ECMAScript 6 入门

作者：阮一峰

目录

其他

- 源码
- 修订历史
- 反馈意见

Module 的加载实现

上一章介绍了模块的语法，本章介绍如何在浏览器和 Node.js 之中加载 ES6 模块，以及实际开发中经常遇到的一些问题（比如循环加载）。

---

1. 浏览器加载

传统方法

HTML 网页中，浏览器通过 `<script>` 标签加载 JavaScript 脚本。

```
<!-- 页面内嵌的脚本 -->
<script type="application/javascript">
  // module code
</script>

<!-- 外部脚本 -->
<script type="application/javascript" src="path/to/myModule.js">
</script>
```

上面代码中，由于浏览器脚本的默认语言是 JavaScript，因此 `type="application/javascript"` 可以省略。

默认情况下，浏览器是同步加载 JavaScript 脚本，即渲染引擎遇到 `<script>` 标签就会停下来，等到执行完脚本，再继续向下渲染。如果是外部脚本，还必须加入脚本下载的时间。

如果脚本体积很大，下载和执行的时间就会很长，因此造成浏览器堵塞，用户会感觉到浏览器“卡死”了，没有任何响应。这显然是很不好的体验，所以浏览器允许脚本异步加载，下面就是两种异步加载的语法。

```
<script src="path/to/myModule.js" defer></script>
<script src="path/to/myModule.js" async></script>
```

上面代码中， `<script>` 标签打开 `defer` 或 `async` 属性，脚本就会异步加载。渲染引擎遇到这一行命令，就会开始下载外部脚本，但不会等它下载和执行，而是直接执行后面的命令。

`defer` 与 `async` 的区别是： `defer` 要等到整个页面在内存中正常渲染结束（DOM 结构完全生成，以及其他脚本执行完成），才会执行； `async` 一旦下载完，渲染引擎就会中断渲染，执行这个脚本以后，再继续渲染。一句话， `defer` 是“渲染完再执行”， `async` 是“下载完就执行”。另外，如果有多个 `defer` 脚本，会按照它们在页面出现的顺序加载，而多个 `async` 脚本是不能保证加载顺序的。

加载规则

浏览器加载 ES6 模块，也使用 `<script>` 标签，但是要加入 `type="module"` 属性。

```
<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。

浏览器对于带有 `type="module"` 的 `<script>`，都是异步加载，不会造成堵塞浏览器，即等到整个页面渲染完，再执行模块脚本，等同于打开了 `<script>` 标签的 `defer` 属性。

```
<script type="module" src="./foo.js"></script>
<!-- 等同于 -->
<script type="module" src="./foo.js" defer></script>
```

如果网页有多个 `<script type="module">`，它们会按照在页面出现的顺序依次执行。

`<script>` 标签的 `async` 属性也可以打开，这时只要加载完成，渲染引擎就会中断渲染立即执行。执行完成后，再恢复渲染。

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

一旦使用了 `async` 属性，`<script type="module">` 就不会按照在页面出现的顺序执行，而是只要该模块加载完成，就执行该模块。

ES6 模块也允许内嵌在网页中，语法行为与加载外部脚本完全一致。

```
<script type="module">
  import utils from "./utils.js";

  // other code
</script>
```

举例来说，jQuery 就支持模块加载。

```
<script type="module">
  import $ from "./jquery/src/jquery.js";
  $('#message').text('Hi from jQuery!');
</script>
```

对于外部的模块脚本（上例是 `foo.js`），有几点需要注意。

- 代码是在模块作用域之中运行，而不是在全局作用域运行。模块内部的顶层变量，外部不可见。
- 模块脚本自动采用严格模式，不管有没有声明 `use strict`。
- 模块之中，可以使用 `import` 命令加载其他模块（`.js` 后缀不可省略，需要提供绝对 URL 或相对 URL），也可以使用 `export` 命令输出对外接口。
- 模块之中，顶层的 `this` 关键字返回 `undefined`，而不是指向 `window`。也就是说，在模块顶层使用 `this` 关键字，是无意义的。
- 同一个模块如果加载多次，将只执行一次。

下面是一个示例模块。

```
import utils from 'https://example.com/js/utils.js';

const x = 1;

console.log(x === window.x); //false
console.log(this === undefined); // true
```

利用顶层的 `this` 等于 `undefined` 这个语法点，可以侦测当前代码是否在 ES6 模块之中。

