

# JavaScript模块化

王红元 coderwhy

# 错误处理方案

- 开发中我们会封装一些工具函数，封装之后给别人使用：
  - 在其他使用人使用的过程中，可能会传递一些参数；
  - 对于函数来说，需要对这些参数进行验证，否则可能得到的是我们不想要的结果；
- 很多时候我们可能验证到不是希望得到的参数时，就会直接return：
  - 但是return存在很大的弊端：调用者不知道是因为函数内部没有正常执行，还是执行结果就是一个undefined；
  - 事实上，正确的做法应该是如果没有通过某些验证，那么应该让外界知道函数内部报错了；
- 如何可以让一个函数告知外界自己内部出现了错误呢？
  - 通过throw关键字，抛出一个异常；
- throw语句：
  - throw语句用于抛出一个用户自定义的异常；
  - 当遇到throw语句时，当前的函数执行会被停止（throw后面的语句不会执行）；
- 如果我们执行代码，就会报错，拿到错误信息的时候我们可以及时的去修正代码。

# throw关键字

- throw表达式就是在throw后面可以跟上一个表达式来表示具体的异常信息：

```
throw expression;
```

- throw关键字可以跟上哪些类型呢？

- 基本数据类型：比如number、string、Boolean

- 对象类型：对象类型可以包含更多的信息

- 但是每次写这么长的对象又有点麻烦，所以我们可以创建一个类：

```
class HYError {  
  constructor(errCode, errMsg) {  
    this.errCode = errCode  
    this.errMessage = errMsg  
  }  
}
```

- 事实上，JavaScript已经给我们提供了一个Error类，我们可以直接创建这个类的对象：

```
function foo() {  
  throw new Error("error message", "123")  
}
```

- Error包含三个属性：

- message：创建Error对象时传入的message；
- name：Error的名称，通常和类的名称一致；
- stack：整个Error的错误信息，包括函数的调用栈，当我们直接打印Error对象时，打印的就是stack；

- Error有一些自己的子类：

- RangeError：下标值越界时使用的错误类型；
- SyntaxError：解析语法错误时使用的错误类型；
- TypeError：出现类型错误时，使用的错误类型；

# 异常的处理

## ■ 我们会发现在之前的代码中，一个函数抛出了异常，调用它的时候程序会被强制终止：

- 这是因为如果我们在调用一个函数时，这个函数抛出了异常，但是我们并没有对这个异常进行处理，那么这个异常会继续传递到上一个函数调用中；
- 而如果到了最顶层（全局）的代码中依然没有对这个异常的处理代码，这个时候就会报错并且终止程序的运行；

## ■ 我们先来看一下这段代码的异常传递过程：

- foo函数在被执行时会抛出异常，也就是我们的bar函数会拿到这个异常；
- 但是bar函数并没有对这个异常进行处理，那么这个异常就会被继续传递到调用bar函数的函数，也就是test函数；
- 但是test函数依然没有处理，就会继续传递到我们的全局代码逻辑中；
- 依然没有被处理，这个时候程序会终止执行，后续代码都不会再执行了；

```
function foo() {  
  throw "coderwhy error message"  
}  
  
function bar() {  
  foo()  
}
```

```
function test() {  
  bar()  
}  
  
test()  
console.log("test后续代码~")
```

# 异常的捕获

■ 但是很多情况下当出现异常时，我们并不希望程序直接推出，而是希望可以正确的处理异常：

□ 这个时候我们就可以使用try catch

```
try {  
  try_statements  
}  
[catch (exception_var_1) {  
  catch_statements_1  
}]  
[finally {  
  finally_statements  
}]
```

```
function foo() {  
  throw "coderwhy error message"  
}  
  
function bar() {  
  try {  
    foo()  
    console.log("foo后续的代码~")  
  } catch (error) {  
    console.log(error)  
  }  
}
```

■ 在ES10 ( ES2019 ) 中，catch后面绑定的error可以省略。

■ 当然，如果有一些必须要执行的代码，我们可以使用finally来执行：

□ finally表示最终一定会被执行的代码结构；

□ 注意：如果try和finally中都有返回值，那么会使用finally当中的返回值；

# 什么是模块化？

## ■ 到底什么是模块化、模块化开发呢？

- 事实上模块化开发最终的目的是将程序划分成一个个小的结构；
- 这个结构中编写属于自己的逻辑代码，有自己的作用域，不会影响到其他的结构；
- 这个结构可以将自己希望暴露的变量、函数、对象等导出给其结构使用；
- 也可以通过某种方式，导入另外结构中的变量、函数、对象等；

