一 Module 的语法

1. 概述

历史上，JavaScript 一直没有模块（module）体系，无法将一个大程序拆分成互相依赖的小文件，再用简单的方法拼装起来。其他语言都有这项功能，比如 Ruby 的require、Python 的import，甚至就连 CSS 都有@import，但是 JavaScript 任何这方面的支持都没有，这对开发大型的、复杂的项目形成了巨大障碍。

在 ES6 之前，社区制定了一些模块加载方案，最主要的有 CommonJS 和 AMD 两种。前者用于服务器，后者用于浏览器。ES6 在语言标准的层面上，实现了模块功能，而且实现得相当简单，完全可以取代 CommonJS 和 AMD 规范，成为浏览器和服务器通用的模块解决方案。

ES6 模块的设计思想，是尽量的静态化，使得编译时就能确定模块的依赖关系，以及输入和输出的变量。CommonJS 和 AMD 模块，都只能在运行时确定这些东西。比如，CommonJS 模块就是对象，输入时必须查找对象属性。

|  |
| --- |
| // CommonJS模块  let { stat, exists, readFile } = require('fs');  // 等同于  let \_fs = require('fs');  let stat = \_fs.stat;  let exists = \_fs.exists;  let readfile = \_fs.readfile; |

上面代码的实质是整体加载fs模块（即加载fs的所有方法），生成一个对象（\_fs），然后再从这个对象上面读取3个方法。这种加载称为“运行时加载”，因为只有运行时才能得到这个对象，导致完全没办法在编译时做“静态优化”。

ES6 模块不是对象，而是通过export命令显式指定输出的代码，再通过import命令输入。

|  |
| --- |
| // ES6模块  import { stat, exists, readFile } from 'fs'; |

上面代码的实质是从fs模块加载3个方法，其他方法不加载。这种加载称为“编译时加载”或者静态加载，即 ES6 可以在编译时就完成模块加载，效率要比 CommonJS 模块的加载方式高。当然，这也导致了没法引用 ES6 模块本身，因为它不是对象。

除了静态加载带来的各种好处，ES6 模块还有以下好处。

◎ 不再需要UMD模块格式了，将来服务器和浏览器都会支持 ES6 模块格式。目前，通过各种工具库，其实已经做到了这一点。

2. 严格模式

ES6 的模块自动采用严格模式，不管你有没有在模块头部加上"use strict";。

严格模式主要有以下限制。

◎ 变量必须声明后在使用

◎ 不能使用with语句

◎ 不能对只读属性赋值，否则报错

◎ 不能使用前缀0表示八进制数，否则报错

◎ 不能删除不可删除的属性，否则报错

◎ 不能删除变量delete prop，会报错，只能删除属性delete global[prop]

◎ eval不会在它的外层作用域引入变量

◎ eval和arguments不能被重新赋值

◎ arguments不会自动反映函数参数的变化

◎ 不能使用arguments.callee

◎ 不能使用arguments.caller

◎ 禁止this指向全局对象

◎ 不能使用fn.caller和fn.arguments获取函数调用的堆栈

◎ 增加了保留字（比如protected、static和interface）

上面这些限制，模块都必须遵守。由于严格模式是 ES5 引入的，不属于 ES6，所以请参阅相关 ES5 书籍，本书不再详细介绍了。

其中，尤其需要注意this的限制。ES6 模块之中，顶层的this指向undefined，即不应该在顶层代码使用this。

3. export命令

模块功能主要由两个命令构成：export和import。export命令用于规定模块的对外接口，import命令用于输入其他模块提供的功能。

一个模块就是一个独立的文件。该文件内部的所有变量，外部无法获取。如果你希望外部能够读取模块内部的某个变量，就必须使用export关键字输出该变量。下面是一个 JS 文件，里面使用export命令输出变量。

|  |
| --- |
| // profile.js  export var firstName = 'Michael';  export var lastName = 'Jackson';  export var year = 1958; |