```
const isNotModuleScript = this !== undefined;
```

2. ES6 模块与 CommonJS 模块的差异

讨论 Node.js 加载 ES6 模块之前，必须了解 ES6 模块与 CommonJS 模块完全不同。

它们有三个重大差异。

- CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用。
- CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。
- CommonJS 模块的 `require()` 是同步加载模块，ES6 模块的 `import` 命令是异步加载，有一个独立的模块依赖的解析阶段。

第二个差异是因为 CommonJS 加载的是一个对象（即 `module.exports` 属性），该对象只有在脚本运行完才会生成。而 ES6 模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成。

下面重点解释第一个差异。

CommonJS 模块输出的是值的拷贝，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。JS 引擎对脚本静态分析的时候，遇到模块加载命令 `import`，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。换句话说，ES6 的 `import` 有点像 Unix 系统的“符号连接”，原始值变了，`import` 加载的值也会跟着变。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。

---

3. Node.js 的模块加载方法

概述

JavaScript 现在有两种模块。一种是 ES6 模块，简称 ESM；另一种是 CommonJS 模块，简称 CJS。

CommonJS 模块是 Node.js 专用的，与 ES6 模块不兼容。语法上面，两者最明显的差异是，CommonJS 模块使用 `require()` 和 `module.exports`，ES6 模块使用 `import` 和 `export`。

它们采用不同的加载方案。从 Node.js v13.2 版本开始，Node.js 已经默认打开了 ES6 模块支持。

Node.js 要求 ES6 模块采用 `.mjs` 后缀文件名。也就是说，只要脚本文件里面使用 `import` 或者 `export` 命令，那么就必须采用 `.mjs` 后缀名。Node.js 遇到 `.mjs` 文件，就认为它是 ES6 模块，默认启用严格模式，不必在每个模块文件顶部指定 `"use strict"`。

如果不希望将后缀名改成 `.mjs`，可以在项目的 `package.json` 文件中，指定 `type` 字段为 `module`。

```
{
   "type": "module"
```

```
}
```

一旦设置了以后，该项目的 JS 脚本，就被解释成 ES6 模块。

```
# 解释成 ES6 模块
$ node my-app.js
```

如果这时还要使用 CommonJS 模块，那么需要将 CommonJS 脚本的后缀名都改成 `.cjs` 。如果没有 `type` 字段，或者 `type` 字段为 `commonjs` ，则 `.js` 脚本会被解释成 CommonJS 模块。

总结为一句话： `.mjs` 文件总是以 ES6 模块加载， `.cjs` 文件总是以 CommonJS 模块加载， `.js` 文件的加载取决于 `package.json` 里面 `type` 字段的设置。

注意，ES6 模块与 CommonJS 模块尽量不要混用。 `require` 命令不能加载 `.mjs` 文件，会报错，只有 `import` 命令才可以加载 `.mjs` 文件。反过来， `.mjs` 文件里面也不能使用 `require` 命令，必须使用 `import` 。

---

package.json 的 main 字段

`package.json` 文件有两个字段可以指定模块的入口文件： `main` 和 `exports` 。比较简单的模块，可以只使用 `main` 字段，指定模块加载的入口文件。

```
// ./node_modules/es-module-package/package.json
{
  "type": "module",
  "main": "./src/index.js"
}
```

上面代码指定项目的入口脚本为 `./src/index.js` ，它的格式为 ES6 模块。如果没有 `type` 字段， `index.js` 就会被解释为 CommonJS 模块。

然后， `import` 命令就可以加载这个模块。

```
// ./my-app.mjs

import { something } from 'es-module-package';
// 实际加载的是 ./node_modules/es-module-package/src/index.js
```

上面代码中，运行该脚本以后，Node.js 就会到 `./node_modules` 目录下面，寻找 `es-module-package` 模块，然后根据该模块 `package.json` 的 `main` 字段去执行入口文件。

这时，如果用 CommonJS 模块的 `require()` 命令去加载 `es-module-package` 模块会报错，因为 CommonJS 模块不能处理 `export` 命令。

---

package.json 的 exports 字段

`exports` 字段的优先级高于 `main` 字段。它有多种用法。

（1）子目录别名

`package.json` 文件的 `exports` 字段可以指定脚本或子目录的别名。

```
// ./node_modules/es-module-package/package.json
{
  "exports": {
    "./submodule": "./src/submodule.js"
  }
}
```

上面的代码指定 `src/submodule.js` 别名为 `submodule`，然后就可以从别名加载这个文件。

