# ES6

## 1 ES6简介

### 1.1 支持查询

各大浏览器对ES6的支持度查询：

<https://kangax.github.io/compat-table/es6/>

http://babeljs.io/  
使用步骤：  
1-初始化项目目录 npm init  
2-安装babel-cli npm install babel-cli -D  
3-在package添加脚本 "build":"babel src -d lib"  
 其中src就是源文件的目录  
 lib就是编译后的文件目录  
 都可以任意修改  
4-创建.babelrc文件并配置  
{  
 "presets":["env"]  
}  
  
5-安装对应的包  
 npm install babel-preset-env -D  
6-执行编译命令 npm run build

## 2 let与const

### 2.0 六种定义变量方式

ES5：

var 可以重复定义，不能限制用户的修改，也没有块级作用域；

function 用来定义函数

ES6：

let 不能重复定义，一般用来定义变量，具备块级作用域；

const 不能重复定义，一般用来定义常量，悲剧块级作用域；

import 用来定义模块

class 用来定义类

### 2.1 let

let用来定义变量，其主要特点：

1. 必须先声明，再使用
2. let声明不存在变量提升
3. let定义的变量只在其所在区域有效

{

var a = 10;

let b = 20;

console.log(a);

console.log(b);

}

console.log(a);

console.log(b);

我们发现最后一个输出b报错：b is not defined。

**注意：let声明不存在变量提升的情况，必须先声明，再使用。**

let与var一样都是用来声明变量，但是let声明的变量只在代码块中有效。let的这一特性让let在for循环中非常实用：

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

}

console.log(i); //这里输出i就会报错。

只要块级作用域内存在let命令，它所声明的变量就会绑定在这个区域，不受外部影响，我们称之为：暂时性死区。

var tmp = 123;

if (true) {

tmp = 'abc'; // ReferenceError

let tmp;

}

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

### 2.2 for循环的let案例

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。

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

let i = 'abc';

console.log(i); //会连续输出3次 abc

}

我们使用var来做一个测试：

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

a[i] = function () {

console.log(i);

};

}

a[6](); // 10

由于用var声明，i在全局都是有效的，所以全局只有一个变量i。每一次循环，变量i的值都会发生改变，而循环内被赋给数组a的函数内部的console.log(i)，里面的i指向的就是全局的i。也就是说，所有数组a的成员里面的i，指向的都是同一个i，导致运行时输出的是最后一轮的i的值，也就是 10。

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。因为let 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。

### 2.3 块级作用域

ES5只有全局作用域和函数作用域，let的特性让ES6拥有了块级作用域。在ES5中：

var tmp = new Date();

function f() {

console.log(tmp);

if (false) {

var tmp = 'hello world';

}

}

f(); // undefined

由于变量提升，内层tmp覆盖了外层的tmp变量，这样就会出现未定义现象，

还有一个问题是上述案例中关于for循环，用var i = 0这样定义让i成为了全局性质的变量，循环结束，i并没有消失，有了let的块级作用域就可以解决上述问题。

ES6允许多个代码块嵌套，且外层均不能获取内层，但是内层作用域可以使用外层作用域的变量名重新定义。

{{{{

let insane = 'Hello World';

{let insane = 'Hello World'

}}}}};

有了块级作用域，其实我们已经不再需要JS常见的立即执行函数了。

在ES6中，允许块级作用域中声明函数，且声明类似let，块级作用域外不可引用，而在ES5中因为变量提升，在哪里声明都会提升，且可被引用。但是实际支持ES6的浏览器仍然会报错（不是支持问题，而是浏览器为了兼容以前的代码，否定了ES6的特性），所以我们要尽量不在代码块中声明函数，确实需要应该写成函数表达式。

// 函数表达式{

let a = 'secret';

let f = function () {

return a;

};

}

### 2.4 const

const用来声明一个只读的常量，一旦声明，不能改变（即声明时就要直接初始化）。

const与let一样：

只在所在作用域有效，

且不存在变量提升，

存在暂时性死区，不可重复声明

const其实保存的不改变值不是变量的值，而是变量的引用地址不可更改，如果保存的是简单数据类型（布尔值、数值、字符串），等同于常量，而如果保存的是复杂数据类型（对象、数组），其实保存的是该数据的指针。

const 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。同理const定义的函数、对象都可以进行内部修改内容，但却不能再次指向一个新的函数、对象。

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, i) => {

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

constantize( obj[key] );

}

});

};