上面代码是profile.js文件，保存了用户信息。ES6 将其视为一个模块，里面用export命令对外部输出了三个变量。

export的写法，除了像上面这样，还有另外一种。

|  |
| --- |
| // profile.js  var firstName = 'Michael';  var lastName = 'Jackson';  var year = 1958;  export {firstName, lastName, year}; |

上面代码在export命令后面，使用大括号指定所要输出的一组变量。它与前一种写法（直接放置在var语句前）是等价的，但是应该优先考虑使用这种写法。因为这样就可以在脚本尾部，一眼看清楚输出了哪些变量。

export命令除了输出变量，还可以输出函数或类（class）。

|  |
| --- |
| export function multiply(x, y) {  return x \* y;  };  //上面代码对外输出一个函数multiply。 |

通常情况下，export输出的变量就是本来的名字，但是可以使用as关键字重命名。

|  |
| --- |
| function v1() { ... }  function v2() { ... }  export {  v1 as streamV1,  v2 as streamV2,  v2 as streamLatestVersion  }; |

需要特别注意的是，export命令规定的是对外的接口，必须与模块内部的变量建立一一对应关系。

|  |
| --- |
| // 报错  export 1;  // 报错  var m = 1;  export m; |

上面两种写法都会报错，因为没有提供对外的接口。第一种写法直接输出1，第二种写法通过变量m，还是直接输出1。1只是一个值，不是接口。正确的写法是下面这样。

|  |
| --- |
| // 写法一  export var m = 1;  // 写法二  var m = 1;  export {m};  // 写法三  var n = 1;  export {n as m}; |

上面三种写法都是正确的，规定了对外的接口m。其他脚本可以通过这个接口，取到值1。它们的实质是，在接口名与模块内部变量之间，建立了一一对应的关系。

|  |
| --- |
| // 报错  function f() {}  export f;  // 正确  export function f() {};  // 正确  function f() {}  export {f}; |

另外，export语句输出的接口，与其对应的值是动态绑定关系，即通过该接口，可以取到模块内部实时的值。

|  |
| --- |
| export var foo = 'bar';  setTimeout(() => foo = 'baz', 500); |

面代码输出变量foo，值为bar，500毫秒之后变成baz。

这一点与 CommonJS 规范完全不同。CommonJS 模块输出的是值的缓存，不存在动态更新，详见下文《ES6模块加载的实质》一节。

最后，export命令可以出现在模块的任何位置，只要处于模块顶层就可以。如果处于块级作用域内，就会报错，下一节的import命令也是如此。这是因为处于条件代码块之中，就没法做静态优化了，违背了ES6模块的设计初衷。

|  |
| --- |
| function foo() {  export default 'bar' // SyntaxError  }  foo()  //上面代码中，export语句放在函数之中，结果报错。 |

4. import

使用export命令定义了模块的对外接口以后，其他 JS 文件就可以通过import命令加载这个模块。

|  |
| --- |
| // main.js  import {firstName, lastName, year} from './profile';  function setName(element) {  element.textContent = firstName + ' ' + lastName;  } |

上面代码的import命令，用于加载profile.js文件，并从中输入变量。import命令接受一对大括号，里面指定要从其他模块导入的变量名。大括号里面的变量名，必须与被导入模块（profile.js）对外接口的名称相同。

如果想为输入的变量重新取一个名字，import命令要使用as关键字，将输入的变量重命名。

|  |
| --- |
| import { lastName as surname } from './profile'; |

import后面的from指定模块文件的位置，可以是相对路径，也可以是绝对路径，.js路径可以省略。如果只是模块名，不带有路径，那么必须有配置文件，告诉 JavaScript 引擎该模块的位置。

|  |
| --- |
| import {myMethod} from 'util'; |

上面代码中，util是模块文件名，由于不带有路径，必须通过配置，告诉引擎怎么取到这个模块。

注意，import命令具有提升效果，会提升到整个模块的头部，首先执行。

|  |
| --- |
| foo();  import { foo } from 'my\_module'; |

上面的代码不会报错，因为import的执行早于foo的调用。这种行为的本质是，import命令是编译阶段执行的，在代码运行之前。

由于import是静态执行，所以不能使用表达式和变量，这些只有在运行时才能得到结果的语法结构。

|  |
| --- |
| // 报错  import { 'f' + 'oo' } from 'my\_module';  // 报错  let module = 'my\_module';  import { foo } from module;  // 报错  if (x === 1) {  import { foo } from 'module1';  } else {  import { foo } from 'module2';  } |

