

# ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证

🔍

## 目录

- 0.前言
- 1.ECMAScript 6简介
- 2.let 和 const 命令
- 3.变量的解构赋值
- 4.字符串的扩展
- 5.正则的扩展
- 6.数值的扩展
- 7.函数的扩展
- 8.数组的扩展
- 9.对象的扩展
- 10.对象的新增方法
- 11.Symbol
- 12.Set 和 Map 数据结构
- 13.Proxy
- 14.Reflect
- 15.Promise 对象
- 16.Iterator 和 for...of 循环
- 17.Generator 函数的语法
- 18.Generator 函数的异步应用
- 19.async 函数
- 20.Class 的基本语法
- 21.Class 的继承
- 22.Decorator
- 23.Module 的语法
- 24.Module 的加载实现
- 25.编程风格
- 26.读懂规格
- 27.ArrayBuffer
- 28.最新提案
- 29.参考链接

## 其他

- 源码
- 修订历史
- 反馈意见

# let 和 const 命令

- 1.let 命令
- 2.块级作用域
- 3.const 命令
- 4.顶层对象的属性
- 5.global 对象

# 1. let 命令

---

## 基本用法

ES6 新增了 `let` 命令，用来声明变量。它的用法类似于 `var`，但是所声明的变量，只在 `let` 命令所在的代码块内有效。

```
{
  let a = 10;
  var b = 1;
}

a // ReferenceError: a is not defined.
b // 1
```

上面代码在代码块之中，分别用 `let` 和 `var` 声明了两个变量。然后在代码块之外调用这两个变量，结果 `let` 声明的变量报错，`var` 声明的变量返回了正确的值。这表明，`let` 声明的变量只在它所在的代码块有效。

`for` 循环的计数器，就很合适使用 `let` 命令。

```
for (let i = 0; i < 10; i++) {
  // ...
}

console.log(i);
// ReferenceError: i is not defined
```

上面代码中，计数器 `i` 只在 `for` 循环体内有效，在循环体外引用就会报错。

下面的代码如果使用 `var`，最后输出的是 `10`。

```
var a = [];
for (var i = 0; i < 10; i++) {
  a[i] = function () {
    console.log(i);
  };
}
a[6](); // 10
```

上面代码中，变量 `i` 是 `var` 命令声明的，在全局范围内都有效，所以全局只有一个变量 `i`。每一次循环，变量 `i` 的值都会发生改变，而循环内被赋给数组 `a` 的函数内部的 `console.log(i)`，里面的 `i` 指向的就是全局的 `i`。也就是说，所有数组 `a` 的成员里面的 `i`，指向的都是同一个 `i`，导致运行时输出的是最后一轮的 `i` 的值，也就是 `10`。

如果使用 `let`，声明的变量仅在块级作用域内有效，最后输出的是 `6`。

```
var a = [];
for (let i = 0; i < 10; i++) {
  a[i] = function () {
    console.log(i);
  };
}
a[6](); // 6
```

上面代码中，变量 `i` 是 `let` 声明的，当前的 `i` 只在本轮循环有效，所以每一次循环的 `i` 其实都是一个新的变量，所以最后输出的是 `6`。你可能会问，如果每一轮循环的变量 `i` 都是重新声明的，那它怎么知道上一轮循环的值，从而计算出本轮循环的值？这是因为 `JavaScript`

引擎内部会记住上一轮循环的值，初始化本轮的变量 `i` 时，就在上一轮循环的基础上进行计算。

另外，`for` 循环还有一个特别之处，就是设置循环变量的那部分是一个父作用域，而循环体内部是一个单独的子作用域。

```
for (let i = 0; i < 3; i++) {  
  let i = 'abc';  
  console.log(i);  
}  
// abc  
// abc  
// abc
```

上面代码正确运行，输出了 3 次 `abc`。这表明函数内部的变量 `i` 与循环变量 `i` 不在同一个作用域，有各自单独的作用域。

---

## 不存在变量提升

`var` 命令会发生“变量提升”现象，即变量可以在声明之前使用，值为 `undefined`。这种现象多多少少是有些奇怪的，按照一般的逻辑，变量应该在声明语句之后才可以使用。

为了纠正这种现象，`let` 命令改变了语法行为，它所声明的变量一定要在声明后使用，否则报错。

```
// var 的情况  
console.log(foo); // 输出undefined  
var foo = 2;  
  
// let 的情况  
console.log(bar); // 报错ReferenceError  
let bar = 2;
```

上面代码中，变量 `foo` 用 `var` 命令声明，会发生变量提升，即脚本开始运行时，变量 `foo` 已经存在了，但是没有值，所以会输出 `undefined`。变量 `bar` 用 `let` 命令声明，不会发生变量提升。这表示在声明它之前，变量 `bar` 是不存在的，这时如果用到它，就会抛出一个错误。

