# ECMAScript 6 入门

## Let和const命令

### Let命令 **基本用法**

{

let a =10;

var b = 1;

}

a // a is not defined

b // 1

var a = [];

for(var i=0;i<10;i++){  
 a[i] = function(){

console.log(i);

}

}

a[6](); //10

var a = [];

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

a[i]=function(){

console.log(i);

};

}

a[6](); //6

for循环语句部分是一个父作用域，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

**不存在变量提升**

// var 的情况

console.log(foo); //输出undefined

var foo = 2;

//let的情况

console.log(bar); //报错RefernceError

let bar =2;

**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在这个区域，不再受外部影响。

var tmp = 123;

if(true){

tmp = ‘abc’; //ReferencesError

let tmp;

}

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”zone,简称TDZ

If(true){

//TDZ开始

tmp = ‘abc’; //RefernceError

console.log(tmp); //RefernceError

let tmp; //TDZ结束

console.log(tmp); //undefined

tmp=123;

console.log(tmp); //123

}

typeof x; //RefernceError

let x;

typeof undefined\_variable //undefined

}

不太容易发现的死区

function bar(x=y,y=2){  
 return [x,y];

}

bar (); //报错 x默认等于y,但是此时y还没有声明，属于“死区”

function bar(x=2,y=x){  
 return[x,y];

}

bar(); //[2,3]

//不报错

var x = x;

//报错

let x=x; //x is not defined

暂时性死区的本质就是只要一进入当前作用域，所有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哪一行代码出现，才可以获取和使用该变量**。**

**不允许重复使用**

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名参数。

function func(arg){

let arg;//报错

}

function func(arg){

{

let arg;//不报错

}

}

### 块级作用域

为什么需要块级作用域

内层变量可能会覆盖外层变量

var tmp = new Date();

fuction f(){

console.log(tmp);

if(false){

var tmp = “hello world”;

}

}

f(); //undefined

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

### ES6的块级作用域

function f1(){  
 let n = 5;

if(true){

let n = 10;

}

console.log(n);//5

}

ES6允许块级作用域的任意嵌套。

**块级作用域与函数声明**

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。

//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;

}

}

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 strct’;

if(true){

function f(){}

}

//报错

‘use strict’;

if(true)

fucntion f(){}

### do表达式（提案，暂时没有）

使得块级作用域有返回值

let x = do{

let t = f();

t \* t+1;

};

变量x 会得到整个块级作用域的返回值。

### 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

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 RefernceError:MAX is not defined

const命令声明的常量也不提升，同样存在暂时性死区，只能在声明的后边使用。

if（true）{

console.log(MAX); //RefernceError

const MAX = 5;

}

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;

对于复核类型的变量，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”

以上代码中，foo存储的是一个地址，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，但对象本事可变，所以依然可以为其添加新属性。

const a = [];

1. push(‘Hello’); //可执行
2. length = 0; //可执行

a=[‘Dave’]; //报错

如果真想将对象本身冻结，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，该行不会报错

foo.prop = 123;

除了将对象本身冻结，对象的属性也应该冻结，下面时将一个对象彻底冻结的函数

var constantize = (obj)=>{

Object.freeze(obj);

Object.keys(obj).forEach((key,vale)=>{

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，import命令和class命令。所以ES6一共有6种声明变量的方法。

### 顶层对象的属性

顶层对象，在浏览器环境中指的是window对象，在Node指的是global对象，ES5之中，顶层对象属性域全局变量是等价的。

window.a = 1;

a //1

a=2;

window.a //2

ES6规定，var命令和function命令声明的全局变量，依旧是顶层对象的属性；另一方面规定，let命令、const命令、cclass命令声明的是全局变量，不属于顶层对象的属性。

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

ES5的顶层对象，在各个实现里面不是不统一的。

# 变量的解构赋值

## 数组的解构赋值

基本用法

ES6允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构。

Let [a,b,c] = [123];

上面代码表示，可以从数组中提取值，按照对应位置，对变量赋值。

本质上，这种写法属于“模式匹配”

let [foo,[[bar],baz]] = [1,[[2],3]];

foo //1

bar //2

baz //3

let[,,third] = [‘foo’,’bar’，‘baz’];

third //’baz’

let [x,,y ]=[1,2,3];

x //1

y //3

let [head,...tail] = [1,2,3,4];

head //1

tail //[2,3,4]

let [x,y,...z] = [‘a’];

x //’a’

y //undefined

z //[]