上面三种写法都会报错，因为它们用到了表达式、变量和if结构。在静态分析阶段，这些语法都是没法得到值的。

最后，import语句会执行所加载的模块，因此可以有下面的写法。

|  |
| --- |
| import 'lodash'; |

上面代码仅仅执行lodash模块，但是不输入任何值。

如果多次重复执行同一句import语句，那么只会执行一次，而不会执行多次。

|  |
| --- |
| import 'lodash';  import 'lodash'; |

上面代码加载了两次lodash，但是只会执行一次。

|  |
| --- |
| import { foo } from 'my\_module';  import { bar } from 'my\_module';  // 等同于  import { foo, bar } from 'my\_module'; |

上面代码中，虽然foo和bar在两个语句中加载，但是它们对应的是同一个my\_module实例。也就是说，import语句是 Singleton 模式。

5. 模块的整体加载

除了指定加载某个输出值，还可以使用整体加载，即用星号（\*）指定一个对象，所有输出值都加载在这个对象上面。

下面是一个circle.js文件，它输出两个方法area和circumference。

|  |
| --- |
| // circle.js  export function area(radius) {  return Math.PI \* radius \* radius;  }  export function circumference(radius) {  return 2 \* Math.PI \* radius;  } |

现在，加载这个模块。

|  |
| --- |
| // main.js  import { area, circumference } from './circle';  console.log('圆面积：' + area(4));  console.log('圆周长：' + circumference(14)); |

上面写法是逐一指定要加载的方法，整体加载的写法如下

|  |
| --- |
| import \* as circle from './circle';  console.log('圆面积：' + circle.area(4));  console.log('圆周长：' + circle.circumference(14)); |

注意，模块整体加载所在的那个对象（上例是circle），应该是可以静态分析的，所以不允许运行时改变。下面的写法都是不允许的。

|  |
| --- |
| import \* as circle from './circle';  // 下面两行都是不允许的  circle.foo = 'hello';  circle.area = function () {}; |

6. export default 命令

从前面的例子可以看出，使用import命令的时候，用户需要知道所要加载的变量名或函数名，否则无法加载。但是，用户肯定希望快速上手，未必愿意阅读文档，去了解模块有哪些属性和方法。

为了给用户提供方便，让他们不用阅读文档就能加载模块，就要用到export default命令，为模块指定默认输出。

|  |
| --- |
| // export-default.js  export default function () {  console.log('foo');  } |

上面代码是一个模块文件export-default.js，它的默认输出是一个函数。

其他模块加载该模块时，import命令可以为该匿名函数指定任意名字。

|  |
| --- |
| // import-default.js  import customName from './export-default';  customName(); // 'foo' |

上面代码的import命令，可以用任意名称指向export-default.js输出的方法，这时就不需要知道原模块输出的函数名。需要注意的是，这时import命令后面，不使用大括号。

export default命令用在非匿名函数前，也是可以的。

|  |
| --- |
| // export-default.js  export default function foo() {  console.log('foo');  }  // 或者写成  function foo() {  console.log('foo');  }  export default foo; |

上面代码中，foo函数的函数名foo，在模块外部是无效的。加载的时候，视同匿名函数加载。

下面比较一下默认输出和正常输出。

|  |
| --- |
| // 第一组  export default function crc32() { // 输出  // ...  }  import crc32 from 'crc32'; // 输入  // 第二组  export function crc32() { // 输出  // ...  };  import {crc32} from 'crc32'; // 输入 |
| 上面代码的两组写法，第一组是使用export default时，对应的import语句不需要使用大括号；第二组是不使用export default时，对应的import语句需要使用大括号。 |

export default命令用于指定模块的默认输出。显然，一个模块只能有一个默认输出，因此export default命令只能使用一次。所以，import命令后面才不用加大括号，因为只可能对应一个方法。