---

## 暂时性死区

只要块级作用域内存在 `let` 命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响。

```
var tmp = 123;  
  
if (true) {  
  tmp = 'abc'; // ReferenceError  
  let tmp;  
}
```

上面代码中，存在全局变量 `tmp`，但是块级作用域内 `let` 又声明了一个局部变量 `tmp`，导致后者绑定这个块级作用域，所以在 `let` 声明变量前，对 `tmp` 赋值会报错。

ES6 明确规定，如果区块中存在 `let` 和 `const` 命令，这个区块对这些命令声明的变量，从一开始就形成了封闭作用域。凡是在声明之前就使用这些变量，就会报错。

总之，在代码块内，使用 `let` 命令声明变量之前，该变量都是不可用的。这在语法上，称为“暂时性死区”（temporal dead zone，简称 TDZ）。

```

if (true) {
  // TDZ开始
  tmp = 'abc'; // ReferenceError
  console.log(tmp); // ReferenceError

  let tmp; // TDZ结束
  console.log(tmp); // undefined

  tmp = 123;
  console.log(tmp); // 123
}

```

上面代码中，在 `let` 命令声明变量 `tmp` 之前，都属于变量 `tmp` 的“死区”。

“暂时性死区”也意味着 `typeof` 不再是一个百分之百安全的操作。

```

typeof x; // ReferenceError
let x;

```

上面代码中，变量 `x` 使用 `let` 命令声明，所以在声明之前，都属于 `x` 的“死区”，只要用到该变量就会报错。因此，`typeof` 运行时就会抛出一个 `ReferenceError`。

作为比较，如果一个变量根本没有被声明，使用 `typeof` 反而不会报错。

```

typeof undeclared_variable // "undefined"

```

上面代码中，`undeclared_variable` 是一个不存在的变量名，结果返回“`undefined`”。所以，在没有 `let` 之前，`typeof` 运算符是百分之百安全的，永远不会报错。现在这一点不成立了。这样的设计是为了让大家养成良好的编程习惯，变量一定要在声明之后使用，否则就报错。

有些“死区”比较隐蔽，不太容易发现。

```

function bar(x = y, y = 2) {
  return [x, y];
}

bar(); // 报错

```

上面代码中，调用 `bar` 函数之所以报错（某些实现可能不报错），是因为参数 `x` 默认值等于另一个参数 `y`，而此时 `y` 还没有声明，属于“死区”。如果 `y` 的默认值是 `x`，就不会报错，因为此时 `x` 已经声明了。

```

function bar(x = 2, y = x) {
  return [x, y];
}
bar(); // [2, 2]

```

另外，下面的代码也会报错，与 `var` 的行为不同。

```

// 不报错
var x = x;

// 报错
let x = x;
// ReferenceError: x is not defined

```

上面代码报错，也是因为暂时性死区。使用 `let` 声明变量时，只要变量在还没有声明完成前使用，就会报错。上面这行就属于这个情况，在变量 `x` 的声明语句还没有执行完成前，就去取 `x` 的值，导致报错“`未定义`”。

ES6 规定暂时性死区和 `let`、`const` 语句不出现变量提升，主要是为了减少运行时错误，防止在变量声明前就使用这个变量，从而导致意料之外的行为。这样的错误在 ES5 是很常见的，现在有了这种规定，避免此类错误就很容易了。

总之，暂时性死区的本质就是，只要一进入当前作用域，所要使用的变量就已经存在了，但是不可获取，只有等到声明变量的那一行代码出现，才可以获取和使用该变量。

---

## 不允许重复声明

`let` 不允许在相同作用域内，重复声明同一个变量。

```
// 报错
function func() {
  let a = 10;
  var a = 1;
}

// 报错
function func() {
  let a = 10;
  let a = 1;
}
```

因此，不能在函数内部重新声明参数。

```
function func(arg) {
  let arg; // 报错
}

function func(arg) {
  {
    let arg; // 不报错
  }
}
```

---

## 2. 块级作用域

---

### 为什么需要块级作用域？

ES5 只有全局作用域和函数作用域，没有块级作用域，这带来很多不合理的场景。

第一种场景，内层变量可能会覆盖外层变量。

```
var tmp = new Date();

function f() {
  console.log(tmp);
  if (false) {
    var tmp = 'hello world';
  }
}

f(); // undefined
```

上面代码的原意是，`if` 代码块的外部使用外层的 `tmp` 变量，内部使用内层的 `tmp` 变量。但是，函数 `f` 执行后，输出结果为 `undefined`，原因在于变量提升，导致内层的 `tmp` 变量覆盖了外层的 `tmp` 变量。

第二种场景，用来计数的循环变量泄露为全局变量。

```
var s = 'hello';

for (var i = 0; i < s.length; i++) {
  console.log(s[i]);
}

console.log(i); // 5
```

