**let命令**

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，就很适合使用let命令。

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

console.log(i);

//ReferenceError: i is not defined

上面的代码的计数器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9。

var a = [];

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

var c = i;

a[i] = function () {

console.log(c);

};

}

a[6](); //9

**let在loop中怎么工作的？**

**参考一：**

How is this working?

If you use that let keyword in the for statement, it will check what names it does bind and then create a new lexical environment with those names for a) the initialiser expression b) each iteration (previosly to evaluating the increment expression) copy the values from all variables with those names from one to the next environment

Your loop statement for (var i = 0; i < 10; i++) { process.nextTick(\_ => console.log(i)) } desugars to a simple

// omitting braces when they don't introduce a block

var i;

i = 0;

if (i < 10)

process.nextTick(\_ => console.log(i))

i++;

if (i < 10)

process.nextTick(\_ => console.log(i))

i++;

…

While  for (let i = 0; i < 10; i++) { process.nextTick(\_ => console.log(i)) }  does "desugar" to the much more complicated

// using braces to explicitly denote block scopes,// using indentation for control flow{ let i;

i = 0;

\_\_status = {i};}{ let {i} = \_\_status;

if (i < 10)

process.nextTick(\_ => console.log(i))

\_\_status = {i};} { let {i} = \_\_status;

i++;

if (i < 10)

process.nextTick(\_ => console.log(i))

\_\_status = {i};

} { let {i} = \_\_status;

i++;

…

the execution context  of let, It's just a scope that contains one variable i. It is referenced by the closure, and it references the scope in which the loop was contained in as its outer link.

1. \_\_status is part of the loop context state, in my syntax denoted by an object, that is used to transport the current value(s) of the block-declared variable(s) in the loop header from one scope to the next. 2) Yes, each iteration creates a new scope with a new variable i. This does not depend on the statements in the block (like the process.nextTick call) though

（转载自https://stackoverflow.com/questions/30899612/explanation-of-let-and-block-scoping-with-for-loops）

**参考二：**

**Loops of the form for (let x...) create a fresh binding for x in each iteration.**

This is a very subtle difference. It means that if a for (let...) loop executes multiple times, and that loop contains a closure, as in our talking cat example, **each closure will capture a different copy of the loop variable, rather than all closures capturing the same loop variable.**

#### **let Loops**

A particular case where let shines is in the for-loop case as we discussed previously.

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

console.log( i );

}

console.log( i ); // ReferenceError

Not only does let in the for-loop header bind the i to the for-loop body, but in fact, it **re-binds it** to each iteration of the loop, making sure to re-assign it the value from the end of the previous loop iteration.

Here's another way of illustrating the per-iteration binding behavior that occurs:

{

let j;

for (j=0; j<10; j++) {

let i = j; // re-bound for each iteration!

console.log( i );

}

}

The reason why this per-iteration binding is interesting will become clear in Chapter 5 when we discuss closures.

Because let declarations attach to arbitrary blocks rather than to the enclosing function's scope (or global), there can be gotchas where existing code has a hidden reliance on function-scoped var declarations, and replacing the var with letmay require additional care when refactoring code.

（转载自https://github.com/getify/You-Dont-Know-JS/blob/master/scope%20%26%20closures/ch3.md#hiding-in-plain-scope）

See [this transpiled version](https://babeljs.io/repl/" \l "?evaluate=true&presets=es2015&code=for (let j = 0; j < 5; j++) {%0D%0A  setTimeout(function() {%0D%0A    console.log(`j: ${j} seconds`);%0D%0A  }, 5000 + j * 1000);%0D%0A}) on Babel's REPL to see this demonstrated. What happens when you declare a let variable in a for loop like that is that a new declarative environment is created to hold that variable ([details here](http://www.ecma-international.org/ecma-262/6.0/index.html" \l "sec-for-statement-runtime-semantics-labelledevaluation)), and then ****for each loop iteration**** *another* declarative environment is created to hold a per-iteration copy of the variable; each iteration's copy is initialized from the previous one's value ([details here](http://www.ecma-international.org/ecma-262/6.0/index.html" \l "sec-forbodyevaluation)), but they're separate variables, as demonstrated in the link by the values output by the closures.

（转载自https://stackoverflow.com/questions/37792934/why-is-let-slower-than-var-in-a-for-loop-in-nodejs）

**Let用法**

（参考：<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let）>

**内用摘要**

When used inside a block, **let limits the variable's scope to that block**. Note the difference between **var** whose scope is inside the function where it is declared.

var a = 1;var b = 2;

if (a === 1) {

var a = 11; // the scope is global

let b = 22; // the scope is inside the if-block

console.log(a); // 11

console.log(b); // 22}

console.log(a); // 11

console.log(b); // 2

At the top level of programs and functions, let, unlike var, **does not create a property on the global object**. For example:

var x = 'global';let y = 'global';

console.log(this.x); // "global"

console.log(this.y); // undefined

**Redeclaring** the same variable within the same function or block scope raises a [SyntaxError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError" \o "The SyntaxError object represents an error when trying to interpret syntactically invalid code.).

if (x) {

let foo;

let foo; // SyntaxError thrown.}

**temporal dead zone**

In ECMAScript 2015, let will hoist the variable declaration to the top of the block, but not the initialization. Referencing the variable in the block before the initialization results in a [ReferenceError](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/ReferenceError) (contrary to a variable declared with [var](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var" \l "var_hoisting), which will just have the undefined value). **The variable is in a "temporal dead zone" from the start of the block until the initialization is processed.**

function do\_something() {

console.log(bar); // undefined

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

var bar = 1;

let foo = 2;}

You may encounter errors in [switch](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/switch) statements because there is only one underlying block.

switch(x) {

case 0:

let foo;

break;

case 1:

let foo; // SyntaxError for redeclaration.

break;}

### Another example of temporal dead zone combined with lexical scoping

Due to lexical scoping, the **identifier "foo"** inside the expression (foo + 55) evaluates to the if block's foo, and **not** the overlying variable foo with the value of 33.  
In that very line, the if block's "foo" has already been defined and hoisted, but has not yet reached (and **terminated**) its declaration statement (which is the statement itself): it's still in the temporal dead zone.

function test(){

var foo = 33;

if (true) {

let foo = (foo + 55); // ReferenceError: foo is not defined

}}test();

This phenomenon may confuse you in a situation like the following. The instruction "let n of n.a" is already inside the private scope of the for loop's block, hence the **identifier "n.a"** is resolved to the property 'a' of the 'n' object located in the first part of the instruction itself ("let n"), which is still in the temporal dead zone since its declaration statement has not been reached and **terminated**.

function go(n) {

// n here is defined!

console.log(n); // Object {a: [1,2,3]}

for (let n of n.a) { // ReferenceError: n is not defined

console.log(n);

}}

go({a: [1, 2, 3]});