require 与 import 的区别

对于 require 和 import 一个是 Node 的导入语法，另一个是 ES6 中引入的模块化方式引入。node 的模块化语法，遵循 Commonjs 规范。接下来我们分别来看看他们之间的基本使用与差异。

CommonJS 模块的加载原理

CommonJs 规范规定，每个模块内部，`module` 变量代表当前模块。这个变量是一个对象，它的 `exports` 属性（即 `module.exports`）是对外的接口，加载某个模块，其实是加载该模块的 `module.exports` 属性。

```
const x = 5;
const addX = function (value) {
  return value + x;
};
module.exports.x = x;
module.exports.addX = addX;
```

上面代码通过 module.exports 输出变量 x 和函数 addX。

require 方法用于加载模块。

```
const example = require('./example.js');

console.log(example.x); // 5
console.log(example.addX(1)); // 6
```

CommonJS 模块的特点如下：

- 所有代码运行在模块作用域，不会污染全局作用域
- 模块可以多次加载，但是只会在第一次加载时运行一次，然后运行结果就被缓存了，以后再加载，就直接读取缓存结果。要想让模块再次运行，必须清除缓存。
- 模块加载的顺序，按照其在代码中出现的顺序

module 对象

Node 内部提供一个 Module 构建函数。所有模块都是 Module 的实例。

```
function Module(id, parent) {
  this.id = id;
  this.exports = {};
  this.parent = parent;
  // ...
}
```

每个模块内部，都有一个 module 对象，代表当前模块。它有以下属性。

- module.id 模块的识别符，通常是带有绝对路径的模块文件名。
- module.filename 模块的文件名，带有绝对路径。
- module.loaded 返回一个布尔值，表示模块是否已经完成加载。
- module.parent 返回一个对象，表示调用该模块的模块。
- module.children 返回一个数组，表示该模块要用到的其他模块。
- module.exports 表示模块对外输出的值。

module.exports 属性表示当前模块对外输出的接口，其他文件加载该模块，实际上就是读取 module.exports 变量。

为了方便，Node 为每个模块提供一个 exports 变量，指向 module.exports。这等同在每个模块头部，有一行这样的命令

```
const exports = module.exports;
```

注意，不能直接将 exports 变量指向一个值，因为这样等于切断了 exports 与 module.exports 的联系。

```
exports = function(x) {console.log(x)};
```

上面这样的写法是无效的，因为 exports 不再指向 module.exports 了。

下面的写法也是无效的。

```
exports.hello = function() {
  return 'hello';
```

```
};

module.exports = 'Hello world';
```

上面代码中，hello 函数是无法对外输出的，因为 module.exports 被重新赋值了。

这意味着，如果一个模块的对外接口，就是一个单一的值，最好不要使用 exports 输出，最好使用 module.exports 输出。

```
module.exports = function (x){ console.log(x);};
```

如果你觉得，exports 与 module.exports 之间的区别很难分清，一个简单的处理方法，就是放弃使用 exports，只使用 module.exports。

模块的缓存

第一次加载某个模块时，Node 会缓存该模块。以后再加载该模块，就直接从缓存取出该模块的 module.exports 属性。

```
require('./example.js');
require('./example.js').message = "hello";
require('./example.js').message
// "hello"
```

上面代码中，连续三次使用 require 命令，加载同一个模块。第二次加载的时候，为输出的对象添加了一个 message 属性。但是第三次加载的时候，这个 message 属性依然存在，这就证明 require 命令并没有重新加载模块文件，而是输出了缓存。

如果想要多次执行某个模块，可以让该模块输出一个函数，然后每次 require 这个模块的时候，重新执行一下输出的函数。

所有缓存的模块保存在 require.cache 之中，如果想删除模块的缓存，可以像下面这样写。

```
// 删除指定模块的缓存
delete require.cache[moduleName];

// 删除所有模块的缓存
Object.keys(require.cache).forEach(function(key) {
  delete require.cache[key];
})
```

注意，缓存是根据绝对路径识别模块的，如果同样的模块名，但是保存在不同的路径，require 命令还是会重新加载该模块。

ES6 模块

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。

export 命令

ES6 的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。

- ES6 模块必须用 export 导出
- export 必须与模块内部的变量建立一一对应关系

1. 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。

```
export const firstName = 'Michael';
export function multiply(x, y) {
  return x * y;
};
```

1. 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
const 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。

```
// 写法一
export const m = 1;

// 写法二
const m = 1;
export {m};

// 写法三
const n = 1;
export {n as m};
```

import 命令

- import 命令输入的变量都是只读的
- import 命令具有提升效果
- import 是静态执行，所以不能使用表达式和变量
- import 语句是 Singleton 模式

1. 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，不要轻易改变它的属性。

1. 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';
```

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。

1. import 是静态执行，所以不能使用表达式和变量

```
// 报错
import { 'f' + 'oo' } from 'my_module';

// 报错
let module = 'my_module';
import { foo } from module;
```

1. 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 { 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。

export default 命令

- `export default` 就是输出一个叫做 default 的变量或方法
- `export default` 所以它后面不能跟变量声明语句

1. 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。

```
// modules.js
function sayHello() {
  console.log('哈哈哈')
}
export { sayHello as default};
// 等同于
// export default sayHello;

// app.js
import { default as sayHello } from 'modules';
// 等同于
// import sayHello from 'modules';
```

1. 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 const a = 1;

// 正确
const a = 1;
export default a;

// 错误
export default const 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，而前一句指定对外接口为 default。

export 和 import 的复合写法

- 在一个模块里导入同时导出模块

```
export { foo, bar } from 'my_module';

// 可以简单理解为
import { foo, bar } from 'my_module';
export { foo, bar };
```

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

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

在平常开发中这种常被用到，有一个 utils 目录，目录下面每个文件都是一个工具函数，这时候经常会创建一个 index.js 文件作为 utils 的入口文件，index.js 中引入 utils 目录下的其他文件，其实这个 index.js 其的作用就是一个对外转发 utils 目录下 所有工具函数的作用，这样其他在使用 utils 目录下文件的时候可以直接 通过 `import { xxx } from './utils'` 来引入。

ES6 模块和 CommonJs 模块主要有以下两大区别

- 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，在代码静态编译阶段就会生成。

在传统编译语言的流程中，程序中的一段源代码在执行之前会经历三个步骤，统称为编译。”分词/词法分析” -> ”解析/语法分析” -> "代码生成"。
下面来解释一下第一个区别
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
const 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
const 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
const 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。

```
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 const 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。

```
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。

这就证明了 x.js 和 y.js 加载的都是 C 的同一个实例。

在平常开发中这种常被用到，有一个 utils 目录，目录下面每个文件都是一个工具函数，这时候经常会创建一个 index.js 文件作为 utils 的入口文件，index.js 中引入 utils 目录下的其他文件，其实这个 index.js 其的作用就是一个对外转发

utils 目录下 所有工具函数的作用，这样其他在使用 utils 目录下文件的时候可以直接 通过 import { xxx } from './utils' 来引入。

-