在一行：

```
const myFunction = () => doSomething()
```

在圆括号中传递参数：

```
const myFunction = (param1, param2) => doSomething(param1, param2)
```

如果只有一个参数，你可以忽略圆括号：

```
const myFunction = param => doSomething(param)
```

多谢这个语法，箭头函数鼓励使用短函数。

隐性返回

箭头函数可以隐性返回值：不用使用 `return` 关键字返回。

它在函数体只有一个语句时有效：

```
const myFunction = () => 'test'
```

```
myFunction() //'test'
```

另一个例子，返回一个对象（记住用圆括号包裹返回值，避免解释器把它看作函数体）：

```
const myFunction = () => ({value: 'test'})
```

```
myFunction() //{value: 'test'}
```

箭头函数中 `this` 如何工作

`this` 是一个很难掌握的理念，上下文造成了它的不同，也受 JavaScript 模式（是否是严格模式）的影响。

理清这个概念很重要，因为箭头函数的表现和常规函数不同。

当定义了对象里的某一方法，在常规函数里 `this` 指向这个对象，所以你可以：

```
const car = {  
  model: 'Fiesta',  
  manufacturer: 'Ford',  
  fullName: function() {  
    return `${this.manufacturer} ${this.model}`  
  }  
}
```

调用 `car.fullName()` 会返回 `Ford Fiesta`。

箭头函数的 `this` 继承自执行上下文。箭头函数根本不会绑定 `this`，所以它的值会在调用栈里查询，所以在这个代码里 `car.fullName()` 无意义，然后返回 `undefined undefined`：

```
const car = {  
  model: 'Fiesta',
```

```

    manufacturer: 'Ford',
    fullName: () => {
      return `${this.manufacturer} ${this.model}`
    }
  }
}

```

因此，箭头函数不适合用于对象方法。

箭头函数也不能用于初始化对象的构造函数，它会抛出 `TypeError`。

当不需要动态上下文，应该使用常规函数替代。

处理事件时也会有问题。DOM 事件监听器会设置 `this` 为目标元素，如果你需要事件处理器的 `this`，应该用常规函数：

```

const link = document.querySelector('#link')
link.addEventListener('click', () => {
  // this === window
})

```

```

const link = document.querySelector('#link')
link.addEventListener('click', function() {
  // this === link
})

```

闭包

这是对闭包话题很友好的介绍，是理解 JavaScript 函数如何工作的关键。

如果你写过 JavaScript 函数，你已经使用了 闭包。这是一个需要理解的关键主题，它会影响你所做的事情。当一个函数运行时，它运行在定义它的作用域中，而不是执行它的位置。

作用域基本上是可见的变量合集。函数会记住它的词法作用域，并且能够访问在父作用域中定义的变量。

简而言之，函数有一整套可以访问的变量。

我们赶紧通过例子验证一下：

```

const bark = dog => {
  const say = `${dog} barked!`
  ;(() => console.log(say))()
}

```

```

bark(`Roger`)

```

这个预期一样打印出：Roger barked!。

如果你想要返回操作，这样做：

```

const prepareBark = dog => {

```



```
const say = `${dog} barked!`
return () => console.log(say)
}

const bark = prepareBark(`Roger`)

bark()
```

这个代码段会打印出 Roger barked!。

最后一个例子，让两种不同的狗 prepareBark：

```
const prepareBark = dog => {
  const say = `${dog} barked!`
  return () => {
    console.log(say)
  }
}

const rogerBark = prepareBark(`Roger`)
const sydBark = prepareBark(`Syd`)

rogerBark()
sydBark()
```

打印：

```
Roger barked!
Syd barked!
```

正如你所见，变量 say 的结果和函数 prepareBark 返回的是相关的。

第二个调用 prepareBark() 时重新定义了新的 say 变量，但是不会影响第一次 prepareBark() 的作用域。

这就是闭包的原理：返回的函数保持作用域里的初始状态。

数组

随着不断地发展，JavaScript 数组有了越来越多的功能，有些时候知道什么使用什么方法是很棘手的。本章节旨在解释截至 2018 年你应该使用什么。

初始化数组

```
const a = []
const a = [1, 2, 3]
const a = Array.of(1, 2, 3)
const a = Array(6).fill(1) //init an array of 6 items of value 1
```

不要使用旧语法（除了类型数组）：

```
const a = new Array() //never use
const a = new Array(1, 2, 3) //never use
```

获取数组长度

```
const l = a.length
```

通过 every 遍历数组

```
a.every(f)
```

遍历 a 直到 f() 返回 false。

通过 some 遍历数组

```
a.some(f)
```

遍历 a 直到 f() 返回 true。

遍历数组并返回函数结果组成的新数组

```
const b = a.map(f)
```

遍历 a，返回每一个 a 元素执行 f() 产生的结果数组。

过滤数组

```
const b = a.filter(f)
```

遍历 a，返回每一个 a 元素执行 f() 都为 true 的新数组。

Reduce

```
a.reduce((accumulator, currentValue, currentIndex, array) => {
  //...
}, initialValue)
```

reduce() 对数组中每一项都调用回调函数，并逐步计算计算结果。如果 initaiValue 存在，accumulator 在第一次迭代时等于这个值。

例子：

foreach

ES6

```
a.forEach(f)
```


遍历 a 执行 f，不能中途停止。

例子：

```
a.forEach(v => {  
  console.log(v)  
})
```

for...of

ES6

```
for (let v of a) {  
  console.log(v)  
}
```

for

```
for (let i = 0; i < a.length; i += 1) {  
  //a[i]  
}
```

遍历 a，可以通过 return 或者 break 中止循环，通过 continue 跳出循环。

@@iterator

ES6

获取数组迭代器的值：

```
const a = [1, 2, 3]  
let it = a[Symbol.iterator]()
```

```
console.log(it.next().value) //1  
console.log(it.next().value) //2  
console.log(it.next().value) //3
```

.entries() 返回一个键值对的迭代器：

```
let it = a.entries()
```

```
console.log(it.next().value) //[0, 1]  
console.log(it.next().value) //[1, 2]  
console.log(it.next().value) //[2, 3]
```

.keys() 返回包含所有键名的迭代器：

```
let it = a.keys()
```

```
console.log(it.next().value) //0
```

```
console.log(it.next().value) //1
console.log(it.next().value) //2
```

数组结束时 `.next()` 返回 `undefined`。你可以通过 `it.next()` 返回的 `value`, `done` 值检测迭代是否结束。当迭代到最后一个元素时 `done` 的值始终为 `true`。

在数组末尾追加值

```
a.push(4)
```

在数组开头添加值

```
a.unshift(0)
a.unshift(-2, -1)
```

移除数组中的值

删除末尾的值

```
a.pop()
```

删除开头的值

```
a.shift()
```

删除任意位置的值

```
a.splice(0, 2) // get the first 2 items
a.splice(3, 2) // get the 2 items starting from index 3
```

不要使用 `remove()`，因为它会留下未定义的值。

移除并插入值

```
a.splice(2, 3, 2, 'a', 'b') //removes 3 items starting from
//index 2, and adds 2 items,
// still starting from index 2
```

合并多个数组

```
const a = [1, 2]
const b = [3, 4]
a.concat(b) // 1, 2, 3, 4
```

查找数组中特定元素

```
a.indexOf()
```

返回匹配到的第一个元素的索引，元素不存在返回 -1。

```
a.lastIndexOf()
```

返回匹配到的最后一个元素的索引，元素不存在返回 -1。

ES6

```
a.find((element, index, array) => {  
  //return true or false  
})
```

返回符合条件的第一个元素，如果不存在返回 undefined。

通常这么用：

```
a.find(x => x.id === my_id)
```

上面的例子会返回数组中 id === my_id 的第一个元素。

findIndex 返回符合条件的第一个元素的索引，如果不存在返回 undefined：

```
a.findIndex((element, index, array) => {  
  //return true or false  
})
```

ES7

```
a.includes(value)
```

如果 a 包含 value 返回 true。

```
a.includes(value, i)
```

如果 a 从位置 i 后包含 value 返回 true。

获取数组的一部分

```
a.slice()
```

数组排序

按字母顺序排序（按照 ASCII 值 - 0-9A-Za-z）：

```
const a = [1, 2, 3, 10, 11]  
a.sort() //1, 10, 11, 2, 3
```

```
const b = [1, 'a', 'Z', 3, 2, 11]
```

```
b = a.sort() //1, 11, 2, 3, Z, a
```

自定义排序

```
const a = [1, 10, 3, 2, 11]
a.sort((a, b) => a - b) //1, 2, 3, 10, 11
```

逆序

```
a.reverse()
```

数组转字符串

```
a.toString()
```

返回字符串类型的值

```
a.join()
```

返回数组元素拼接的字符串。传递参数以自定义分隔符：

```
a.join(',')
```

复制所有值

```
const b = Array.from(a)
const b = Array.of(...a)
```

复制部分值

```
const b = Array.from(a, x => x % 2 == 0)
```

将值复制到本身其它位置

```
const a = [1, 2, 3, 4]
a.copyWithin(0, 2) // [3, 4, 3, 4]
const b = [1, 2, 3, 4, 5]
b.copyWithin(0, 2) // [3, 4, 5, 4, 5]
//0 is where to start copying into,
// 2 is where to start copying from
const c = [1, 2, 3, 4, 5]
c.copyWithin(0, 2, 4) // [3, 4, 3, 4, 5]
//4 is an end index
```

循环

JavaScript 提供了许多种循环方法。这个章节通过小例子和主要属性讲解现代 JavaScript 中的所有循环方法。

for

```
const list = ['a', 'b', 'c']
for (let i = 0; i < list.length; i++) {
  console.log(list[i]) //value
  console.log(i) //index
}
```

- 可以通过 break 中断 for 循环
- 可以通过 continue 跳过当前 for 循环

forEach

ES5 中引入。给你一个数组，你可以通过 `list.forEach()` 遍历它的属性：

```
const list = ['a', 'b', 'c']
list.forEach((item, index) => {
  console.log(item) //value
  console.log(index) //index
})
//index is optional
list.forEach(item => console.log(item))
```

不幸的是，你不能中断这个循环。

do...while

```
const list = ['a', 'b', 'c']
let i = 0
do {
  console.log(list[i]) //value
  console.log(i) //index
  i = i + 1
} while (i < list.length)
```