上面代码中，变量 `i` 只用来控制循环，但是循环结束后，它并没有消失，泄露成了全局变量。

---

## ES6 的块级作用域

`let` 实际上为 JavaScript 新增了块级作用域。

```
function f1() {
  let n = 5;
  if (true) {
    let n = 10;
  }
  console.log(n); // 5
}
```

上面的函数有两个代码块，都声明了变量 `n`，运行后输出 `5`。这表示外层代码块不受内层代码块的影响。如果两次都使用 `var` 定义变量 `n`，最后输出的值才是 `10`。

ES6 允许块级作用域的任意嵌套。

```
{{{{let insane = 'Hello World'}}}}};
```

上面代码使用了一个五层的块级作用域。外层作用域无法读取内层作用域的变量。

```
{{{
  {let insane = 'Hello World'}
  console.log(insane); // 报错
}}}};
```

内层作用域可以定义外层作用域的同名变量。

```
{{{{
  let insane = 'Hello World';
  {let insane = 'Hello World'}
}}}};
```

块级作用域的出现，实际上使得获得广泛应用的立即执行函数表达式（`IIFE`）不再必要了。

```
// IIFE 写法
(function () {
  var tmp = ...;
  ...
})();
```

```
// 块级作用域写法
{
  let tmp = ...;
  ...
}
```

---

## 块级作用域与函数声明

函数能不能在块级作用域之中声明？这是一个相当令人混淆的问题。

ES5 规定，函数只能在顶层作用域和函数作用域之中声明，不能在块级作用域声明。

```
// 情况一
if (true) {
  function f() {}
}

// 情况二
try {
  function f() {}
} catch(e) {
  // ...
}
```

上面两种函数声明，根据 ES5 的规定都是非法的。

但是，浏览器没有遵守这个规定，为了兼容以前的旧代码，还是支持在块级作用域之中声明函数，因此上面两种情况实际都能运行，不会报错。

ES6 引入了块级作用域，明确允许在块级作用域之中声明函数。ES6 规定，块级作用域之中，函数声明语句的行为类似于 `let`，在块级作用域之外不可引用。

```
function f() { console.log('I am outside!'); }

(function () {
  if (false) {
    // 重复声明一次函数f
    function f() { console.log('I am inside!'); }
  }

  f();
})();
```

上面代码在 ES5 中运行，会得到“I am inside!”，因为在 `if` 内声明的函数 `f` 会被提升到函数头部，实际运行的代码如下。

```
// ES5 环境
function f() { console.log('I am outside!'); }

(function () {
  function f() { console.log('I am inside!'); }
  if (false) {
  }
  f();
})();
```

ES6 就完全不一样了，理论上会得到“I am outside!”。因为块级作用域内声明的函数类似于 `let`，对作用域之外没有影响。但是，如果你真的在 ES6 浏览器中运行一下上面的代码，是会报错的，这是为什么呢？

原来，如果改变了块级作用域内声明的函数的处理规则，显然会对老代码产生很大影响。为了减轻因此产生的不兼容问题，ES6 在附录 B 里面规定，浏览器的实现可以不遵守上面的规定，有自己的行为方式。

- 允许在块级作用域内声明函数。
- 函数声明类似于 `var`，即会提升到全局作用域或函数作用域的头部。
- 同时，函数声明还会提升到所在的块级作用域的头部。

注意，上面三条规则只对 ES6 的浏览器实现有效，其他环境的实现不用遵守，还是将块级作用域的函数声明当作 `let` 处理。

根据这三条规则，在浏览器的 ES6 环境中，块级作用域内声明的函数，行为类似于 `var` 声明的变量。

```
// 浏览器的 ES6 环境
function f() { console.log('I am outside!'); }

(function () {
  if (false) {
    // 重复声明一次函数f
    function f() { console.log('I am inside!'); }
  }

  f();
})();
// Uncaught TypeError: f is not a function
```

上面的代码在符合 ES6 的浏览器中，都会报错，因为实际运行的是下面的代码。

```
// 浏览器的 ES6 环境
function f() { console.log('I am outside!'); }
(function () {
  var f = undefined;
  if (false) {
    function f() { console.log('I am inside!'); }
  }

  f();
})();
// Uncaught TypeError: f is not a function
```

考虑到环境导致的行为差异太大，应该避免在块级作用域内声明函数。如果确实需要，也应该写成函数表达式，而不是函数声明语句。

```
// 函数声明语句
{
  let a = 'secret';
  function f() {
    return a;
  }
}

// 函数表达式
{
  let a = 'secret';
  let f = function () {
    return a;
  };
}
```

另外，还有一个需要注意的地方。ES6 的块级作用域允许声明函数的规则，只在使用大括号的情况下成立，如果没有使用大括号，就会报错。

```
// 不报错
'use strict';
if (true) {
```



```
function f() {}  
}  
  
// 报错  
'use strict';  
if (true)  
  function f() {}
```

