

node.js基础

node.js的运行

当我们安装好了node.js的环境以后，我们就可以在控制台输入 `node -v` 来查看环境了，如果显示正常则代表安装成功

安装成功以后，如果我们要运行node.js需要使用一些命令去完成

```
1 $ node 文件名.js
```

如

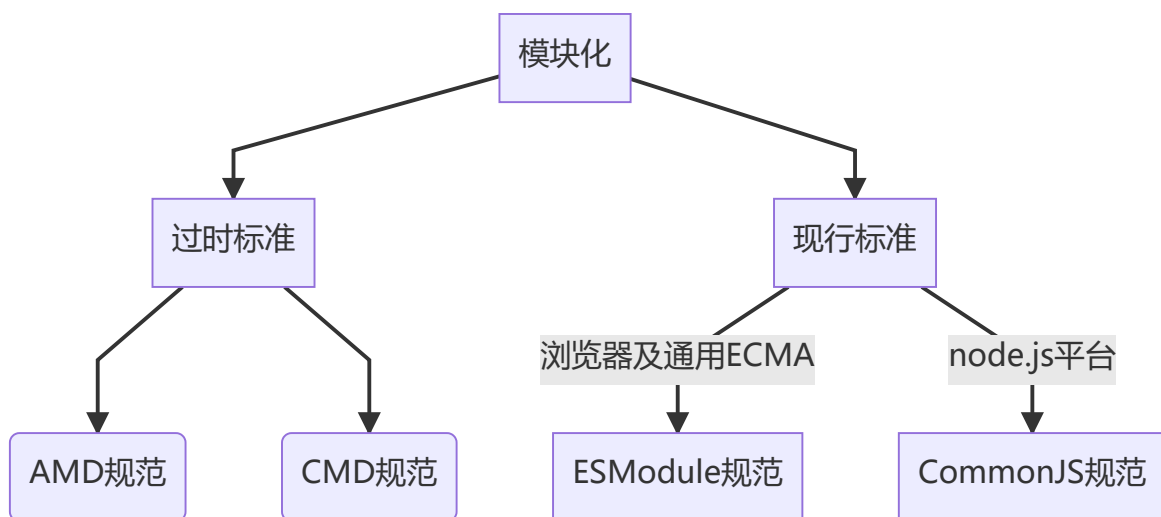
```
1 $ node 01.js
```

在node.js里面，只能执行ES的代码，不能执行DOM与BOM

```
1 console.log("hello node.js");
2 console.log("标哥哥");
3 window.alert("hello 标哥哥");           //报错，因为node.js没有包含BOM
                                           里面的
```

CommonJs模块化

在之前的ES6里面，我们讲模块化的时候，提到过模块的分类应用



在现行的模块化标准里面是有2个标准的，分为ESModule及CommonJS，ESModule主要针对的是ECMA及浏览器，而CommonJS主要针对的就j node.js

1. 在ESModule里面，我们使用 `export` 导出，使用 `import` 导入
2. 在nodejs里面，我们使用 `module.exports` 导出，使用 `require` 导入

| | 导出 | 导入 |
|-------------|----------------|--------------|
| ESModule 规范 | export 关键字 | import 关键字 |
| CommonJS 规范 | module.exports | require() 方法 |

现在我们先看下面的代码

Person.js

```
1 class Person{
2   constructor(userName){
3     this.userName = userName;
4   }
5   sayHello(){
6     console.log(`我叫${this.userName}`);
7   }
8 }
9
10
11 // export default Person;
12 module.exports = Person;
```

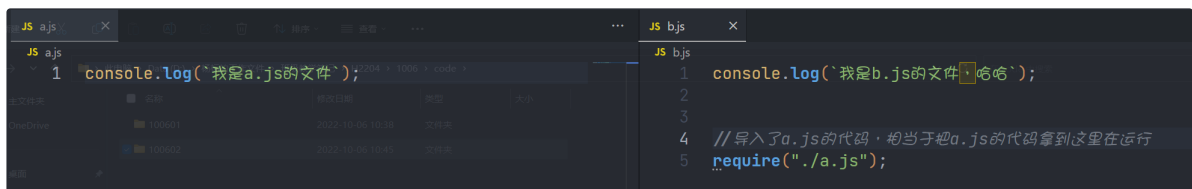
Student.js

```
1 // import Person from "./Person.js";
2 const Person = require("./Person.js");
3
4
5 class Student extends Person{
6   constructor(userName,sex){
7     super(userName);
8     this.sex = sex;
9   }
10 }
11
12
13 let s1 = new Student("张珊","女");
14 s1.sayHello();
```

