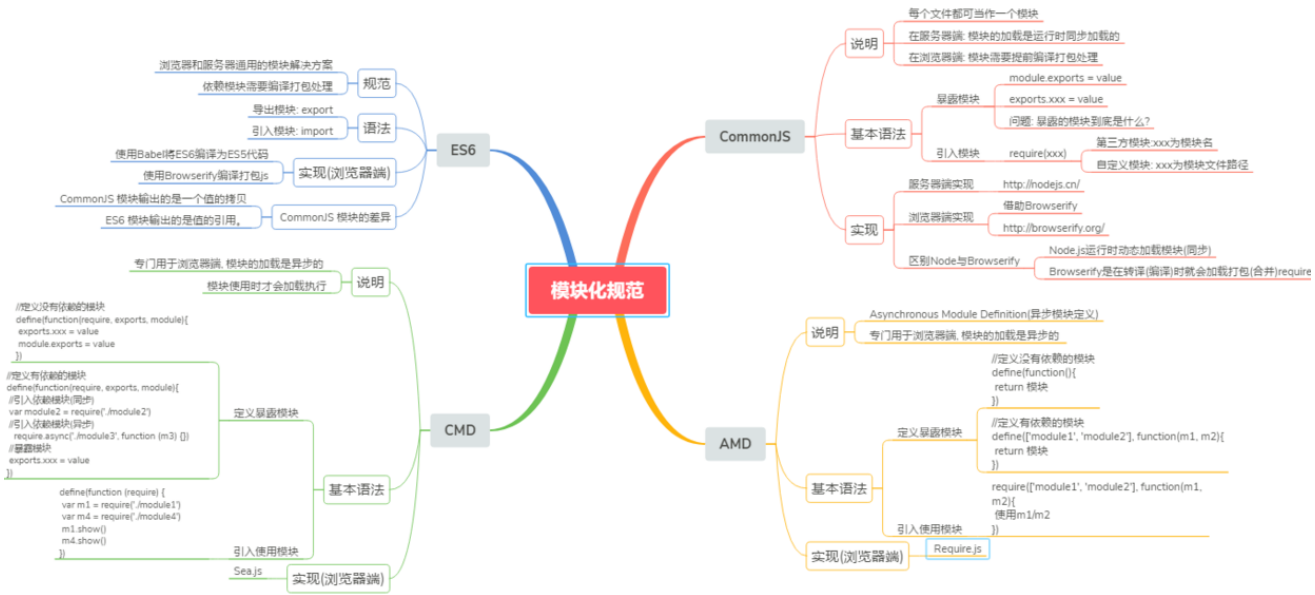
前言

在JavaScript发展初期就是为了实现简单的页面交互逻辑，寥寥数语即可；如今CPU、浏览器性能得到了极大的提升，很多页面逻辑迁移到了客户端（表单验证等），随着web2.0时代的到来，Ajax技术得到广泛应用，jQuery等前端库层出不穷，前端代码日益膨胀，此时在JS方面就会考虑使用模块化规范去管理。 本文内容主要有理解模块化，为什么要模块化，模块化的优缺点以及模块化规范,并且介绍下开发中最流行的CommonJS, AMD, ES6、CMD规范。本文试图站在小白的角度，用通俗易懂的笔调介绍这些枯燥无味的概念，希望诸君阅读后，对模块化编程有个全新的认识和理解！

建议下载本文源代码，自己动手敲一遍，请猛戳GitHub个人博客(全集)



一、模块化的理解

1.什么是模块?

- 将一个复杂的程序依据一定的规则(规范)封装成几个块(文件), 并进行组合在一起
- 块的内部数据与实现是私有的, 只是向外部暴露一些接口(方法)与外部其它模块通信

2.模块化的进化过程

- 全局function模式 : 将不同的功能封装成不同的全局函数
  - 编码: 将不同的功能封装成不同的全局函数
  - 问题: 污染全局命名空间, 容易引起命名冲突或数据不安全，而且模块成员之间看不出直接关系

```
function m1(){
  //...
}
function m2(){
  //...
}
```