本质上，export default就是输出一个叫做default的变量或方法，然后系统允许你为它取任意名字。所以，下面的写法是有效的。

|  |
| --- |
| // modules.js  function add(x, y) {  return x \* y;  }  export {add as default};  // 等同于  // export default add;  // app.js  import { default as xxx } from 'modules';  // 等同于  // import xxx from 'modules'; |

正是因为export default命令其实只是输出一个叫做default的变量，所以它后面不能跟变量声明语句。

|  |
| --- |
| // 正确  export var a = 1;  // 正确  var a = 1;  export default a;  // 错误  export default var a = 1; |

上面代码中，export default a的含义是将变量a的值赋给变量default。所以，最后一种写法会报错。

同样地，因为export default本质是将该命令后面的值，赋给default变量以后再默认，所以直接将一个值写在export default之后。

|  |
| --- |
| // 正确  export default 42;  // 报错  export 42; |
| 上面代码中，后一句报错是因为没有指定对外的接口，而前一句指定外对接口为default。 |

有了export default命令，输入模块时就非常直观了，以输入 lodash 模块为例。

|  |
| --- |
| import \_ from 'lodash'; |

如果想在一条import语句中，同时输入默认方法和其他变量，可以写成下面这样。

|  |
| --- |
| import \_, { each } from 'lodash'; |
| 对应上面代码的export语句如下。  export default function (obj) {  // ···  }  export function each(obj, iterator, context) {  // ···  }  export { each as forEach }; |

上面代码的最后一行的意思是，暴露出forEach接口，默认指向each接口，即forEach和each指向同一个方法。

export default也可以用来输出类。

|  |
| --- |
| // MyClass.js  export default class { ... }  // main.js  import MyClass from 'MyClass';  let o = new MyClass(); |

7. export 与 import 的复合写法

如果在一个模块之中，先输入后输出同一个模块，import语句可以与export语句写在一起。

|  |
| --- |
| export { foo, bar } from 'my\_module';  // 等同于  import { foo, bar } from 'my\_module';  export { foo, bar }; |
| 上面代码中，export和import语句可以结合在一起，写成一行。 |

模块的接口改名和整体输出，也可以采用这种写法。

|  |
| --- |
| // 接口改名  export { foo as myFoo } from 'my\_module';  // 整体输出  export \* from 'my\_module'; |

默认接口的写法如下：

|  |
| --- |
| export { default } from 'foo'; |

具名接口改为默认接口的写法如下：

|  |
| --- |
| export { es6 as default } from './someModule';  // 等同于  import { es6 } from './someModule';  export default es6; |

同样地，默认接口也可以改名为具名接口。

|  |
| --- |
| export { default as es6 } from './someModule'; |

下面三种import语句，没有对应的复合写法。

|  |
| --- |
| import \* as someIdentifier from "someModule";  import someIdentifier from "someModule";  import someIdentifier, { namedIdentifier } from "someModule"; |

为了做到形式的对称，现在有提案，提出补上这三种复合写法。

|  |
| --- |
| export \* as someIdentifier from "someModule";  export someIdentifier from "someModule";  export someIdentifier, { namedIdentifier } from "someModule"; |

8. 模块的继承

模块之间也可以继承。

假设有一个circleplus模块，继承了circle模块。

|  |
| --- |
| // circleplus.js  export \* from 'circle';  export var e = 2.71828182846;  export default function(x) {  return Math.exp(x);  } |
| 上面代码中的export \*，表示再输出circle模块的所有属性和方法。注意，export \*命令会忽略circle模块的default方法。然后，上面代码又输出了自定义的e变量和默认方法。 |

这时，也可以将circle的属性或方法，改名后再输出。

|  |
| --- |
| // circleplus.js  export { area as circleArea } from 'circle'; |
| 上面代码表示，只输出circle模块的area方法，且将其改名为circleArea。 |

加载上面模块的写法如下：

|  |
| --- |
| // main.js  import \* as math from 'circleplus';  import exp from 'circleplus';  console.log(exp(math.e)); |
| 上面代码中的import exp表示，将circleplus模块的默认方法加载为exp方法。 |

