



# 第44课：Module的基本语法

主讲老师：万章



## 目录

什么是Module

Module的基本语法

模块的整体加载

export 的其他命令

模块的其他细节



什么是module

## module(模块的历史)

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(多个script标签引入的js默认是相互独立的, 而不是依赖的)。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。

```
// 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”。

CommonJS 模块就是对象，输入时必须查找对象属性。



我举个栗子

## 什么是module

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';
```

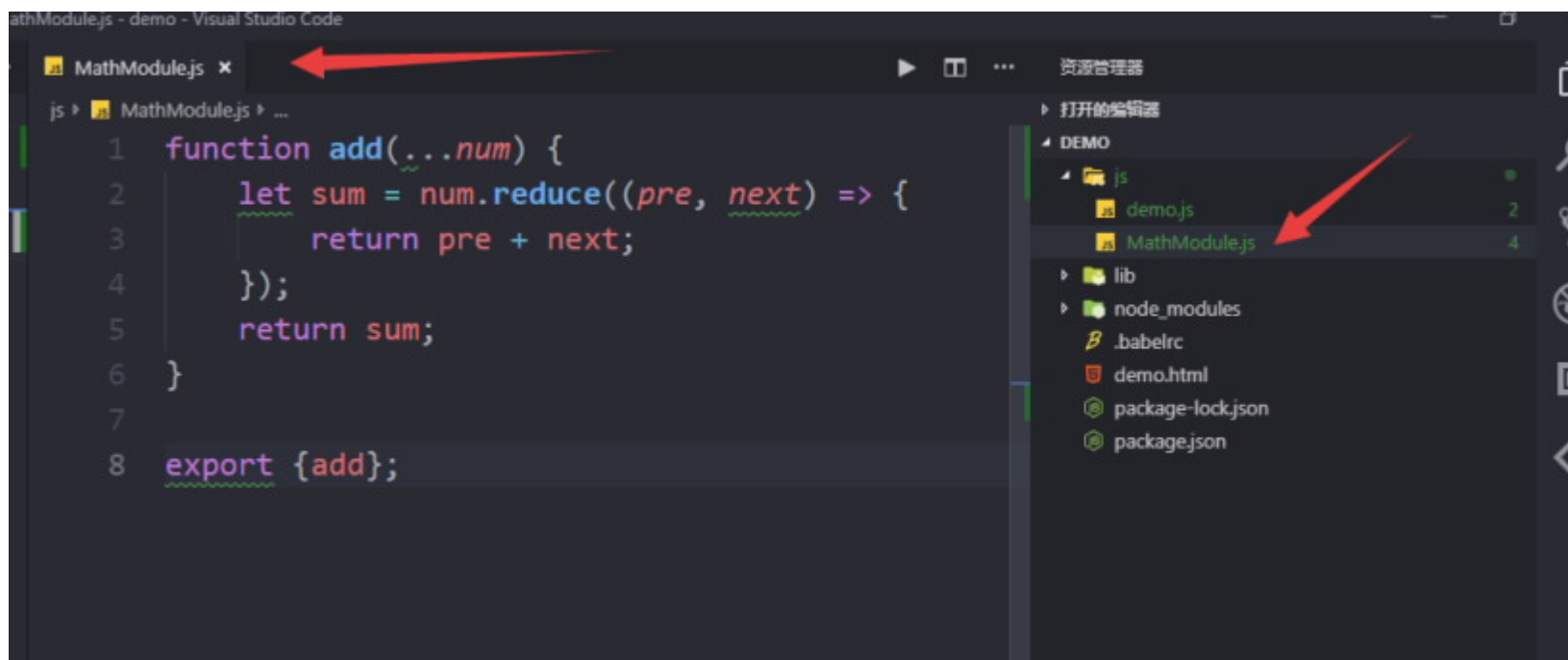
上面代码的实质是从fs模块加载 3 个方法，其他方法不加载。这种加载称为“编译时加载”或者静态加载，即 ES6 可以在编译时就完成模块加载，效率要比 CommonJS 模块的加载方式高。当然，这也导致了没法引用 ES6 模块本身，因为它不是对象。