- namespace模式 : 简单对象封装
  - 作用: 减少了全局变量，解决命名冲突
  - 问题: 数据不安全(外部可以直接修改模块内部的数据)

```
let myModule = {
  data: 'www.baidu.com',
  foo() {
    console.log(`foo() ${this.data}`)
  },
  bar() {
    console.log(`bar() ${this.data}`)
  }
}
myModule.data = 'other data' //能直接修改模块内部的数据
myModule.foo() // foo() other data
```

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。

- IIFE模式：匿名函数自调用(闭包)
  - 作用: 数据是私有的, 外部只能通过暴露的方法操作
  - 编码: 将数据和行为封装到一个函数内部, 通过给window添加属性来向外暴露接口
  - 问题: 如果当前这个模块依赖另一个模块怎么办?

```
// index.html文件
<script type="text/javascript" src="module.js"></script>
<script type="text/javascript">
    myModule.foo()
    myModule.bar()
    console.log(myModule.data) //undefined 不能访问模块内部数据
    myModule.data = 'xxxx' //不是修改的模块内部的data
    myModule.foo() //没有改变
</script>
// module.js文件
(function(window) {
  let data = 'www.baidu.com'
  //操作数据的函数
  function foo() {
    //用于暴露有函数
    console.log(`foo() ${data}`)
  }
  function bar() {
    //用于暴露有函数
    console.log(`bar() ${data}`)
    otherFun() //内部调用
  }
  function otherFun() {
    //内部私有的函数
    console.log('otherFun()')
  }
  //暴露行为
  window.myModule = { foo, bar } //ES6写法
```

```
})(window)
```

最后得到的结果：

```
foo() www.baidu.com
bar() www.baidu.com
otherFun()
undefined
foo() www.baidu.com
```

- **IIFE模式增强：引入依赖**

这就是现代模块实现的基石

```
// module.js文件
(function(window, $) {
  let data = 'www.baidu.com'
  //操作数据的函数
  function foo() {
    //用于暴露有函数
    console.log(`foo() ${data}`)
    $('body').css('background', 'red')
  }
  function bar() {
    //用于暴露有函数
    console.log(`bar() ${data}`)
    otherFun() //内部调用
  }
  function otherFun() {
    //内部私有的函数
    console.log('otherFun()')
  }
  //暴露行为
  window.myModule = { foo, bar }
})(window, jQuery)
 // index.html文件
  <!-- 引入的js必须有一定顺序 -->
  <script type="text/javascript" src="jquery-1.10.1.js"></script>
  <script type="text/javascript" src="module.js"></script>
  <script type="text/javascript">
    myModule.foo()
  </script>
```

上例子通过jquery方法将页面的背景颜色改成红色，所以必须先引入jQuery库，就把这个库当作参数传入。**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。**

3. 模块化的好处

- 避免命名冲突(减少命名空间污染)
- 更好的分离, 按需加载
- 更高复用性
- 高可维护性

4. 引入多个`<script>`后出现出现问题

- 请求过多

首先我们要依赖多个模块，那样就会发送多个请求，导致请求过多

- 依赖模糊

我们不知道他们的具体依赖关系是什么，也就是说很容易因为不了解他们之间的依赖关系导致加载先后顺序出错。

- 难以维护

以上两种原因就导致了很难维护，很可能出现牵一发而动全身的情况导致项目出现严重的问题。 模块化固然有多个好处，然而一个页面需要引入多个js文件，就会出现以上这些问题。而这些问题可以通过模块化规范来解决，下面介绍开发中最流行的commonjs, AMD, ES6, CMD规范。

二、模块化规范

1.CommonJS

(1)概述

Node 应用由模块组成，采用 CommonJS 模块规范。每个文件就是一个模块，有自己的作用域。在一个文件里面定义的变量、函数、类，都是私有的，对其他文件不可见。**在服务器端，模块的加载是运行时同步加载的；在浏览器端，模块需要提前编译打包处理。**

(2)特点

- 所有代码都运行在模块作用域，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。

(3)基本语法

- 暴露模块：`module.exports = value`或`exports.xxx = value`
- 引入模块：`require(xxx)`,如果是第三方模块，xxx为模块名；如果是自定义模块，xxx为模块文件路径

