module

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。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。

由于 ES6 模块是编译时加载，使得静态分析成为可能。

- 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。

ES6 的模块自动采用严格模式，不管有没有在模块头部加上 `"use strict";` 。

一个模块就是一个独立的文件。 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。

1. export

输出变量：

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

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

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

输出函数或类：

```
export function multiply(x, y) {
  return x * y;
};
```

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
};
```

重命名后，`v2` 用不同的名字输出了两次。

`export` 命令能够对外输出的就是三种接口：函数（Functions），类（Classes），`var`、`let`、`const` 声明的变量（Variables）。

`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。 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。

2. import

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;
}
```

上例的 `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，必须与导出模块（profile.js）对外接口的名称相同。**

将输入的变量重命名，`import` 命令要使用 `as` 关键字：

```
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;
```

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，不要轻易改变它的属性。

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

a.foo = 'hello'; // 合法操作
```

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。在静态分析阶段，这些语法都是没法得到值的。

```
// 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';
}
```

3. 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。

```
// circle.js
export function area(radius) {
  return Math.PI * radius * radius;
}
export function circumference(radius) {
  return 2 * Math.PI * radius;
}
```

```
// 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 () {};
```

4. export default

`export default` 命令，为模块指定默认输出。

```
// export-default.js
export default function () {
  console.log('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。

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

上例的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。**这时import命令后面，不使用大括号。**

```
// 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
```

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。

```
// 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';
```

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;
```

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, forEach } from 'lodash';
```

5. 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 };
```

`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';
```

```
export * as ns from "mod";

// 等同于
import * as ns from "mod";
export {ns};
```

6. import() 函数

`import()` **返回一个 Promise 对象。**

6.1. 按需加载

```
button.addEventListener('click', event => {
  import('./dialogBox.js')
  .then(dialogBox => {
    dialogBox.open();
  })
  .catch(error => {
    /* Error handling */
  })
});
```

只有用户点击了按钮，才会加载这个模块。

6.2. 条件加载

```
if (condition) {
  import('moduleA').then(...);
} else {
  import('moduleB').then(...);
}
```

放在if代码块，根据不同的情况，加载不同的模块。

6.3. 动态的模块路径

```
import(f()).then(...);
```

根据函数 `f` 的返回结果，加载不同的模块。

7. import.meta

开发者使用一个模块时，有时需要知道模板本身的一些信息（比如模块的路径）。`import.meta` 返回当前模块的元信息。

`import.meta` 只能在模块内部使用，如果在模块外部使用会报错。

这个属性返回一个对象，该对象的各种属性就是当前运行的脚本的元信息。具体包含哪些属性，标准没有规定，由各个运行环境自行决定。一般来说，`import.meta` 至少会有下面两个属性。

（1）`import.meta.url`

`import.meta.url` 返回当前模块的 URL 路径。举例来说，当前模块主文件的路径是 `https://foo.com/main.js`，`import.meta.url` 就返回这个路径。如果模块里面还有一个数据文件 `data.txt`，那么就可以用下面的代码，获取这个数据文件的路径。

```
new URL('data.txt', import.meta.url)
```

Node.js 环境中，`import.meta.url` 返回的总是本地路径，即 `file:URL` 协议的字符串，比如 `file:///home/user/foo.js`。

（2）`import.meta.scriptElement`

`import.meta.scriptElement` 是浏览器特有的元属性，返回加载模块的那个 `<script>` 元素，相当于 `document.currentScript` 属性。

```
// HTML 代码为：<script type="module" src="my-module.js" data-foo="abc"></script>

// my-module.js 内部执行下面的代码
import.meta.scriptElement.dataset.foo
// "abc"
```

8. 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>
```

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>
```

9. 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"
}
```

上例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
```

上例中，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。