当我们去执行 `node Student.js` 的时候，就不会报错了，因为我们已经改用 **CommonJS** 的规范来导出与导入

CommonJS的模块导入

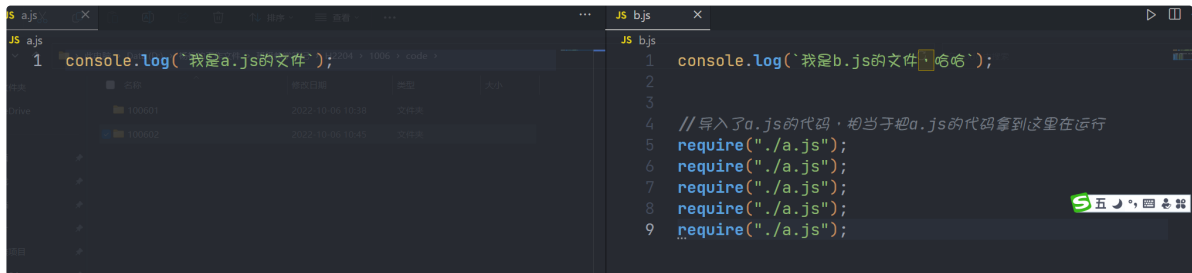
在上一个章节我们已经了解了基本的 **CommonJS** 的导出与导入，我们也知道了如果要导入一个JS文件我们可以使用 `require()` 方法，现在仔细看一下它的使用



```
JS a.js
1 console.log('我是a.js的文件');
```

```
JS b.js
1 console.log('我是b.js的文件');
2
3
4 // 导入了a.js的代码，相当于把a.js的代码拿到这里来运行
5 require("./a.js");
```

require其实就相当于把一个JS文件拿到另一个JS文件里面去执行



```
JS a.js
1 console.log('我是a.js的文件');
```

```
JS b.js
1 console.log('我是b.js的文件');
2
3
4 // 导入了a.js的代码，相当于把a.js的代码拿到这里来运行
5 require("./a.js");
6 require("./a.js");
7 require("./a.js");
8 require("./a.js");
9 require("./a.js");
```

同时我们发现一个特点，当我们执行多次导入的时候，最终a.js里面的代码也只会执行一次

require()方法在导入一个模块以后会把这个模块缓存下来，下次再导入的时候就直接从缓存里面拿出来使用，所以在上面的代码当中虽然我们导入了5次，但是真正导入的只有第一次导入，后面的几次导入都是直接从缓存里面在拿东西

CommonJS的缓存

如果希望一个模块被导入以后，不要有缓存，可以在一个模块的最后面添加下在面的代码



```
1 delete require.cache[module.filename];
```

```
JS a.js
1 console.log('我是a.js的文件');
```

```
JS b.js
1 console.log('我是b.js的文件');
2
3
4 // 导入了a.js的代码，相当于把a.js的代码拿到这里来运行
5 require("./a.js");
6 require("./a.js");
7 require("./a.js");
8 require("./a.js");
9 require("./a.js");
```

CommonJS模块导出

导入与导出是一对，在nodejs里面如果想导出可以使用下面的2种方式

1. module.exports直接导出
2. exports指针导出

首先我们先来看一下最基本的导入与导出



```
a.js
1 let userName = "标哥哥";
2 // 导出了userName
3 module.exports = userName;
```

```
JS b.js
1 const userName = require("./a.js");
2 console.log('我导入的是${userName}');
```

