[**http://es6.ruanyifeng.com/#docs/module#export**](http://es6.ruanyifeng.com/#docs/module#export)

**him.ts**

export let HisHouse = "His House";

export default "sHim Default Yes";

export class HOUSE {

constructor(public name: string) { }

show(): void {

alert("Him.ts House Show: " + this.name);

}

}

**her.ts**

export { default as HH, HisHouse, HOUSE } from "./him";

export var google = "google";

// error 不能重复定义

export class HOUSE {

constructor(public addr: string) { }

}

**him.ts**

export let HisHouse = "His House";

export default "Him Default Yes";

export class HOUSE {

constructor(public name: string) { }

show(): void {

alert("Him.ts House Show: " + this.name);

}

}

**her.ts**

export \* from "./him";

export var google = "google";

export class HOUSE1 {

constructor(public addr: string) { }

}

// default 会覆盖掉 default from “./him”

// 如果没有定义default 则使用导入的default

export default "Her Default replace";

**注意：**

1. export herdef, { HOUSE, HisHouse } from "./him"; -- 语法出错

export .. from 不支持这种引用导出

1. 引用导出的对象， 不能被使用。

export { default as HH, HOUSE, HisHouse } from "./him";

// HH, HOUSE, HisHouse 都不能被使用

import HH from "./him"; -- 如果想使用，必须使用import导入以后再深造。

export default HH;

案例：

Wliu2.ts

export let s2 = ()=>console.log("s2 from wliu2");

export let url = "wliu2 url";

export default ()=>console.log("s1 from wliu2");

console.log("Wliu2.ts loading");

Wliu1.ts

export let url = "www.d3security.com"; - 这里的 url 优先级更高，即使wliu2模块是最后被导入

var sub = (a:number, b:number) => a-b;

export { sub }

function add(a:number, b:number) {

return a+b;

}

export {add};

export { default as default } from "./wliu2"; 如果想把wliu2 里的default 当做default 输出

export { default } from "./wliu2"; 效果同上

export \* from "./wliu2"; 导出同一个模块，可以放在多个语句里导出同一

console.log("Wliu1.ts loading");

Test.ts

import dd, \* as gg from "./lib/wliu1";

console.log(dd());

console.log(gg.url);

console.log(gg.sub(123, 100) );

console.log(gg.s2());

import dd from "./lib/wliu1"; - **ok** 只输出默认的，给默认赋予dd的名字，其他的不输出

import ww, \* as gg from "./lib/wliu1"; - **ok**, ww是承接default, gg是承接整个模块里输出函数

import ww, {default as gg, url, sub} from "./lib/wliu1"; - **ok**, ww和gg都是default, 其他单独导入

import ww, {default as gg, url as uu, sub} from "./lib/wliu1"; - **ok**，可以有别名

import \* from "./lib/wliu1"; - 语法错误，用于 export 可以， 但是import 则出错

import ww, {\* as gg} from "./lib/wliu1"; - 语法错误

import ww, url, sub from "./lib/wliu1"; - 语法错误

-----------------------------------

import "./lib/wliu1"; - ok, 只是执行模块里的代码，但是并不导入任何方法。

运行结果：

[Running] ts-node "d:\GitTest\test.ts"

Wliu2.ts loading

Wliu1.ts loading

想使用模块里的方法，则会找不到方法而出错

console.log(add(20,40));

console.log(sub(122,80));

-----------------------------------

Wliu1.ts

var google = "Google Website";

console.log("Wliu1.ts loading: " + google);

import "./lib/wliu1"; - ok, 只是执行模块里的代码，但是并不导入任何方法。

console.log(google); - 但是访问变量google 出错 , 因为没有被导入, 也就是说模块有自己的封装域

[Running] ts-node "d:\GitTest\test.ts"

Wliu2.ts loading: Yahoo Website

Wliu1.ts loading: Google Website

-------------------------------------------------------------

import \* as gg from "./lib/wliu1"; 把整个输出给变量 gg , 默认也可以使用 gg.default 即可

gg.default()

console.log(gg.url);

console.log(gg.sub(122,80));

---------------------------------------------------------------

import \* as gg from "./lib/wliu1";

import ww from "./lib/wliu1"; - 可以多次导入同一个模块，分别调用不同，但是加载模块只是静态执行一次

ww();

gg.default();

console.log(gg.url);

console.log(gg.sub(122,80));

**Module 的语法**

1. [概述](http://es6.ruanyifeng.com/#docs/module#概述)
2. [严格模式](http://es6.ruanyifeng.com/#docs/module#严格模式)
3. [export 命令](http://es6.ruanyifeng.com/#docs/module#export 命令)
4. [import 命令](http://es6.ruanyifeng.com/#docs/module#import 命令)
5. [模块的整体加载](http://es6.ruanyifeng.com/#docs/module#模块的整体加载)
6. [export default 命令](http://es6.ruanyifeng.com/#docs/module#export default 命令)
7. [export 与 import 的复合写法](http://es6.ruanyifeng.com/#docs/module#export 与 import 的复合写法)
8. [模块的继承](http://es6.ruanyifeng.com/#docs/module#模块的继承)
9. [跨模块常量](http://es6.ruanyifeng.com/#docs/module#跨模块常量)
10. [import()](http://es6.ruanyifeng.com/#docs/module#import())

**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。

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，不再必须做成全局变量或者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。

本章介绍 ES6 模块的语法，下一章介绍如何在浏览器和 Node 之中，加载 ES6 模块。

**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。

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。

**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《Module 的加载实现》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。

**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。

// main.js

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

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.js';

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。但是，如果a是一个对象，改写a的属性是允许的。

import {a} from './xxx.js'

a.foo = 'hello'; // 合法操作

上面代码中，a的属性可以成功改写，并且其他模块也可以读到改写后的值。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。

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';

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。

目前阶段，通过 Babel 转码，CommonJS 模块的require命令和 ES6 模块的import命令，可以写在同一个模块里面，但是最好不要这样做。因为import在静态解析阶段执行，所以它是一个模块之中最早执行的。下面的代码可能不会得到预期结果。

require('core-js/modules/es6.symbol');

require('core-js/modules/es6.promise');

import React from 'React';

**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 () {};

**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，因为只可能唯一对应export default命令。

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 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。

// 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;

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接口，可以写成下面这样。

import \_, { each, each as forEach } 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();

**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';

// 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。

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](https://github.com/leebyron/ecmascript-export-default-from)，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";

**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。

**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 './index';

**import()**

**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。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语句做不到这一点。

因此，有一个[提案](https://github.com/tc39/proposal-dynamic-import)，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。

**适用场合**

下面是import()的一些适用场合。

（1）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。

（2）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。

（3）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。

**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();