

let 和 const 命令

1. let 命令

1.1. 基本用法

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

上例中，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
```

上例中，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。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。

`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
```

上例正确运行，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（同一个作用域不可使用 `let` 重复声明同一个变量）。

1.2. 暂时性死区

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。

“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: x is not defined
let x;
```

上例中，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。

如果一个变量根本没有被声明，使用 `typeof` 反而不会报错。

```
typeof x; // "undefined"
```

养成良好的编程习惯，变量一定要在声明之后使用。

1.3. 不允许重复申明

`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;
}
```

不能在函数内部重新声明参数。

```
function func(arg) {
  let arg; // SyntaxError: Identifier 'arg' has already been declared
}

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

```
}  
}
```

2. 块级作用域

块级作用域的出现，使得匿名立即执行函数表达式（匿名 IIFE）不再必要了。

```
// 立即执行函数表达式（IIFE）写法，下面两种写法同理  
(function () { var tmp = 3; console.log(tmp); })(); // 3  
(function () { var tmp = 3; console.log(tmp); })(); // 3  
  
// 块级作用域写法  
{  
  let tmp = 4;  
  console.log(tmp);  
}  
// 4
```

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。

```
// 块级作用域内部的函数声明语句，建议不要使用  
{  
  let a = 'secret';  
  function f() {  
    return a;  
  }  
}  
  
// 块级作用域内部，优先使用函数表达式  
{  
  let a = 'secret';  
  let f = function () {  
    return a;  
  };  
}
```

ES6 的块级作用域必须有大括号，如果没有大括号，JavaScript 引擎就认为不存在块级作用域。

```
// 第一种写法，报错  
if (true) let x = 1;  
// SyntaxError: Lexical declaration cannot appear in a single-statement context 词  
法声明不能出现在单语句上下文中  
  
// 第二种写法，不报错  
if (true) {
```

```
let x = 1;
}
```

上例中，第一种写法没有大括号，所以不存在块级作用域，而 **let** 只能出现在当前作用域的顶层，所以报错。第二种写法有大括号，所以块级作用域成立。

3. const 命令

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.
```

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
```

3.1. ES6 声明变量的方法

ES6 声明变量的方法有六种：**var**、**function**、**let**、**const**、**import**、**class**。

- 使用 **let** 关键字。**let** 是块级作用域，只在声明它的块或子块中可见。允许在同一作用域中多次声明相同的变量。

```
let x = 10;
let x = 20; // 重新声明 x
```

- 使用 **const** 关键字。**const** 也是块级作用域，但一旦赋值就不能改变。这使得 **const** 很适合用于不想被改变的值。

```
const PI = 3.14159;
```

- 使用 **var** 关键字。这是在 ES6 之前的主要变量声明方法，但在 ES6 中，**var** 的使用已经被 **let** 和 **const** 取代。**var** 的作用域是函数级的，而不是块级的。

```
var x = 10;
```

- 使用 **function** 声明来定义变量。这种方式是在函数的参数列表中使用变量名，并在函数体内部使用该变量。

使用函数声明定义了一个名为 `myVariable` 的变量，然后，可以通过调用 `myFunction` 函数并将参数传递给它来设置 `myVariable` 的值：

```
function myFunction(myVariable) {  
  // 在函数体内部使用 myVariable  
  console.log(myVariable);  
}  
myFunction("Hello, world!");  
// myVariable 的值将被设置为字符串 "Hello, world!"
```

- 使用 `class` 关键字声明类，并在类中声明变量。这是在 ES6 中引入的新特性，允许使用面向对象编程。

```
class MyClass {  
  constructor() {  
    this.x = 10;  
  }  
}
```

- 使用 `import` 和 `export` 进行模块化的变量声明和导出。这也是在 ES6 中引入的新特性，允许创建独立的模块并在需要时导入。

```
// moduleA.js  
export const x = 10;  
  
// moduleB.js  
import { x } from './moduleA';
```

4. globalThis 对象

JavaScript 语言存在一个顶层对象，它提供全局环境（即全局作用域），所有代码都是在这个环境中运行。但是，顶层对象在各种实现里面是不统一的。

- 浏览器环境：顶层对象是 `window` 和 `self`。
- `Web Worker` 环境：顶层对象是 `self`。
- `Node` 环境：顶层对象是 `global`。

ES2020 在语言标准的层面，引入 `globalThis` 作为顶层对象。也就是说，任何环境下，`globalThis` 都是存在的，都可以从它拿到顶层对象，指向全局环境下的 `this`。

```
// 在浏览器环境下  
window === self; // true  
window === globalThis; // true
```