9. 跨模块常量

本书介绍const命令的时候说过，const声明的常量只在当前代码块有效。如果想设置跨模块的常量（即跨多个文件），或者说一个值要被多个模块共享，可以采用下面的写法。

|  |
| --- |
| // constants.js 模块  export const A = 1;  export const B = 3;  export const C = 4;  // test1.js 模块  import \* as constants from './constants';  console.log(constants.A); // 1  console.log(constants.B); // 3  // test2.js 模块  import {A, B} from './constants';  console.log(A); // 1  console.log(B); // 3 |

如果要使用的常量非常多，可以建一个专门的constants目录，将各种常量写在不同的文件里面，保存在该目录下。

|  |
| --- |
| // constants/db.js  export const db = {  url: 'http://my.couchdbserver.local:5984',  admin\_username: 'admin',  admin\_password: 'admin password'  };  // constants/user.js  export const users = ['root', 'admin', 'staff', 'ceo', 'chief', 'moderator']; |

然后，将这些文件输出的常量，合并在index.js里面。

|  |
| --- |
| // constants/index.js  export {db} from './db';  export {users} from './users'; |

使用的时候，直接加载index.js就可以了。

|  |
| --- |
| // script.js  import {db, users} from './constants'; |

10. import()

10.1 简介

前面介绍过，import命令会被 JavaScript 引擎静态分析，先于模块内的其他模块执行（叫做”连接“更合适）。所以，下面的代码会报错。

|  |
| --- |
| // 报错  if (x === 2) {  import MyModual from './myModual';  } |

上面代码中，引擎处理import语句是在编译时，这时不会去分析或执行if语句，所以import语句放在if代码块之中毫无意义，因此会报句法错误，而不是执行时错误。也就是说，import和export命令只能在模块的顶层，不能在代码块之中（比如，在if代码块之中，或在函数之中）。

这样的设计，固然有利于编译器提高效率，但也导致无法在运行时加载模块。从语法上，条件加载就不可能实现。如果import命令要取代 Node 的require方法，这就形成了一个障碍。因为require是运行时加载模块，import命令无法取代require的动态加载功能。

|  |
| --- |
| const path = './' + fileName;  const myModual = require(path); |
| 上面的语句就是动态加载，require到底加载哪一个模块，只有运行时才知道。import语句做不到这一点。 |

因此，有一个提案，建议引入import()函数，完成动态加载。

|  |
| --- |
| import(specifier) |

上面代码中，import函数的参数specifier，指定所要加载的模块的位置。import命令能够接受什么参数，import()函数就能接受什么参数，两者区别主要是后者为动态加载。

import()返回一个 Promise 对象。下面是一个例子。

|  |
| --- |
| const main = document.querySelector('main');  import(`./section-modules/${someVariable}.js`)  .then(module => {  module.loadPageInto(main);  })  .catch(err => {  main.textContent = err.message;  }); |

import()函数可以用在任何地方，不仅仅是模块，非模块的脚本也可以使用。它是运行时执行，也就是说，什么时候运行到这一句，也会加载指定的模块。另外，import()函数与所加载的模块没有静态连接关系，这点也是与import语句不相同。

import()类似于 Node 的require方法，区别主要是前者是异步加载，后者是同步加载。

10.2 适用场合

下面是import()的一些适用场合。

◎ 按需加载

import()可以在需要的时候，再加载某个模块。

|  |
| --- |
| button.addEventListener('click', event => {  import('./dialogBox.js')  .then(dialogBox => {  dialogBox.open();  })  .catch(error => {  /\* Error handling \*/  })  }); |
| 上面代码中，import()方法放在click事件的监听函数之中，只有用户点击了按钮，才会加载这个模块。 |

◎ 条件加载

import()可以放在if代码块，根据不同的情况，加载不同的模块。

|  |
| --- |
| if (condition) {  import('moduleA').then(...);  } else {  import('moduleB').then(...);  } |
| 上面代码中，如果满足条件，就加载模块 A，否则加载模块 B。 |