如果结构不成功，变量的值就等于undefined。

let [foo] = [];

let [bar,foo] = [1];

以上两种情况都属于解构不成功,foo的值都会等于undefined。

另一种是不完全解构，即等号左边的模式，只匹配一部分的等号右边的数组。

let [x,y]= [1,2,3];

x //1

y //2

let [a,[b],d] = [1,[2,3],4];

a //1

b//2

d //4

如果等号的右边不是数组（或者严格的说，不是可遍历的结构）

//报错

let [foo] = 1;

let [foo] = false;

let [foo] = NaN;

let [foo] = undefined;

let [foo] = null

let [foo] = {};

对于Set结构 ，也可以使用数组的解构赋值。

let [x,y,z] = new Set([‘a’,’b’,’c’]);

x // ‘a’

事实上，只要某种数据结构具有Iterator接口，都可以采用数组形式的解构赋值。

事实上，只要某种数据结构具有Iterator接口，都可以采用数组形式的解构赋值。

function\* fibs(){

let a = 0;

let b = 1;

while(true){

yield a;

[a,b] = [b,a+b];

}

}

let [first,second,third,fourth,fifth,sixth] = fibs();

sixth //5

### 默认值

解构赋值允许指定默认值

let [foo = true] = [];

foo // true

let [x,y = ‘b’] = [‘a’]; //x=’a’,y = ‘b’

let [x,y = ‘b’]= [‘a’,undefined]; //x = ‘a’,y = ‘b’

ES6内部使用严格相等运算符有（ = = = ），判断一个位置是否有值，所以，如果数组成员不严格等于undefined，默认值不会生效。

let [x=1]= [undefined];

x//1

let [x=1] = [null];

x //null

如果默认值是一个表达式，那么这个表达式是惰性求值的，即只有在用到的时候，才会求值。

function f(){  
 console.log(‘aaa’);

}

let [x=f()] = [1];

上面代码中，因为x能取到值，所以函数f根本不会执行。等价于下面的代码

let x;

if([1][0] === undeffined){

x=xf();

}else{

x=[1][0];

}

默认值可以引用解构赋值的其他变量，但该变量必须已经声明

let [x = 1,y=x] = []; //x=1;y =1

let [x=1,y = x] =[2]; //x=x2;y=2

let [x=1,y=x] = [1,2]; //x=1,y=2

let [x=y,y=1] = []; //RefernceError

对象的解构赋值

解构不仅可以用于数组，还可以用于对象。

let {foo,bar} = {foo:’aaa’,bar:bbb”};

foo //’aaa’

bar //’bbb’

对象的解构与数组有一个重要的不同，数组的元素书按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。

let {bar,foo} = {foo,’aaa’,bar:’bbb’};

foo //’aaa’

bar //’bbb’

let {baz} = {foo:’aaa’,bar:bbb”};

baz // undefined

var {foo:baz} = {foo:’aaa’,bar:’bbb’};

baz //’aaa’

let obj = {first:’hello’,last:’world’};

let {first:f,last:l} = obj;

f //’hello’

l //’world’

实际上说明，对象的解构赋值时下面形式的简写

let {foo:foo,bar:bar} = {foo:’aaa’,bar:’bbb’};

也就是说，对象的解构赋值得得内部机制，是先找到同名属性，然后再赋给对应变量，真正被赋值的是后者，而不是前者。

let {foo:baz } = {foo:’aaa’,bar:’bbb’};

baz //’aaa’

foo //Error:foo is not defined

上面代码中，foo是匹配的模式，baz才是变量。

采用这种写法时，变量的声明和赋值时一体的，对于let和const来说，变量不能重复声明，所以一旦赋值的变量以前声明过，就会报错

let foo;

let{foo} = {foo:1}; //SyntaxError:Duplicate declaration ‘foo’

let baz

let {bar:baz} = {bar:1}; //SyntaxError: Duplicate declaration ‘baz’

let foo;

({foo} = {foo:1}); //成功

let baz;

({bar:baz} = {bar:1}); //成功

上面代码中，let命令下面一行的圆括号时必须的，否则会报错。因为解析器会将起首的大括号理解成一个代码块。

和数组一样，解构也可以用于嵌套解构的对象。

let obj = {

p:[

‘Hello’,

{y:World}

]

};

let{ p:[x,{y}]} = obj;

x //’Hello’

y //’World’

