**第2章 let和const命令**

2.1 let命令

**基本用法**

ES6新增了let命令，用于声明变量。其用法类似于var，但是所声明的变量只在let命令所在的代码块内有效。

1. {

2.     let a = 10;

3.     var b = 1;

4. }

5.

6. a // ReferenceError: a is not defined

7. b // 1

8.

上面的代码在代码块中分别用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声明的变量只在其所在代码块有效。

for循环的计数器，就很适合使用let命令。

1. for (let i = 0; i < arra.length; i++) {}

2.

3. console.log(i);

4. // ReferenceError

5.

以上代码中的计数器i，只在for循环体内有效。

下面的代码如果使用var，最后将输出10。

1. var a = [];

2. for (var i = 0; i < 10; i++) {

3.     a[i] = function () {

4.         console.log(i);

5.     };

6. }

7.

8. a[6](); // 10

9.

上面的代码中，变量i是var声明的，在全局范围内都有效。所以每一次循环，新的i值都会覆盖旧值，导致最后输出的是最后一轮的i值。

如果使用let，声明的变量仅在块级作用域内有效，最后将输出6。

1. var a = [];

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

3.     a[i] = function () {

4.         console.log(i);

5.     };

6. }

7.

8. a[6](); // 6

9.

上面的代码中，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，于是最后输出的是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。

1. for (let i = 0; i < 3; i++) {

2. let i = 'abc';

3. console.log(i);

4. }

5. // abc

6. // abc

7. // abc

8.

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（同一个作用域不可使用 let 重复声明同一个变量）。

**不存在变量提升**

let不像var那样会发生“变量提升”现象。所以，变量一定要在声明后使用，否则报错。

1. console.log(foo); // ReferenceError

2. let foo = 2;

3.

上面的代码在声明foo之前就使用这个变量，结果会抛出一个错误。

这也意味着typeof不再是一个百分百安全的操作。

1. typeof x; // ReferenceError

2. let x;

3.

上面的代码中，由于typeof运行时x尚未声明，所以会抛出一个ReferenceError。

**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。

1. var tmp = 123;

2. if (true) {

3. tmp = ‘abc’; // ReferenceError

4. let tmp;

5. }

6.

上面的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，如果在区块中存在let和const命令，则这个区块对这些命令声明的变量从一开始就形成封闭作用域。只要在声明之前使用这些变量，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）。

1. if (true) {

2.     // TDZ开始

3.     tmp = 'adc';    // ReferenceError

4.     console.log(tmp);   // ReferenceError

5.

6.     let tmp;    //  TDZ结束

7.     console.log(tmp);   // undefined

8.

9.     tmp = 123;

10.     console.log(tmp);   // 123

11. }

12.

上面的代码中，在let命令声明变量tmp之前，都属于变量tmp的“死区”。

有些“死区”比较隐蔽，不太容易发现。

1. function bar(x = y, y = 2) {

2.     return [x, y];

3. }

4. bar();  // [2, 2] ReferenceError: Cannot access 'y' before initialization

5.

上面的代码中，调用bar函数之所以报错，是因为参数x的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已声明。

1. function bar(x = 2, y = x) {

2.     return [x, y];

3. }

4. bar();  // [2, 2]

5.

ES6规定暂时性死区和不存在变量提升，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中是很常见的，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存在，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。

1. // 报错

2. function() {

3. let a = 10;

4. var a = 1;

5. }

6. // 报错

7. function() {

8. let a = 10;

9. let a = 1;

10. }

11.

因此，不能在函数内部重新声明参数。

1. function(arg) {

2. let arg; // 报错

3. }

4.

5. function (arg) {

6. {

7. let arg; // 报错

8. }

9. }

10.

2.2 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，这带来了很多不合理的场景。

第一种场景，内层变量可能会覆盖外层变量。

1. var tmp = new Date();

2.

3. function f() {

4.

5.   console.log(tmp);

6.

7.   if (false) {

8.     var tmp = "hello world";

9.   }

10. }

11.

12. f(); // undefined

13.

上面的代码中，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漏为全局变量。

1. var s = "hello";

2. for (var i = 0; i < s.length; i++) {

3.   console.log(s[i]);

4. }

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

6.

上面的代码中，变量i只用来控制循环。但是循环结束后，它并没有消失，而是泄漏成了全局变量。

**ES6的块级块级作用域**

let实际上为JavaScript新增了块级作用域。

1. function f1() {

2.

3.   let n = 5;

4.   if (true) {

5.     let n = 10;

6.   }

7.   console.log(n); // 5

8. }

9.

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var定义变量n，最后输出的值就是10。

ES6允许块级作用域任意嵌套。

{{{{{ let insane = “hello world” }}}}};

上面的代码使用了一个5层的块级作用域。外层作用域无法读取内层作用域的变量。

1. {{{

2.   { let insane = "Hello World!" }

3.   console.log(insane); // 报错

4. }}}

5.

内层作用域可以定义外层作用域的同名变量。

1. {{{

2.   { let insane = "Hello World!" }

3.   console.log(insane); // 报错

4. }}}

5.

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。

1. // IFEE

2. (function () {

3.   var tmp = ...;

4.   ...

5. })();

6.