## 3 顶层对象

### 3.1 不同环境的顶层对象

顶层对象，在浏览器环境指的是window对象，在 Node 指的是global对象。

* 浏览器顶层对象是window，但 Node 和 Web Worker 没有window。
* 浏览器和 Web Worker 里self也指向顶层对象，但 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。

两种勉强可以使用的方法：（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）

// 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');

};

### 3.2 顶层对象属性

ES5 之中，顶层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，顶层对象的属性赋值与全局变量的赋值，是同一件事，这是JS最大的设计败笔之一。这种做法会让很多地方出现全局变量，全局变量是顶层对象的属性，这个属性又是可读写的，不利于模块化编程，且会造成大量运行时错误。

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，依旧是顶层对象的属性；另一方面规定，let命令、const命令、class命令声明的全局变量，不属于顶层对象的属性。也就是说，从 ES6 开始，全局变量将逐步与顶层对象的属性脱钩。

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.awindow.a // 1let b = 1;

window.b // undefined

上面代码中，全局变量a由var命令声明，所以它是顶层对象的属性；全局变量b由let命令声明，所以它不是顶层对象的属性，返回undefined。

## 4 变量的解构赋值

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（Destructuring）。

let [a,b,c] = [12,{a:1,b:2,c:3},53];

注意： 解构赋值的左右两边必须一样，且右边是合法的数据。

解构赋值的定义和赋值必须同步完成。

错误写法：

let [a,b];  
[a,b] = [1,2];

### 4.1 数组解构

常见的解构案例：

let [a, b, c] = [1, 2, 3];

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 // []

let [x, y, z] = new Set(['a', 'b', 'c']);

x // "a"

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

//解构不完全

a // 1

b // 2

d // 4

### 4.2 解构失败

如果解构不成功，变量的值就等于undefined。

### 4.3 解构默认值

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'

### 4.4 字符串解构

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

### 4.5 解构赋值的作用

****（1）交换变量的值****

let x = 1;let y = 2;

[x, y] = [y, x];

上面代码交换变量x和y的值，这样的写法不仅简洁，而且易读，语义非常清晰。

****（2）从函数返回多个值****

函数只能返回一个值，如果要返回多个值，只能将它们放在数组或对象里返回。有了解构赋值，取出这些值就非常方便。

// 返回一个数组function example() {

return [1, 2, 3];}let [a, b, c] = example();

// 返回一个对象function example() {

return {

foo: 1,

bar: 2

};}let { foo, bar } = example();

****（3）函数参数的定义****

解构赋值可以方便地将一组参数与变量名对应起来。

// 参数是一组有次序的值function f([x, y, z]) { ... }f([1, 2, 3]);

// 参数是一组无次序的值function f({x, y, z}) { ... }f({z: 3, y: 2, x: 1});

****（4）提取 JSON 数据****

解构赋值对提取 JSON 对象中的数据，尤其有用。

let jsonData = {

id: 42,

status: "OK",

data: [867, 5309]};

let { id, status, data: number } = jsonData;

console.log(id, status, number);

// 42, "OK", [867, 5309]

上面代码可以快速提取 JSON 数据的值。

****（5）函数参数的默认值****

jQuery.ajax = function (url, {

async = true,

beforeSend = function () {},

cache = true,

complete = function () {},

crossDomain = false,

global = true,

// ... more config}) {

// ... do stuff};

指定参数的默认值，就避免了在函数体内部再写var foo = config.foo || 'default foo';这样的语句。

****（6）遍历 Map 结构****

任何部署了 Iterator 接口的对象，都可以用for...of循环遍历。Map 结构原生支持 Iterator 接口，配合变量的解构赋值，获取键名和键值就非常方便。

const map = new Map();

map.set('first', 'hello');

map.set('second', 'world');

for (let [key, value] of map) {

console.log(key + " is " + value);}

// first is hello// second is world

如果只想获取键名，或者只想获取键值，可以写成下面这样。

// 获取键名for (let [key] of map) {

// ...}

// 获取键值for (let [,value] of map) {

// ...}

****（7）输入模块的指定方法****

加载模块时，往往需要指定输入哪些方法。解构赋值使得输入语句非常清晰。

const { SourceMapConsumer, SourceNode } = require("source-map");

## 5 字符串扩展

### 5.1 字符串遍历

ES6 为字符串添加了遍历器接口，使得字符串可以被for...of循环遍历。

for (let codePoint of 'foo') {

console.log(codePoint)