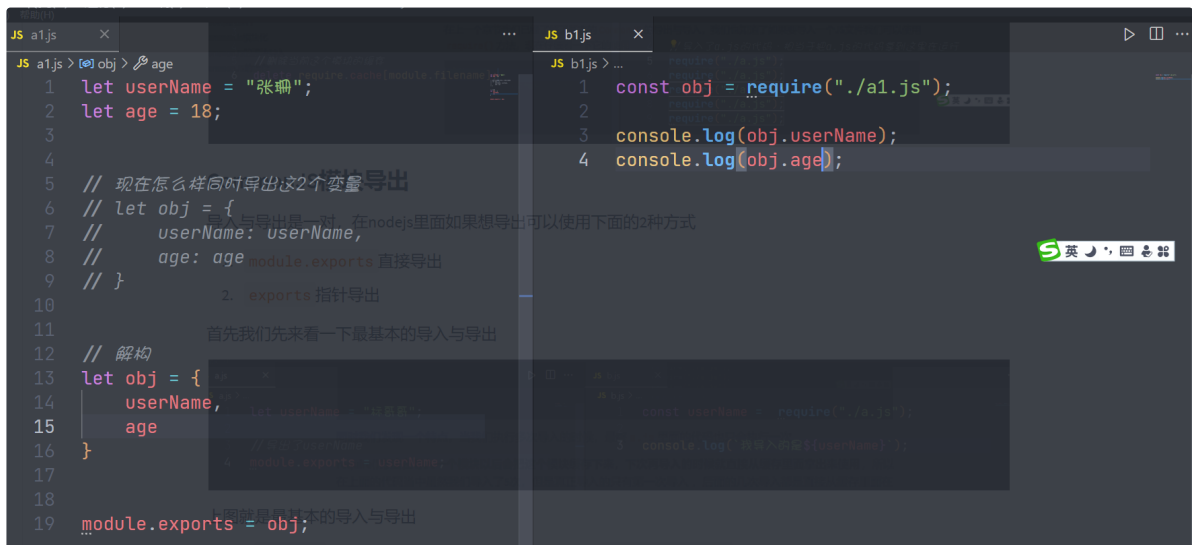
上图就是最基本的导入与导出

在每个node.js运行的文件里面，默认都会一个属性叫module.exports。这个是专门用于负责导出模块的，同时还有一个exports指向了module.exports

当前先不考虑exports

module.exports

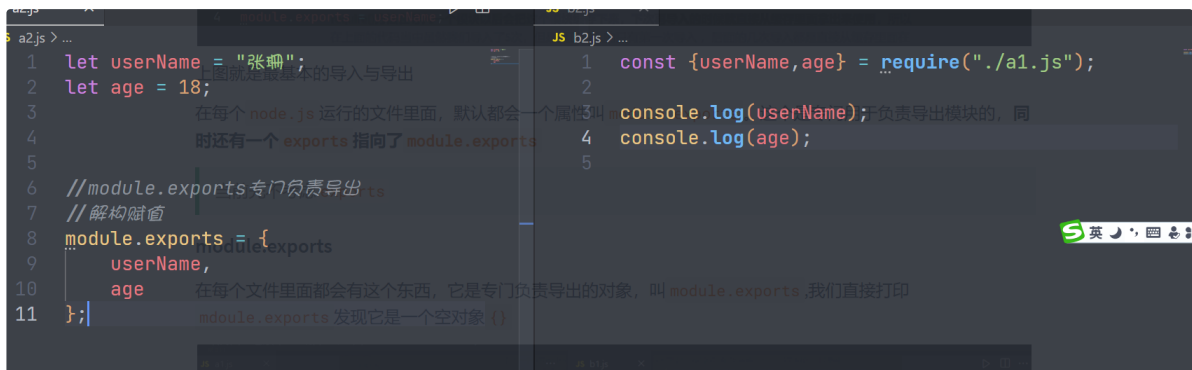
在每个文件里面都会有这个东西，它是专门负责导出的对象，叫 `module.exports`，我们直接打印 `module.exports` 发现它是一个空对象 {}



```
JS a1.js x
1 let userName = "张珊";
2 let age = 18;
3
4
5 // 现在怎么样同时导出这2个变量
6 // let obj = {
7 //   userName: userName,
8 //   age: age
9 // }
10
11 // 解构
12 let obj = {
13   userName,
14   age
15 }
16
17
18
19 module.exports = obj;
```

```
JS b1.js x
1 const obj = require("./a1.js");
2
3 console.log(obj.userName);
4 console.log(obj.age);
```