注意，这时p是模式，不是变量，因此不会被赋值

var node = {

loc:{

loc:{

start:{

line:1,

column:5

}

}

}

};

var { loc: { start: {line} } } = node;

line //1

loc //error:loc is undefined

strat //error:start is undefined

上面代码中，只有line是变量，loc和start都是模式，不会被赋值

下面是嵌套赋值的例子

let obj = {};

let arr = [];

({foo:obj.prop,bar:arr[0]} = {foo:123,bar:true});

obj //{prop:123}

arr //[true]

对象的解构也可以指定默认值

var {x = 3} = {};

x //3

var {x,y = 5} ={x:1}

x //1

y //5

var {x:y = 3} = {};

y //3

var {x:y = 3} = {x:5};

y //5

var {message :msg = ‘Something went wrong’} = {};

msg //’Something went wrong’

默认值生效的条件是对象的属性值严格等于undefined。

var {x = 3} = {x:null}

x // null

如果解构失败，变量的值等于undefined

let {foo} = {bar:’baz’};

foo // undefined

如果解构模式是嵌套的对象，而且子对象所在的父属性不存在，那么将会报错

//报错

let {foo:{bar}} = {baz:’baz’};

\_tmp.foo.bar //报错

如果要将一个已经声明的变量用于解构赋值，必须非常小心。

//错误的写法

let x;

{x} = {x:1};

//SyntaxError:syntax error

上面代码的写法会报错，因为JavaScript引擎会将{x}理解成一个代码块，从而发生语法错误。只有不将大括号写在行首，避免JavaScript将其解释为代码块，才能解决这个问题

//正确的写法

（{x} = {x:1}）;

解构赋值允许等号左边的模式之中，不放置任何变量名。

（{} = [true,false]）;

({}= ‘abc’);

({} = []);

上面的表达式虽然毫无意义，但是语法是合法的，可以执行

对象的解构赋值，可以很方便地将现有对象的方法，赋值到某个变量

let {log， sin， cos} = Math;

上面代码将Math对象那个的对数、正弦、余弦三个方法，赋值到对应的变量上，使用起来就会方便很多。

由于数组本质是特殊的对象，因此可以对数组进行对象属性的解构。

let arr = [1,2,3];

let {0:first,[arr.length-1:last]} = arr;

first //1

last //3

方括号的这种写法，属于属性名表达式

## 字符串的解构赋值

字符串也可以解构赋值。这是因为此时，字符串被转换成了一个类似数组的对象。

const [a,b,c,d,e] = ‘hello’;

a //’h’

b //’e’  
c //’l’

d //’l’

e //’o’

类似数组的对象都有一个length属性，因此还可以对这个属性解构赋值

let {length:len} = ‘hello’;

len //5

## 数值和布尔值的解构赋值

解构赋值时，如果等号右边是数值和布尔值，则会先转为对象。

let {toString:s} = 123;

s === Number.prototype.toString //true

let {toString:s} = true;

s === Boolean.prototype.toString //true

上面代码中，数值和布尔值的包装对象都有toString属性，因此s都能取到值。

解构赋值的规则是，只要等号右边的值不是对象，就先将其转为对象，由于undefined和null无法转为对象，所以对他们进行解构赋值，都会报错。

let {prop:x} = undefined; //TypeError;

let {prop:y } = null; //TypeError

## 函数参数的解构赋值

函数的参数也可以使用解构赋值。

function add([x,y]){

return x+y;

}

add([1,2]); //3

[[1,2],[3,4]].map(([a,b])=>a+b)

//[3,7

函数参数的解构也可以使用默认值。

function move ({x=0,y=0}){

return [x,y];

}

move({x:3,y:8}); //[3,8]

move({x:3}); //[3,0]

move({}); //[0,0]

move(); //[0,0s]

上面代码中，函数move的参数是一个对象，通过对这个对象进行解构，得到变量x和y的值。如果解构失败，x和y等于默认值。

## 圆括号问题

建议只要有可能，就不要在模式中放置圆括号

不能使用圆括号的情况

以下三种解构赋值不得使用圆括号。

1. 变量声明语句中，不能带有圆括号

//全部报错

let [(a)] = [1];

let {x:（c）}={};

let{(x:c)} = {};

let {(x:c)} = {};

let {（x）:c} = {};

let {o：（{p:p}）} = {o:{p:2}};

上面三个语句都会报错平，因为他们都是变量声明语句，模式不能使用圆括号