}

### 5.2 includes() startWith() endsWith()

* ****includes()****：返回布尔值，表示是否找到了参数字符串。
* ****startsWith()****：返回布尔值，表示参数字符串是否在原字符串的头部。
* ****endsWith()****：返回布尔值，表示参数字符串是否在原字符串的尾部。

let s = 'Hello world!';

s.startsWith('Hello')

// true

s.endsWith('!') // true

s.includes('o') // true

这三个方法都支持第二个参数，表示开始搜索的位置。

let s = 'Hello world!';

s.startsWith('world', 6) // true

s.endsWith('Hello', 5) // true

s.includes('Hello', 6) // false

上面代码表示，使用第二个参数n时，endsWith的行为与其他两个方法有所不同。它针对前n个字符，而其他两个方法针对从第n个位置直到字符串结束。

### 5.3 模板字符串

传统的 JavaScript 语言，输出模板通常是这样写的。

$('#result').append(

'There are <b>' + basket.count + '</b> ' +

'items in your basket, ' +

'<em>' + basket.onSale +

'</em> are on sale!');

上面这种写法相当繁琐不方便，ES6 引入了模板字符串解决这个问题。

$('#result').append(`

There are <b>${basket.count}</b> items

in your basket, <em>${basket.onSale}</em>

are on sale!

`);

模板字符串（template string）是增强版的字符串，用反引号（`）标识。它可以当作普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量。

// 普通字符串`In JavaScript '\n' is a line-feed.`

// 多行字符串`In JavaScript this is

not legal.`

console.log(`string text line 1

string text line 2`);

// 字符串中嵌入变量let name = "Bob", time = "today";

`Hello ${name}, how are you ${time}?`

上面代码中的模板字符串，都是用反引号表示。如果在模板字符串中需要使用反引号，则前面要用反斜杠转义。

let greeting = `\`Yo\` World!`;

如果使用模板字符串表示多行字符串，所有的空格和缩进都会被保留在输出之中。

$('#list').html(`<ul>

<li>first</li>

<li>second</li></ul>

`);

上面代码中，所有模板字符串的空格和换行，都是被保留的，比如<ul>标签前面会有一个换行。如果你不想要这个换行，可以使用trim方法消除它。

$('#list').html(`<ul>

<li>first</li>

<li>second</li></ul>

`.trim());

模板字符串中嵌入变量，需要将变量名写在${}之中。

function authorize(user, action) {

if (!user.hasPrivilege(action)) {

throw new Error(

// 传统写法为 // 'User ' // + user.name // + ' is not authorized to do ' // + action // + '.' `User ${user.name} is not authorized to do ${action}.`);

}}

大括号内部可以放入任意的 JavaScript 表达式，可以进行运算，以及引用对象属性。

let x = 1;let y = 2;

`${x} + ${y} = ${x + y}`

// "1 + 2 = 3"

`${x} + ${y \* 2} = ${x + y \* 2}`

// "1 + 4 = 5"let obj = {x: 1, y: 2};

`${obj.x + obj.y}`

// "3"

模板字符串之中还能调用函数。

function fn() {

return "Hello World";}

`foo ${fn()} bar`

// foo Hello World bar

如果大括号中的值不是字符串，将按照一般的规则转为字符串。比如，大括号中是一个对象，将默认调用对象的toString方法。

如果模板字符串中的变量没有声明，将报错。

// 变量place没有声明let msg = `Hello, ${place}`;

// 报错

由于模板字符串的大括号内部，就是执行 JavaScript 代码，因此如果大括号内部是一个字符串，将会原样输出。

`Hello ${'World'}`

// "Hello World"

模板字符串甚至还能嵌套。

const tmpl = addrs => `

<table>

${addrs.map(addr => `

<tr><td>${addr.first}</td></tr>

<tr><td>${addr.last}</td></tr>

`).join('')}

</table>

`;

上面代码中，模板字符串的变量之中，又嵌入了另一个模板字符串，使用方法如下。

const data = [

{ first: '<Jane>', last: 'Bond' },

{ first: 'Lars', last: '<Croft>' },];

console.log(tmpl(data));

// <table>//// <tr><td><Jane></td></tr>// <tr><td>Bond</td></tr>//// <tr><td>Lars</td></tr>// <tr><td><Croft></td></tr>//// </table>

如果需要引用模板字符串本身，在需要时执行，可以像下面这样写。

// 写法一let str = 'return ' + '`Hello ${name}!`';let func = new Function('name', str);func('Jack') // "Hello Jack!"

// 写法二let str = '(name) => `Hello ${name}!`';let func = eval.call(null, str);func('Jack') // "Hello Jack!"