可以通过 break 中断 do...while 循环：

```
do {
  if (something) break
} while (true)
```

可以通过 continue 跳过当前 do...while 循环：

```
do {
  if (something) continue
  //do something else
} while (true)
```

while

```
const list = ['a', 'b', 'c']
```

```
let i = 0
while (i < list.length) {
  console.log(list[i]) //value
  console.log(i) //index
  i = i + 1
}
```

可以通过 break 中断 while 循环：

```
while (true) {
  if (something) break
}
```

可以通过 continue 跳过当前 while 循环：

```
while (true) {
  if (something) continue
  //do something else
}
```

和 do...while 不同的是 do...while 至少会循环一次。

for...in

遍历对象的所有可迭代属性名。

```
for (let property in object) {
  console.log(property) //property name
  console.log(object[property]) //property value
}
```

for...of

ES2015 中引入了 for...of 循环，它结合了 forEach 的易用性和不能中断的特性：

```
//iterate over the value
for (const value of ['a', 'b', 'c']) {
  console.log(value) //value
}

//get the index as well, using `entries()`
for (const [index, value] of ['a', 'b', 'c'].entries()) {
  console.log(index) //index
  console.log(value) //value
}
```

注意使用 const。这个循环在每次迭代都创建了一个新的作用域，所以我们可以安全的使用它替代 let。

for...in vs for...of

和 `for...in` 不同的是：

- `for...of` 迭代属性值
- `for...in` 迭代属性名

事件

浏览器中的 JavaScript 使用事件驱动编程模型。万物始于事件。这个章节介绍了 JavaScript 事件以事件处理器的工作原理。

事件可能是 DOM 加载完成，或者是异步请求结束，或者是用户点击了元素或是滚动了页面，或者是用户按下键盘。

有很多种不同的事件。

事件处理器

你可以通过事件处理器响应所有事件，就是事件发生时调用对应函数。

你可以对同一个事件注册多个处理器，它们都会在事件发生时被调用。

JavaScript 提供了三种方法注册事件处理器：

行内事件处理器

这种方法由于自身限制现在已经很少使用，但在早期的 JavaScript 中是唯一的方法：

```
<a href="site.com" onclick="dosomething();">A link</a>
```

DOM 事件处理器

当一个对象只有一个事件处理器时这种方法很常用，在这个例子中没办法添加多个处理器：

```
window.onload = () => {  
  //window loaded  
}
```

在处理 XHR 请求时这也很常见：

```
const xhr = new XMLHttpRequest()  
xhr.onreadystatechange = () => {  
  //.. do something  
}
```

你可以通过 `if ('onsomething' in window) {}` 检查处理器是否已经分配给某个属性。

使用 `addEventListener()`

这是很现代的方法。这个方法允许我们按需注册多个事件处理器，你会发现它是最流行的：

```
window.addEventListener('load', () => {  
  //window loaded  
})
```

注意：IE8 及以下版本不支持这个方法，可以使用 `attachEvent()` 代替。如果你需要兼容旧浏览器这很重要。

监听不同的元素

你可以监听 `window` 拦截“全局”事件，比如键盘的使用。你也可以监听特定元素上发生的事件，比如鼠标点击了某个按钮。

这也是为什么 `addEventListener` 有时候在 `window` 上调用，有时间在某个 DOM 元素上。

事件对象

事件处理器会获得一个 `Event` 对象作为第一个参数：

```
const link = document.getElementById('my-link')  
link.addEventListener('click', event => {  
  // link clicked  
})
```

这个对象包含很多有用的属性和方法，比如：

- `target`，事件发生的目标 DOM 元素
- `type`，事件类型
- `stopPropagation()`，调用以阻止 DOM 事件传播

([查看完整清单](#))

其它属性提供给特定的事件，`Event` 只是不同事件的一个接口：

- [MouseEvent](#)
- [KeyboardEvent](#)
- [DragEvent](#)
- [FetchEvent](#)
- 等等

上面的每一个都链接到了 MDN 页面，你可以在那查看它们所有的属性。

举个例子，当一个键盘事件发生时，你可以检查哪个键被按下，通过 `key` 属性值得到一个易读的值（`Escape`, `Enter` 等等）：

```
window.addEventListener('keydown', event => {  
  // key pressed  
  console.log(event.key)
```



```
})
```

在鼠标事件中我们可以直到哪个按钮被按下：

```
const link = document.getElementById('my-link')
link.addEventListener('mousedown', event => {
  // mouse button pressed
  console.log(event.button) //0=left, 2=right
})
```

事件冒泡和事件捕捉

事件冒泡和事件捕捉是事件传播的两个模型。

假设你的 DOM 结构是这样的：

```
<div id="container">
  <button>Click me</button>
</div>
```

你希望跟踪用户什么时候点击了这个按钮，你有两个事件处理器，一个在 `button` 上，一个在 `#container` 上。记住，子元素上的点击事件也会传播到它的父元素上，除非你阻止了事件传播（稍后详解）。

这些事件处理器会按照顺序调用，这个顺序通过事件冒泡/事件捕捉模型决定。

冒泡意味着事件从被点击的元素（子元素）一直向上传播到所有祖先元素，从最近的一个开始。

在我们的例子中，`button` 上的处理器会在 `#container` 之前发生。

捕捉恰恰相反：最外部的的事件会在特定处理器之前发生，比如 `button`。

默认采用事件冒泡模型。

你也可以选择使用事件捕捉，通过将 `addEventListener` 的第三个参数设为 `true`：

```
document.getElementById('container').addEventListener(
  'click',
  () => {
    //window loaded
  },
  true
)
```

注意：首先运行的是捕捉阶段的事件处理器，然后才是冒泡的事件处理器。

这个顺序遵循这个原则：DOM 从 Window 对象开始遍历所有元素，直到找到被点击的对象。执行此操作时，会调用任何绑定的事件处理器（捕捉阶段）。一旦找到目标元素，它会重复这个过程直到回到 Window 对象，此时调用相应的事件处理器（冒泡阶段）。

阻止传播

DOM 元素事件会一直在它的母树上传播，除非手动阻止它：

```
<html>
  <body>
    <section>
      <a id="my-link" ...>
```

a 上的点击事件会传播到 section 然后是 body。

你可以调用 `stopPropagation()` 方法阻止事件传播，一般放在事件处理器的末尾：

```
const link = document.getElementById('my-link')
link.addEventListener('mousedown', event => {
  // process the event
  // ...

  event.stopPropagation()
})
```

常见事件

这是一个你经常会用到的事件清单。

load

window 和 body 元素的 load 事件在页面加载完成时触发。

鼠标事件

click 事件在鼠标单击时触发。dblclick 事件在双击鼠标时触发，当然，在这种情况下会先触发 click 事件。mousedown, mousemove 和 mouseup 可以和拖动事件结合在一起。小心使用 mousemove，它会在鼠标移动过程中触发很多次（稍后会看到节流）。

键盘事件

keydown 事件在按下键盘时触发（并在处于按下状态时持续触发）。keyup 事件在松开键盘时触发。

滚动

scroll 事件在每一次滚动页面时触发。在这个事件处理器内部，你可以通过 `window.scrollY`（Y轴）查看当前滚动位置。

注意这个事件不是一次性的，它会在滚动过程中持续发生，不仅仅是在滚动开始和滚动结束，所以不要在处理事件时进行大量计算和操作 - 使用节流代替。

节流

正如上面提到的，`mousemove` 和 `scroll`都不是一次性事件，它们在操作发生期间持续调用事件处理器。这是因为它们需要提供你需要知道的坐标。

如果你在这些事件处理器中进行复杂的操作，将会影响性能给你的网页用户带来糟糕的体验。

像 [Lodash](#) 这样的库提供了 100 行代码实现的节流函数来帮助解决这个问题。一个简单又容易理解的实现是使用定时器每隔 100ms 缓存一次滚动事件：

```
let cached = null
window.addEventListener('scroll', event => {
  if (!cached) {
    setTimeout(() => {
      //you can access the original event at `cached`
      cached = null
    }, 100)
  }
  cached = event
})
```

事件循环

事件循环是 JavaScript 中最重要的内容。

我已经使用 JavaScript 好多年了，但是也没有完全理解它的工作原理。当然不了解这些细枝末节也没有什么关系，但通常来讲，知道它的工作原理是很有帮助的，你可能也很好奇这个内容。

这个章节致力于解释 JavaScript 如何是单线程工作以及如何处理异步函数的内在细节。

你的 JavaScript 代码运行在单线程，同一时间只会发生一件事情。这是一个非常有用的限制，它简化了很多程序，你不用再为并发问题担忧。你只需要关注于

如何书写代码，避免造成线程堵塞的内容，比如同步网络请求或者无限[循环](#)。

通常，大多数浏览器的每一个浏览标签都有独立的事件循环，以使进程隔离避免有无限循环或者繁重处理的页面阻塞整个浏览器。浏览器管理多个并发的[事件循环](#)来解决 API 的调用。Web Workers 也运行在自己的事件循环里。

你只需要明白你的代码运行在单一事件循环，并在写代码时考虑到这一点，避免阻塞它。

阻塞事件循环

任何执行时间过长不能将控制权返回给事件循环的 JavaScript 代码都会阻塞页面内其它代码的执行，甚至阻塞 UI 线程，导致用户不能点击、滚动页面等等。

大多数 JavaScript 原语是非阻塞的，比如网络请求，Node.js 文件系统操作等等。发生阻塞是意外的，这也是为什么 JavaScript 基于大量的回调以及最近的 promises 和 `async/await`。

调用堆栈

调用堆栈是 LIFO 队列（Last In, First Out）。

事件循环不断检查调用堆栈里是否仍有函数需要运行。于此同时，它将找到的函数加入调用堆栈，然后按照顺序执行。

你了解调试器或者浏览器控制台里的错误堆栈跟踪信息吗？浏览器在调用堆栈中查询函数名字，然后标记出当前调用由哪个函数触发：

```
> const bar = () => {  
    throw new DOMException()  
}  
  
const baz = () => console.log('baz')  
  
const foo = () => {  
    console.log('foo')  
    bar()  
    baz()  
}  
  
foo()  
foo
```

```
✖ ▼ Uncaught DOMException  
    bar      @ VM570:2  
    foo      @ VM570:9  
    (anonymous) @ VM570:13
```

```
> |
```

一个简单的事件循环说明

举个例子：

```
const bar = () => console.log('bar')  
const baz = () => console.log('baz')  
  
const foo = () => {  
    console.log('foo')  
    bar()  
}
```

```
    baz()  
}
```

```
foo()
```