如果我们需要导出多个变量，我们可以把它封装成对象，然后再导出。但是这么做还不够简洁，我们进一步简化



```
a2.js x
1 let userName = "张珊";
2 let age = 18;
3
4 // module.exports 专门负责导出
5 // 解构赋值
6 module.exports = {
7   userName,
8   age
9 };
10
11
```

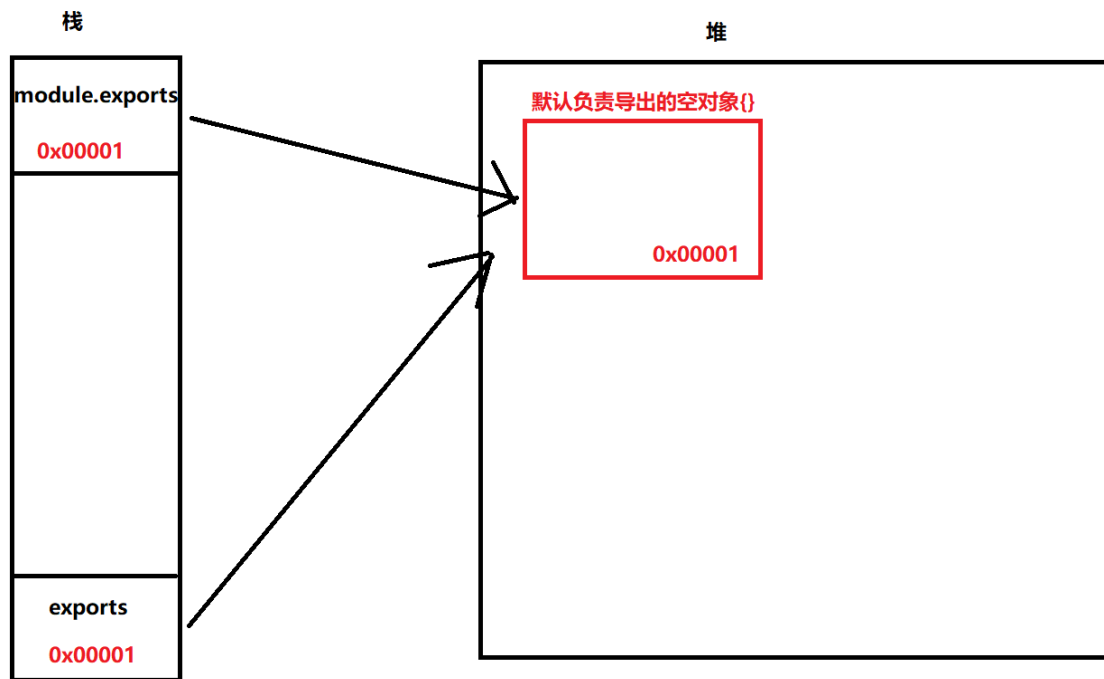
```
JS b2.js x
1 const {userName, age} = require("./a1.js");
2
3 console.log(userName);
4 console.log(age);
5
```

如果要导出多个变量，我们可以使用解构的赋值与取值的方式快速的完成

exports

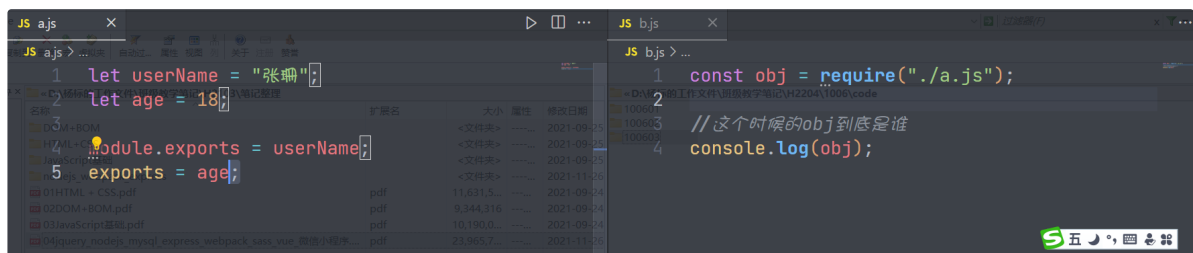
在CommonJS的模块化规范里面，负责导出的只有一个叫 `module.exports`，但是还有一个默认指向了 `module.exports`，这个就是 `exports`