## ■ 上面说提到的结构，就是模块；按照这种结构划分开发程序的过程，就是模块化开发的过程；

## ■ 无论你多么喜欢JavaScript，以及它现在发展的有多好，它都有很多的缺陷：

- 比如var定义的变量作用域问题；
- 比如JavaScript的面向对象并不能像常规面向对象语言一样使用class；
- 比如JavaScript没有模块化的问题；

## ■ Brendan Eich本人也多次承认过JavaScript设计之初的缺陷，但是随着JavaScript的发展以及标准化，存在的缺陷问题基本都得到了完善。

## ■ 无论是web、移动端、小程序端、服务器端、桌面应用都被广泛的使用；

# 模块化的历史

- 在网页开发的早期，*Brendan Eich*开发JavaScript仅仅作为一种**脚本语言**，做一些简单的表单验证或动画实现等，那个时候代码还是很少的：
  - 这个时候我们只需要讲JavaScript代码写到**<script>标签**中即可；
  - 并没有必要放到多个文件中来编写；甚至流行：通常来说 JavaScript 程序的**长度只有一行**。
- 但是随着前端和JavaScript的快速发展，JavaScript代码变得越来越复杂了：
  - ajax的出现，**前后端开发分离**，意味着后端返回数据后，我们需要通过**JavaScript进行前端页面的渲染**；
  - SPA的出现，前端页面变得更加复杂：包括**前端路由**、**状态管理**等等一系列复杂的需求需要通过JavaScript来实现；
  - 包括Node的实现，JavaScript编写**复杂的后端程序**，没有模块化是致命的硬伤；
- 所以，模块化已经是JavaScript一个非常迫切的需求：
  - 但是JavaScript本身，直到**ES6**（2015）才推出了自己的模块化方案；
  - 在此之前，为了让JavaScript支持模块化，涌现出了很多不同的模块化规范：**AMD**、**CMD**、**CommonJS**等；
- 在我们的课程中，我将详细讲解JavaScript的模块化，尤其是CommonJS和ES6的模块化。



# 没有模块化带来的问题

- 早期没有模块化带来了很多问题：比如命名冲突的问题
- 当然，我们有办法可以解决上面的问题：立即函数调用表达式（IIFE）
  - IIFE (Immediately Invoked Function Expression)
- 但是，我们其实带来了新的问题：
  - 第一，我必须记得**每一个模块中返回对象的命名**，才能在其他模块使用过程中正确的使用；
  - 第二，代码写起来**混乱不堪**，每个文件中的代码都需要**包裹在一个匿名函数中来编写**；
  - 第三，在**没有合适的规范**情况下，每个人、每个公司都可能会任意命名、甚至出现模块名称相同的情况；
- 所以，我们会发现，虽然实现了模块化，但是我们的实现过于简单，并且是没有规范的。
  - 我们需要制定一定的规范来约束每个人都**按照这个规范去编写模块化的代码**；
  - 这个规范中应该包括核心功能：**模块本身可以导出暴露的属性，模块又可以导入自己需要的属性**；
  - JavaScript社区为了解决上面的问题，涌现出**一系列好用的规范**，接下来我们就学习具有代表性的一些规范。



# CommonJS规范和Node关系

- 我们需要知道CommonJS是一个规范，最初提出来是在浏览器以外的地方使用，并且当时被命名为ServerJS，后来为了体现它的广泛性，修改为CommonJS，平时我们也会简称为CJS。
  - Node是CommonJS在服务器端一个具有代表性的实现；
  - Browserify是CommonJS在浏览器中的一种实现；
  - webpack打包工具具备对CommonJS的支持和转换；
- 所以，Node中对CommonJS进行了支持和实现，让我们在开发node的过程中可以方便的进行模块化开发：
  - 在Node中每一个js文件都是一个单独的模块；
  - 这个模块中包括CommonJS规范的核心变量：exports、module.exports、require；
  - 我们可以使用这些变量来方便的进行模块化开发；
- 前面我们提到过模块化的核心是导出和导入，Node中对其进行了实现：
  - exports和module.exports可以负责对模块中的内容进行导出；
  - require函数可以帮助我们导入其他模块（自定义模块、系统模块、第三方库模块）中的内容；