此处我们有个疑问：**CommonJS暴露的模块到底是什么?** 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。**

```
// example.js
var x = 5;
var 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。

```
var example = require('./example.js');//如果参数字符串以“./”开头，则表示加载的是一个位于相对路径
console.log(example.x); // 5
console.log(example.addX(1)); // 6
```

require命令用于加载模块文件。**require命令的基本功能是，读入并执行一个JavaScript文件，然后返回该模块的exports对象。如果没有发现指定模块，会报错。**

(4)模块的加载机制

CommonJS模块的加载机制是，输入的是被输出的值的拷贝。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。这点与ES6模块化有重大差异（下文会介绍），请看下面这个例子：

```
// lib.js
var 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。

```
// main.js
var counter = require('./lib').counter;
var incCounter = require('./lib').incCounter;

console.log(counter);  // 3
incCounter();
console.log(counter); // 3
```

上面代码说明，counter输出以后，lib.js模块内部的变化就影响不到counter了。**这是因为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。**

(5)服务器端实现

①下载安装node.js

②创建项目结构

注意：用npm init 自动生成package.json时，package name(包名)不能有中文和大写

```
|-modules
  |-module1.js
  |-module2.js
  |-module3.js
|-app.js
|-package.json
  {
    "name": "commonJS-node",
    "version": "1.0.0"
  }
```

③下载第三方模块

```
npm install uniq --save // 用于数组去重
```

④定义模块代码

```
//module1.js
module.exports = {
  msg: 'module1',
  foo() {
    console.log(this.msg)
  }
}
//module2.js
module.exports = function() {
  console.log('module2')
}
//module3.js
exports.foo = function() {
  console.log('foo() module3')
}
exports.arr = [1, 2, 3, 3, 2]
// app.js文件
// 引入第三方库，应该放置在最前面
let uniq = require('uniq')
let module1 = require('./modules/module1')
let module2 = require('./modules/module2')
let module3 = require('./modules/module3')

module1.foo() //module1
module2() //module2
module3.foo() //foo() module3
console.log(uniq(module3.arr)) //[ 1, 2, 3 ]
```

⑤通过node运行app.js

命令行输入`node app.js`，运行JS文件

(6)浏览器端实现(借助Browserify)

①创建项目结构

```
|-js
  |-dist //打包生成文件的目录
  |-src //源码所在的目录
    |-module1.js
    |-module2.js
    |-module3.js
    |-app.js //应用主源文件
|-index.html //运行于浏览器上
|-package.json
  {
    "name": "browserify-test",
    "version": "1.0.0"
  }
```

②下载browserify

- 全局: npm install browserify -g
- 局部: npm install browserify --save-dev

③定义模块代码(同服务器端)

注意：`index.html`文件要运行在浏览器上，需要借助browserify将`app.js`文件打包编译，如果直接在`index.html`引入`app.js`就会报错!

④打包处理js

根目录下运行`browserify js/src/app.js -o js/dist/bundle.js`

⑤页面使用引入

在index.html文件中引入`<script type="text/javascript" src="js/dist/bundle.js"></script>`

2.AMD

CommonJS规范加载模块是同步的，也就是说，只有加载完成，才能执行后面的操作。AMD规范则是非同步加载模块，允许指定回调函数。由于Node.js主要用于服务器编程，模块文件一般都已经存在于本地硬盘，所以加载起来比较快，不用考虑非同步加载的方式，所以CommonJS规范比较适用。但是，**如果是浏览器环境，要从服务器端加载模块，这时就必须采用非同步模式，因此浏览器端一般采用AMD规范。**此外AMD规范比CommonJS规范在浏览器端实现要来着早。

(1)AMD规范基本语法

定义暴露模块:

```
//定义没有依赖的模块
define(function(){
   return 模块
})
//定义有依赖的模块
define(['module1', 'module2'], function(m1, m2){
   return 模块
})
```

