2019-1-17

|  |
| --- |
| Web班教学 | 李创 |



|  |  |
| --- | --- |
| 前端开发 | JS模块化知识 |

目录

[模块化演变 2](#_Toc536870913)

[一、 基础写法 3](#_Toc536870914)

[二、 对象写法 3](#_Toc536870915)

[三、立即执行函数写法 4](#_Toc536870916)

[四、放大模式写法 4](#_Toc536870917)

[五、宽放大模式 4](#_Toc536870918)

[六、输入全局变量写法 5](#_Toc536870919)

[七、 模块化规范 6](#_Toc536870920)

[八、 CommonJS 6](#_Toc536870921)

[九、 AMD 6](#_Toc536870922)

[es6中的模块化 8](#_Toc536870923)

[export的用法 8](#_Toc536870924)

[默认导出（default export） 10](#_Toc536870925)

[重命名export和import 10](#_Toc536870926)

# 模块化演变

https://baijiahao.baidu.com/s?id=1573985394423490&wfr=spider&for=pc

目前，通行的js模块规范有：

CommonJS（nodeJS）和AMD（requireJS）和CMD（seaJS）和ES6模块化。

AMD是"Asynchronous Module Definition"的缩写，意思就是"异步模块定义"。它是一个在浏览器端模块化开发的规范，服务器端的规范是CommonJS模块将被异步加载，模块加载不影响后面语句的运行。所有依赖某些模块的语句均放置在回调函数中。

随着应用的规模越来越庞大，嵌入网页的Js代码越来越庞大，越来越复杂。如果任由其发展，那么就会像下图所示。后期增加一个功能，对应用来的维护性来说将是指数级别的递增。



程序开发者不得不使用软件工程的方法，管理网页的业务逻辑。Js的模块化，封装化已经成为了一个迫切的需求。

## 一、 基础写法

封装是一种思想，它把特定的功能封装成一组方法。

只要把不同的函数以及变量简单地放在一起，就算是一个封装了的功能。

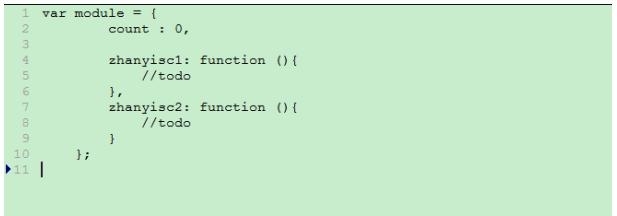


上面的函数zhanyisc1()和zhanyisc2()，组成一个模块。使用的时候，直接调用就行了。

这种做法的**缺点：**很明显："污染"了全局变量，无法保证不与其他模块发生变量名冲突，而且模块成员之间看不出直接关系。

## 二、 对象写法

为了解决上面的缺点，可以把模块写成一个对象，所有的模块成员都封装放到这个对象里面。



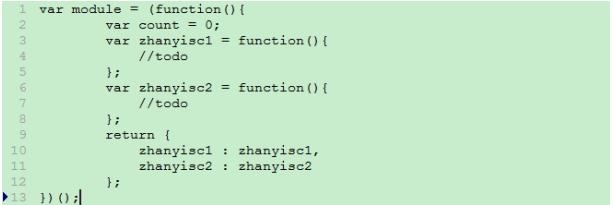
上面的函数zhanyisc1()和zhanyisc2()，都封装在module对象里。使用的时候，就是调用这个对象的属性。

例如调用函数zhanyisc1的写法：module.zhanyisc1()

**缺点：**这样的写法会暴露所有模块成员，内部状态可以被外部改写。比如，外部代码可以直接改变内部计数器的值。

## 三、立即执行函数写法

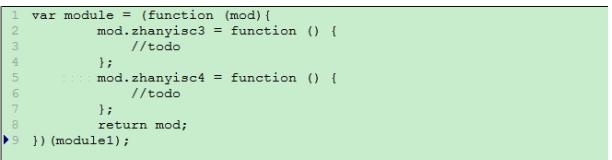
使用"立即执行函数"（Immediately-Invoked Function Expression，IIFE），可以达到不暴露私有成员的目的。



**缺点：**上面的写法无法直接从外部获取count变量，上面的封装就是js的常用封装方法。

## 四、放大模式写法

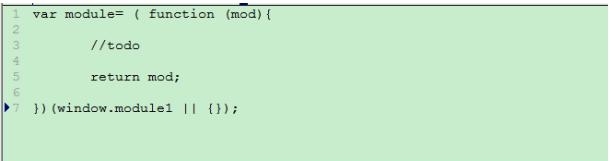
如果一个模块很大，必须分成几个部分，或者一个模块需要继承另一个模块，这时就有必要采用"放大模式"（augmentation）。