# 模块化案例

■ 我们来看一下两个文件：

JS main.js

05\_javascript-module > 02\_commonjs > JS main.js

```
1 console.log(name);  
2 console.log(age);  
3  
4 sayHello('kobe');  
5
```

main必须进行导入

JS bar.js

05\_javascript-module > 02\_commonjs > JS bar.js > ...

```
1 const name = 'coderwhy';  
2 const age = 18;  
3  
4 function sayHello(name) {  
5   console.log("Hello " + name);  
6 }
```

bar必须导出自己的内容



# exports导出

- **注意**：exports是一个对象，我们可以在这个对象中添加很多个属性，添加的属性会导出；

```
exports.name = name;  
...  
exports.age = age;  
exports.sayHello = sayHello;
```

- **另外一个文件中可以导入**：

```
const bar = require('./bar');
```

- 上面这行完成了什么操作呢？理解下面这句话，Node中的模块化一目了然

- 意味着main中的bar变量等于exports对象；
- 也就是require通过各种查找方式，最终找到了exports这个对象；
- 并且将这个exports对象赋值给了bar变量；
- bar变量就是exports对象了；



# module.exports

■ 但是Node中我们经常导出东西的时候，又是通过module.exports导出的：

□ module.exports和exports有什么关系或者区别呢？

■ 我们追根溯源，通过维基百科中对CommonJS规范的解析：

□ CommonJS中是没有module.exports的概念的；

□ 但是为了实现模块的导出，Node中使用的是Module的类，每一个模块都是Module的一个实例，也就是module；

□ 所以在Node中真正用于导出的其实根本不是exports，而是module.exports；

□ 因为module才是导出的真正实现者；

■ 但是，为什么exports也可以导出呢？

□ 这是因为module对象的exports属性是exports对象的一个引用；

□ 也就是说 `module.exports = exports = main中的bar`；

# 改变代码发生了什么？

■ 我们这里从几个方面来研究修改代码发生了什么？

□ 1.在三者项目引用的情况下，修改exports中的name属性到底发生了什么？

□ 2.在三者引用的情况下，修改了main中的bar的name属性，在bar模块中会发生什么？

□ 3.如果module.exports不再引用exports对象了，那么修改export还有意义吗？

■ 先画出在内存中发生了什么，再得出最后的结论。



# require细节

- 我们现在已经知道，**require是一个函数**，可以帮助我们引入一个文件（模块）中导出的对象。
- 那么，require的查找规则是怎么样的呢？
  - [https://nodejs.org/dist/latest-v14.x/docs/api/modules.html#modules\\_all\\_together](https://nodejs.org/dist/latest-v14.x/docs/api/modules.html#modules_all_together)
- 这里我总结比较常见的查找规则：导入格式如下：require(X)
- 情况一：X是一个Node核心模块，比如path、http
  - 直接返回核心模块，并且停止查找

# 情况二：

- 情况二：X是以 ./ 或 ../ 或 /（根目录）开头的
- 第一步：将X当做一个文件在对应的目录下查找；
  - 1.如果有后缀名，按照后缀名的格式查找对应的文件
  - 2.如果没有后缀名，会按照如下顺序：
    - ✓ 1> 直接查找文件X
    - ✓ 2> 查找X.js文件
    - ✓ 3> 查找X.json文件
    - ✓ 4> 查找X.node文件
- 第二步：没有找到对应的文件，将X作为一个目录
  - 查找目录下面的index文件
    - ✓ 1> 查找X/index.js文件
    - ✓ 2> 查找X/index.json文件
    - ✓ 3> 查找X/index.node文件
- 如果没有找到，那么报错：not found



- 情况三：直接是一个X（没有路径），并且X不是一个核心模块
- /Users/coderwhy/Desktop/Node/TestCode/04\_learn\_node/05\_javascript-module/02\_commonjs/main.js中编写 `require('why')`

```
paths: [
  '/Users/coderwhy/Desktop/Node/TestCode/04_learn_node/05_javascript-module/02_commonjs/node_modules',
  '/Users/coderwhy/Desktop/Node/TestCode/04_learn_node/05_javascript-module/node_modules',
  '/Users/coderwhy/Desktop/Node/TestCode/04_learn_node/node_modules',
  '/Users/coderwhy/Desktop/Node/TestCode/node_modules',
  '/Users/coderwhy/Desktop/Node/node_modules',
  '/Users/coderwhy/Desktop/node_modules',
  '/Users/coderwhy/node_modules',
  '/Users/node_modules',
  '/node_modules'
]
```