这个代码打印出：

```
foo  
bar  
baz
```

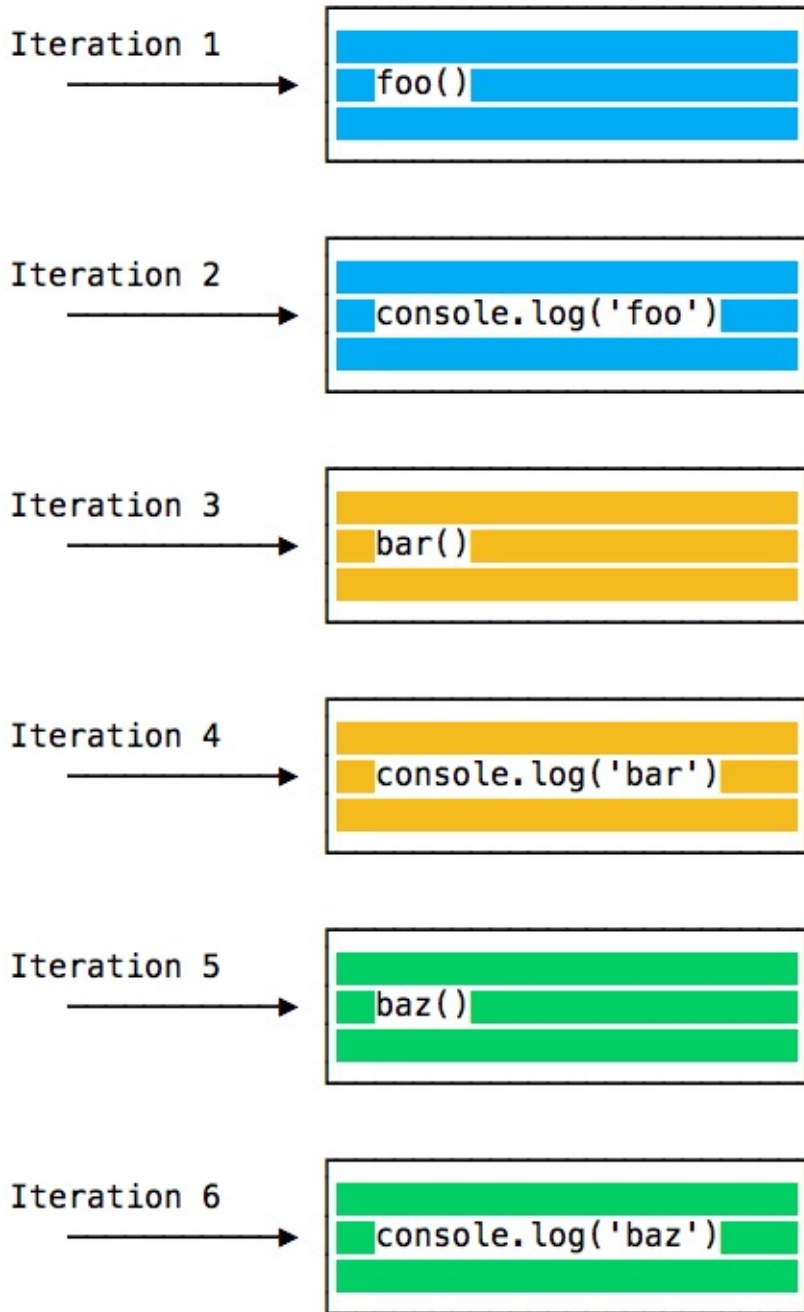
和预期一样。

当这个代码运行时，最开始 `foo()` 被调用，在 `foo()` 内部先调用 `bar()`，然后调用 `baz()`。

这时我们的调用栈看起来就是这样：



每次迭代的事件循环都会查看调用堆栈中是否还有内容，并执行它：



直到整个调用堆栈是空的。

函数执行队列

上面的例子很普通，也没有什么特殊之处：JavaScript 发现需要执行的内容然后按照顺序执行。

让我们看看如何延迟函数执行直到清空调用栈。

使用 `setTimeout(() => {}, 0)` 调用一个函数，会在其它函数全部执行完毕那一刻执行这个函数。

举个例子：

```
const bar = () => console.log('bar')
const baz = () => console.log('baz')

const foo = () => {
  console.log('foo')
  setTimeout(bar, 0)
  baz()
}

foo()
```

结果令人惊讶：

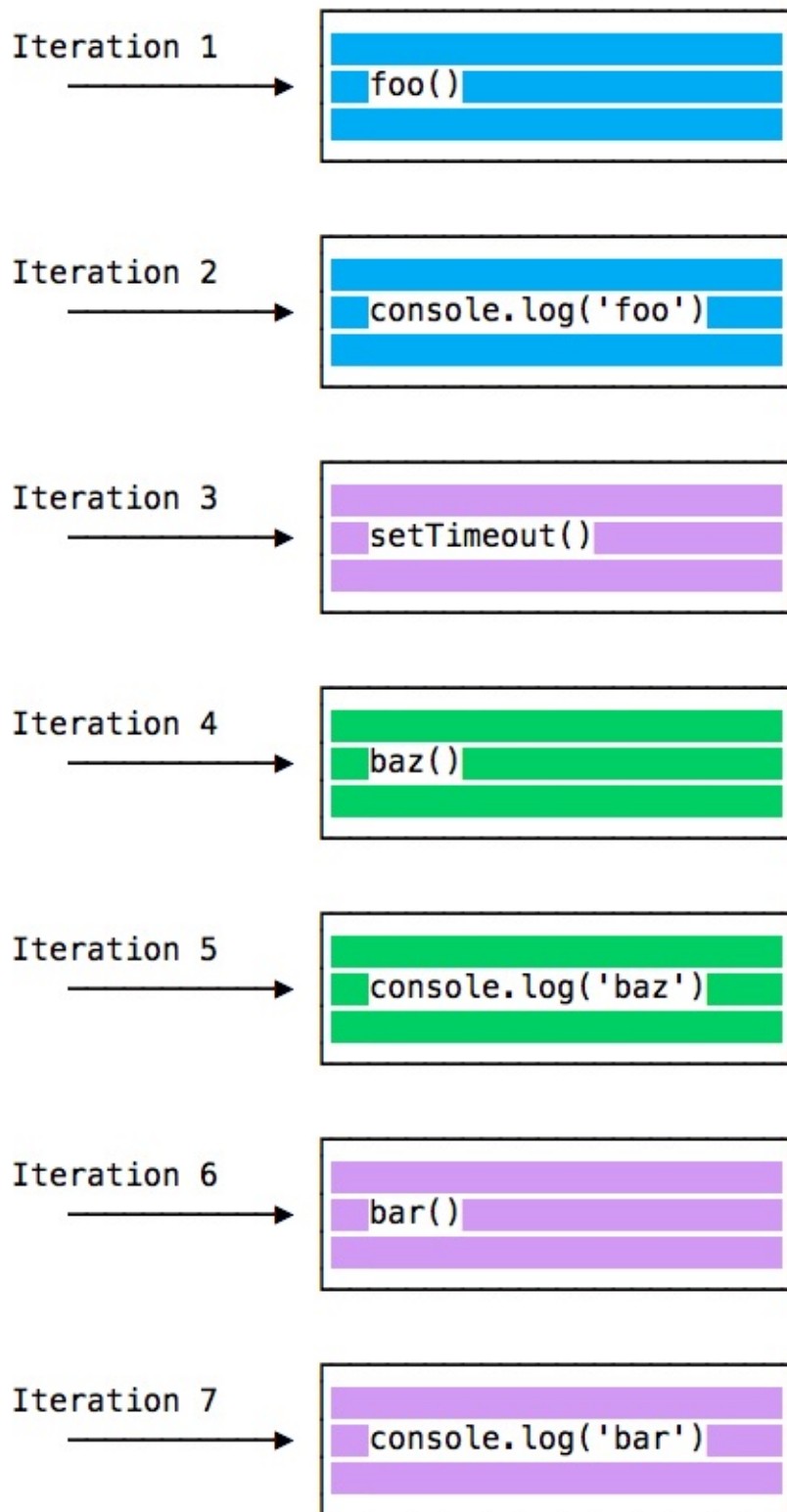
```
foo
baz
bar
```

当这个代码运行时，`foo()` 先被调用。`foo()` 内部先调用 `setTimeout`，将 `bar` 作为参数传入定时器，我们传入 `0` 指示它立即执行，然后调用 `baz()`。

这时调用栈时这样的：



我们的程序中所有函数的执行顺序：



为什么这样？

消息队列

当 `setTimeout()` 调用时，浏览器或者 Node.js 开始计时。在这个例子中，我们将 0 作为延时时间，时间一到，回调函数就会被推入消息队列。

消息队列也包含用户发出的点击或者键盘事件，或者是在你的代码之前已经存在的获取响应的队列，或者是像 `onLoad` 这样的 DOM 事件。

整个循环会优先进行调用堆栈，它会先处理调用堆栈里找到的所有内容，一旦没有内容，它就会在事件队列里拾取内容。

我们不必等待像 `setTimeout`，`fetch` 等其它自己完成工作的函数，因为它们由浏览器提供，具有自己的线程。举个例子，如果你设置一个延时 2 秒的 `setTimeout` 定时器，你不用等待 2 秒 - 等待在别的地方完成。

ES6 工作队列（Job Queue）

ECMAScript 2015 引入了工作队列概念，用在 Promises（同样在 ES6/ES2015 中引入）中。这是一种尽快执行异步函数的方法，而不是放在调用堆栈的末尾。

在当前函数结束前完成的 Promises 将会在当前函数之后执行。

我发现在游乐场坐过山车可以很好的解释这个内容：消息队列将你放在其它游客队列之后，工作队列是一张快速通行证，能让你插队提前坐上过山车。

例子：

```
const bar = () => console.log('bar')
const baz = () => console.log('baz')

const foo = () => {
  console.log('foo')
  setTimeout(bar, 0)
  new Promise((resolve, reject) =>
    resolve('should be right after baz, before bar')
  ).then(resolve => console.log(resolve))
  baz()
}

foo()
```

这会打印出：

```
foo
baz
should be right after baz, before bar
bar
```

这是 Promises（包括基于 promises 的 Async/await）和原生的旧异步函数 `setTimeout()` 或者其它平台 API 之间最大的不同。