◎ 动态的模块路径

import()允许模块路径动态生成。

|  |
| --- |
| import(f())  .then(...); |
| 上面代码中，根据函数f的返回结果，加载不同的模块。 |

10.3 注意点

import()加载模块成功以后，这个模块会作为一个对象，当作then方法的参数。因此，可以使用对象解构赋值的语法，获取输出接口。

|  |
| --- |
| import('./myModule.js')  .then(({export1, export2}) => {  // ...·  }); |
| 上面代码中，export1和export2都是myModule.js的输出接口，可以解构获得。 |

如果模块有default输出接口，可以用参数直接获得。

|  |
| --- |
| import('./myModule.js')  .then(myModule => {  console.log(myModule.default);  }); |
| 上面的代码也可以使用具名输入的形式。 |
| import('./myModule.js')  .then(({default: theDefault}) => {  console.log(theDefault);  }); |

如果想同时加载多个模块，可以采用下面的写法。

|  |
| --- |
| Promise.all([  import('./module1.js'),  import('./module2.js'),  import('./module3.js'),  ])  .then(([module1, module2, module3]) => {  ···  }); |

import()也可以用在 async 函数之中。

|  |
| --- |
| async function main() {  const myModule = await import('./myModule.js');  const {export1, export2} = await import('./myModule.js');  const [module1, module2, module3] =  await Promise.all([  import('./module1.js'),  import('./module2.js'),  import('./module3.js'),  ]);  }  main(); |

二 let命令和const命令

1. let命令

1.1 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 |
| 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 < 10; i++) {}  console.log(i);  //ReferenceError: i is not defined |
| 上面代码中，计数器i只在for循环体内有效，在循环体外引用就会报错。  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function在运行时，会通过闭包读到这同一个变量i，导致最后输出的是最后一轮的i的值，也就是10。  而如果使用let，声明的变量仅在块级作用域内有效，最后输出的是6。 |

另外，for循环还有一个特别之处，就是循环语句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++) { //一个作用域  let i = 'abc'; // 循环体，另一个作用域  console.log(i);  }  // abc  // abc  // abc  //有必要进行测试  上面代码输出了3次abc，这表明函数内部的变量i和外部的变量i是分离的。 |

1.2 不存在变量提升

var命令会发生”变量提升“现象，即变量可以在声明之前使用，值为undefined。这种现象多多少少是有些奇怪的，按照一般的逻辑，变量应该在声明语句之后才可以使用。

为了纠正这种现象，let命令改变了语法行为，它所声明的变量一定要在声明后使用，否则报错。

|  |
| --- |
| // var 的情况  console.log(foo); // 输出undefined  var foo = 2;  // let 的情况  console.log(bar); // 报错ReferenceError  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。 |

1.3 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。  //理解： x 在 y 之前，此时 y 还未声明，属于死区 |
| function bar(x = 2, y = x) {  return [x, y];  }  bar(); // [2, 2] |

另外，下面的代码也会报错，与var的行为不同。

|  |
| --- |
| // 不报错  var x = x;  // 报错  let x = x;  // ReferenceError: x is not defined |
| 上面代码报错，也是因为暂时性死区。使用let声明变量时，只要变量在还没有声明完成前使用，就会报错。上面这行就属于这个情况，在变量x的声明语句还没有执行完成前，就去取x的值，导致报错”x 未定义“。  //理解：声明时引用自身，所以出错 |

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。

1.4 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

2.1 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 |
| 上面代码的原意是，if代码块的外部使用外层的tmp变量，内部使用内层的tmp变量。但是，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。 |

2.2 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函数表达式（IIFE）不再必要了。

|  |
| --- |
| // IIFE 写法  (function () {  var tmp = ...;  ...  }());  // 块级作用域写法  {  let tmp = ...;  ...  } |

2.3 块级作用域和函数声明

函数能不能在块级作用域之中声明？这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，为了兼容以前的旧代码，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。

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 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();  }());  // => I am inside |

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 环境  function f() { console.log('I am outside!'); }  (function () {  function f() { console.log('I am inside!'); }  if (false) {  }  f();  }()); |