上面的代码为module模块添加了一个新方法zhanyisc3()，然后返回新的module模块。

## 五、宽放大模式

在浏览器环境中，模块的各个部分通常都是从网上获取的，有时无法知道哪个部分会先加载。如果采用上一节的写法，第一个执行的部分有可能加载一个不存在空对象，这时就要采用"宽放大模式"。

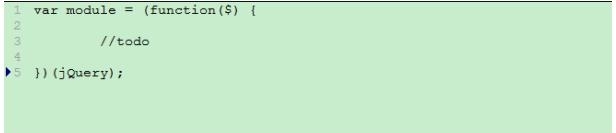


与"放大模式"相比，＂宽放大模式＂就是"立即执行函数"的参数可以是空对象。

## 六、输入全局变量写法

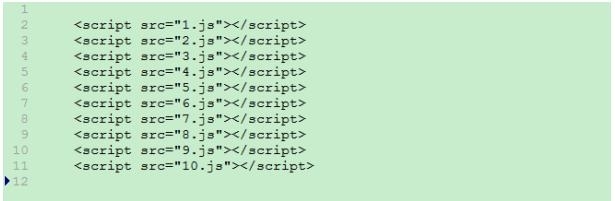
独立性是模块的重要特点，模块内部最好不与程序的其他部分直接交互。

为了在模块内部调用全局变量，必须显式地将其他变量输入模块。



上面的module模块需要使用jQuery库，就把该库当作参数输入module。这样做除了保证模块的独立性，还使得模块之间的依赖关系变得明显。上面介绍的是一个js文件内部的封装过程。

但是如果一个应用中一个页面需要加载很多个js文件，常见的写法如下：



这样的写法有很大的缺点。首先，加载的时候，浏览器会停止网页渲染，加载文件越多，网页失去响应的时间就会越长；其次，由于js文件之间存在依赖关系，因此必须严格保证加载顺序（比如上例的1.js要在2.js的前面），依赖性最大的模块一定要放到最后加载，当依赖关系很复杂的时候，代码的编写和维护都会变得困难。

有没有办法来解决上面所提到的问题呢？答案是有的，这就是多个js协作工作的问题，请接着往下看。

## 七、 模块化规范

俗话说没有规矩不成方圆，模块化没有规范，每个人都按照自己的想法来写，对别人理解起来就很费时费力，不利于沟通传播和学习。因此js就制定了这样一个规范，用于模块的加载，别人想要什么，按照规范来加载使用就好了。

目前，通行的js模块规范有CommonJS和AMD。下面就来说说这两种加载方式。

## 八、 CommonJS

在CommonJS规范中，有一个全局性方法require()，用于加载模块。假定有一个模块hello.js，就可以像下面这样加载。

上面的伪代码中，必须等hello.js加载完成。才能调用该模块的方法，也就是说，如果加载时间很长，整个应用就会停在那里等。

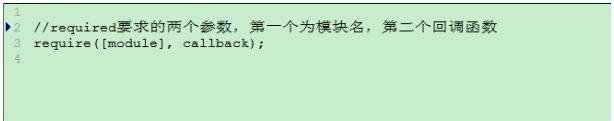
这个重大的局限，使得CommonJS规范不适用于浏览器环境，如果网速很慢，加载js的等待时间过长，浏览器处于"假死"状态。

因此，浏览器端的模块，不能采用"同步加载"（synchronous），只能采用"异步加载"（asynchronous）。这也是AMD规范诞生的背景。

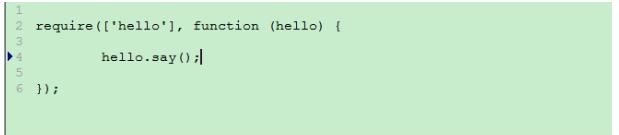
## 九、 AMD

AMD是"Asynchronous Module Definition"的缩写，意思就是"异步模块定义"。它采用异步方式加载模块，模块的加载不影响它后面语句的运行。所有依赖这个模块的语句，都定义在一个回调函数中，等到加载完成之后，这个回调函数才会运行。

AMD也采用require()语句加载模块，但是不同于CommonJS，它要求两个参数如下所示：



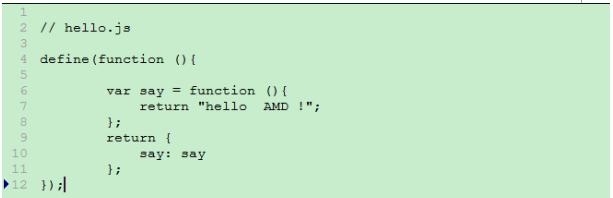
将上面的方法改为AMD方式异步加载写法如下：



它更适合浏览器环境。

上面提到了AMD加载其他模块，那么所谓的其他模块如hello.js这样的独立的js文件在AMD规范中该如何写呢？

这里也要使用定义的规范函数来定义它就是define()函数。如下hello.js的定义为：



如果这个hello.js模块还依赖其他模块，那么define()函数的第一个参数，必须是一个数组，指明该模块的依赖性。



至此一步一步封装js到这里就告一段落啦。喜欢的小伙伴，记得点个关注哦！

<head>

<script src="./requrie.min.2.3.6.js"></script>

<script src="./a.js"></script>

</head>

a.js

require(["./b"],function(){

alert('a');