异步编程和回调

JavaScript 默认就是异步和单线程的。这意味着代码不能创建新线程并且并行运行。让我们了解异步代码是什么。

编程语言里的异步

计算机在设计上是异步的。

异步是某些东西可以独立于主程序流程发生。

在当前的消费计算机中，每一个程序都运行在一个特殊的时间段，然后停止之后让其它程序开始运行。这一切发生的很快以至于我们无法注意到，我们认为我们的计算机同时运行许多程序，但这是一种误解（除了多进程机器）。

程序内部使用中断，一种提交给处理器获得系统注意的信号。

我不会深入介绍它，但是要记住这对异步程序来说很普遍，在它们被注意之前停止执行，同时计算机可以执行其它内容。当一个程序等待网络响应时，只有等请求结束才能终止运行。

一般，编程语言是异步的，其中一些也会提供异步操作的办法，这些语言或者库，C 语言，Java，C#，PHP，Go，Ruby，Swift，Python 默认都是异步的，其中一些通过使用线程处理异步，产生一个新进程。

JavaScript

JavaScript 默认就是异步和单线程的。这意味着代码不能创建新线程并且并行运行。

每行代码都是一个接着一个执行，举个例子：

```
const a = 1
const b = 2
const c = a * b
console.log(c)
doSomething()
```

但是 JavaScript 是为了浏览器而生的，一开始它的主要工作是响应用户的操作，比如 onClick，onMouseOver，onChnage，onSubmit 等等。它怎么能使用异步编程模式？

答案在它的环境中。浏览器通过提供一系列可以处理这种功能的 API 解决了这个问题。

尤其最近 Node.js 引入了非阻塞 I/O 环境将这个理念扩展到文件访问，网络调用等等方面。

回调函数

你无法知道用户何时要单击按钮，因此你要为 click 事件定义事件处理程序。此事件处理程序接受一个函数，该函数将在触发事件时调用：

```
document.getElementById('button').addEventListener('click', () => {
  //item clicked
})
```

这也被叫做回调函数。

回调是一个简单的函数，作为值传入其它函数，只会在事件发生时被调用。我们可以这么是因为 JavaScript 的头等函数可以和变量绑定并传入其它函数（称作高阶函数）。

把你所有的代码包裹在 window 对象上的 load 事件监听器里是很常见的，这样代码只会在页面准备好时运行：

```
window.addEventListener('load', () => {  
  //window loaded  
  //do what you want  
})
```

回调函数任何地方都会用到，不仅仅是 DOM 事件。

一个常见的例子是使用定时器：

```
setTimeout(() => {  
  // runs after 2 seconds  
}, 2000)
```

XHR 请求同样接受一个回调函数。在这个例子中，将一个函数分配给一个属性，这样当特定事件（这里是请求状态发生变化）发生时就会调用这个函数：

```
const xhr = new XMLHttpRequest()  
xhr.onreadystatechange = () => {  
  if (xhr.readyState === 4) {  
    xhr.status === 200 ? console.log(xhr.responseText) : console.error('error')  
  }  
}  
xhr.open('GET', 'https://yoursite.com')  
xhr.send()
```

处理回调中的错误

你如何处理回调错误？一个常见的策略是采用 Node.js 的方法：回调函数的第一个参数始终是错误对象：错误优先回调。

如果没有错误，这个对象为 null。如果发生错误，它包含描述错误和其它信息的内容。

```
fs.readFile('/file.json', (err, data) => {  
  if (err !== null) {  
    //handle error  
    console.log(err)  
    return  
  }  
  //no errors, process data  
  console.log(data)  
})
```

回调伴随的问题

回调让简单代码更简单！

然而每一个回调函数都会添加一级嵌套，如果你有很多个回调函数，代码会变得十分复杂：

```
window.addEventListener('load', () => {
  document.getElementById('button').addEventListener('click', () => {
    setTimeout(() => {
      items.forEach(item => {
        //your code here
      })
    }, 2000)
  })
})
```

这仅仅是一个简单的四级代码，但是我看到很多层嵌套，这并不有趣。

怎么解决这个问题？

回调的替代方案

从 ES6 开始，JavaScript 引入了很多功能让我们不用回调就能优雅的写异步代码：

- Promises (ES6)
- Async/Await (ES8)

Promises

Promises 是 JavaScript 解决异步代码中需要写太多回调函数的一种方法。

Promises 通常被定义为一个最终可用的值的代理（a proxy for a value that will eventually become available）。

Promises 是一个解决异步代码的方法，不用在代码中写太多回调函数。尽管这些年已经变得很流行，也在 ES2015 中被引入并成为标准，在 ES2017 中也被异步函数（async functions）代替。

promises 工作原理（简短）

一旦一个 promise 被调用，它就会变成 **pending** 状态。这意味调用者会持续执行，同时等待自身处理结果，然后给调用函数一些反馈。

此时，该调用函数等待这个 promise 返回 **resolved** 状态 或者 **rejected** 状态，但是你知道 [JavaScript](#) 是异步的，所以这个函数会在 *promise* 工作时继续执行其它代码。

哪些 JS API 使用 promises？

除了你自己的代码和一些库之外，现在 Web API 也使用 promises：

- 电池 API
- [Fetch API](#)
- [Service Workers](#)

在现代 JavaScript 中你不太可能不使用 promises，所以让我们深挖一下它。

创建 promise

Promise API 暴露出一个 Promise 构造函数，你可以通过 `new Promise()` 初始化：

```
let done = true
```

```
const isItDoneYet = new Promise(
  (resolve, reject) => {
    if (done) {
      const workDone = 'Here is the thing I built'
      resolve(workDone)
    } else {
      const why = 'Still working on something else'
      reject(why)
    }
  }
)
```

你可以看到这个 promise 在全局常量 `done` 为 `true` 时，返回一个已解决的 promise，否则返回一个被拒绝的 promise。

使用 `resolve` 和 `reject` 我们可以回传一个值，在上面的例子中，我们返回了一个字符串，也可以返回一个对象。

使用 promise

在上个章节中，我们介绍了如何创建一个 promise。

现在让我们看看如何使用 promise。

```
const isItDoneYet = new Promise(
  //...
)
```

```
const checkIfItsDone = () => {
  isItDoneYet
    .then((ok) => {
      console.log(ok)
    })
    .catch((err) => {
      console.error(err)
    })
}
```

运行 `checkIfItsDone()` 会执行 `isItDoneYet()` promise 然后等待它 resolve 调用 `then` 回调，如果发生错误，会在 `catch` 回调中处理错误。

链式 promise

promise 可以返回另一个 promise，形成链式 promise。

Fetch API (XMLHttpRequest API 上层 API) 提供了一个很好的例子。我们可以用它获取资源并用 promise 链式操作资源。

Fetch API 基于 promise 机制，调用 `fetch()` 等同于我们通过 `new promise()` 定义一个 promise。

链式 promise 例子

```
const status = (response) => {
  if (response.status >= 200 && response.status < 300) {
    return Promise.resolve(response)
  }
  return Promise.reject(new Error(response.statusText))
}

const json = (response) => response.json()
fetch('/todos.json')
  .then(status)
  .then(json)
  .then((data) => { console.log('Request succeeded with JSON response', data) })
  .catch((error) => { console.log('Request failed', error) })
```

在这个例子中，我们调用 `fetch()` 从根域名中的 `todos.json` 文件获取一个 TODO 清单，我们创建了一个 promises 链。

运行 `fetch()` 返回一个包含很多个属性的[响应](#)，我们引用了其中的：

- `status`，反应 HTTP 状态的数值
- `statusText`，状态消息，请求成功时为 OK

`response` 也有一个 `json()` 方法，可以将成功获取的响应内容转化为 JSON 并作为 promise 返回。

基于此：链中第一个 promise 是我们定义的函数 `status()`，它用来检查响应状态，当结果不在 200 和 299 之间时拒绝这个 promise。这样会导致 promise 链跳过所有链表直接进入末尾的 `catch()` 语句，打印出 Request failed 和错误信息。

如果成功，它会调用我们定义的 `json()` 函数。当成功时，上一个 promise 返回 `response` 对象，作为第二个 promise 的输入。

在这个过程中，我们返回处理过的 JSON 数据，所以第三个 promise 直接获得 JSON 对象：

```
.then((data) => {
  console.log('Request succeeded with JSON response', data)
})
```

在控制台会打印出这些内容。

处理错误

在上面的例子中，promises 链后面有一个 `catch` 块。当链式 promises 有任何错误发生，亦或手

动返回 rejects, 程序控制权会交给错误代码后距离最近的 catch() 语句。

```
new Promise((resolve, reject) => {  
  throw new Error('Error')  
})  
  .catch((err) => { console.error(err) })
```

// or

```
new Promise((resolve, reject) => {  
  reject('Error')  
})  
  .catch((err) => { console.error(err) })
```

级联错误

如果在 catch() 内部又抛出一个错误, 你可以添加第二个 catch() 处理它, 以此类推。

```
new Promise((resolve, reject) => {  
  throw new Error('Error')  
})  
  .catch((err) => { throw new Error('Error') })  
  .catch((err) => { console.error(err) })
```

编排 promises

Promise.all()

如果你需要同时处理多个 promises, Promise.all() 可以帮助你定义一组 promises, 等待它们全部完成之后再执行某些操作。比如：

```
const f1 = fetch('/something.json')  
const f2 = fetch('/something2.json')
```

```
Promise.all([f1, f2]).then((res) => {  
  console.log('Array of results', res)  
})  
  .catch((err) => {  
    console.error(err)  
  })
```

ES2015 的结构语法也允许你这么做：

```
Promise.all([f1, f2]).then(([res1, res2]) => {  
  console.log('Results', res1, res2)  
})
```

这不仅限于 fetch, 任何 promise 都可以处理。

Promise.race()

当传入的 `promise` 有一个完成 `Promise.race()` 就开始运行，且只会运行一次附加的回调函数，传入首先运行完的 `promise` 返回的结果。例子：

```
const first = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, 'first')
})
const second = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'second')
})

Promise.race([first, second]).then((result) => {
  console.log(result) // second
})
```

Async 和 Await

现在，我们将探讨 JavaScript 中更现代的异步函数方法。JavaScript 从回调函数发展为 `Promise` 只用了很短的时间，从 ES2017 开始，`async` 和 `await` 让异步 JavaScript 变得更加简单。

异步函数结合了 `promise` 和生成器，基本上可以看作是 `promises` 之上的抽象方法。我再重复一遍：`async` 和 `await` 基于 `promises`。