---

## 3. const 命令

---

### 基本用法

`const` 声明一个只读的常量。一旦声明，常量的值就不能改变。

```
const PI = 3.1415;  
PI // 3.1415  
  
PI = 3;  
// TypeError: Assignment to constant variable.
```

上面代码表明改变常量的值会报错。

`const` 声明的变量不得改变值，这意味着，`const` 一旦声明变量，就必须立即初始化，不能留到以后赋值。

```
const foo;  
// SyntaxError: Missing initializer in const declaration
```

上面代码表示，对于 `const` 来说，只声明不赋值，就会报错。

`const` 的作用域与 `let` 命令相同：只在声明所在的块级作用域内有效。

```
if (true) {  
  const MAX = 5;  
}  
  
MAX // Uncaught ReferenceError: MAX is not defined
```

`const` 命令声明的常量也是不提升，同样存在暂时性死区，只能在声明的位置后面使用。

```
if (true) {  
  console.log(MAX); // ReferenceError  
  const MAX = 5;  
}
```

上面代码在常量 `MAX` 声明之前就调用，结果报错。

`const` 声明的常量，也与 `let` 一样不可重复声明。

```
var message = "Hello!";  
let age = 25;  
  
// 以下两行都会报错
```

```
const message = "Goodbye!";  
const age = 30;
```

---

## 本质

`const` 实际上保证的，并不是变量的值不得改动，而是变量指向的那个内存地址所保存的数据不得改动。对于简单类型的数据（数值、字符串、布尔值），值就保存在变量指向的那个内存地址，因此等同于常量。但对于复合类型的数据（主要是对象和数组），变量指向的内存地址，保存的只是一个指向实际数据的指针，`const` 只能保证这个指针是固定的（即总是指向另一个固定的地址），至于它指向的数据结构是不是可变的，就完全不能控制了。因此，将一个对象声明为常量必须非常小心。

```
const foo = {};  
  
// 为 foo 添加一个属性，可以成功  
foo.prop = 123;  
foo.prop // 123  
  
// 将 foo 指向另一个对象，就会报错  
foo = {}; // TypeError: "foo" is read-only
```

上面代码中，常量 `foo` 储存的是一个地址，这个地址指向一个对象。不可变的只是这个地址，即不能把 `foo` 指向另一个地址，但对象本身是可变的，所以依然可以为其添加新属性。

下面是另一个例子。

```
const a = [];  
a.push('Hello'); // 可执行  
a.length = 0;    // 可执行  
a = ['Dave'];    // 报错
```

上面代码中，常量 `a` 是一个数组，这个数组本身是可写的，但是如果将另一个数组赋值给 `a`，就会报错。

如果真的想将对象冻结，应该使用 `Object.freeze` 方法。

```
const foo = Object.freeze({});  
  
// 常规模式时，下面一行不起作用；  
// 严格模式时，该行会报错  
foo.prop = 123;
```

上面代码中，常量 `foo` 指向一个冻结的对象，所以添加新属性不起作用，严格模式时还会报错。

除了将对象本身冻结，对象的属性也应该冻结。下面是一个将对象彻底冻结的函数。

```
var constantize = (obj) => {  
  Object.freeze(obj);  
  Object.keys(obj).forEach( (key, i) => {  
    if ( typeof obj[key] !== 'object' ) {  
      constantize( obj[key] );  
    }  
  });  
};
```

ES5 只有两种声明变量的方法：`var` 命令和 `function` 命令。ES6 除了添加 `let` 和 `const` 命令，后面章节还会提到，另外两种声明变量的方法：`import` 命令和 `class` 命令。所以，ES6 一共有 6 种声明变量的方法。

---

## 4. 顶层对象的属性

顶层对象，在浏览器环境指的是 `window` 对象，在 `Node` 指的是 `global` 对象。ES5 之中，顶层对象的属性与全局变量是等价的。

```
window.a = 1;
a // 1

a = 2;
window.a // 2
```

上面代码中，顶层对象的属性赋值与全局变量的赋值，是同一件事。

顶层对象的属性与全局变量挂钩，被认为是 `JavaScript` 语言最大的设计败笔之一。这样的设计带来了几个很大的问题，首先是没法在编译时就报出变量未声明的错误，只有运行时才能知道（因为全局变量可能是顶层对象的属性创造的，而属性的创造是动态的）；其次，程序员很容易不知不觉地就创建了全局变量（比如打字出错）；最后，顶层对象的属性是到处可以读写的，这非常不利于模块化编程。另一方面，`window` 对象有实体含义，指的是浏览器的窗口对象，顶层对象是一个有实体含义的对象，也是不合适的。

