**LET命令**

1. 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。

在for循环的时候很适合用let

var a = [];

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

a[i] = function () {

console.log(i);

};

}

a[6](); // 6而不是10

1. 不存在变量提升

console.log(foo); // 输出undefined

console.log(bar); // 报错ReferenceError

var foo = 2;

let bar = 2;

1. 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。

if (true) {

tmp = 'abc'; // ReferenceError

let tmp;

}

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）。“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而以前会输出’underfined’

let x;

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，在y定义之前使用y

1. 不允许重复声明

// 报错

function () {

let a = 10;

var a = 1;

}

// 报错

function () {

let a = 10;

let a = 1;

}

// 报错

function () {

var a = 10;

let a = 1;

}

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

**}**

}

**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，因为在函数里，var声明提前了

第二种场景，用来计数的循环变量泄露为全局变量。

For循环的i变量没有被销毁。

1. Es6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。如果使用都用var定义变量n，最后输出的值就是10。

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

// IIFE写法

(function () {

var tmp = ...;

...

}());

1. 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，是一个相当令人混淆的问题。

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。

ES6的运行结果就完全不一样了，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。且不提升头部。

这种行为差异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现代浏览器规定：

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。

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;

};

}

**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

const foo;

// 只声明不赋值也报错。SyntaxError: Missing initializer in const declaration

PI = 3;

// 报错。TypeError: Assignment to constant variable.

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。

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。

const声明的常量，也与let一样不可重复声明。

对于复合类型的变量，变量名不指向数据，而是指向数据所在的地址。

const foo = {};

foo.prop = 123;

foo.prop

// 123

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;

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, value) => {

if ( typeof obj[key] === 'object' ) {

constantize( obj[key] );

}

});

};

**全局对象的属性**

全局对象是最顶层的对象，在浏览器环境指的是window对象，在Node.js指的是global对象。ES5之中，全局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，全局变量a由var命令声明，所以它是全局对象的属性；全局变量b由let命令声明，所以它不是全局对象的属性，返回undefined。

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

**Global对象**

* 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，内容安全政策），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](https://github.com/tc39/proposal-global)，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](https://github.com/ljharb/System.global)（https://github.com/ljharb/System.global）模拟了这个提案，可以在所有环境拿到global。