为什么引入 `async` 和 `await`？

它们减少了 `promises` 的固定样板和链式 `promise` “不能切断链式”的限制。

ES2015 引入 `promises` 时，只是为了解决异步代码的问题，这一点做得很好，但是，ES2015 和 ES2017 之间的这两年人们发现 `promises` 并不是最终的解决方案。

引入 `Promises` 是为了解决著名的回调地狱问题，但其自身也很复杂，引入了更复杂的语法。

它们是很好的开始，但是应该有更好的语法供开发者使用，所以异步函数（`async functions`）诞生了。它让代码看起来是同步的，但其实它们是异步的并不会阻塞后面的代码。

工作原理

一个异步函数返回一个 `promise`，就像这个例子：

```
const doSomethingAsync = () => {
  return new Promise((resolve) => {
    setTimeout(() => resolve('I did something'), 3000)
  })
}
```

你在前面加上 `await` 然后调用这个函数，这样代码会暂停执行直到这个 `promise` 变成 `resolved` 或者 `rejected`。需要注意的是：委托函数必须定义为 `async`。例子：

```
const doSomething = async () => {
  console.log(await doSomethingAsync())
}
```

```
}
```

例子

这是一个使用 `async/await` 异步运行函数的例子：

```
const doSomethingAsync = () => {  
  return new Promise((resolve) => {  
    setTimeout(() => resolve('I did something'), 3000)  
  })  
}
```

```
const doSomething = async () => {  
  console.log(await doSomethingAsync())  
}
```

```
console.log('Before')  
doSomething()  
console.log('After')
```

上面的代码在浏览器控制台中打印出：

```
Before  
After  
I did something //after 3s
```

一切都是 **promise**

在任何函数前加上 `async` 前缀都使这个函数返回一个 `promise`。即使它没有明确这么做，它也会在内部让它返回一个 `promise`。这也是为什么这个代码是合法的：

```
const aFunction = async () => {  
  return 'test'  
}
```

```
aFunction().then(alert) // This will alert 'test'
```

它和这个类似：

```
const aFunction = async () => {  
  return Promise.resolve('test')  
}
```

```
aFunction().then(alert) // This will alert 'test'
```

代码更易读

你看我们上面的代码和原生的 `promise` 链式回调比起来是多么的简单。这仅仅是一个非常简单的例子，代码越复杂，越能凸显它的优势。

拿 promises 举个例子，你需要获取一个 JSON 资源并且对它进行解析：

```
const getFirstUserData = () => {  
  return fetch('/users.json') // get users list  
    .then(response => response.json()) // parse JSON  
    .then(users => users[0]) // pick first user  
    .then(user => fetch(`/users/${user.name}`)) // get user data  
    .then(userResponse => response.json()) // parse JSON  
}
```

getFirstUserData()

用 async/await 实现同样的需求：

```
const getFirstUserData = async () => {  
  const response = await fetch('/users.json') // get users list  
  const users = await response.json() // parse JSON  
  const user = users[0] // pick first user  
  const userResponse = await fetch(`/users/${user.name}`) // get user data  
  const userData = await user.json() // parse JSON  
  return userData  
}
```

getFirstUserData()

串联多个异步函数

异步函数可以很容易的串联起来，语法也比原生 promise 更易读：

```
const promiseToDoSomething = () => {  
  return new Promise(resolve => {  
    setTimeout(() => resolve('I did something'), 10000)  
  })  
}  
const watchOverSomeoneDoingSomething = async () => {  
  const something = await promiseToDoSomething()  
  return something + ' and I watched'  
}  
const watchOverSomeoneWatchingSomeoneDoingSomething = async () => {  
  const something = await watchOverSomeoneDoingSomething()  
  return something + ' and I watched as well'  
}
```

```
watchOverSomeoneWatchingSomeoneDoingSomething().then((res) => {  
  console.log(res)  
})
```

打印出：

I did something and I watched and I watched as well

更容易调试

调试 promise 很难，因为调试器不能跳过异步代码。

Async/await 就很简单了，因为对编译器来说它就是同步代码。

循环作用域

对开发者来说，JavaScript 的循环作用域可能会让人感到头疼。我们将学习循环作用域中和 var, let 有关的技巧。

例子：

```
const operations = []

for (var i = 0; i < 5; i++) {
  operations.push(() => {
    console.log(i)
  })
}

for (const operation of operations) {
  operation()
}
```

它遍历 5 次，每次添加一个函数到 operations 数组。这个函数能够打印出循环时的索引变量 i。稍后运行这些函数。

期望的结果是：

```
0
1
2
3
4
```

但真实结果却是：

```
5
5
5
5
5
```

为什么这样？原因在于 var。

由于 var 声明会被提升，上面的代码等同于：

```
var i;
const operations = []

for (i = 0; i < 5; i++) {
  operations.push(() => {
```

```

        console.log(i)
    })
}

for (const operation of operations) {
    operation()
}

```

所以，在 for-of 循环中，i 始终等于 5。在这个函数中每一次调用 i 都等于 5。

所以我们怎么做能让它按照我们的需要工作呢？

最简单的方法是用 let 声明。在 ES2015 引入，它可以避免 var 声明带来一些奇怪的事情。

把循环中的 var 改成 let 一切就正常了：

```

const operations = []

for (let i = 0; i < 5; i++) {
    operations.push(() => {
        console.log(i)
    })
}

for (const operation of operations) {
    operation()
}

```

输出：

```

0
1
2
3
4

```

这怎么可能？原因是每一次循环迭代 i 都创建了一个新的作用域，每个添加到 operations 数组的函数都获取当时的 i 副本。

另一个解决这个问题的办法在 ES6 之前很常用，使用立即执行函数表达式（IIFE）。

在这个问题中，你可以包裹整个函数并绑定 i。这样每次都创建了一个立即执行的函数，并返回了一个新函数，所以我们可以稍后执行：

```

const operations = []

for (var i = 0; i < 5; i++) {
    operations.push(((j) => {
        return () => console.log(j)
    })(i))
}

```

```
for (const operation of operations) {  
  operation()  
}
```

定时器

写 JavaScript 代码时，你可能想要延时执行某个函数。本节我们将讨论如何使用 `setTimeout` 和 `setInterval` 安排将来的函数。

`setTimeout()`

写 JavaScript 代码时，你可能想要延时执行某个函数。这个工作交给 `setTimeout`。你可以指定一个延时执行的函数和需要的延时时间，以毫秒计：

```
setTimeout(() => {  
  // runs after 2 seconds  
}, 2000)  
  
setTimeout(() => {  
  // runs after 50 milliseconds  
}, 50)
```

这个语法定义了一个新函数。你可以在任何位置调用它，你也可以传递一个已经存在的函数名，也可以设置一些参数：

```
const myFunction = (firstParam, secondParam) => {  
  // do something  
}  
  
// runs after 2 seconds  
setTimeout(myFunction, 2000, firstParam, secondParam)
```

`setTimeout` 返回一个定时器标识符。通常不会用到它，但是你可以保存这个 `id`，在需要删除这个定时函数时清空它：

```
const id = setTimeout(() => {  
  // should run after 2 seconds  
}, 2000)  
  
// I changed my mind  
clearTimeout(id)
```

零延时

如果你指定延时时间为 0，这个回调函数将会尽可能快的执行，但是必须等当前函数执行完毕：

```
setTimeout(() => {  
  console.log('after ')
```

```
} , 0)
```

```
console.log(' before ')
```

会打印出 before after。

通过对调度程序中的函数进行排序，这对于避免在密集型任务上阻塞 CPU 并在执行繁重计算时让其他函数能够执行特别有用。

一些浏览器（IE 和 Edge）实现了 `setImmediate()` 方法达到上述目的，但是没有成为标准，[不能在其它浏览器中使用](#)。但在 Node.js 可用。

setInterval()

`setInterval` 和 `setTimeout` 类似，区别在于：和只运行一次回调函数不同，它能够在指定的特定时间间隔（以毫秒为单位）一直运行它：

```
setInterval(() => {  
  // runs every 2 seconds  
}, 2000)
```

上面的函数每隔两秒运行一次，除非你用 `clearInterval` 停止它，传入 `setInterval` 返回的间隔 id：

```
const id = setInterval(() => {  
  // runs every 2 seconds  
}, 2000)
```

```
clearInterval(id)
```

在 `setInterval` 回调函数中调用 `clearInterval` 很常见，这让它自己决定是否需要继续运行。例子中的代码会一直运行，除非 `App.somethingIWait` 的值等于 `arrived`：

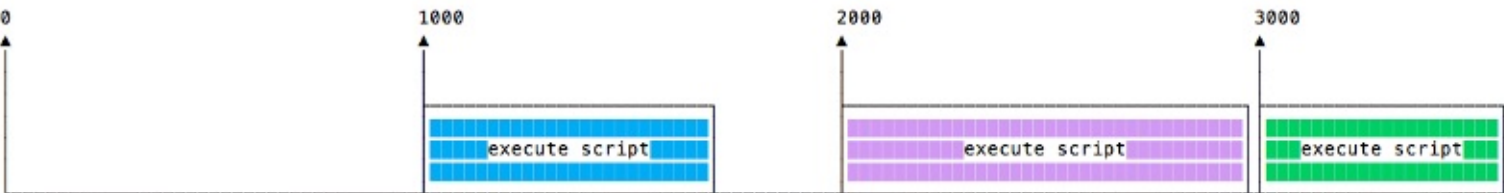
```
const interval = setInterval(() => {  
  if (App.somethingIWait === 'arrived') {  
    clearInterval(interval)  
    return  
  }  
  // otherwise do things  
, 100)
```