```
import submodule from 'es-module-package/submodule';
// 加载 ./node_modules/es-module-package/src/submodule.js
```

下面是子目录别名的例子。

```
// ./node_modules/es-module-package/package.json
{
  "exports": {
    "./features/": "./src/features/"
  }
}

import feature from 'es-module-package/features/x.js';
// 加载 ./node_modules/es-module-package/src/features/x.js
```

如果没有指定别名，就不能用“模块+脚本名”这种形式加载脚本。

```
// 报错
import submodule from 'es-module-package/private-module.js';

// 不报错
import submodule from './node_modules/es-module-package/private-module.js';
```

（2）main 的别名

`exports` 字段的别名如果是 `.`，就代表模块的主入口，优先级高于 `main` 字段，并且可以直接简写成 `exports` 字段的值。

```
{
  "exports": {
    ".": "./main.js"
  }
}

// 等同于
{
  "exports": "./main.js"
}
```

由于 `exports` 字段只有支持 ES6 的 Node.js 才认识，所以可以用来兼容旧版本的 Node.js。

```
{
  "main": "./main-legacy.cjs",
  "exports": {
    ".": "./main-modern.cjs"
  }
}
```

上面代码中，老版本的 Node.js （不支持 ES6 模块）的入口文件是 `main-legacy.cjs`，新版本的 Node.js 的入口文件是 `main-modern.cjs`。

（3）条件加载

利用 `.` 这个别名，可以为 ES6 模块和 CommonJS 指定不同的入口。目前，这个功能需要在 Node.js 运行的时候，打开 `--experimental-conditional-exports` 标志。

```
{
  "type": "module",
  "exports": {
    ".": {
      "require": "./main.cjs",
      "default": "./main.js"
    }
  }
}
```

上面代码中，别名 `.` 的 `require` 条件指定 `require()` 命令的入口文件（即 CommonJS 的入口）， `default` 条件指定其他情况的入口（即 ES6 的入口）。

上面的写法可以简写如下。

```
{
  "exports": {
    "require": "./main.cjs",
    "default": "./main.js"
  }
}
```

注意，如果同时还有其他别名，就不能采用简写，否则会报错。

```
{
  // 报错
  "exports": {
    "./feature": "./lib/feature.js",
    "require": "./main.cjs",
    "default": "./main.js"
  }
}
```

CommonJS 模块加载 ES6 模块

CommonJS 的 `require()` 命令不能加载 ES6 模块，会报错，只能使用 `import()` 这个方法加载。

```
(async () => {
  await import('./my-app.mjs');
})();
```

上面代码可以在 CommonJS 模块中运行。

`require()` 不支持 ES6 模块的一个原因是，它是同步加载，而 ES6 模块内部可以使用顶层 `await` 命令，导致无法被同步加载。

ES6 模块加载 CommonJS 模块

ES6 模块的 `import` 命令可以加载 CommonJS 模块，但是只能整体加载，不能只加载单一的输出项。

```
// 正确
import packageMain from 'commonjs-package';

// 报错
import { method } from 'commonjs-package';
```

这是因为 ES6 模块需要支持静态代码分析，而 CommonJS 模块的输出接口是 `module.exports`，是一个对象，无法被静态分析，所以只能整体加载。

加载单一的输出项，可以写成下面这样。

```
import packageMain from 'commonjs-package';
const { method } = packageMain;
```

还有一种变通的加载方法，就是使用 Node.js 内置的 `module.createRequire()` 方法。

```
// cjs.cjs
module.exports = 'cjs';

// esm.mjs
import { createRequire } from 'module';

const require = createRequire(import.meta.url);

const cjs = require('./cjs.cjs');
cjs === 'cjs'; // true
```

上面代码中，ES6 模块通过 `module.createRequire()` 方法可以加载 CommonJS 模块。但是，这种写法等于将 ES6 和 CommonJS 混在一起了，所以不建议使用。

同时支持两种格式的模块

一个模块同时要支持 CommonJS 和 ES6 两种格式，也很容易。

如果原始模块是 ES6 格式，那么需要给出一个整体输出接口，比如 `export default obj`，使得 CommonJS 可以用 `import()` 进行加载。

如果原始模块是 CommonJS 格式，那么可以加一个包装层。

