ECMAScript 6 入门

作者：阮一峰

目录

Module

ES6的Class只是面向对象编程的语法糖，升级了ES5的构造函数的原型链继承的写法，并没有解决模块化问题。Module功能就是为了解决这个问题而提出的。

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在语言规格的层面上，实现了模块功能，而且实现得相当简单，完全可以取代现有的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, exists = _fs.exists, 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，输入时也采用静态命令的形式。

```
// 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”，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。

由于ES6模块是编译时加载，使得静态分析成为可能。有了它，就能进一步拓宽JavaScript的语法，比如引入宏（macro）和类型检验（type system）这些只能靠静态分析实现的功能。

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。
- 将来浏览器的新API就能用模块格式提供，不再必要做成全局变量或者`navigator`对象的属性。
- 不再需要对象作为命名空间（比如`Math`对象），未来这些功能可以通过模块提供。

浏览器使用ES6模块的语法如下。

```
<script type="module" src="foo.js"></script>
```

上面代码在网页中插入一个模块`foo.js`，由于`type`属性设为`module`，所以浏览器知道这是一个ES6模块。

Node的默认模块格式是CommonJS，目前还没决定怎么支持ES6模块。所以，只能通过Babel这样的转码器，在Node里面使用ES6模块。

---

1. 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。

- 变量必须声明后再使用
- 函数的参数不能有同名属性，否则报错
- 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。

---

2. 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
};
```

上面代码使用`as`关键字，重命名了函数`v1`和`v2`的对外接口。重命名后，`v2`可以用不同的名字输出两次。

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。

同样的，`function`和`class`的输出，也必须遵守这样的写法。

```
// 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);
```

上面代码输出变量`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。

---

3. import命令

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（文件）。

```
// main.js

import {firstName, lastName, year} from './profile';
```

```
function setName(element) {
  element.textContent = firstName + ' ' + lastName;
}
```

上面代码的`import`命令，就用于加载`profile.js`文件，并从中输入变量。`import`命令接受一个对象（用大括号表示），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';
```

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。

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 { es6 as default } from './someModule';

// 等同于
import { es6 } from './someModule';
```

```
export default es6;
```

上面代码中，`export`和`import`语句可以结合在一起，写成一行。但是从可读性考虑，不建议采用这种写法，而应该采用标准写法。

另外，ES7有一个提案，简化先输入后输出的写法，拿掉输出时的大括号。

```
// 提案的写法
export v from 'mod';

// 现行的写法
export {v} from 'mod';
```

`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。

---

4. 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));
```

---

5. 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。

```
// 输出
export default function crc32() {
  // ...
}
// 输入
import crc32 from 'crc32';

// 输出
export function crc32() {
  // ...
};
// 输入
import {crc32} from 'crc32';
```

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 deault`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。

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

```
import $ from 'jquery';
```

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 customName, { otherMethod } from './export-default';
```

如果要输出默认的值，只需将值跟在`export default`之后即可。

```
export default 42;
```

`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();
```

---

6. 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。

---

7. ES6模块加载的实质

ES6模块加载的机制，与CommonJS模块完全不同。CommonJS模块输出的是一个值的拷贝，而ES6模块输出的是值的引用。

CommonJS模块输出的是被输出值的拷贝，也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。请看下面这个模块文件`lib.js`的例子。

```
// lib.js
var counter = 3;
function incCounter() {
  counter++;
}
module.exports = {
  counter: counter,
  incCounter: incCounter,
};
```

上面代码输出内部变量`counter`和改写这个变量的内部方法`incCounter`。然后，在`main.js`里面加载这个模块。

```
// main.js
var mod = require('./lib');
```

```
console.log(mod.counter);  // 3
mod.incCounter();
console.log(mod.counter); // 3
```

上面代码说明，`lib.js`模块加载以后，它的内部变化就影响不到输出的`mod.counter`了。这是因为`mod.counter`是一个原始类型的值，会被缓存。除非写成一个函数，才能得到内部变动后的值。

```
// lib.js
var counter = 3;
function incCounter() {
  counter++;
}
module.exports = {
  get counter() {
    return counter
  },
  incCounter: incCounter,
};
```

上面代码中，输出的`counter`属性实际上是一个取值器函数。现在再执行`main.js`，就可以正确读取内部变量`counter`的变动了。

```
$ node main.js
3
4
```

ES6模块的运行机制与CommonJS不一样，它遇到模块加载命令`import`时，不会去执行模块，而是只生成一个动态的只读引用。等到真的需要用到时，再到模块里面去取

值，换句话说，ES6的输入有点像Unix系统的"符号连接"，原始值变了，`import`输入的值也会跟着变。因此，ES6模块是动态引用，并且不会缓存值，模块里面的变量绑定其所在的模块。

还是举上面的例子。