递归 setTimeout

`setInterval` 每隔 n 毫秒执行一次函数，不会考虑函数是否执行完毕。如果函数执行都花费相同的时间，这没有任何问题：



但存在执行时间不一致的可能，比如网络条件导致的：



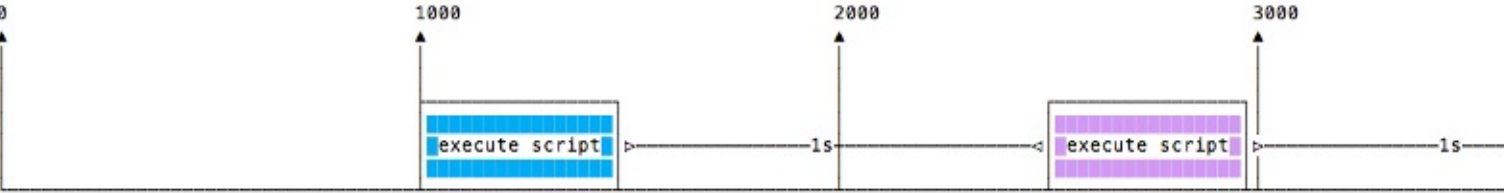
还有执行时间与下一个重合：



为了避免这个情况，你可以使用递归 setTimeout，在回调函数结束后调用：

```
const myFunction = () => {  
  // do something  
  setTimeout(myFunction, 1000)  
}  
  
setTimeout(  
  myFunction()  
, 1000)
```

实现这种情况：



setTimeout 和 setInterval 在 Node.js 的 Timers 模块里可用。

Node.js 也提供了 setImmediate(), 和使用 setTimeout(() => {}, 0) 效果一样，主要用在 Node.js 的事件循环。

This

this 的值取决于在哪里使用它。不了解这个小细节会让人头大，所以花五分钟时间学习一下它

吧。

严格模式的 **this**

严格模式，在对象外的 **this** 始终是 `undefined`。

注意我提到了严格模式。如果严格模式没有开启（默认关闭，除非你在文件头部显式添加 `use strict`），即草率模式（*sloppy mode*），没有特别说明，下面提到的 **this** 都指向全局对象。在浏览器环境里是 `window` 对象。

方法里的 **this**

方法是附加在对象里的函数。

你可以看到不同的格式，比如：

```
const car = {
  maker: 'Ford',
  model: 'Fiesta',
  drive() {
    console.log(`Driving a ${this.maker} ${this.model} car!`)
  }
}
```

```
car.drive()
//Driving a Ford Fiesta car!
```

在这个例子中，使用了常规函数，**this** 自动绑定了这个对象。

注意：上面的方法声明和 `drive: function() {...}` 相同，只是更短：

```
const car = {
  maker: 'Ford',
  model: 'Fiesta',

  drive: function() {
    console.log(`Driving a ${this.maker} ${this.model} car!`)
  }
}
```

在这个例子里也一样：

```
const car = {
  maker: 'Ford',
  model: 'Fiesta'
}

car.drive = function() {
  console.log(`Driving a ${this.maker} ${this.model} car!`)
}
```

```
car.drive()  
//Driving a Ford Fiesta car!
```

箭头函数与此不同，因为它属于词法绑定：

```
const car = {  
  maker: 'Ford',  
  model: 'Fiesta',  
  
  drive: () => {  
    console.log(`Driving a ${this.maker} ${this.model} car!`)  
  }  
}
```

```
car.drive()  
//Driving a undefined undefined car!
```

绑定箭头函数

你不能像给普通函数那样给箭头函数绑定值。这是因为它们的原理不同，`this` 是词法绑定，这意味着它的值来自定义它们的上下文。

显式传递 `this` 指向的对象

JavaScript 提供了一种映射 `this` 和对象的方法。

在函数声明阶段使用 `bind()`：

```
const car = {  
  maker: 'Ford',  
  model: 'Fiesta'  
}  
  
const drive = function() {  
  console.log(`Driving a ${this.maker} ${this.model} car!`)  
}.bind(car)  
  
drive()  
//Driving a Ford Fiesta car!
```

你也可以重新映射一个已经存在的对象作为 `this` 值：

```
const car = {  
  maker: 'Ford',  
  model: 'Fiesta',  
  drive() {  
    console.log(`Driving a ${this.maker} ${this.model} car!`)  
  }  
}  
  
const anotherCar = {
```

```

    maker: 'Audi',
    model: 'A4'
}

car.drive.bind(anotherCar)()
//Driving a Audi A4 car!

```

在函数调用阶段使用 `call()` 和 `apply()` :

```

const car = {
  maker: 'Ford',
  model: 'Fiesta'
}

const drive = function(kmh) {
  console.log(`Driving a ${this.maker} ${this.model} car at ${kmh} km/h!`)
}

drive.call(car, 100)
//Driving a Ford Fiesta car at 100 km/h!

drive.apply(car, [100])
//Driving a Ford Fiesta car at 100 km/h!

```

传入 `call()` 或者 `apply()` 的第一个参数始终绑定 `this`。 `call()` 和 `apply()` 不同之处在于 `apply()` 传入的作为函数参数的参数是一个数组，而 `call()` 可以接受多个参数。

浏览器事件处理中的特殊情况

在事件处理器的回调中。`this` 指向收到事件的 HTML 元素：

```

document.querySelector('#button').addEventListener('click', function(e) {
  console.log(this) //HTMLElement
})

```

你可以这样绑定：

```

document.querySelector('#button').addEventListener(
  'click',
  function(e) {
    console.log(this) //Window if global, or your context
  }.bind(this)
)

```

严格模式

严格模式是 ES5 的功能，它使 JavaScript 表现得更好 - 开启严格模式可以改变 JavaScript 语言的语义。知道严格模式和一般模式，也经常被称作草率模式，在 JavaScript 代码的不同十分重要。

严格模式主要移除了 ES3 中存在的功能，从 ES5 开始弃用这些功能（考虑到向后兼容的需要没

有被删除）。

如何开启严格模式

严格模式是可选的。伴随着不兼容的变化，我们不能简单的改变语言的默认行为，这会破坏大量的 JavaScript 代码，而且 JavaScript 花费了巨大的努力确保 1996 年的代码在今天仍然能够生效。这也是它能成功的关键。

所以在我们需要开启严格模式的时候，有了 `use strict` 指令。你可以把它放在文件的开头，把它应用到整个文件：

```
'use strict'  
  
const name = 'Flavio'  
const hello = () => 'hey'  
  
//...
```

你也可以在独立的函数中启动严格模式，只需要把 `use strict` 放在函数体开始的位置：

```
function hello() {  
  'use strict'  
  
  return 'hey'  
}
```

这对操作那些你没有时间测试或者没有信心在整个文件开启严格模式的历史遗留代码很有用。

严格模式的变化

意外的全局变量

如果你把值绑定在未声明的变量上，JavaScript 默认会在全局对象上创建这个变量：

```
;(function() {  
  variable = 'hey'  
})()(() => {  
  name = 'Flavio'  
})()  
  
variable // 'hey'  
name // 'Flavio'
```

打开严格模式，这样做就会抛出错误：

```
;(function() {  
  'use strict'  
  variable = 'hey'  
})()(() => {  
  'use strict'
```

```
myname = 'Flavio'  
}}())
```

```
> (function() {  
  'use strict'  
  variable = 'hey'  
}}())
```

```
✖ ▶ Uncaught ReferenceError: variable is not defined  
    at <anonymous>:3:12  
    at <anonymous>:4:3
```

```
> (() => {  
  'use strict'  
  myname = 'Flavio'  
}}())
```

```
✖ ▶ Uncaught ReferenceError: myname is not defined  
    at <anonymous>:3:10  
    at <anonymous>:4:3
```

分配错误

JavaScript 默默处理了一些转换错误。

在严格模式，这些错误会被展示出来：

```
const undefined = 1(() => {  
  'use strict'  
  undefined = 1  
}}())
```

```
> undefined = 1
```

```
< 1
```

```
>  
  (() => {  
    'use strict'  
    undefined = 1  
  })()
```

```
✖ ▶ Uncaught TypeError: Cannot assign to read only property  
  'undefined' of object '#<Window>'  
    at <anonymous>:4:13  
    at <anonymous>:5:3
```

Infinity, NaN, eval, arguments等等同样如此。

在 JavaScript 中，你可以定义一个不可写的对象属性，比如：

```
const car = {}
Object.defineProperty(car, 'color', { value: 'blue', writable: false })
```

在严格模式下，你不能覆盖这个值，但在草率模式下可以这么做：

```
const car = {}
Object.defineProperty(car, 'color', { value: 'blue', writable: false })
< ▶ {color: "blue"}
> car.color = 'test'
< "test"
>
(() => {
  'use strict'
  car.color = 'yellow'
})()
✖ ▶ Uncaught TypeError: Cannot assign to read only property 'color' of object '#<Object>'
    at <anonymous>:4:13
    at <anonymous>:5:3
```

和 getters 的原理一样：

草率模式下可以扩展一个不可扩展对象：

```
const car = { color: 'blue' }
Object.preventExtensions(car)

car.model = 'Fiesta'
//ok
() => {
  'use strict'
  car.owner = 'Flavio' //TypeError: Cannot add property owner, object is not extensib.
}
>()
```

而且可以设置原始值的属性，没有任何错误提示，但是也不生效：

```
true.false = ''
//''
1
).name =
'xxx' //'xxx'
var test = 'test' //undefined
test.testing = true //true
```



```
test.testing //undefined
```

严格模式下这些都不被允许：

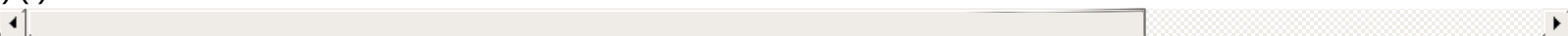
```
;(() => {
  'use strict'
  true.false = ''(
    //TypeError: Cannot create property 'false' on boolean 'true'
    1
  ).name =
    'xxx' //TypeError: Cannot create property 'name' on number '1'
  'test'.testing = true //TypeError: Cannot create property 'testing' on string 'test'
})();
```

删除错误

草率模式下，如果你尝试删除不能删除的属性值，JavaScript 只会返回 false，但在严格模式，它会抛出 TypeError：

```
delete Object.prototype(
  //false

  () => {
    'use strict'
    delete Object.prototype //TypeError: Cannot delete property 'prototype' of function
  }
)()
```



同名函数参数

在一般函数中，可能有冲突的参数名：

```
(function(a, a, b) {
  console.log(a, b)
})(1, 2, 3)
//2 3

(function(a, a, b) {
  'use strict'
  console.log(a, b)
})(1, 2, 3)
//Uncaught SyntaxError: Duplicate parameter name not allowed in this context
```

在这个例子中，箭头函数始终抛出 SyntaxError：

```
((a, a, b) => {
  console.log(a, b)
})(1, 2, 3)
//Uncaught SyntaxError: Duplicate parameter name not allowed in this context
```