引入使用模块:

```
require(['module1', 'module2'], function(m1, m2){
   使用m1/m2
})
```

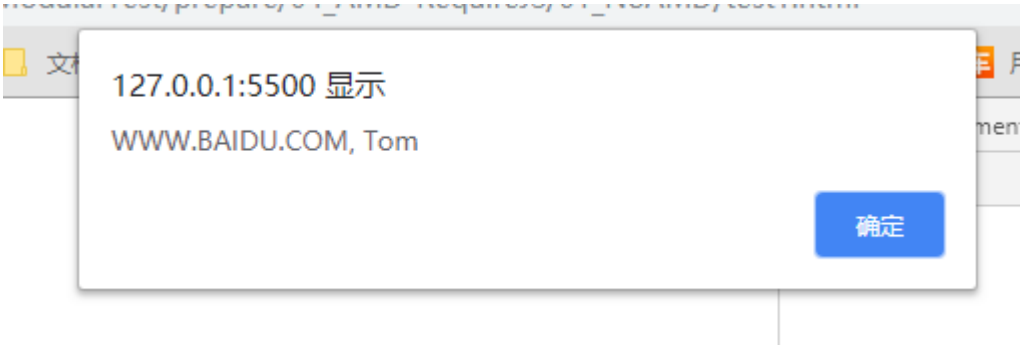
(2)未使用AMD规范与使用require.js

通过比较两者的实现方法，来说明使用AMD规范的好处。

- 未使用AMD规范

```
// dataService.js文件
(function (window) {
  let msg = 'www.baidu.com'
  function getMsg() {
    return msg.toUpperCase()
  }
  window.dataService = {getMsg}
})(window)
// alerter.js文件
(function (window, dataService) {
  let name = 'Tom'
  function showMsg() {
    alert(dataService.getMsg() + ', ' + name)
  }
  window.alerter = {showMsg}
})(window, dataService)
// main.js文件
(function (alerter) {
  alerter.showMsg()
})(alerter)
// index.html文件
<div><h1>Modular Demo 1: 未使用AMD(require.js)</h1></div>
<script type="text/javascript" src="js/modules/dataService.js"></script>
<script type="text/javascript" src="js/modules/alerter.js"></script>
<script type="text/javascript" src="js/main.js"></script>
```

最后得到如下结果：



这种方式缺点很明显：**首先会发送多个请求，其次引入的js文件顺序不能搞错，否则会报错！**

- 使用require.js

RequireJS是一个工具库，主要用于客户端的模块管理。它的模块管理遵守AMD规范，**RequireJS的基本思想是，通过define方法，将代码定义为模块；通过require方法，实现代码的模块加载。** 接下来介绍AMD规范在浏览器实现的步骤：

①下载require.js, 并引入

- 官网: `http://www.requirejs.cn/`
- github : `https://github.com/requirejs/requirejs`

然后将require.js导入项目: js/libs/require.js

②创建项目结构

```
|-js
  |-libs
    |-require.js
  |-modules
    |-alerter.js
    |-dataService.js
  |-main.js
|-index.html
```

③定义require.js的模块代码

```
// dataService.js文件
// 定义没有依赖的模块
define(function() {
  let msg = 'www.baidu.com'
  function getMsg() {
    return msg.toUpperCase()
  }
  return { getMsg } // 暴露模块
})
//alerter.js文件
// 定义有依赖的模块
define(['dataService'], function(dataService) {
  let name = 'Tom'
  function showMsg() {
    alert(dataService.getMsg() + ', ' + name)
```

```
  }
  // 暴露模块
  return { showMsg }
})
// main.js文件
(function() {
  require.config({
    baseUrl: 'js/', //基本路径 出发点在根目录下
    paths: {
      //映射: 模块标识名: 路径
      alerter: './modules/alerter', //此处不能写成alerter.js,会报错
      dataService: './modules/dataService'
    }
  })
  require(['alerter'], function(alerter) {
    alerter.showMsg()
  })
})()
// index.html文件
<!DOCTYPE html>
<html>
  <head>
    <title>Modular Demo</title>
  </head>
  <body>
    <!-- 引入require.js并指定js主文件的入口 -->
    <script data-main="js/main" src="js/libs/require.js"></script>
  </body>
</html>
```

④页面引入require.js模块:

在index.html引入 `<script data-main="js/main" src="js/libs/require.js"></script>`

此外在项目中如何引入第三方库? 只需在上面代码的基础稍作修改:

```
// alerter.js文件
define(['dataService', 'jquery'], function(dataService, $) {
  let name = 'Tom'
  function showMsg() {
    alert(dataService.getMsg() + ', ' + name)
  }
  $('body').css('background', 'green')
  // 暴露模块
  return { showMsg }
})
// main.js文件
(function() {
  require.config({
    baseUrl: 'js/', //基本路径 出发点在根目录下
    paths: {
      //自定义模块
      alerter: './modules/alerter', //此处不能写成alerter.js,会报错
```

```
      dataService: './modules/dataService',
      // 第三方库模块
      jquery: './libs/jquery-1.10.1' //注意：写成jQuery会报错
    }
  })
  require(['alerter'], function(alerter) {
    alerter.showMsg()
  })
})()
```

上例是在alerter.js文件中引入jQuery第三方库，main.js文件也要有相应的路径配置。 **小结**：通过两者的比较，可以得出**AMD模块定义的方法非常清晰，不会污染全局环境，能够清楚地显示依赖关系。**AMD模式可以用于浏览器环境，并且允许非同步加载模块，也可以根据需要动态加载模块。

3.CMD

CMD规范专门用于浏览器端，模块的加载是异步的，模块使用时才会加载执行。CMD规范整合了CommonJS和AMD规范的特点。在 Sea.js 中，所有 JavaScript 模块都遵循 CMD模块定义规范。

(1)CMD规范基本语法

定义暴露模块：

```
//定义没有依赖的模块
define(function(require, exports, module){
  exports.xxx = value
  module.exports = value
})
//定义有依赖的模块
define(function(require, exports, module){
  //引入依赖模块(同步)
  var module2 = require('./module2')
  //引入依赖模块(异步)
    require.async('./module3', function (m3) {
    })
  //暴露模块
  exports.xxx = value
})
```

引入使用模块：

```
define(function (require) {
  var m1 = require('./module1')
  var m4 = require('./module4')
  m1.show()
  m4.show()
})
```

(2)sea.js简单使用教程

①下载sea.js, 并引入

- 官网: http://seajs.org/
- github : https://github.com/seajs/seajs

然后将sea.js导入项目: js/libs/sea.js

②创建项目结构

```
|-js
  |-libs
    |-sea.js
  |-modules
    |-module1.js
    |-module2.js
    |-module3.js
    |-module4.js
    |-main.js
|-index.html
```

③定义sea.js的模块代码

```
// module1.js文件
define(function (require, exports, module) {
  //内部变量数据
  var data = 'atguigu.com'
  //内部函数
  function show() {
    console.log('module1 show() ' + data)
  }
  //向外暴露
  exports.show = show
})
// module2.js文件
define(function (require, exports, module) {
  module.exports = {
    msg: 'I Will Back'
  }
})
// module3.js文件
define(function(require, exports, module) {
  const API_KEY = 'abc123'
  exports.API_KEY = API_KEY
})
// module4.js文件
define(function (require, exports, module) {
  //引入依赖模块(同步)
  var module2 = require('./module2')
  function show() {
    console.log('module4 show() ' + module2.msg)
  }
  exports.show = show
  //引入依赖模块(异步)
  require.async('./module3', function (m3) {
```

```
    console.log('异步引入依赖模块3  ' + m3.API_KEY)
  })
})
// main.js文件
define(function (require) {
  var m1 = require('./module1')
  var m4 = require('./module4')
  m1.show()
  m4.show()
})
```

④在index.html中引入

```
<script type="text/javascript" src="js/libs/sea.js"></script>
<script type="text/javascript">
  seajs.use('./js/modules/main')
</script>
```

最后得到结果如下：

```
Live reload enabled.
module1 show() atguigu.com
module4 show() I Will Back
异步引入依赖模块3  abc123
```

4.ES6模块化

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。

(1)ES6模块化语法

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。

```
/** 定义模块 math.js **/
var basicNum = 0;
var add = function (a, b) {
    return a + b;
};
export { basicNum, add };
/** 引用模块 **/
import { basicNum, add } from './math';
function test(ele) {
    ele.textContent = add(99 + basicNum);
}
```