```
import cjsModule from '../index.js';
export const foo = cjsModule.foo;
```

上面代码先整体输入 CommonJS 模块，然后再根据需要输出具名接口。

你可以把这个文件的后缀名改为 `.mjs`，或者将它放在一个子目录，再在这个子目录里面放一个单独的 `package.json` 文件，指明 `{ type: "module" }`。

另一种做法是在 `package.json` 文件的 `exports` 字段，指明两种格式模块各自的加载入口。

```
"exports": {
  "require": "./index.js",
  "import": "./esm/wrapper.js"
}
```

上面代码指定 `require()` 和 `import`，加载该模块会自动切换到不一样的入口文件。

Node.js 的内置模块

Node.js 的内置模块可以整体加载，也可以加载指定的输出项。

```
// 整体加载
import EventEmitter from 'events';
const e = new EventEmitter();

// 加载指定的输出项
import { readFile } from 'fs';
readFile('./foo.txt', (err, source) => {
  if (err) {
    console.error(err);
  } else {
    console.log(source);
  }
});
```

加载路径

ES6 模块的加载路径必须给出脚本的完整路径，不能省略脚本的后缀名。 `import` 命令和 `package.json` 文件的 `main` 字段如果省略脚本的后缀名，会报错。

```
// ES6 模块中将报错
import { something } from './index';
```

为了与浏览器的 `import` 加载规则相同，Node.js 的 `.mjs` 文件支持 URL 路径。

```
import './foo.mjs?query=1'; // 加载 ./foo 传入参数 ?query=1
```

上面代码中，脚本路径带有参数 `?query=1`，Node 会按 URL 规则解读。同一个脚本只要参数不同，就会被加载多次，并且保存成不同的缓存。由于这个原因，只要文件名中含有 `:`、`%`、`#`、`?` 等特殊字符，最好对这些字符进行转义。

目前，Node.js 的 `import` 命令只支持加载本地模块（`file:` 协议）和 `data:` 协议，不支持加载远程模块。另外，脚本路径只支持相对路径，不支持绝对路径（即以 `/` 或 `//` 开头的路径）。

内部变量

ES6 模块应该是通用的，同一个模块不用修改，就可以用在浏览器环境和服务器环境。为了达到这个目标，Node.js 规定 ES6 模块之中不能使用 CommonJS 模块的特有的一些内部变量。

首先，就是 `this` 关键字。ES6 模块之中，顶层的 `this` 指向 `undefined`；CommonJS 模块的顶层 `this` 指向当前模块，这是两者的一个重大差异。

其次，以下这些顶层变量在 ES6 模块之中都是不存在的。

- arguments
- require
- module
- exports
- __filename
- __dirname

---

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

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

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

```
$ node --experimental-modules a.mjs
b.mjs
ReferenceError: foo is not defined
```

上面代码中，执行 `a.mjs` 以后会报错， `foo` 变量未定义，这是为什么？

让我们一行行来看，ES6 循环加载是怎么处理的。首先，执行 `a.mjs` 以后，引擎发现它加载了 `b.mjs` ，因此会优先执行 `b.mjs` ，然后再执行 `a.mjs` 。接着，执行 `b.mjs` 的时候，已知它从 `a.mjs` 输入了 `foo` 接口，这时不会去执行 `a.mjs` ，而是认为这个接口已经存在了，继续往下执行。执行到第三行 `console.log(foo)` 的时候，才发现这个接口根本没定义，因此报错。

解决这个问题的方法，就是让 `b.mjs` 运行的时候， `foo` 已经有定义了。这可以通过将 `foo` 写成函数来解决。

```
// a.mjs
import {bar} from './b';
console.log('a.mjs');
console.log(bar());
function foo() { return 'foo' }
export {foo};
```

```
// b.mjs
import {foo} from './a';
console.log('b.mjs');
console.log(foo());
function bar() { return 'bar' }
export {bar};
```

这时再执行 a.mjs 就可以得到预期结果。

```
$ node --experimental-modules a.mjs
b.mjs
foo
a.mjs
bar
```

这是因为函数具有提升作用，在执行 import {bar} from './b' 时，函数 foo 就已经有定义了，所以 b.mjs 加载的时候不会报错。这也意味着，如果把函数 foo 改写成函数表达式，也会报错。

```
// a.mjs
import {bar} from './b';
console.log('a.mjs');
console.log(bar());
const foo = () => 'foo';
export {foo};
```

上面代码的第四行，改成了函数表达式，就不具有提升作用，执行就会报错。

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， 上一章 un 下一章 。

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 `undefined` ，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
```

留言