我们现在通过内存图来看下看



```
1 console.log(module.exports === exports);
```

场景一

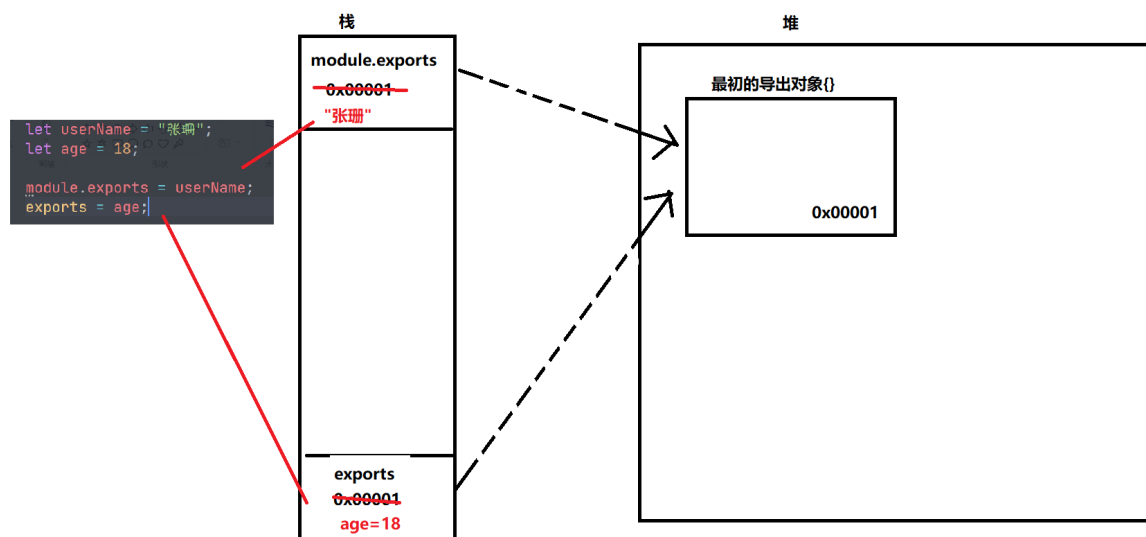


在上面的代码里面，我们通过 `module.exports` 导出，又通过 `exports` 导出，请问 `b.js` 导入的 `obj` 到底是谁？

为了弄清楚上面的东西，我们一定要谨记几个点

1. 真正负责导出的是 `module.exports`
2. `exports` 指向了最初的 `module.exports` 的堆里面的地址

现在我们根据上面的代码来画一个内存图



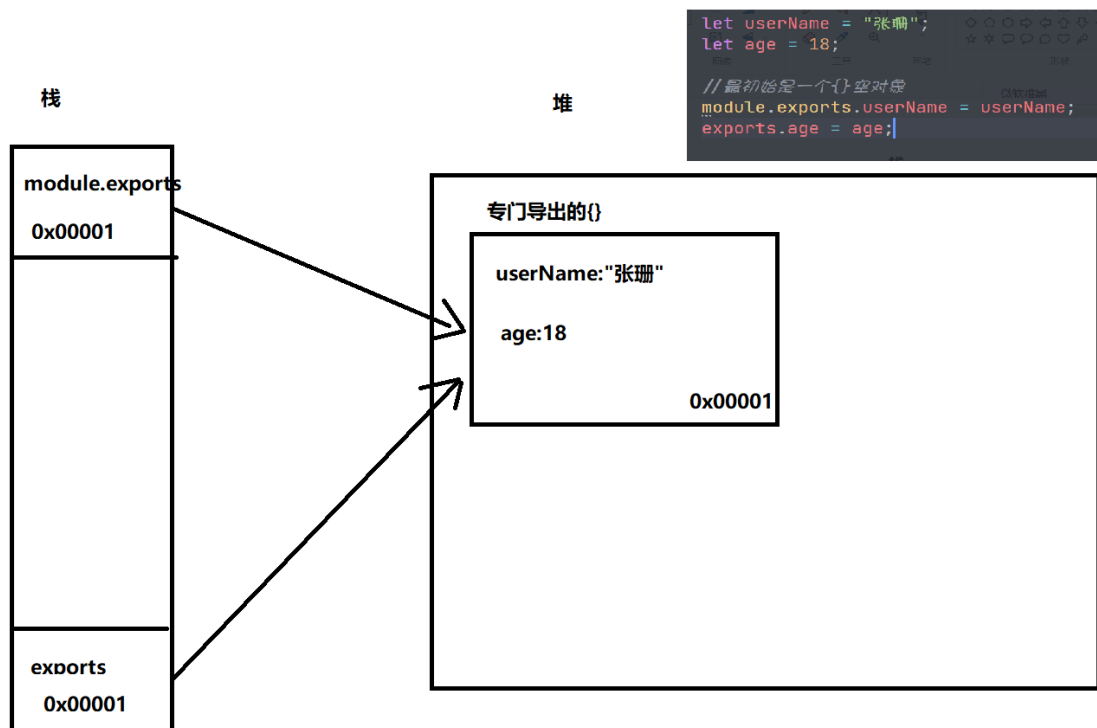
始终记得一句话，直接负责导出的是 `module.exports`，所以 `b.js` 导入的其实就是 `userName` 的值，结果就是“张珊”