如上例所示，使用import命令的时候，用户需要知道所要加载的变量名或函数名，否则无法加载。为了给用户提供方便，让他们不用阅读文档就能加载模块，就要用到export default命令，为模块指定默认输出。

```
// export-default.js
export default function () {
  console.log('foo');
}
// import-default.js
import customName from './export-default';
customName(); // 'foo'
```

模块默认输出, 其他模块加载该模块时，import命令可以为该匿名函数指定任意名字。

(2)ES6 模块与 CommonJS 模块的差异

它们有两个重大差异:

① CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用。

② 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，在代码静态解析阶段就会生成。

下面重点解释第一个差异，我们还是举上面那个CommonJS模块的加载机制例子:

```
// 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
```

ES6 模块的运行机制与 CommonJS 不一样。**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。**

(3) ES6-Babel-Browserify使用教程

简单来说就一句话：**使用Babel将ES6编译为ES5代码，使用Browserify编译打包js。**

①定义package.json文件

```
{
  "name" : "es6-babel-browserify",
  "version" : "1.0.0"
}
```

②安装babel-cli, babel-preset-es2015和browserify

- npm install babel-cli browserify -g
- npm install babel-preset-es2015 --save-dev
- preset 预设(将es6转换成es5的所有插件打包)

③定义.babelrc文件

```
  {
    "presets": ["es2015"]
  }
```

④定义模块代码

```
//module1.js文件
// 分别暴露
export function foo() {
  console.log('foo() module1')
}
export function bar() {
  console.log('bar() module1')
}
//module2.js文件
// 统一暴露
function fun1() {
  console.log('fun1() module2')
}
function fun2() {
  console.log('fun2() module2')
}
export { fun1, fun2 }
//module3.js文件
// 默认暴露 可以暴露任意数据类项，暴露什么数据，接收到就是什么数据
export default () => {
  console.log('默认暴露')
}
// app.js文件
import { foo, bar } from './module1'
import { fun1, fun2 } from './module2'
import module3 from './module3'
foo()
bar()
fun1()
fun2()
module3()
```

⑤ 编译并在index.html中引入

- 使用Babel将ES6编译为ES5代码(但包含CommonJS语法)：`babel js/src -d js/lib`
- 使用Browserify编译js：`browserify js/lib/app.js -o js/lib/bundle.js`

然后在index.html文件中引入

```
  <script type="text/javascript" src="js/lib/bundle.js"></script>
```

最后得到如下结果：

```
foo() module1
bar() module1
fun1() module2
fun2() module2
默认暴露
```

此外第三方库(以jQuery为例)如何引入呢？ 首先安装依赖`npm install jquery@1`然后在app.js文件中引入

```
//app.js文件
import { foo, bar } from './module1'
import { fun1, fun2 } from './module2'
import module3 from './module3'
import $ from 'jquery'

foo()
bar()
fun1()
fun2()
module3()
$('body').css('background', 'green')
```

三、总结

- CommonJS规范主要用于服务端编程，加载模块是同步的，这并不适合在浏览器环境，因为同步意味着阻塞加载，浏览器资源是异步加载的，因此有了AMD CMD解决方案。
- AMD规范在浏览器环境中异步加载模块，而且可以并行加载多个模块。不过，AMD规范开发成本高，代码的阅读和书写比较困难，模块定义方式的语义不顺畅。
- CMD规范与AMD规范很相似，都用于浏览器编程，依赖就近，延迟执行，可以很容易在Node.js中运行。不过，依赖SPM 打包，模块的加载逻辑偏重
- **ES6 在语言标准的层面上，实现了模块功能，而且实现得相当简单，完全可以取代 CommonJS 和 AMD 规范，成为浏览器和服务器通用的模块解决方案。**

参考文章

- 前端模块化开发那点历史
- CommonJS，AMD，CMD区别
- AMD 和 CMD 的区别有哪些?
- Javascript模块化编程
- Javascript标准参考教程
- CMD 模块定义规范
- 理解CommonJS、AMD、CMD三种规范