});

b.js

define(['c.js'],function(c){

function fun1(){

alert(c);

}

fun1();

})

c.js

define(function(){

var a=6;

return a;

})

# 十、UMD模块打包

按理说，分别讲完非模块化、AMD、CommonJS的打包后，并没有必要再专门引入一篇讲UMD打包的。但因为UMD的实现中依赖一些特殊的变量，因此还是提一下。

UMD的模块定义：

(function (root, factory) {

if (typeof define === 'function' && define.amd) {

// AMD. Register as an anonymous module.

define(['b'], factory);

} else if (typeof module === 'object' && module.exports) {

// Node. Does not work with strict CommonJS, but

// only CommonJS-like environments that support module.exports,

// like Node.

module.exports = factory(require('b'));

} else {

// Browser globals (root is window)

root.returnExports = factory(root.b);

}

}(this, function (b) {

//use b in some fashion.

// Just return a value to define the module export.

// This example returns an object, but the module

// can return a function as the exported value.

return {};

}));

https://webpack.toobug.net/zh-cn/chapter2/umd.html

# es6中的模块化

在之前的javascript中是没有模块化概念的。如果要进行模块化操作，需要引入第三方的类库。随着技术的发展，前后端分离，前端的业务变的越来越复杂化。直至ES6带来了模块化，才让javascript第一次支持了module。ES6的模块化分为导出（export）与导入（import）两个模块。

作者：张培跃  
链接：https://www.jianshu.com/p/9e5f39e4792b  
來源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

## export的用法

在ES6中每一个模块即是一个文件，在文件中定义的变量，函数，对象在外部是无法获取的。如果你希望外部可以读取模块当中的内容，就必须使用export来对其进行暴露（输出）。先来看个例子，来对一个变量进行模块化。我们先来创建一个test.js文件，来对这一个变量进行输出：

export let myName="laowang";

然后可以创建一个index.js文件，以import的形式将这个变量进行引入:

import {myName} from "./test.js";

console.log(myName);//laowang

如果要输出多个变量可以将这些变量包装成对象进行模块化输出：

let myName="laowang";

let myAge=90;

let myfn=function(){

return "我是"+myName+"！今年"+myAge+"岁了"

}

export {

myName,

myAge,

myfn

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*接收的代码调整为\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import {myfn,myAge,myName} from "./test.js";

console.log(myfn());//我是laowang！今年90岁了

console.log(myAge);//90

console.log(myName);//laowang

如果你不想暴露模块当中的变量名字，可以通过as来进行操作:

let myName="laowang";

let myAge=90;

let myfn=function(){

return "我是"+myName+"！今年"+myAge+"岁了"

}

export {

myName as name,

myAge as age,

myfn as fn

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*接收的代码调整为\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import {fn,age,name} from "./test.js";

console.log(fn());//我是laowang！今年90岁了

console.log(age);//90

console.log(name);//laowang

也可以直接导入整个模块，将上面的接收代码修改为：

import \* as info from "./test.js";//通过\*来批量接收，as 来指定接收的名字

console.log(info.fn());//我是laowang！今年90岁了

console.log(info.age);//90

console.log(info.name);//laowang

## 默认导出（default export）

一个模块只能有一个默认导出，对于默认导出，导入的名称可以和导出的名称不一致。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*导出\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

export default function(){

return "默认导出一个方法"

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*引入\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import myFn from "./test.js";//注意这里默认导出不需要用{}。

console.log(myFn());//默认导出一个方法

可以将所有需要导出的变量放入一个对象中，然后通过export default进行导出

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*导出\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

export default {

myFn(){

return "默认导出一个方法"

},

myName:"laowang"

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*引入\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import myObj from "./test.js";

console.log(myObj.myFn(),myObj.myName);//默认导出一个方法 laowang

同样也支持混合导出

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*导出\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

export default function(){

return "默认导出一个方法"

}

export var myName="laowang";

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*引入\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import myFn,{myName} from "./test.js";

console.log(myFn(),myName);//默认导出一个方法 laowang

## 重命名export和import

如果导入的多个文件中，变量名字相同，即会产生命名冲突的问题，为了解决该问题，ES6为提供了重命名的方法，当你在导入名称时可以这样做：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*test1.js\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

export let myName="我来自test1.js";

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*test2.js\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

export let myName="我来自test2.js";

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*index.js\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import {myName as name1} from "./test1.js";

import {myName as name2} from "./test2.js";

console.log(name1);//我来自test1.js

console.log(name2);//我来自test1.js