ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证

目录

其他

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 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

上一章 下一章

传统方法

在 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：前者要等到整个页面正常渲染结束，才会执行；后者一旦下载完，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>
```

`<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>
```

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>
```

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

delete x; // 句法错误，严格模式禁止删除变量
```

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

它们有两个重大差异。

- 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。

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 加载

概述

Node 对 ES6 模块的处理比较麻烦，因为它有自己的 CommonJS 模块格式，与 ES6 模块格式是不兼容的。目前的解决方案是，将两者分开，ES6 模块和 CommonJS 采用各自的加载方案。

Node 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。 `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` 。

目前，这项功能还在试验阶段。安装 Node v8.5.0 或以上版本，要用 `--experimental-modules` 参数才能打开该功能。

```
$ node --experimental-modules my-app.mjs
```

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

```
import './foo?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 的 `import` 命令只支持加载本地模块（`file:` 协议），不支持加载远程模块。

如果模块名不含路径，那么 `import` 命令会去 `node_modules` 目录寻找这个模块。

```
import 'baz';
import 'abc/123';
```

如果模块名包含路径，那么 `import` 命令会按照路径去寻找这个名字的脚本文件。

```
import 'file:///etc/config/app.json';
import './foo';
import './foo?search';
import '../bar';
import '/baz';
```

如果脚本文件省略了后缀名，比如 `import './foo'`，Node 会依次尝试四个后缀名：`./foo.mjs`、`./foo.js`、`./foo.json`、`./foo.node`。如果这些脚本文件都不存在，Node 就会去加载 `./foo/package.json` 的 `main` 字段指定的脚本。如果 `./foo/package.json` 不存在或者没有 `main` 字段，那么就会依次加载 `./foo/index.mjs`、`./foo/index.js`、`./foo/index.json`、`./foo/index.node`。如果以上四个文件还是都不存在，就会抛出错误。

最后，Node 的 `import` 命令是异步加载，这一点与浏览器的处理方法相同。

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 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`

如果你一定要使用这些变量，有一个变通方法，就是写一个 CommonJS 模块输出这些变量，然后再用 ES6 模块加载这个 CommonJS 模块。但是这样一来，该 ES6 模块就不能直接用于浏览器环境了，所以不推荐这样做。

```
// expose.js
module.exports = {__dirname};

// use.mjs
import expose from './expose.js';
const {__dirname} = expose;
```

上面代码中，`expose.js` 是一个 CommonJS 模块，输出变量 `__dirname`，该变量在 ES6 模块之中不存在。ES6 模块加载 `expose.js`，就可以得到 `__dirname`。

ES6 模块加载 CommonJS 模块

CommonJS 模块的输出都定义在 `module.exports` 这个属性上面。Node 的 `import` 命令加载 CommonJS 模块，Node 会自动将 `module.exports` 属性，当作模块的默认输出，即等同于 `export default xxx` 。

下面是一个 CommonJS 模块。

```
// a.js
module.exports = {
  foo: 'hello',
  bar: 'world'
};

// 等同于
export default {
  foo: 'hello',
  bar: 'world'
};
```

`import` 命令加载上面的模块， `module.exports` 会被视为默认输出，即 `import` 命令实际上输入的是这样一个对象 `{ default: module.exports }` 。

所以，一共有三种写法，可以拿到 CommonJS 模块的 `module.exports` 。

```
// 写法一
import baz from './a';
// baz = {foo: 'hello', bar: 'world'};

// 写法二
import {default as baz} from './a';
// baz = {foo: 'hello', bar: 'world'};

// 写法三
import * as baz from './a';
// baz = {
//   get default() {return module.exports;},
//   get foo() {return this.default.foo}.bind(baz),
//   get bar() {return this.default.bar}.bind(baz)
// }
```

上面代码的第三种写法，可以通过 `baz.default` 拿到 `module.exports` 。 `foo` 属性和 `bar` 属性就是可以通过这种方法拿到了 `module.exports` 。

下面是一些例子。

```
// b.js
module.exports = null;

// es.js
import foo from './b';
// foo = null;

import * as bar from './b';
// bar = { default:null };
```

上面代码中， `es.js` 采用第二种写法时，要通过 `bar.default` 这样的写法，才能拿到 `module.exports` 。

```
// c.js
module.exports = function two() {
  return 2;
};

// es.js
import foo from './c';
foo(); // 2

import * as bar from './c';
bar.default(); // 2
bar(); // throws, bar is not a function
```

上面代码中， `bar` 本身是一个对象，不能当作函数调用，只能通过 `bar.default` 调用。

CommonJS 模块的输出缓存机制，在 ES6 加载方式下依然有效。

```
// foo.js
module.exports = 123;
setTimeout(_ => module.exports = null);
```

上面代码中，对于加载 `foo.js` 的脚本， `module.exports` 将一直是 `123` ，而不会变成 `null` 。

由于 ES6 模块是编译时确定输出接口，CommonJS 模块是运行时确定输出接口，所以采用 `import` 命令加载 CommonJS 模块时，不允许采用下面的写法。

```
// 不正确
import { readfile } from 'fs';
```

上面的写法不正确，因为 `fs` 是 CommonJS 格式，只有在运行时才能确定 `readfile` 接口，而 `import` 命令要求编译时就确定这个接口。解决方法就是改为整体输入。

```
// 正确的写法一
import * as express from 'express';
const app = express.default();

// 正确的写法二
import express from 'express';
const app = express();
```

CommonJS 模块加载 ES6 模块

CommonJS 模块加载 ES6 模块，不能使用 `require` 命令，而要使用 `import()` 函数。ES6 模块的所有输出接口，会成为输入对象的属性。

```
// es.mjs
let foo = { bar:'my-default' };
export default foo;
foo = null;

// cjs.js
const es_namespace = await import('./es');
// es_namespace = {
//   get default() {
//     ...
//   }
// }
console.log(es_namespace.default);
// { bar:'my-default' }
```

上面代码中， `default` 接口变成了 `es_namespace.default` 属性。另外，由于存在缓存机制， `es.js` 对 `foo` 的重新赋值没有在模块外部反映出来。

下面是另一个例子。

```
// es.js
export let foo = { bar:'my-default' };
export { foo as bar };
export function f() {};
export class c {};

// cjs.js
const es_namespace = await import('./es');
// es_namespace = {
//   get foo() {return foo;}
//   get bar() {return foo;}
//   get f() {return f;}
```

```
//   get c() {return c;}
// }
```

---

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.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 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，[illegible]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
```

---

5. 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，安装这个转码器。

```
$ 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