# ES Module

## 仅导出

- named exports: 命名导出，每次可以导出一个或者多个。
- default exports: 默认导出，每次只能存在一个。

以上两者可以混合导出。

```
 // 命名导出
  export const b = 'b'
  // 默认导出
  export default {
    a: 1
  };

  const c = 'c'
  export { c }

// 以上内容会合并导出，即导出为:  {b:'b', c:'c', default: {a:1}}
```

## 重导出（re-exporting / aggregating）

算是一个导入再导出的一个语法糖吧。

```
export {
  default as function1,
  function2,
} from 'bar.js';

// 等价于
```

```
import { default as function1, function2 } from 'bar.js';
export { function1, function2 };
```

然而这种语法是会报错的：

```
export DefaultExport from 'bar.js'; // Invalid
```

正确的语法应该是：

```
export { defaultasDefaultExport } from'bar.js'; // valid复制代码
```

我猜是因为export 本身支持的export xxx这种语法必须是要导出一个对象，然而import xxx可能是任意类型，两者冲突了，所以从编译层面就不让这种语法生效会更好。

### 嵌入式脚本

嵌入式脚本不可以使用export。

## 引入

### 语法

- import all exports: `import * as allVar`，所有导出内容，包含命名导出及默认导出。allVar会是一个对象，默认导出会作为allVar的key名为default对应的值。
- import named exports: `import {var1, var2}`，引入命名导出的部分。没找到，对应的值就为undefined。个人觉得可以看做是"import all exports"的解构语法。
- import default exports: `import defaultVar`，引入默认导出的部分。
- import side effects: `import "xxx./js"`，仅运行这个js，可能是为了获取其副作用。

参考nodejs进阶视频讲解：进入学习

```
// test.js
export const b = 'b'      // 命名导出
export default {    // 默认导出
  a: 1
};

// index.js
import { b, default as _defaultModule } from './test.js'
import defaultModule from './test.js'
import * as allModule from './test.js'

console.log('name export', b) // 'b'
console.log('default export', defaultModule) // {a:1}
console.log(_defaultModule === defaultModule) // true
console.log('all export', allModule) // {b:'b', default: {a:1}}
```

一个之前老记错的case

```
// test.js
export default {    // 默认导出
  a: 1
};

// index.js
import { a } from './test.js'
console.log('name export', a) // undefined

// index.js
import defaultModule from './test.js'
import * as allModule from './test.js'
console.log('default export', defaultModule) // {a:1}
console.log('all export', allModule) // {default: {a:1}}
```

## 嵌入式脚本

嵌入式脚本引入modules时，需要在script上增加 type="module"。

## 特点

- live bindings

通过export在mdn上的解释，export导出的是live bindings，再根据其他文章综合判断，应该是引用的意思。即**export导出的是引用**。

模块内的值更新了之后，所有使用export导出值的地方都能使用最新值。

- read-only

通过import在mdn上的解释，import使用的是通过export导出的**不可修改的引用**。

- strict-mode

被引入的模块都会以严格模式运行。

- 静态引入、动态引入

`import x from`这种语法有syntactic rigid，需要编译时置于顶部且无法做到动态引入加载。如果需要动态引入，则需要`import ()`语法。有趣的是，在mdn上，前者分类到了 **Statements & declarations**，后者分类到了 **Expressions & operators**。这俩是根据什么分类的呢?

```
true && import test from "./a.js";
// SyntaxError: import can only be used in import() or import.meta
// 这里应该是把import当成了动态引入而报错
```

- 示例

```
// a.js
```

```
// a.js
const test = {
  a: 1
};
export default test;
// 改动模块内部的值
setTimeout(() => {
  test.a = 2;
}, 1000);

// index.js
import test from './index.js'

/* live bindings */
console.log(test) // {a:1}
setTimeout(()=>{
  console.log(test) // {a:2}
}, 2000)

/* read-only */
test= { a: 3 } // 报错, Error: "test" is read-only.

/* syntactically rigid */
if(true){
  import test from './index.js' // 报错, SyntaxError: 'import' and 'export' may
}
```

## commonJS

## 导出

在 Node.js 模块系统中，每个文件都被视为独立的模块。模块导入导出实际是由nodejs的模块封装器实现，通过为`module.exports`分配新的值来实现导出具体内容。

module.exports有个简写变量exports，其实就是个引用复制。exports作用域只限于模块文件内部。原理类似于：

```
// nodejs内部
exports = module.exports

console.log(exports, module.exports) // {}, {}
console.log(exports === module.exports) // true
```

注意，nodejs实际导出的是module.exports，以下几种经典case单独看一下：

case1

```
// ✅使用exports
exports.a = xxx
console.log(exports === module.exports) // true

// ✅等价于
module.exports.a = xxx
```

case2:

```
// ✅这么写可以导出，最终导出的是{a:'1'}
module.exports = {a:'1'}

console.log(exports, module.exports) // {}, {a:'1'}
console.log(exports === module.exports) // false


// ❌不会将{a:'1'}导出，最终导出的是{}
```

```
exports = {a:'1'}

console.log(exports, module.exports) // {a:'1'}, {}
console.log(exports === module.exports) // false
```

## 引入

通过require语法引入：

```
// a是test.js里module.exports导出的部分
const a = require('./test.js')
```

原理伪代码：

```
function require(/* ... */) {
  const module = { exports: {} };
  ((module, exports) => {
    // Module code here. In this example, define a function.
    function someFunc() {}
    exports = someFunc;
    // At this point, exports is no longer a shortcut to module.exports, and
    // this module will still export an empty default object.
    module.exports = someFunc;
    // At this point, the module will now export someFunc, instead of the
    // default object.
  })(module, module.exports);
  return module.exports;
}
```

## 特点

## 值拷贝

```
// test.js
let test = {a:'1'}
setTimeout(()=>{
  test = {a:'2'}
},1000)
module.exports = test

// index.js
const test1 = require('./test.js')
console.log(test1) // {a:1}
setTimeout(()=>{
  console.log(test1) // {a:1}
},2000)
```

# ES Module和 commonJS区别

1. 语法

`exports`、`module.exports`和`require` 是**Node.js模块系统**关键字。

`export`、`export default`和`import` 则是**ES6模块系统**的关键字:

1. 原理

`exports`、`module.exports`导出的模块为值复制。

`export`、`export default`为引用复制。

1. 时机

ES Module静态加载是编译时确定，ES Module动态加载是运行时确定。

CommonJS是运行时确定。