场景二

```

a.js
1 let userName = "张珊";
2 let age = 18;
3
4 // 最初是一个 {} 空对象
5 module.exports.userName = userName;
6 exports.age = age;

JS b.js
1 const obj = require("./a.js");
2
3 // 这个时候的obj是谁
4 console.log(obj);
  
```



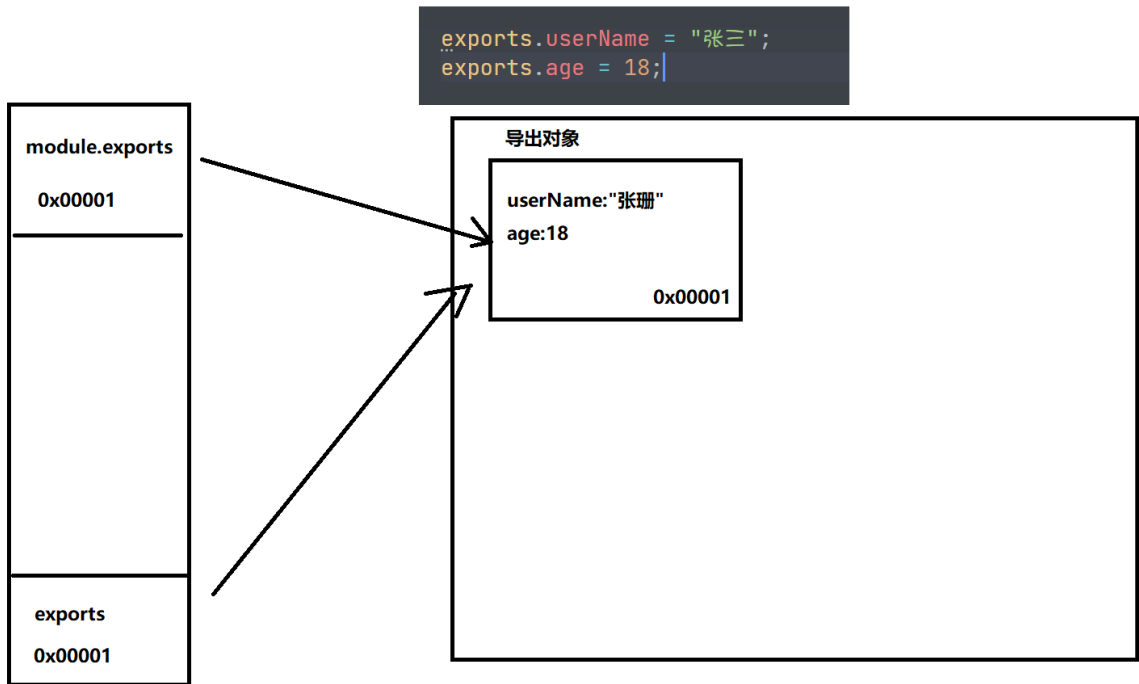
真正负责导出的仍然是 `module.exports`，这个对象被修改了2次，第一次是通过 `module.exports.userName` 来修改的，第二次是通过 `exports.age` 来修改的