八进制

八进制语法在严格模式下是禁用的。默认情况下，在数字前加上兼容八进制格式的 `0` 可以把它解释为八进制数字（有时看起来很困惑）：

```
((() => {  
  console.log(010)  
})();  
//8
```

```
((() => {  
  'use strict'  
  console.log(010)  
})();  
//Uncaught SyntaxError: Octal literals are not allowed in strict mode.
```

你仍然可以通过 `0oXX` 语法在严格模式下使用八进制数字：

```
;(()) => {  
  'use strict'  
  console.log(0o10)  
})();  
//8
```

移除了 `with`

严格模式不能使用 `with` 关键字，移除了一些边界情况，以使编译器层面可以更好的优化。

立即执行函数（IIFE）

立即执行函数在它们被创建的时候就会立即执行。

立即执行函数非常有用，因为它们不会污染全局变量，而且能够隔离变量声明。

立即执行函数的语法是这样的：

```
;(function() {  
  /* */  
})();
```

立即执行函数也可以使用箭头函数：

```
;(()) => {  
  /* */  
})();
```

基本上，我们把函数定义在圆括号内，然后再后面加上一对 `()` 执行这个函数：`(/* function */)()`。

这些括号实际上是使我们的函数在内部被视为表达式的原因，否则，函数声明将是无效的，因

为我们没有指定任何名称：

```
>> function() {  
    /* */  
}
```

▲ **SyntaxError: function statement requires a name** [\[Learn More\]](#)

```
>> (function() {  
    /* */  
})();
```

```
← undefined
```

函数声明需要一个名字，而函数表达式不需要。

你也可以把调用括号放在表达式括号里面，这是一样的，仅仅是写法不同：

```
(function() {  
    /* */  
})();
```

```
((() => {  
    /* */  
})();
```

使用一元运算符

在使用 IIFE 时使用任意一元运算符很奇怪，但是在实际运用中却非常有用：

```
;(function() {  
    /* */  
})();
```

```
+(function() {  
    /* */  
})();
```

```
~(function() {  
    /* */  
})();
```

```
!(function() {  
    /* */  
})();
```

（箭头函数上无效）

命名的 IIFE

IIFE 也可以命名常规函数（不是箭头函数）。这不会导致函数“泄露”到全局作用域，而且在它

执行之后也不能再次调用：

```
;(function doSomething() {  
    /* */  
})();
```

IIFE 前的分号

你会看到：

```
;(function() {  
    /* */  
})();
```

这是为了防止把两个文件拼合在一起时出现问题。因为 JavaScript 不强制使用分号，你可能会在最后一行中使用某些语句连接一个文件，从而导致语法错误。

这个问题可以通过一种“聪明”的代码打包工具解决，比如[webpack](#)。

数学运算符

对任何编程语言执行数学运算都是很常见的事情。JavaScript 提供了几个操作符来帮助我们处理数字。

算术运算符

加 (+)

```
const three = 1 + 2  
const four = three + 1
```

+ 运算符也会拼接字符串，所以注意：

```
const three = 1 + 2  
three + 1 // 4  
'three' + 1 // three1
```

减 (-)

```
const two = 4 - 2
```

除 (/)

返回第一个数字和第二个数字之间商：

```
const result = 20 / 5 //result === 4  
const result = 20 / 7 //result === 2.857142857142857
```

如果除以 0，JavaScript 不会抛出任何错误，而是返回 Infinity（如果是负值返回 -Infinity）。

```
1 / 0 //Infinity
-1 / 0 //-Infinity
```

取余 (%)

取余在很多情况下都很有用：

```
const result = 20 % 5 //result === 0
const result = 20 % 7 //result === 6
```

对 0 取余始终是 NaN，意思是“不是一个数字”：

```
1 % 0 //NaN
-1 % 0 //NaN
```

乘 (*)

```
1 * 2 //2
-1 * 2 //-2
```

求幂 (**)

将第一个操作数乘第二个操作数次数：

```
1 ** 2 //1
2 ** 1 //2
2 ** 2 //4
2 ** 8 //256
8 ** 2 //64
```

一元运算符

递增 (++)

递增数字。这是一个一元运算符，如果把它放在数字前面，返回自增后的值。如果把它放在数字后面，先返回初始值，然后递增。

```
let x = 0
x++ //0
x //1
++x //2
```

递减 (–)

和递增操作符相似，不过它递减值。

```
let x = 0
x -= //0
x //-1
--x //-2
```

一元减 (-)

返回负值

```
let x = 2
-x //-2
x //2
```

一元加 (+)

如果目标不是数字，它会尝试转化它。否则什么也不做。

```
let x = 2
+x //2
```

```
x = '2'
+x //2
```

```
x = '2a'
+x //NaN
```

快速赋值

常规的赋值操作符 = 对所有的数学运算符都有一个快捷方式让你能组合赋值，将第一个操作数和第二个操作数的结果赋值给第一个操作数。

它们是：

- += ：加法赋值
- -= ：减法赋值
- *= ：乘法赋值
- /= ：除法赋值
- %= ：取余赋值
- **= ：求幂赋值

例子：

```
const a = 0
a += 5 //a === 5
a -= 2 //a === 3
a *= 2 //a === 6
a /= 2 //a === 3
a %= 2 //a === 1
```

优先级

每个复杂的语句都会引入优先问题。看这个：

```
const a = 1 * 2 + 5 / 2 % 2
```

结果等于 2.5。但是为什么呢？哪个运算先执行，哪个后执行？

有些运算符的优先级比其它的高。其优先级遵循以下规则：

- - + ++ -- 一元运算符，递增，递减
- * / % 乘法/除法
- + - 加法/减法
- = += -= *= /= %= **= 赋值运算

同级运算符（比如 + 和 -）按照顺序执行。

根据上面的顺序，我们可以计算这个式子：

```
const a = 1 * 2 + 5 / 2 % 2
const a = 1 * 2 + 5 / 2 % 2
const a = 2 + 2.5 % 2
const a = 2 + 0.5
const a = 2.5
```

Math 对象

Math 对象包含很多数学相关的工具。让我们看看都有什么。

常量

Item	Description
Math.E	The constant e, base of the natural logarithm (means ~2.71828)
Math.LN10	The constant that represents the base e (natural) logarithm of 10
Math.LN2	The constant that represents the base e (natural) logarithm of 2
Math.LOG10E	The constant that represents the base 10 logarithm of e
Math.LOG2E	The constant that represents the base 2 logarithm of e
Math.PI	The π constant (~3.14159)
Math.SQRT1_2	The constant that represents the reciprocal of the square root of 2
Math.SQRT2	The constant that represents the square root of 2

函数

所有函数方法都是静态的，Math 不能被继承。

Math.abs()

返回数字的绝对值

```
Math.abs(2.5) //2.5  
Math.abs(-2.5) //2.5
```

Math.acos()

返回反余弦值，参数必须在 -1 到 1 之间。

```
Math.acos(0.8) //0.6435011087932843
```

Math.asin()

返回反正弦值，参数必须在 -1 到 1 之间。

```
Math.asin(0.8) //0.9272952180016123
```

Math.atan()

返回反正切值

```
Math.atan(30) //1.5374753309166493
```

Math.atan2()

返回其参数商的反正切值

```
Math.atan2(30, 20) //0.982793723247329
```

Math.ceil()

向上取整

```
Math.ceil(2.5) //3  
Math.ceil(2) //2  
Math.ceil(2.1) //3  
Math.ceil(2.99999) //3
```

Math.cos()

用弧度表示角度的余弦值

```
Math.cos(0) //1
Math.cos(Math.PI) //-1
```

Math.exp()

返回 Math.E 的参数次方

```
Math.exp(1) //2.718281828459045
Math.exp(2) //7.38905609893065
Math.exp(5) //148.4131591025766
```

Math.floor()

向下取整

```
Math.ceil(2.5) //2
Math.ceil(2) //2
Math.ceil(2.1) //2
Math.ceil(2.99999) //2
```

Math.log()

返回基数 e 的自然对数

```
Math.log(10) //2.302585092994046
Math.log(Math.E) //1
```

Math.max()

返回传入的一系列数字中的最大值

```
Math.max(1, 2, 3, 4, 5) //5
Math.max(1) //1
```

Math.min()

返回传入的一系列数字中的最小值

```
Math.max(1, 2, 3, 4, 5) //1
Math.max(1) //1
```

Math.pow()

返回第一个参数的第二参数次方

```
Math.pow(1, 2) //1
Math.pow(2, 1) //2
Math.pow(2, 2) //4
```



```
Math.pow(2, 4) //16
```

Math.random()

返回 0.0 到 1.0 之间的伪随机值

```
Math.random() //0.9318168241227056
Math.random() //0.35268950194094395
```

Math.round()

四舍五入

```
Math.round(1.2) //1
Math.round(1.6) //2
```

Math.sin()

用弧度计算角度的正弦值

```
Math.sin(0) //0
Math.sin(Math.PI) //1.2246467991473532e-16)
```

Math.sqrt()

开方

```
Math.sqrt(4) //2
Math.sqrt(16) //4
Math.sqrt(5) //2.23606797749979
```

Math.tan()