ES6 就完全不一样了，理论上会得到“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。但是，如果你真的在 ES6 浏览器中运行一下上面的代码，是会报错的，这是为什么呢？

原来，如果改变了块级作用域内声明的函数的处理规则，显然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。

根据这三条规则，在浏览器的 ES6 环境中，块级作用域内声明的函数，行为类似于var声明的变量。

|  |
| --- |
| // 浏览器的 ES6 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 |
| 上面的代码在符合 ES6 的浏览器中，都会报错，因为实际运行的是下面的代码。 |
| // 浏览器的 ES6 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() {} |

2.4 do表达式

本质上，块级作用域是一个语句，将多个操作封装在一起，没有返回值。

|  |
| --- |
| {  let t = f();  t = t \* t + 1;  } |

上面代码中，块级作用域将两个语句封装在一起。但是，在块级作用域以外，没有办法得到t的值，因为块级作用域不返回值，除非t是全局变量。

现在有一个提案，使得块级作用域可以变为表达式，也就是说可以返回值，办法就是在块级作用域之前加上do，使它变为do表达式。

|  |
| --- |
| let x = do {  let t = f();  t \* t + 1;  };  上面代码中，变量x会得到整个块级作用域的返回值。 |

3. const 命令

3.1 基本用法

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; |

3.2 const 本质

const实际上保证的，并不是变量的值不得改动，而是变量指向的那个内存地址不得改动。对于简单类型的数据（数值、字符串、布尔值），值就保存在变量指向的那个内存地址，因此等同于常量。但对于复合类型的数据（主要是对象和数组），变量指向的内存地址，保存的只是一个指针，const只能保证这个指针是固定的，至于它指向的数据结构是不是可变的，就完全不能控制了。因此，将一个对象声明为常量必须非常小心。

|  |
| --- |
| const foo = {};  // 为 foo 添加一个属性，可以成功  foo.prop = 123;  foo.prop // 123  // 将 foo 指向另一个对象，就会报错  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。 |
| 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, i) => {  if ( typeof obj[key] === 'object' ) {  constantize( obj[key] );  }  });  }; |

4. ES 6 的六种变量声明方法

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。

5. 顶层对象的属性

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

顶层对象的属性与全局变量挂钩，被认为是JavaScript语言最大的设计败笔之一。这样的设计带来了几个很大的问题，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（因为全局变量可能是顶层对象的属性创造的，而属性的创造是动态的）；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）；最后，顶层对象的属性是到处可以读写的，这非常不利于模块化编程。另一方面，window对象有实体含义，指的是浏览器的窗口对象，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.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。 |

6. global 对象

ES5的顶层对象，本身也是一个问题，因为它在各种实现里面是不统一的。

- 浏览器里面，顶层对象是window，但 Node 和 Web Worker 没有window。

- 浏览器和 Web Worker 里面，self也指向顶层对象，但是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。

综上所述，很难找到一种方法，可以在所有情况下，都取到顶层对象。下面是两种勉强可以使用的方法。

|  |
| --- |
| // 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');  }; |

现在有一个提案，在语言标准的层面，引入global作为顶层对象。也就是说，在所有环境下，global都是存在的，都可以从它拿到顶层对象。

垫片库system.global模拟了这个提案，可以在所有环境拿到global。

|  |
| --- |
| // CommonJS的写法  require('system.global/shim')();  // ES6模块的写法  import shim from 'system.global/shim'; shim(); |
| 上面代码可以保证各种环境里面，global对象都是存在的。 |
| // CommonJS的写法  var global = require('system.global')();  // ES6模块的写法  import getGlobal from 'system.global';  const global = getGlobal(); |
| 上面代码将顶层对象放入变量global。 |

三 变量的解构赋值

1. 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，这种被称为”解构”.

以前变量赋值，只能直接指定值，如下：

|  |
| --- |
| let a = 1;  let b = 2;  let c = 3; |

ES6 允许写成如下:

|  |
| --- |
| let [a, b, c] = [1, 2, 3];  // a -> 1  // b -> 2  // c -> 3 |