ES6 为了改变这一点，一方面规定，为了保持兼容性，`var` 命令和 `function` 命令声明的全局变量，依旧是顶层对象的属性；另一方面规定，`let` 命令、`const` 命令、`class` 命令声明的全局变量，不属于顶层对象的属性。也就是说，从 ES6 开始，全局变量将逐步与顶层对象的属性脱钩。

```
var a = 1;
// 如果在 Node 的 REPL 环境，可以写成 global.a
// 或者采用通用方法，写成 this.a
window.a // 1

let b = 1;
window.b // undefined
```

上面代码中，全局变量 `a` 由 `var` 命令声明，所以它是顶层对象的属性；全局变量 `b` 由 `let` 命令声明，所以它不是顶层对象的属性，返回 `undefined`。

---

## 5. global 对象

ES5 的顶层对象，本身也是一个问题，因为它在各种实现里面是不统一的。

- 浏览器里面，顶层对象是 `window`，但 `Node` 和 `Web Worker` 没有 `window`。
- 浏览器和 `Web Worker` 里面，`self` 也指向顶层对象，但是 `Node` 没有 `self`。
- `Node` 里面，顶层对象是 `global`，但其他环境都不支持。

同一段代码为了能够在各种环境，都能取到顶层对象，现在一般是使用 `this` 变量，但是有局限性。

- 全局环境中，`this` 会返回顶层对象。但是，`Node` 模块和 `ES6` 模块中，`this` 返回的是当前模块。
- 函数里面的 `this`，如果函数不是作为对象的方法运行，而是单纯作为函数运行，`this` 会指向顶层对象。但是，严格模式下，这时 `this` 会返回 `undefined`。

- 不管是严格模式，还是普通模式，`new Function('return this')()`，总是会返回全局对象。但是，如果浏览器用了 CSP（Content Security Policy，内容安全策略），那么 `eval`、`new Function` 这些方法都可能无法使用。

综上所述，很难找到一种方法，可以在所有情况下，都取到顶层对象。下面是两种勉强可以使用的方法。

```
// 方法一
(typeof window !== 'undefined'
  ? window
  : (typeof process === 'object' &&
    typeof require === 'function' &&
    typeof global === 'object')
    ? global
    : this);

// 方法二
var getGlobal = function () {
  if (typeof self !== 'undefined') { return self; }
  if (typeof window !== 'undefined') { return window; }
  if (typeof global !== 'undefined') { return global; }
  throw new Error('unable to locate global object');
};
```

现在有一个提案，在语言标准的层面，引入 `global` 作为顶层对象。也就是说，在所有环境下，`global` 都是存在的，都可以从它拿到顶层对象。

垫片库 `system.global` 模拟了这个提案，可以在所有环境拿到 `global`。

```
// CommonJS 的写法
require('system.global/shim')();

// ES6 模块的写法
import shim from 'system.global/shim'; shim();
```

上面代码可以保证各种环境里面，`global` 对象都是存在的。

```
// CommonJS 的写法
var global = require('system.global')();

// ES6 模块的写法
import getGlobal from 'system.global';
const global = getGlobal();
```

上面代码将顶层对象放入变量 `global`。

---

留言

222 Comments

ECMAScript 6 入门

 Login ▾

 Recommend 32

 Tweet

 Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name

[上一章](#)

[下一章](#)



**william** • 4 years ago

阮老师，还有个问题，“函数参数的作用域与函数体的作用域是分离的”，既然他们的作用域是分离的，那么为什么这里会报错呢，从理论上说参数`arg`应该和函数内部的`let`声明已经分离了，而`let`在不同的作用域内可以重新声明的呀。

```
function func(arg) {  
  let arg; // 报错  
}
```

43 ^ | v • Reply • Share ›



**ruanyf** Mod → **william** • 4 years ago

谢谢指出这个严重错误，已经更正了。

^ | v • Reply • Share ›



**Bendan Xiao** → **ruanyf** • 2 years ago

你好，在后面的“函数扩展”一章中，你和一个提问者讨论过这个问题，说是函数的参数有默认值时，会为其创建一个参数作用域，但是我为什么执行这段代码还是会显示错误，按你的回答来看参数作用域，和函数体作用域，是分离的。重新`let`声明`arg`



应该不冲突啊

^ | v • Reply • Share ›



**ruanyf** Mod → **Bendan Xiao** • 2 years ago

参数作用域会带入函数体，所以参数已经声明的变量，函数体内不得用`let`再次声明。

2017年1月23日 下午7:00, "Disqus" <notifications@disqus.net>写道:

^ | v • Reply • Share ›



**Bendan Xiao** → **ruanyf** • 2 years ago



还有一个问题就是：参数作用域里面声明变量是用`var`声明的？还是`let`？因为参数作用域只有用`var`声明的变量 才可以再函数体里面继续声明，否则的话，如果参数作用域用`let`声明的变量，函数体里面`var`声明应该也是报错的，原因是不可重复定义`let`变量。不知道这样理解对不对