# Module的基本语法

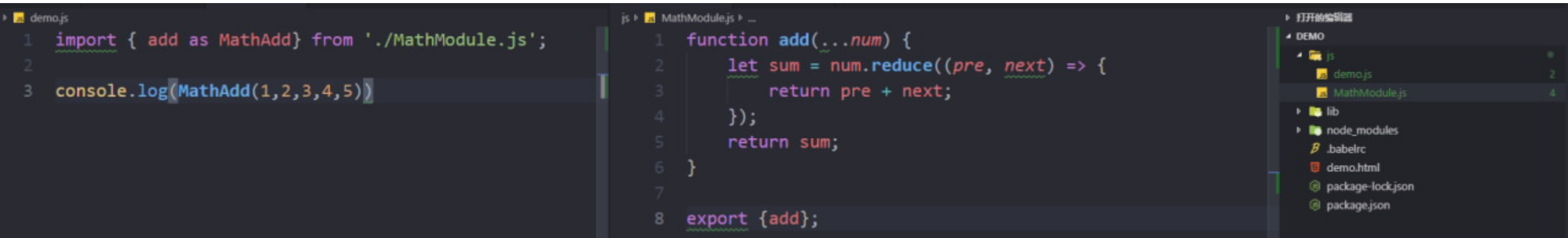
## Module的基本组成

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。

# Module的基本结构



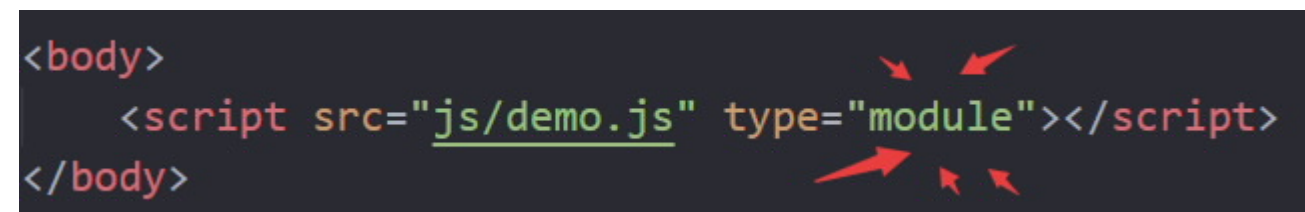
```
demo.js
1 import { add as MathAdd } from './MathModule.js';
2
3 console.log(MathAdd(1,2,3,4,5));
```

```
MathModule.js
1 function add(...num) {
2   let sum = num.reduce((pre, next) => {
3     return pre + next;
4   });
5   return sum;
6 }
7
8 export { add };
```

调用一个模块都要用import来  
调用其他模块

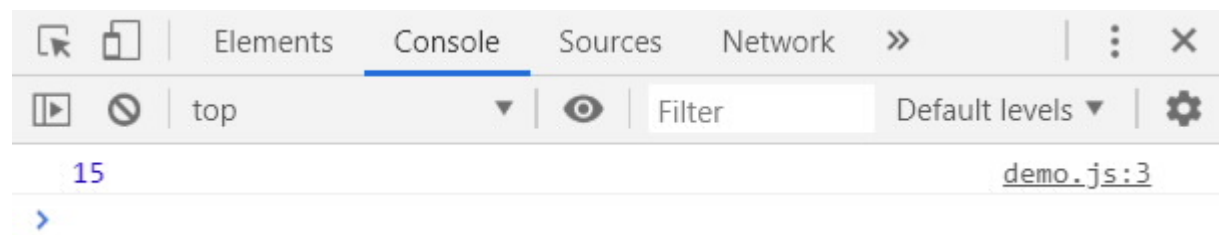
每个模块都要有输出export接  
口

单模块单文件存储



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

在网页上使用这种模块系统的时候，需要  
在script标签里面写上module



```
15
```

运行结果



## Module的export模块之变量输出

用export命令对外部输出了三个变量。

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

写法1

```
// profile.js
var firstName = 'Michael';
var lastName = 'Jackson';
var year = 1958;

export { firstName, lastName, year };
```

写法2

```
function v1() { ... }
function v2() { ... }

export {
  v1 as streamV1,
  v2 as streamV2,
  v2 as streamLatestVersion
};
```

写法3

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。

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。

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。

## Module的export模块之变量输出

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。

## Module的export模块之变量输出

```
// 报错
function f() {}
export f;

// 正确
export function f() {};

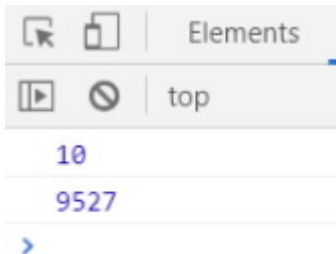
// 正确
function f() {}
export {f};
```

```
MathModule.js
1 function add(...num) {
2   let sum = num.reduce((pre, next) => {
3     return pre + next;
4   });
5   return sum;
6 }
7
8 let a=10;
9
10 setTimeout(()=>{
11   a=9527;
12 },5000)
13 export {add,a};

import { add as MathAdd, a } from './MathModule.js';

console.log(a)

setTimeout(()=>{
  console.log(a);
},6000)
```



同样的，function和class的输出，也必须遵守这样的写法。

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。

## Module的export模块之变量输出

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。

## Module的import模块之模块加载

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。

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 { add as MathAdd, a } from './MathModule.js';  
  
console.log(a)  
  
setTimeout(()=>{  
  console.log(a);  
}, 6000)
```

```
MathModule.js  
1 function add(...num) {  
2   let sum = num.reduce((pre, next) => {  
3     return pre + next;  
4   });  
5   return sum;  
6 }  
7  
8 let a=10;  
9  
10 setTimeout(()=>{  
11   a=9527;  
12 }, 5000)  
13 export {add, a};
```

上面代码的import命令，用于加载MathModule.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（MathModule.js）对外接口的名称相同

## Module的import模块之模块加载

import命令输入的变量都是只读的，因为它的本质是输入接口。也就是说，**不允许在加载模块的脚本里面，改写接口。**