- 如果上面的路径中都没有找到，那么报错：not found

# 模块的加载过程

■ 结论一：模块在被第一次引入时，模块中的js代码会被运行一次

■ 结论二：模块被多次引入时，会缓存，最终只加载（运行）一次

□ 为什么只会加载运行一次呢？

□ 这是因为每个模块对象module都有一个属性：loaded。

□ 为false表示还没有加载，为true表示已经加载；

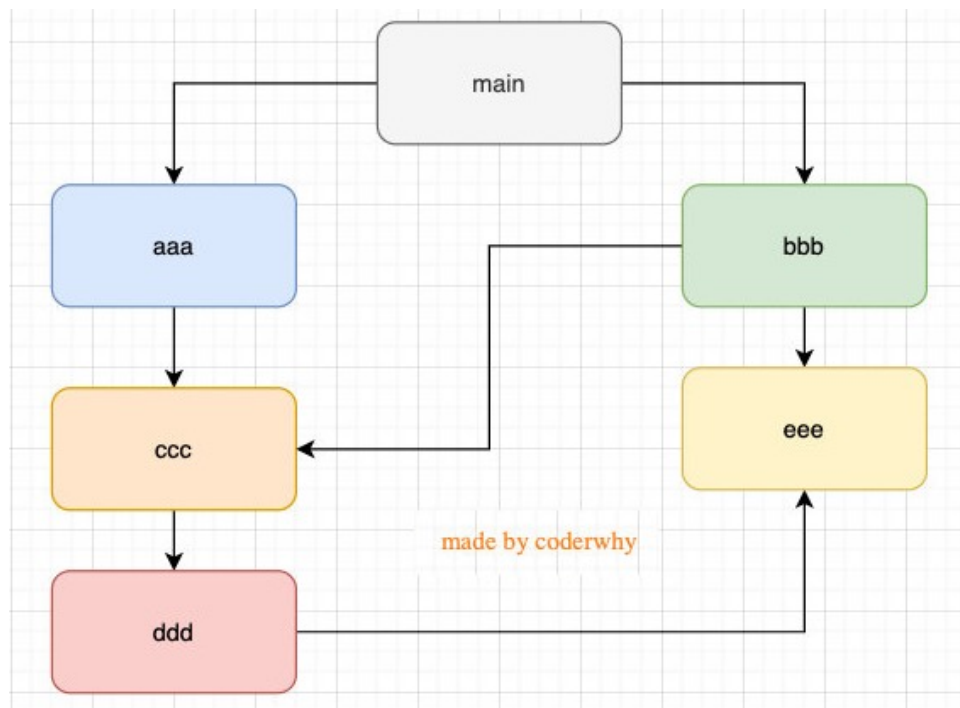
■ 结论三：如果有循环引入，那么加载顺序是什么？

■ 如果出现右图模块的引用关系，那么加载顺序是什么呢？

□ 这个其实是一种数据结构：图结构；

□ 图结构在遍历的过程中，有深度优先搜索（DFS, depth first search）和广度优先搜索（BFS, breadth first search）；

□ Node采用的是深度优先算法：main -> aaa -> ccc -> ddd -> eee -> bbb





# CommonJS规范缺点

## ■ CommonJS加载模块是同步的：

- 同步的意味着只有等到对应的模块加载完毕，当前模块中的内容才能被运行；
- 这个在服务器不会有什么问题，因为服务器加载的js文件都是本地文件，加载速度非常快；

## ■ 如果将它应用于浏览器呢？

- 浏览器加载js文件需要先从服务器将文件下载下来，之后再加载运行；
- 那么采用同步的就意味着后续的js代码都无法正常运行，即使是一些简单的DOM操作；

## ■ 所以在浏览器中，我们通常不使用CommonJS规范：

- 当然在webpack中使用CommonJS是另外一回事；
- 因为它会将我们的代码转成浏览器可以直接执行的代码；

## ■ 在早期为了可以在浏览器中使用模块化，通常会采用AMD或CMD：

- 但是目前一方面现代的浏览器已经支持ES Modules，另一方面借助于webpack等工具可以实现对CommonJS或者ES Module代码的转换；
- AMD和CMD已经使用非常少了，所以这里我们进行简单的演练；