```
// lib.js
export let counter = 3;
export function incCounter() {
  counter++;
}

// main.js
import { counter, incCounter } from './lib';
console.log(counter); // 3
incCounter();
console.log(counter); // 4
```

上面代码说明，ES6模块输入的变量`counter`是活的，完全反应其所在模块`lib.js`内部的变化。

再举一个出现在`export`一节中的例子。

```
// m1.js
export var foo = 'bar';
setTimeout(() => foo = 'baz', 500);

// m2.js
import {foo} from './m1.js';
console.log(foo);
setTimeout(() => console.log(foo), 500);
```

上面代码中，`m1.js`的变量`foo`，在刚加载时等于`bar`，过了500毫秒，又变为等于`baz`。

让我们看看，`m2.js`能否正确读取这个变化。

```
$ babel-node m2.js

bar
baz
```

上面代码表明，ES6模块不会缓存运行结果，而是动态地去被加载的模块取值，并且变量总是绑定其所在的模块。

由于ES6输入的模块变量，只是一个“符号连接”，所以这个变量是只读的，对它进行重新赋值会报错。

```
// lib.js
export let obj = {};

// main.js
import { obj } from './lib';

obj.prop = 123; // OK
obj = {}; // TypeError
```

上面代码中，`main.js`从`lib.js`输入变量`obj`，可以对`obj`添加属性，但是重新赋值就会报错。因为变量`obj`指向的地址是只读的，不能重新赋值，这就好比`main.js`创造了一个名为`obj`的const变量。

最后，`export`通过接口，输出的是同一个值。不同的脚本加载这个接口，得到的都是同样的实例。

```
// mod.js
function C() {
  this.sum = 0;
  this.add = function () {
    this.sum += 1;
  };
  this.show = function () {
    console.log(this.sum);
  };
}

export let c = new C();
```

上面的脚本`mod.js`，输出的是一个`C`的实例。不同的脚本加载这个模块，得到的都是同一个实例。

```
// x.js
import {c} from './mod';
c.add();

// y.js
import {c} from './mod';
c.show();

// main.js
import './x';
import './y';
```

现在执行`main.js`，输出的是1。

```
$ babel-node main.js
1
```

这就证明了`x.js`和`y.js`加载的都是`C`的同一个实例。

---

8. 循环加载

"循环加载"（circular dependency）指的是，`a`脚本的执行依赖`b`脚本，而`b`脚本的执行又依赖`a`脚本。

```
// a.js
var b = require('b');

// b.js
var a = require('a');
```

通常，"循环加载"表示存在强耦合，如果处理不好，还可能导致递归加载，使得程序无法执行，因此应该避免出现。

但是实际上，这是很难避免的，尤其是依赖关系复杂的大项目，很容易出现`a`依赖`b`，`b`依赖`c`，`c`又依赖`a`这样的情况。这意味着，模块加载机制必须考虑"循环加载"的情况。

对于JavaScript语言来说，目前最常见的两种模块格式CommonJS和ES6，处理"循环加载"的方法是不一样的，返回的结果也不一样。

CommonJS模块的加载原理

介绍ES6如何处理"循环加载"之前，先介绍目前最流行的CommonJS模块格式的加载原理。

CommonJS的一个模块，就是一个脚本文件。`require`命令第一次加载该脚本，就会执行整个脚本，然后在内存生成一个对象。

```
{
  id: '...',
  exports: { ... },
  loaded: true,
  ...
}
```

上面代码就是Node内部加载模块后生成的一个对象。该对象的`id`属性是模块名，`exports`属性是模块输出的各个接口，`loaded`属性是一个布尔值，表示该模块的脚本是否执行完毕。其他还有很多属性，这里都省略了。

以后需要用到这个模块的时候，就会到`exports`属性上面取值。即使再次执行`require`命令，也不会再次执行该模块，而是到缓存之中取值。也就是说，CommonJS模块无论加载多少次，都只会在第一次加载时运行一次，以后再加载，就返回第一次运行的结果，除非手动清除系统缓存。

CommonJS模块的循环加载

CommonJS模块的重要特性是加载时执行，即脚本代码在`require`的时候，就会全部执行。一旦出现某个模块被"循环加载"，就只输出已经执行的部分，还未执行的部分不会输出。

让我们来看，Node官方文档里面的例子。脚本文件`a.js`代码如下。