```
import {a} from './xxx.js'
a = {}; // Syntax Error : 'a' is read-only;
```

上面代码中，脚本加载了变量a，对其重新赋值就会报错，因为a是一个只读的接口。

```
import {a} from './xxx.js'
a.foo = 'hello'; // 合法操作
```

但是，如果a是一个对象，改写a的属性是允许的。不过，这种写法很难查错，建议凡是输入的变量，都当作完全只读，轻易不要改变它的属性。

## Module的import模块之模块加载

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后缀可以省略。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';
```

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。这种行为本质是，import命令是编译阶段执行的，在代码运行之前。

## Module的import模块之模块加载

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。



## Module的import模块之模块加载

最后，import语句会执行所加载的模块，因此可以有右面的写法。

```
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';
```

```
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。



# 模块的整体加载

## 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。

```
js ▶ MathModule.js ▶ circumference
1  export function area(radius) {
2      return Math.PI * radius * radius;
3  }
4
5  export function circumference(radius) {
6      return 2 * Math.PI * radius;
7  }
```

模块文件代码

```
js demo.js
1  import * as circle from './MathModule.js';
2
3  console.log('圆面积: ' + circle.area(4));
4  console.log('圆周长: ' + circle.circumference(14));
```

主模块文件代码

圆面积: 50.26548245743669	demo.js:3
圆周长: 87.96459430051421	demo.js:4

执行结果

```
demo.js ▶ ...
1  import * as circle from './circle';
2
3  // 下面两行都是不允许的
4  circle.foo = 'hello';
5  circle.area = function () {};
```

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。上面的写法都是不允许的。



export 其他命名

## 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

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

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。

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

## export default命令

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

// 或者写成

function foo() {
  console.log('foo');
}

export default foo;
```

export default命令用在非匿名函数前，也是可以的。  
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。

比较一下默认输出和正常输出

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**。

```
// 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
```

## export default命令

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，因为只可能唯一对应export default命令。

```
// modules.js
function add(x, y) {
  return x * y;
}
export {add as default};
// 等同于
// export default add;

// app.js
import { default as foo } from 'modules';
// 等同于
// import foo 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。

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，左侧的写法是有效的。

```
// 正确
export var a = 1;

// 正确
var a = 1;
export default a;

// 错误
export default var a = 1;
```

## export default命令

同样地，因为export default命令的本质是将后面的值，赋给default变量，所以可以直接将一个值写在export default之后。

```
// 正确
export default 42;

// 报错
export 42;
```

左侧代码中，后一句报错是因为没有指定对外的接口，而前一句指定对外接口为default。

```
demo.js
1 import _, { each, forEach } from './MathModule.js';
2
3 console.log(_);
4 console.log(each);
5 console.log(forEach);
```

```
js ▶ MathModule.js ▶ ...
1 export default function (obj) {
2   // ...
3 }
4
5 export function each(obj, iterator, context) {
6   // ...
7 }
8
9 export {
10   each as forEach
11 };
```

```
f (obj) {
  // ...
}
demo.js:3
f each(obj, iterator, context) {
  // ...
}
demo.js:4
f each(obj, iterator, context) {
  // ...
}
demo.js:5
```

如果想在一条import语句中，同时输入默认方法和其他接口，可以写成上面这样。没放在花括号里面的就是接收default输出的值，花括号里面的就是接收变量匹配的



## export 和import 的复合写法

一个模块既可以引入也可以输出，甚至可以两者一起实现



```
export { foo, bar } from 'my_module';  
  
// 可以简单理解为  
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。但需要注意的是，写成一行以后，foo和bar实际上并没有被导入当前模块，只是相当于对外转发了这两个接口，导致当前模块不能直接使用foo和bar。

## export 和import 的复合写法

模块的接口改名和整体输出，也可以采用这种写法。

```
export { default } from 'foo';
```

默认接口的写法

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

具名接口改为默认接口的写法如下。

```
// 接口改名  
export { foo as myFoo } from 'my_module';  
  
// 整体输出  
export * from 'my_module';
```

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

同样地，默认接口也可以改名为具名接口。



## 模块的其他细节

## 模块的继承

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/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'];
```

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。

然后，将这些文件输出的常量，合并到index.js里面。使用的时候，直接加载index.js就可以了。

```
// constants/index.js
export {db} from './db';
export {users} from './users';
// script.js
import {db, users} from './constants/index';
```

## import 当前的缺憾

前面介绍过，import命令会被 JavaScript 引擎静态分析，先于模块内的其他语句执行（import命令叫做“连接” binding 其实更合适）

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

**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。

在语法上，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命令做不到这一点。

这有个新的ES6提案，可以有效解决这个问题，但是还需要时间

import() <https://github.com/tc39/proposal-dynamic-import>

## 模块的几个注意点

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。

1. 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。
2. **模块脚本自动采用严格模式，不管有没有声明use strict。**
3. 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。
4. 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。
5. 同一个模块如果加载多次，将只执行一次。