用弧度计算角度的正切值

```
Math.tan(0) //0
Math.tan(Math.PI) //-1.2246467991473532e-16
```

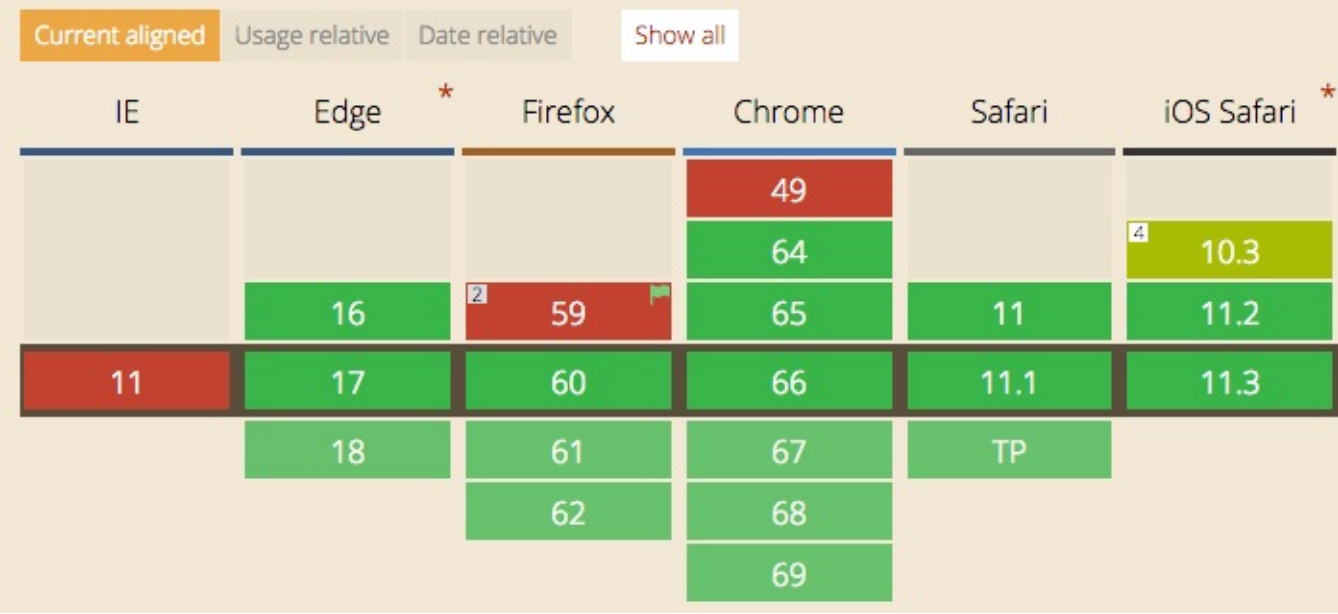
ES 模块

ES 模块是用于处理模块的 ECMAScript 标准。Node.js 长期以来一直使用 CommonJS，然而浏览器还没有模块系统。每个主要决策（如模块系统）必须首先先由 ECMAScript 标准化，然后由浏览器实现。

这个标准化过程在 ES6 中完成，与此同时，浏览器开始逐步实现这个标准，尽力使工作方式保持一致。现在 ES 模块被 Chrome，Safari，Edge 和 Firefox（从 60 版本开始）支持。

JavaScript modules via script tag - LS

Loading JavaScript module scripts using `<script type="module">`
Includes support for the `nomodule` attribute.



模块很酷，它们允许你封装任意功能，然后作为库暴露给其它 JavaScript 文件。

ES 模块语法

导入一个模块可以用：

```
import package from 'module-name'
```

然而 CommonJS 使用：

```
const package = require('module-name')
```

一个模块是一个通过 `export` 导出一个或多个值（对象，函数或变量）的 JavaScript 文件。举个例子，这个模块导出了一个大大写字字符串的函数：

```
uppercase.js  
  
export default str => str.toUpperCase()
```

在这个例子中，模块定义了一个唯一的，默认导出（`default export`），所以它可以是一个匿名函数。否则它需要一个名字来区分其它的导出。现在，其它任何的 JavaScript 模块都可以通过导入 `uppercase.js` 导入这个函数。

一个 HTML 页面可以通过 `<script>` 标签上特殊的 `type="module"` 属性添加一个模块：

```
<script type="module" src="index.js"></script>
```

注意：模块导入行为和 `defer` 脚本加载类似。查看[使用延迟和异步高效加载 JavaScript](#)。

重要的是任何有 `type="module"` 的模块都是以严格模式加载的。

在这个例子中，`uppercase.js` 模块定义了一个默认导出，所以我们可以导入这个模块，还能够给它绑定一个我们喜欢的名字：

```
import toUpperCase from './uppercase.js'
```

然后我们可以使用它：

```
toUpperCase('test') // 'TEST'
```

你也可以使用绝对路径导入模块，以便引用其它域名中定义的模块：

```
import toUpperCase from 'https://flavio-es-modules-example.glitch.me/uppercase.js'
```

这种语法也是有效的：

```
import { foo } from '/uppercase.js'
import { foo } from '../uppercase.js'
```

这种不行：

```
import { foo } from 'uppercase.js'
import { foo } from 'utils/uppercase.js'
```

路径必须是绝对路径，或在名字前加上 `./` 或者 `/`。

其它导入/导出方法

我们已经知道这个例子：

```
export default str => str.toUpperCase()
```

这创建了一个默认的导出。然而在一个文件中你可以导出多个内容，比如：

```
const a = 1
const b = 2
const c = 3
```

```
export { a, b, c }
```

另一个模块可以导入所有：

```
import * from 'module'
```

你可以只选择导入部分内容，使用解构绑定：

```
import { a } from 'module'
import { a, b } from 'module'
```

你也可以导入默认导出，然后通过名字导入没有默认导出的内容，就像常见的导入 React：

```
import React, { Component } from 'react'
```

在[这里](#)查看 ES 模块的示例。

跨域资源共享 CORS

通过 [CORS](#) 获取模块，意味着如果你从其它域名引入脚本，他们必须有一个有效的 CORS 头允许跨网页加载（像 Access-Control-Allow-Origin）。

不支持模块的浏览器怎么办？

结合使用 type=module 和 nomodule：

```
<script type="module" src="module.js"></script>
<script nomodule src="fallback.js"></script>
```

总结

ES 模块是现代浏览器引入的最重要的功能。它们是 ES6 的一部分，但实现它们的过程很漫长。

现在我们可以使用它们！但是我们也必须知道多个模块会影响页面的性能，因为这是浏览器必须执行的一步。

即使 ES 模块已经在浏览器端登陆，[Webpack](#) 也将扮演重要的角色，但是直接用语言构建这样的功能对于统一模块在客户端和 Node.js 上的工作方式来说任务繁重。

CommonJS

CommonJS 模块规范是 Node.js 中的模块标准。

浏览器端的 JavaScript 采用 ES 模块。

它们让你能够创建易于分割且可复用的代码片段，每个片段都可以独立测试。

庞大的 npm 生态环境就建立在 CommonJS 格式之上。

导入模块的语法如下：

```
const package = require('module-name')
```

在 CommonJS 中，模块同步加载，按照 JavaScript 运行时找到的顺序执行。这个系统原本为服务器端 JavaScript 而生，不兼容客户端（这也是为什么引入 ES 模块）。

JavaScript 文件是导出一个或多个其中定义的符号的模块，它们可以是变量，函数，对象：

```
uppercase.js
```

```
exports.uppercase = str => str.toUpperCase()
```

任何 JavaScript 文件都可以导入并且使用这个模块：

```
const uppercaseModule = require('uppercase.js')
uppercaseModule.uppercase('test')
```

在 [Glitch](#) 上的简单例子。

你可以导出多个值：

```
exports.a = 1
exports.b = 2
exports.c = 3
```

通过解构赋值单独导入每一个值：

```
const { a, b, c } = require('./uppercase.js')
```

或者只导出一个值：

```
//file.js
module.exports = value
```

然后导入：

```
const value = require('./file.js')
```

词汇表

最后，一些前端开发的术语可能和你理解的不同。

异步

异步代码是当你启动某个内容，可以先忘掉它，无需刻意等待，当结果准备就绪你会自动得到它。典型的例子就是 AJAX 调用，可能会占用很多秒，与此同时你可以完成其它事情，当获取响应，回调函数会被调用。Promises 和 async/await 是处理异步代码更现代的方法。

块级作用域

块级作用域里任何定义在块内的变量对整个块可见，可以在内部访问，不能在块外部访问。

回调

回调是一个事件发生时调用的函数。元素绑定的点击事件在用户点击了元素时调用。一个 fetch 请求的回调在资源下载完成后调用。

声明

声明方法是你说明细节，告诉机器你需要做什么。React 被认为是声明式的，理由是更注重抽象

推理而不是直接编辑 DOM。每个高级编程语言都比低级编程语言，如 Assembler，更具声明性。JavaScript 比 C 语言更加声明性，HTML 也是声明性的。

回退

回退用来在用户无法访问特定功能时提供更好的体验。比如浏览器关闭了 JavaScript，用户应该得到一个原生版本的 HTML 页面。或者遇到浏览器没有实现的 API，你应该有一个可靠的办法避免严重影响用户体验。

函数作用域

函数作用域里任何定义在函数内的变量对整个函数可见，可以在函数内部访问，不能在函数外部访问。

一致性

一个变量的值在创建之后不能改变，我们就称这变量是不变的。可变变量是可变的。这在数组和对象里都一样。

词汇范围

词汇范围是一个特定作用域，父函数的变量可以在内部函数使用。内部函数的作用域也包括父函数的作用域。

Polyfill

Polyfill 用来为旧浏览器提供它没有原生支持的现代浏览器具有的新功能。Polyfill 是一种特殊的 shim。

纯粹函数（pure function）

纯粹函数是一个没有副作用（不会修改外部资源），而且参数决定输出的函数。你可以调用函数一百万次，每次都使用相同的参数，输出始终保持一致。

重新赋值

var 和 let 允许你无数次地为变量重新赋值。const 声明了一个不可变的字符串，整型，布尔值，对象（但是你仍然可以通过提供的方法修改它）。

范围

范围是一系列变量对于程序可见的部分。

作用域

作用域是编程语言里定义的决定变量值的一系列规则。

Shim

Shim 包含了许多功能或者 API。它通常用于抽象内容，预填充参数或为不支持某些功能的浏览器添加 polyfill。你可以把它看作兼容层。哈哈

副作用

副作用是一个函数与其它函数或者对象进行交互。与网络、文件系统或者是 UI 交互都有副作用。

状态（state）

谈到组件时，不得不提到状态。一个组件管理数据是有状态的，否则是无状态的。

有状态（stateful）

一个有状态的组件，函数或者类会自己管理自己的状态（数据）。它可以存储一个数组，一个计时器或者其它的东西。

无状态（stateless）

一个无状态的组件，函数或者类也被称作 *dumb*，因为它无法使用自己的数据做出决定，所以它的输出或者展示完成基于它的参数。这意味着纯粹函数无状态的。

严格模式

严格模式是 ECMAScript 5.1 的新功能，它会导致 JavaScript 运行时捕捉更多的错误，但是它可以通过拒绝未声明的变量和冲突的对象属性等其他容易被忽视的问题来帮助你改进 JavaScript 代码。建议：使用严格模式。另一个“草率模式”看名字就知道不是什么好东西。

Tree Shaking

Tree Shaking 意味着从你打包发送给用户的代码中删除“死代码”。如果你在很重要语句中添加了从来不会用到的代码，它不会发送给你的应用用户，以此减少文件体积和加载时间。

感谢阅读！

注意：你可以获取这篇 JavaScript 指南的 [PDF](#), [ePub](#), [Mobi](#) 版本以便在 Kindle 或平板上阅读。