```
exports.done = false;
var b = require('./b.js');
console.log('在 a.js 之中，b.done = %j', b.done);
exports.done = true;
console.log('a.js 执行完毕');
```

上面代码之中，`a.js`脚本先输出一个`done`变量，然后加载另一个脚本文件`b.js`。注意，此时`a.js`代码就停在这里，等待`b.js`执行完毕，再往下执行。

再看`b.js`的代码。

```
exports.done = false;
var a = require('./a.js');
console.log('在 b.js 之中，a.done = %j', a.done);
exports.done = true;
console.log('b.js 执行完毕');
```

上面代码之中，`b.js`执行到第二行，就会去加载`a.js`，这时，就发生了“循环加载”。系统会去`a.js`模块对应对象的`exports`属性取值，可是因为`a.js`还没有执行完，从`exports`属性只能取回已经执行的部分，而不是最后的值。

`a.js`已经执行的部分，只有一行。

```
exports.done = false;
```

因此，对于`b.js`来说，它从`a.js`只输入一个变量`done`，值为`false`。

然后，`b.js`接着往下执行，等到全部执行完毕，再把执行权交还给`a.js`。于是，`a.js`接着往下执行，直到执行完毕。我们写一个脚本`main.js`，验证这个过程。

```
var a = require('./a.js');
var b = require('./b.js');
console.log('在 main.js 之中, a.done=%j, b.done=%j', a.done, b.done);
```

执行`main.js`，运行结果如下。

```
$ node main.js

在 b.js 之中，a.done = false
b.js 执行完毕
在 a.js 之中，b.done = true
a.js 执行完毕
在 main.js 之中, a.done=true, b.done=true
```

上面的代码证明了两件事。一是，在`b.js`之中，`a.js`没有执行完毕，只执行了第一行。二是，`main.js`执行到第二行时，不会再次执行`b.js`，而是输出缓存的`b.js`的执行结果，即它的第四行。

```
exports.done = true;
```

总之，CommonJS输入的是被输出值的拷贝，不是引用。

另外，由于CommonJS模块遇到循环加载时，返回的是当前已经执行的部分的值，而不是代码全部执行后的值，两者可能会有差异。所以，输入变量的时候，必须非常小心。

```
var a = require('a'); // 安全的写法
var foo = require('a').foo; // 危险的写法

exports.good = function (arg) {
  return a.foo('good', arg); // 使用的是 a.foo 的最新值
};

exports.bad = function (arg) {
  return foo('bad', arg); // 使用的是一个部分加载时的值
};
```

上面代码中，如果发生循环加载，`require('a').foo`的值很可能后面会被改写，改用`require('a')`会更保险一点。

ES6模块的循环加载

ES6处理"循环加载"与CommonJS有本质的不同。ES6模块是动态引用，如果使用`import`从一个模块加载变量（即`import foo from 'foo'`），那些变量不会被缓存，而是成为一个指向被加载模块的引用，需要开发者自己保证，真正取值的时候能够取到值。

请看下面这个例子。

```
// a.js如下
import {bar} from './b.js';
console.log('a.js');
console.log(bar);
export let foo = 'foo';

// b.js
import {foo} from './a.js';
console.log('b.js');
console.log(foo);
export let bar = 'bar';
```

上面代码中，`a.js`加载`b.js`，`b.js`又加载`a.js`，构成循环加载。执行`a.js`，结果如下。

```
$ babel-node a.js
b.js
undefined
a.js
bar
```

上面代码中，由于`a.js`的第一行是加载`b.js`，所以先执行的是`b.js`。而`b.js`的第一行又是加载`a.js`，这时由于`a.js`已经开始执行了，所以不会重复执行，而是继续往下执行`b.js`，所以第一行输出的是`b.js`。

接着，`b.js`要打印变量`foo`，这时`a.js`还没执行完，取不到`foo`的值，导致打印出来是`undefined`。`b.js`执行完，开始执行`a.js`，这时就一切正常了。

再看一个稍微复杂的例子（摘自 Dr. Axel Rauschmayer 的《Exploring ES6》）。

```
// a.js
import {bar} from './b.js';
export function foo() {
  console.log('foo');
  bar();
  console.log('执行完毕');
}
foo();

// b.js
import {foo} from './a.js';
export function bar() {
  console.log('bar');
  if (Math.random() > 0.5) {
    foo();
  }
}
```

按照CommonJS规范，上面的代码是没法执行的。`a`先加载`b`，然后`b`又加载`a`，这时`a`还没有任何执行结果，所以输出结果为`null`，即对于`b.js`来说，变量`foo`的值等于`null`，后面的`foo()`就会报错。

但是，ES6可以执行上面的代码。