^ | v • Reply • Share ›



**jerry** → **Bendan** : 上一章 a 下一章

在函数体内用`let`或`const`重定义变量是一个Early Error 见: <https://www.ecma>

在函数体内用let或const重定义变量是一个 Early Error，见：<https://www.ecma-international.org/ecma-262/6.0/#sec-early-errors>。当函数参数有默认值得时候，参数会有一个独立的作用域，见：<https://stackoverflow.com/a/11810144>

1 ^ | v • Reply • Share ›



宋增 ➔ Bendan Xiao • a year ago

函数体内参数应该属于 var 声明的

^ | v • Reply • Share ›



ruanyf Mod ➔ Bendan Xiao • 2 years ago

基本上，这个只能作为一条单独规则，进行记忆了。

函数体内使用 var 声明同一个参数变量，不报错，并绑定函数体作用域。

使用 let 声明同一个参数变量，就会报句法错误。

2017-01-24 10:00 GMT+08:00 Disqus <notifications@disqus.net>:

^ | v • Reply • Share ›



宋增 ➔ ruanyf • a year ago

函数体内的参数应该属于 var 声明的，再次使用 var 声明，不会报错，使用 let 声明会报语法错误。

^ | v • Reply • Share ›



Seaborn Lee • 3 years ago

...

上面代码中，变量 i 是 let 声明的，当前的 i 只在本轮循环有效，所以每一次循环的 i 其实都是一个新的变量，所以最后输出的是 6。

...

如果每一次循环的 i 都是一个新变量，那它的值是怎么得到传递的呢？

如果您说的是正确的，那它应该永远是 0 啊。

53 ^ | v • Reply • Share ›



Albert Yu ➔ Seaborn Lee • 3 years ago

这里的说法的确不完整，let i = 0 在整个循环过程中其实只执行了一次（你可以在循环开始处打断点步进观察），由于 let 创建的变量是尊重块级作用域的，所以的确每次循环都是一个新的 i，但是赋值为 0 的动作只有第一次，后面每次的值都是 i++ 的结果。因为每次循环的 i 都是一个独立的变量（内存里的唯一地址），因此闭包记录的值都是唯一的，所以才能得到最终的结果。

如果用 var 的话，变量 i 是一个全局变量，虽然循环体内每次都创建了一个函数来打印 i 的值，但是当时当刻仅仅是一个指向全局变量 i 的指针，当循环结束之后无论你用哪一个下标去访问循环创建的闭包函数，打印的变量 i 都是全局的那一个，所以全部都是 9。

12 ^ | v • Reply • Share ›



夏末星辰 ➔ Albert Yu • 9 months ago

应该是10

3 ^ | v • Reply • Share ›



Xie Min ➔ Albert Yu • 2 years ago

That's right : )

[上一章](#)

[下一章](#)

^ | v • Reply • Share ›



**ruanyf** Mod → Seaborn Lee • 3 years ago

JavaScript 引擎会记住每一次 `let` 的值，下一次创建的时候，直接在这个值上递进。具体过程，规格里面有写。 <http://www.ecma-international...>

6 ^ | v • Reply • Share ›



刘峰 → ruanyf • 2 months ago

```
let j = 0;
for(let i = 0,k=j+1; i < 10; i++,j++) {
  !function () {
    console.log('i:',i);
    console.log('k:',k);
    console.log('j:',j);
  }()
}
// k的值总是为1, 是否说明 let i = 0,k=j+1; 只执行了一次
```

^ | v • Reply • Share ›



crazyshoubin → 刘峰 • 2 months ago

我理解`k=j+1`不是在循环的块内部

^ | v • Reply • Share ›



遁地龙卷风 • 2 years ago

感觉文章讲的过于详细了些，入门还是突出重点的好，较为深冷的知识点作为扩展或深入阅读

3 ^ | v • Reply • Share ›



Scott Xing → 遁地龙卷风 • a year ago

阮老师的风格就是类似《C++ Primer》的风格，这是一种风格。 @ruanyf

^ | v • Reply • Share ›



J.W • 5 years ago

用 `let` 声明的变量的是从声明的地方开始才能引用的，这和 `var` 不一样，应该注明。

```
(function () { 'use strict';
  console.log(x); // ReferenceError: x is not defined
  let x = 3;
  console.log(x);
})();
```