7. // 块级作用域写法

8. {

9.   let tmp = ...;

10.   ...

11. }

12.

另外，ES6也规定，函数本身的作用域在其所在的块级作用域之内。

1. function f() {

2.   console.log("I am outside");

3. }

4.

5. (function () {

6.   if (false) {

7.     // 重复声明一次函数f

8.     function f() {

9.       console.log("I am inside");

10.     }

11.   }

12.   f();

13. }());

14.

上面的代码在ES5中运行，会得到I am inside！，但是在ES6中运行，会得到I am outside！。这是因为ES5存在函数提升，不管会不会进入if代码块，函数声明都会提升到当前作用域的顶部而得到执行；而ES6支持块级作用域，不管会不会进入if代码块，其内部声明的函数皆不会影响到作用域的外部。

1. {

2.   let a = "secret";

3.   function f() {

4.     return a;

5.   }

6. }

7.

8. f(); // 报错

9.

上面的代码中，块级作用域外部无法调用块级作用域内部定义的函数。如果确实需要调用，则要像下面这样处理。

1. let f;

2. {

3. let a = “secret”;

4. f = function () {

5. return a;

6. }

7. }

8. f(); // secret

9.

需要注意的是，如果在严格模式下，函数只能在顶层作用域和函数内声明，其他情况（比如if代码块、循环代码块）下的声明会报错。

2.3 const命令

const用来声明常量。一旦声明，其值就不能改变。

1. const PI = 3.1415;

2. PI // 3.1415

3. PI = 3;

4. // TypeError: “PI” is read-only

5.

上面的代码表明改变常量的值会报错。

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

1. const foo;

2. // SyntaxError: Missing initializer in const declaration

3.

上面的代码表示，对于const而言，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。

1. if (true) {

2.   const MAX = 5;

3. }

4.

5. MAX // ReferenceError: MAX is not defined

6.

const命令声明的常量也不提升，同样存在暂时性死区，只能在声明后使用。

1. if (true) {

2.   console.log(MAX); // ReferenceError

3.   const MAX = 5;

4. }

5.

上面的代码在常量MAX声明之前就调用，结果报错。

与let命令一样，const也不可重复声明常量。

1. var message = "Hello";

2. let age = 25;

3.

4. // 以下两行都会报错

5. const message = "Goodbye!";

6. const age = 30;

7.

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命令只是保证变量名指向的地址不变，并不保证改地址的数据不变，所以将一个对象声明为常量必须非常小心。

1. const foo = {};

2. foo.prop = 123;

3.

4. console.log(foo.prop); // 123

5.

6. foo = {}; //  ReferenceError: "foo" is read-only不起作用 [TypeError: Assignment to constant variable.]

7.

上面的代码中，常量foo存储的是一个地址，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。

1. const a = [];

2. a.push("Hello"); // 可执行

3. a.length = 0;   // 可执行

4. a = ["Dave"];   // 报错

5.

上面的代码中，常量a是一个数组，这个数组本身是可写的，但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。

1. const foo = Object.freeze({});

2. foo.prop = 123; // 不起作用

3.

上面的代码中，常量foo指向一个冻结的对象，所以添加新属性不起作用。

除了将对象本身冻结，对象的属性也应该冻结。下面是将一个对象彻底冻结的函数。

1. var constantize = (obj) => {

2.     Object.freeze(obj);

3.     Object.keys(obj).forEach( (key) => {

4.         if (typeof obj[key] == "object") {

5.             constantize(obj[key]);

6.         }

7.     } );

8. };

9.

ES5只有两种声明变量的方法：var命令和function命令。ES6除了添加了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中声明变量的方法。

2.4 跨模块常量

上面说过，const声明的常量只在当前代码块有效。如果像设置跨模块的常量，可以采用下面的写法。

1. // constants.js模块

2. export const A = 1;

3. export const B = 2;

4. export const C = 3;

5.

6. // test1.js模块

7. import \* as constants from './constants.js';

8. console.log(constants.A);

9. console.log(constants.B);

10. console.log(constants.C);

11.

12. // test2.js模块

13. import {A, B} from "./constants.js";

14. console.log(A);

15. console.log(B);

16.

2.5 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中指的是global对象。在ES5中，全局对象的属性和全局变量是等价的。

1. window.a = 1;

2. a // 1

3.

4. a = 2;

5. window.a; // 2

6.

上面的代码中，全局对象的属性赋值与全局变量的赋值，是同一件事。（对于Node.js来说，这一条只对REPL环境适用，模块环境中，全局变量必须显示声明成global对象的属性。）这种规定被视为JavaScript语言的一大问题，因为很容易不知不觉就创建了全局变量。ES6为了改变这一点，一方面规定，var命令和function命令声明的全局变量依旧是全局对象的属性；另一方面，let命令、const命令和class命令声明的全局变量不属于全局对象的属性。

1. var a = 1;

2. // 如果在Node.js的REPL环境，可以写成global.a

3. // 或者采用通用方法，写成this.a

4. window.a // 1

5.

6. let b = 1;

7. window.b // undefined

8.

上面的代码中，全局变量a由var命令声明，所以是全局对象的属性；全局变量b是由let命令声明，所以不是全局对象的属性，返回undefined。