```
$ babel-node a.js
foo
bar
执行完毕

// 执行结果也有可能是
```

```
foo
bar
foo
bar
执行完毕
执行完毕
```

上面代码中，`a.js`之所以能够执行，原因就在于ES6加载的变量，都是动态引用其所在的模块。只要引用存在，代码就能执行。

下面，我们详细分析这段代码的运行过程。

```
// a.js

// 这一行建立一个引用，
// 从`b.js`引用`bar`
import {bar} from './b.js';

export function foo() {
  // 执行时第一行输出 foo
  console.log('foo');
  // 到 b.js 执行 bar
  bar();
  console.log('执行完毕');
}
foo();

// b.js

// 建立`a.js`的`foo`引用
import {foo} from './a.js';

export function bar() {
```

```
  // 执行时，第二行输出 bar
  console.log('bar');
  // 递归执行 foo，一旦随机数
  // 小于等于0.5，就停止执行
  if (Math.random() > 0.5) {
    foo();
  }
}
```

我们再来看ES6模块加载器SystemJS给出的一个例子。

```
// even.js
import { odd } from './odd'
export var counter = 0;
export function even(n) {
  counter++;
  return n == 0 || odd(n - 1);
}

// odd.js
import { even } from './even';
export function odd(n) {
  return n != 0 && even(n - 1);
}
```

上面代码中，`even.js`里面的函数`even`有一个参数`n`，只要不等于0，就会减去1，传入加载的`odd()`。`odd.js`也会做类似操作。

运行上面这段代码，结果如下。

```
$ babel-node
> import * as m from './even.js';
```

```
> m.even(10);
true
> m.counter
6
> m.even(20)
true
> m.counter
17
```

上面代码中，参数`n`从10变为0的过程中，`even()`一共会执行6次，所以变量`counter`等于6。第二次调用`even()`时，参数`n`从20变为0，`even()`一共会执行11次，加上前面的6次，所以变量`counter`等于17。

这个例子要是改写成CommonJS，就根本无法执行，会报错。

```
// even.js
var odd = require('./odd');
var counter = 0;
exports.counter = counter;
exports.even = function(n) {
  counter++;
  return n == 0 || odd(n - 1);
}

// odd.js
var even = require('./even').even;
module.exports = function(n) {
  return n != 0 && even(n - 1);
}
```

上面代码中，`even.js`加载`odd.js`，而`odd.js`又去加载`even.js`，形成“循环加载”。

这时，执行引擎就会输出`even.js`已经执行的部分（不存在任何结果），所以在`odd.js`之中，变量`even`等于`null`，等到后面调用`even(n-1)`就会报错。

```
$ node
> var m = require('./even');
> m.even(10)
TypeError: even is not a function
```

---

9. 跨模块常量

上面说过，`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），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
```

10. ES6模块的转码

浏览器目前还不支持ES6模块，为了现在就能使用，可以将转为ES5的写法。除了Babel可以用来转码之外，还有以下两个方法，也可以用来转码。

ES6 module transpiler

ES6 module transpiler是square公司开源的一个转码器，可以将ES6模块转为CommonJS模块或AMD模块的写法，从而在浏览器中使用。

首先，安装这个转玛器。

```
$ npm install -g es6-module-transpiler
```

然后，使用`compile-modules convert`命令，将ES6模块文件转码。

```
$ compile-modules convert file1.js file2.js
```

`-o`参数可以指定转码后的文件名。

```
$ compile-modules convert -o out.js file1.js
```

SystemJS

另一种解决方法是使用SystemJS。它是一个垫片库（polyfill），可以在浏览器内加载ES6模块、AMD模块和CommonJS模块，将其转为ES5格式。它在后台调用的是Google的Traceur转码器。

使用时，先在网页内载入system.js文件。

```
<script src="system.js"></script>
```

然后，使用`System.import`方法加载模块文件。

```
<script>
  System.import('./app.js');
</script>
```

上面代码中的`./app`，指的是当前目录下的app.js文件。它可以是ES6模块文件，`System.import`会自动将其转码。

需要注意的是，`System.import`使用异步加载，返回一个Promise对象，可以针对这个对象编程。下面是一个模块文件。

```
// app/es6-file.js:

export class q {
  constructor() {
    this.es6 = 'hello';
  }
}
```

然后，在网页内加载这个模块文件。

```
<script>

System.import('app/es6-file').then(function(m) {
  console.log(new m.q().es6); // hello
});

</script>
```

上面代码中，`System.import`方法返回的是一个Promise对象，所以可以用then方法指定回调函数。

留言