所以导入的 `obj` 结果就是有2个属性

场景三

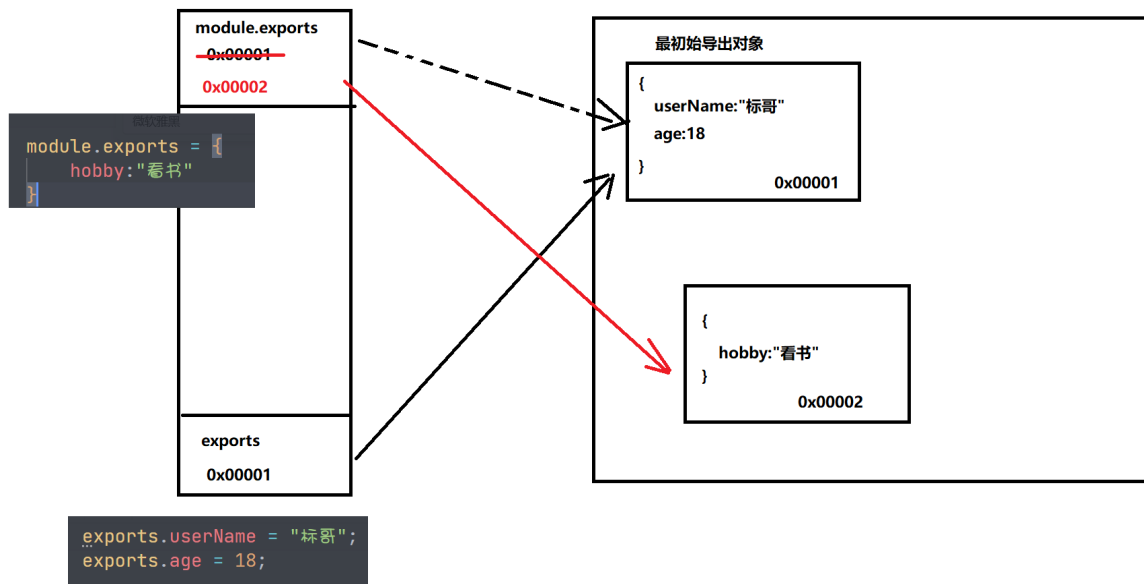
```
JS a1.js x 大纲 JS b1.js x
JS a1.js > ... 真正负责导出的仍然是 module.exports，这个对象被修改了2次
1 console.log(module.exports === exports);
2
3 exports.userName = "张三";
4 exports.age = 18;
5
JS b1.js > (e) obj
1 const obj = require("./a1.js");
2
3 //obj又是谁
4 console.log(obj);

exports.userName = "张三";
exports.age = 18;
```



场景四

```
a2.js x JS b2.js x
a2.js > (e) <unknown> hobby JS b2.js > ...
1 //最初的module.exports和exports指向了同一个对象
2
3
4 exports.userName = "标哥";
5 exports.age = 18;
6
7
8 module.exports = {
9   hobby: "看书"
10 }
11
1 const obj = require("./a2.js");
2
3
4 //obj是谁
5 console.log(obj);
```



真正负责导出的是 `module.exports`，所以最终得到的结果是 `hobby:看书`

为了避免后期开发的时候有歧意，我在后面的代码当中优先使用 `module.exports`