## ■ AMD主要是应用于浏览器的一种模块化规范：

- AMD是Asynchronous Module Definition（异步模块定义）的缩写；
- 它采用的是异步加载模块；
- 事实上AMD的规范还要早于CommonJS，但是CommonJS目前依然在被使用，而AMD使用的较少了；

## ■ 我们提到过，规范只是定义代码的应该如何去编写，只有有了具体的实现才能被应用：

- AMD实现的比较常用的库是require.js和curl.js；

# require.js的使用

## ■ 第一步：下载require.js

- 下载地址：<https://github.com/requirejs/requirejs>
- 找到其中的require.js文件；

## ■ 第二步：定义HTML的script标签引入require.js和定义入口文件：

- data-main属性的作用是在加载完src的文件后会加载执行该文件

`<script src="./lib/require.js" data-main="./index.js"></script>`

```
(function() {  
    require.config({  
        baseUrl: '',  
        paths: {  
            foo: './modules/foo',  
            bar: './modules/bar'  
        }  
    })  
  
    require(['foo'], function(foo) {  
  
    })  
})();
```

```
define(function() {  
    const name = "coderwhy";  
    const age = 18;  
    const sayHello = function(name) {  
        console.log("Hello " + name);  
    }  
  
    return {  
        name,  
        age,  
        sayHello  
    }  
});
```

```
define(['bar'], function(bar) {  
    console.log(bar.name);  
    console.log(bar.age);  
    bar.sayHello('kobe');  
});
```

- CMD规范也是应用于浏览器的一种模块化规范：
  - CMD 是Common Module Definition ( 通用模块定义 ) 的缩写；
  - 它也采用了异步加载模块，但是它将CommonJS的优点吸收了过来；
  - 但是目前CMD使用也非常少了；
- CMD也有自己比较优秀的实现方案：
  - SeaJS

# SeaJS的使用

## ■ 第一步：下载SeaJS

- 下载地址：<https://github.com/seajs/seajs>
- 找到dist文件夹下的sea.js

## ■ 第二步：引入sea.js和使用主入口文件

- sea.js是指定主入口文件的

```
<script src="./lib/sea.js"></script>
<script>
  seajs.use('./index.js');
</script>
```

```
define(function(require, exports, module) {
  const foo = require('./modules/foo');
})
```

```
define(function(require, exports, module) {
  const bar = require('./bar');

  console.log(bar.name);
  console.log(bar.age);
  bar.sayHello("韩梅梅");
})
```

```
define(function(require, exports, module) {
  const name = 'lilei';
  const age = 20;
  const sayHello = function(name) {
    console.log("你好" + name);
  }

  module.exports = {
    name,
    age,
    sayHello
  }
})
```



# 认识 ES Module



- JavaScript没有模块化一直是它的痛点，所以才会产生我们前面学习的社区规范：CommonJS、AMD、CMD等，所以在ES推出自己的模块化系统时，大家也是兴奋异常。
- ES Module和CommonJS的模块化有一些不同之处：
  - 一方面它使用了import和export关键字；
  - 另一方面它采用编译期的静态分析，并且也加入了动态引用的方式；
- ES Module模块采用export和import关键字来实现模块化：
  - export负责将模块内的内容导出；
  - import负责从其他模块导入内容；
- 了解：采用ES Module将自动采用严格模式：**use strict**
  - 如果你不熟悉严格模式可以简单看一下MDN上的解析；
  - [https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Strict\\_mode](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Strict_mode)

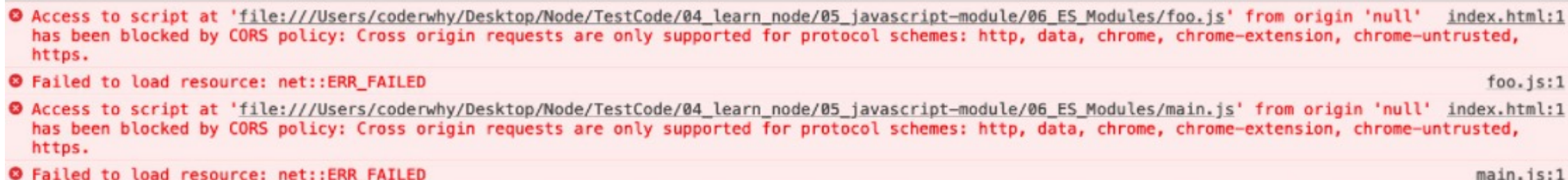


# 案例代码结构组件

- 这里我在浏览器中演示ES6的模块化开发：

```
<script src="./modules/foo.js" type="module"></script>
<script src="main.js" type="module"></script>
```