3 ^ | v • Reply • Share ›



ruanyf Mod → J.W • 5 years ago

谢谢提醒，已经加上这段了。用了MDN中的例子。

^ | v • Reply • Share ›



Genffy → J.W • 4 years ago

`var` 声明也是这样的吧？

^ | v • Reply • Share ›




木马人 → Genffy • 3 years ago


文中写的很清楚了，`var` 是全局变量，在云狐 上一章 `undefined` 错误，而是初始值是 `undefined` 下一章

^ | v • Reply • Share ›

 **Imbai** ➔ 木马人 • a year ago  
haha  
^ | v • Reply • Share ›

 **郑建兵** ➔ 木马人 • 3 years ago  
初始值是undefined  
^ | v • Reply • Share ›

 **木马人** ➔ 郑建兵 • 3 years ago  
yeah, you are right, i don't know why i write 'null', huaaaa!  
^ | v • Reply • Share ›

 **Amor** ➔ 木马人 • 3 years ago  
huaaaa!  
^ | v • Reply • Share ›




**John Youi** • 5 years ago


我不喜欢看翻译书的原因之一是会常出现中文词配英文缩写的不友好。例如var是variable，variable是变量，却基本没有中文书会注意这点的。正文里通篇变量二字，接下来就直接在代码，注释，还是文件名里写var。老外懒得打字用缩写的基础是他们能在上下文确保读者知道相应的单词，中国人没有些编程经验的人，哪里看出fn是函数的意思。

好吧，其实我被let惹了，这是什么缩写还是取了let something be some thing的意思。


3 ^ | v • Reply • Share ›

 **jerryppy** ➔ John Youi • a year ago  
这里有原因，官方也引用了这个回答：<https://stackoverflow.com/q...>  
^ | v • Reply • Share ›

 **Baiyi Liao** ➔ John Youi • 2 years ago  
我觉得阮一峰大神的这本书主要是面向那些有其他语言基础的人的（例如我就有一点C++的基础）  
^ | v • Reply • Share ›

 **JeffOwOSun** ➔ John Youi • 3 years ago  
英文中let就是“令”的意思了  
^ | v • Reply • Share ›

 **dou4cc** ➔ John Youi • 4 years ago  
BASIC中就用let声明变量  
^ | v • Reply • Share ›

 **hbrls** ➔ John Youi • 4 years ago  
习惯就是这样，从 LISP 抄得吧  
^ | v • Reply • Share ›

 **Frank Fang** ➔ John Youi • 5 years ago  
我也觉得 let 是这个意思  
上一章 下一章



我也觉得 let 是这么回事  
^ | v • Reply • Share ›



杨伟荣 • 2 years ago

阮老师，根据stackoverflow上的一篇文章的说法，let, const等也都是存在变量提升的，不过他们提升后，并不会像var等用undefined进行初始化，而是维持一个未初始化的状态，直到赋值语句执行，在此之前引用同样会报ReferenceError错误。附链接：<http://stackoverflow.com/qu...>

1 ^ | v • Reply • Share ›



LeeJunhui → 杨伟荣 • 2 years ago

我跑到sf上看了一下，确实是这么回事，不论是var,还是let,const声明语句，都会发生hoisted(变量提升)。

^ | v • Reply • Share ›



ruanyf Mod → LeeJunhui • 2 years ago

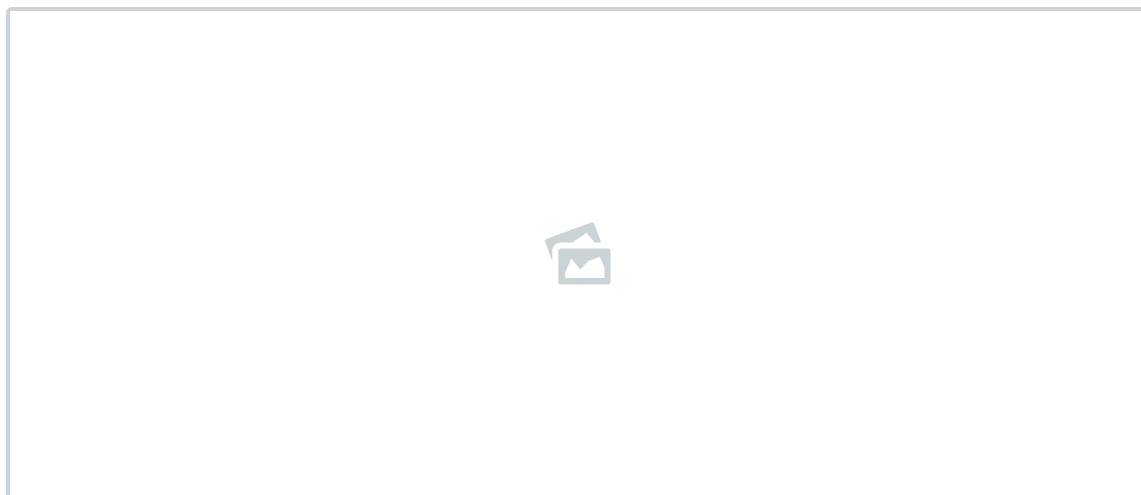
不能这么理解。在`const`声明之前，使用该变量，会报错的，这就不能称为“变量提升”。

所谓“变量提升”，是指代码不报错，还能运行。

^ | v • Reply • Share ›



LeeJunhui → ruanyf • 2 years ago



那阮老师您看这上面的说法是不是有误呢？

^ | v • Reply • Share ›



ruanyf Mod → LeeJunhui • 2 years ago

那篇帖子里面的说法，我认为是不合适的。

^ | v • Reply • Share ›



MagiCarbon → ruanyf • 2 years ago

<https://developer.mozilla.o...>

> In ECMAScript 2015, let will hoist the variable to the top of the block. However, referencing the variable in the block before the variable declaration results in a ReferenceError. The variable is in a "temporal dead zone" from the start of the block until the declaration is processed.

