let和const命令

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
```

其他

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 < arr.length; i++) {}

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

上面代码的计数器`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`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`的值。

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。

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。

```
console.log(foo); // 输出undefined
console.log(bar); // 报错ReferenceError

var foo = 2;
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]
```

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 () {
  let a = 10;
  var a = 1;
}
```

```
// 报错
function () {
  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匿名函数（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，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) {
}
```

上面代码的两种函数声明，根据ES5的规定都是非法的。

但是，浏览器没有遵守这个规定，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。不过，"严格模式"下还是会报错。

```
// ES5严格模式
'use strict';
if (true) {
  function f() {}
}
// 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严格模式
'use strict';
if (true) {
  function f() {}
}
// 不报错
```

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

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，实际运行的代码如下。

```
// ES6版本
function f() { console.log('I am outside!'); }
(function () {
  f();
}());
```

很显然，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，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。

前面那段代码，在Chrome环境下运行会报错。

```
// ES6的浏览器环境
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
```

上面的代码报错，是因为实际运行的是下面的代码。

```
// ES6的浏览器环境
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

`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;
```

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，并不保证该地址的数据不变，所以将一个对象声明为常量必须非常小心。

```
const foo = {};
```

```
foo.prop = 123;

foo.prop
// 123

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, value) => {
    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. 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，全局对象的属性赋值与全局变量的赋值，是同一件事。（对于Node来说，这一条只对REPL环境适用，模块环境之中，全局变量必须显式声明成`global`对象的属性。）

未声明的全局变量，自动成为全局对象`window`的属性，这被认为是JavaScript语言最大的设计败笔之一。这样的设计带来了两个很大的问题，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，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）。另一方面，从语义上讲，语言的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，依旧是全局对象的属性；另一方面规定，`let`命令、`const`命令、`class`命令声明的全局变量，不属于全局对象的属性。也就是说，从ES6开始，全局变量将逐步与全局对象的属性脱钩。

```
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`。

留言

上一章 下一章