- 如果直接在浏览器中运行代码，会报如下错误：



The screenshot shows a browser console with three error messages:

- ✖ Access to script at 'file:///Users/coderwhy/Desktop/Node/TestCode/04\_learn\_node/05\_javascript-module/06\_ES\_Modules/foo.js' from origin 'null' index.html:1 has been blocked by CORS policy: Cross origin requests are only supported for protocol schemes: http, data, chrome, chrome-extension, chrome-untrusted, https.
- ✖ Failed to load resource: net::ERR\_FAILED foo.js:1
- ✖ Access to script at 'file:///Users/coderwhy/Desktop/Node/TestCode/04\_learn\_node/05\_javascript-module/06\_ES\_Modules/main.js' from origin 'null' index.html:1 has been blocked by CORS policy: Cross origin requests are only supported for protocol schemes: http, data, chrome, chrome-extension, chrome-untrusted, https.
- ✖ Failed to load resource: net::ERR\_FAILED main.js:1

- 这个在MDN上面有给出解释：

- ❑ <https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Modules>

- ❑ 你需要注意本地测试 — 如果你通过本地加载Html 文件 (比如一个 file:// 路径的文件), 你将会遇到 CORS 错误，因为 Javascript 模块安全性需要。

- ❑ 你需要通过一个服务器来测试。

- 我这里使用的VSCode，VSCode中有一个插件：Live Server



# exports关键字

- export关键字将一个模块中的变量、函数、类等导出；
- 我们希望将其他中内容全部导出，它可以有如下的方式：
  - 方式一：在语句声明的前面直接加上export关键字
  - 方式二：将所有需要导出的标识符，放到export后面的 {}中
    - 注意：这里的 {}里面不是ES6的对象字面量的增强写法，{}也不是表示一个对象的；
    - 所以： `export {name: name}`，是错误的写法；
  - 方式三：导出时给标识符起一个别名



# import关键字

- import关键字负责从另外一个模块中导入内容
- 导入内容的方式也有多种：
  - 方式一：import {标识符列表} from '模块'；
    - 注意：这里的{}也不是一个对象，里面只是存放导入的标识符列表内容；
  - 方式二：导入时给标识符起别名
  - 方式三：通过 \* 将模块功能放到一个模块功能对象（a module object）上



# export和import结合使用

## ■ 补充：export和import可以结合使用

```
export { sum as barSum } from './bar.js';
```

## ■ 为什么要这样做呢？

- 在开发和封装一个功能库时，通常我们希望将暴露的所有接口放到一个文件中；
- 这样方便指定统一的接口规范，也方便阅读；
- 这个时候，我们就可以使用export和import结合使用；



# default用法

■ 前面我们学习的导出功能都是有名字的导出（named exports）：

- 在导出export时指定了名字；
- 在导入import时需要知道具体的名字；

■ 还有一种导出叫做默认导出（default export）

- 默认导出export时不需要指定名字；
- 在导入时不需要使用 {}，并且可以自己来指定名字；
- 它也方便我们和现有的CommonJS等规范相互操作；

■ 注意：在一个模块中，只能有一个默认导出（default export）；

# import函数

- 通过import加载一个模块，是不可以在其放到逻辑代码中的，比如：
- 为什么会出现这个情况呢？
  - 这是因为ES Module在被JS引擎解析时，就必须知道它的依赖关系；
  - 由于这个时候js代码没有任何的运行，所以无法在进行类似于if判断中根据代码的执行情况；
  - 甚至下面的这种写法也是错误的：因为我们必须到运行时能确定path的值；
- 但是某些情况下，我们确确实实希望动态的来加载某一个模块：
  - 如果根据不懂的条件，动态来选择加载模块的路径；
  - 这个时候我们需要使用 import() 函数来动态加载；

```
if (true) {  
  import sub from './modules/foo.js';  
}
```

```
let flag = true;  
if (flag) {  
  import('./modules/aaa.js').then(aaa => {  
    aaa.aaa();  
  })  
} else {  
  import('./modules/bbb.js').then(bbb => {  
    bbb.bbb();  
  })  
}
```