^ | v • Reply • Share ›



ruanyf Mod → MagiCarbon • 2 years ago



ruanyf mod → magicearon • 2 years ago  
 谢谢指出，官方文档这样说，当然以它为准。

2017年1月23日 下午3:11, "Disqus" <notifications@disqus.net>写道:

^ | v • Reply • Share ›



jerryppy → ruanyf • a year ago

> In ECMAScript 2015, let bindings are not subject to Variable Hoisting, which means that let declarations do not move to the top of the current execution context.

官方已经修改了说辞，并不会提升。

^ | v • Reply • Share ›



Bendan Xiao → ruanyf • 2 years ago

貌似文章中“let变量声明不会提升”的错误还未更改~

^ | v • Reply • Share ›



ruanyf Mod → Bendan Xiao • 2 years ago

想了想，“变量提升”的定义应该是：变量可以在声明之前使用。

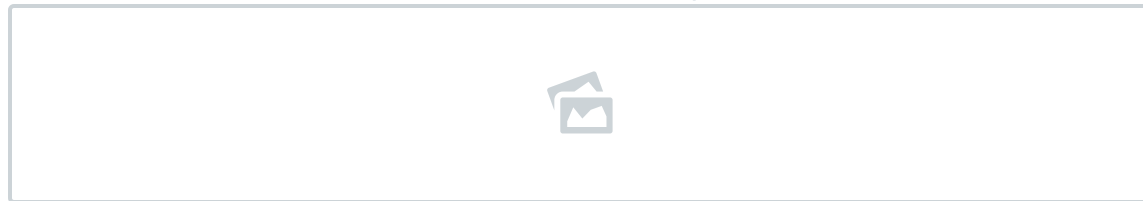
按照这种定义，let 声明的变量如果在声明语句前使用，是报错的，所以不算“变量提升”。因为它是无法运行的，所以不能说这种现象存在。

^ | v • Reply • Share ›



ruanyf Mod → Bendan Xiao • 2 years ago

规格里面并没有说到“提升”，我再想想怎么修改。<http://www.ecma-international...>



^ | v • Reply • Share ›



jerryppy → ruanyf • a year ago

根据规格里说的，其实这些变量都是“提升”（在声明语句之前就被创建）了的，但是使用let和const特殊的一点是存在TDZ，导致了在声明语句真正执行之前无法直接访问。

^ | v • Reply • Share ›



Frank Fang → ruanyf • 2 years ago

在文档里搜索 let hoisting 试试。

^ | v • Reply • Share ›



xgqfrms • 2 years ago

```
{
let a = 10;
var b = 1;
}
//Uncaught SyntaxError: Unexpected identifier (这句，为什么会报错呀？)
a
// Uncaught ReferenceError: a is not d    上一章    a    下一章    function)
b
```

//1

1 ^ | v • Reply • Share ›



**ruanyf** Mod ➔ xgqfrms • 2 years ago

最前面加一个分号。

^ | v • Reply • Share ›



**xgqfrms** ➔ ruanyf • 2 years ago

```

;{
let a = 10;
var b = 1;
}
//undefined
{
let a = 10;
var b = 1;
}
// Uncaught SyntaxError: Unexpected identifier
(; 这是为什么，什么意思，有什么区别，不太明白！)
```

^ | v • Reply • Share ›



**李鑫** ➔ xgqfrms • 2 years ago

不报错呀，你在什么环境写的，找到什么原因了吗？

^ | v • Reply • Share ›

Load more comments

ALSO ON ECMAScript 6 入门

## Class和Module

116 comments • 5 years ago

刘亮 — `class Point { constructor() { // ... } toString() { // ... } toValue() { // ... }}` 等同于 `Point.prototype = { ...`

## 函数的扩展

188 comments • 5 years ago

Doujun Zhang —

## 数组的扩展

71 comments • 5 years ago

liyun — 不好意思，之前是我理解错了浅拷贝和深拷贝的概念

## Class 的继承

38 comments • 2 years ago

Francie Who —

Subscribe Add Disqus to your siteAdd DisqusAdd Disqus' Privacy PolicyPrivacy PolicyPrivacy Policy

