[js模块化](http://www.cnblogs.com/Leo_wl/p/4869090.html)

[**js模块化**](http://www.cnblogs.com/lvdabao/p/js-modules-develop.html)

这是一篇关于js模块化历程的长长的流水账，记录js模块化思想的诞生与变迁，展望ES6模块化标准的未来。经历过这段历史的人或许会感到沧桑，没经历过的人也应该知道这段历史。

**无模块时代**

在ajax还未提出之前，js还只是一种“玩具语言”，由Brendan Eich花了不到十天时间发明，用来在网页上进行表单校验、实现简单的动画效果等等，你可以回想一下那个网页上到处有公告块飘来飘去的时代。

这个时候并没有前端工程师，服务端工程师只需在页面上随便写写js就能搞定需求。那个时候的前端代码大概像这样：

[复制代码](javascript:void(0);)

复制代码

if(xx){

//.......

}

else{

//xxxxxxxxxxx

}

for(var i=0; i<10; i++){

//........

}

element.onclick = function(){

//.......

}

复制代码

[复制代码](javascript:void(0);)

代码简单的堆在一起，只要能从上往下依次执行就可以了。

**模块萌芽时代**

2006年，ajax的概念被提出，前端拥有了主动向服务端发送请求并操作返回数据的能力，随着Google将此概念的发扬光大，传统的网页慢慢的向“富客户端”发展。前端的业务逻辑越来越多，代码也越来越多，于是一些问题就暴漏了出来：

**1. 全局变量的灾难**

小明定义了 i=1

小刚在后续的代码里：i=0

小明在接下来的代码里：if(i==1){...} //悲剧

**2. 函数命名冲突**

项目中通常会把一些通用的函数封装成一个文件，常见的名字有utils.js、common.js...

小明定义了一个函数：function formatData(){   }

小刚想实现类似功能，于是这么写：function formatData2(){   }

小光又有一个类似功能，于是：function formatData3(){   }

......

避免命名冲突就只能这样靠丑陋的方式人肉进行。

**3. 依赖关系不好管理**

b.js依赖a.js，标签的书写顺序必须是

<script type="text/javascript" src="a.js"></script>

<script type="text/javascript" src="b.js"></script>

顺序不能错，也不能漏写某个。在多人开发的时候很难协调。

萌芽时代的解决方案：

**1. 用自执行函数来包装代码**

[复制代码](javascript:void(0);)

复制代码

modA = function(){

var a,b; //变量a、b外部不可见

return {

add : function(c){

a + b + c;

},

format: function(){

//......

}

}

}()

复制代码

[复制代码](javascript:void(0);)

这样function内部的变量就对全局隐藏了，达到是封装的目的。但是这样还是有缺陷的，modA这个变量还是暴漏到全局了，随着模块的增多，全局变量还是会越来越多。

**2. java风格的命名空间**

为了避免全局变量造成的冲突，人们想到或许可以用多级命名空间来进行管理，于是，代码就变成了这个风格：

app.util.modA = xxx;

app.tools.modA = xxx;

app.tools.modA.format = xxx;

Yahoo的YUI早期就是这么做的，调用的时候不得不这么写：

app.tools.modA.format();

这样调用函数，写写都会觉得恶心，所以这种方式并没有被很多人采用，YUI后来也不用这种方式了。

**3. jQuery风格的匿名自执行函数**

[复制代码](javascript:void(0);)

复制代码

(function(window){

//代码

window.jQuery = window.$ = jQuery;//通过给window添加属性而暴漏到全局

})(window);

复制代码

[复制代码](javascript:void(0);)

jQuery的封装风格曾经被很多框架模仿，通过匿名函数包装代码，所依赖的外部变量传给这个函数，在函数内部可以使用这些依赖，然后在函数的最后把模块自身暴漏给window。

如果需要添加扩展，则可以作为jQuery的插件，把它挂载到$上。

这种风格虽然灵活了些，但并未解决根本问题：所需依赖还是得外部提前提供、还是增加了全局变量。

**模块化面临什么问题**

从以上的尝试中，可以归纳出js模块化需要解决那些问题：

1. 如何安全的包装一个模块的代码？（不污染模块外的任何代码）

2. 如何唯一标识一个模块？

3. 如何优雅的把模块的API暴漏出去？（不能增加全局变量）

4. 如何方便的使用所依赖的模块？

围绕着这些问题，js模块化开始了一段艰苦而曲折的征途。

**源自nodejs的规范CommonJs**

2009年，nodejs横空出世，开创了一个新纪元，人们可以用js来编写服务端的代码了。如果说浏览器端的js即便没有模块化也可以忍的话，那服务端是万万不能的。

大牛云集的CommonJs社区发力，制定了Modules/1.0（<http://wiki.commonjs.org/wiki/Modules/1.0>）规范，首次定义了一个模块应该长啥样。具体来说，Modules/1.0规范包含以下内容：

1. 模块的标识应遵循的规则（书写规范）

2. 定义全局函数require，通过传入模块标识来引入其他模块，执行的结果即为别的模块暴漏出来的API

3. 如果被require函数引入的模块中也包含依赖，那么依次加载这些依赖

4. 如果引入模块失败，那么require函数应该报一个异常

5. 模块通过变量exports来向往暴漏API，exports只能是一个对象，暴漏的API须作为此对象的属性。

此规范一出，立刻产生了良好的效果，由于其简单而直接，在nodejs中，这种模块化方案立刻被推广开了。

遵循commonjs规范的代码看起来是这样的：（来自官方的例子）

[复制代码](javascript:void(0);)

复制代码

//math.js

exports.add = function() {

var sum = 0, i = 0, args = arguments, l = args.length;

while (i < l) {

sum += args[i++];

}

return sum;

};

复制代码

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

复制代码

//increment.js

var add = require('math').add;

exports.increment = function(val) {

return add(val, 1);

};

复制代码

[复制代码](javascript:void(0);)

//program.js

var inc = require('increment').increment;

var a = 1;

inc(a); // 2

**服务端向前端进军**

Modules/1.0规范源于服务端，无法直接用于浏览器端，原因表现为：

1. 外层没有function包裹，变量全暴漏在全局。如上面例子中increment.js中的add。

2. 资源的加载方式与服务端完全不同。服务端require一个模块，直接就从硬盘或者内存中读取了，消耗的时间可以忽略。而浏览器则不同，需要从服务端来下载这个文件，然后运行里面的代码才能得到API，需要花费一个http请求，也就是说，require后面的一行代码，需要资源请求完成才能执行。由于浏览器端是以插入<script>标签的形式来加载资源的（ajax方式不行，有跨域问题），没办法让代码同步执行，所以像commonjs那样的写法会直接报错。

所以，社区意识到，要想在浏览器环境中也能模块化，需要对规范进行升级。顺便说一句，CommonJs原来是叫ServerJs，从名字可以看出是专攻服务端的，为了统一前后端而改名CommonJs。（论起名的重要性~）

而就在社区讨论制定下一版规范的时候，内部发生了比较大的分歧，分裂出了三个主张，渐渐的形成三个不同的派别：

**1.Modules/1.x派**

这一波人认为，在现有基础上进行改进即可满足浏览器端的需要，既然浏览器端需要function包装，需要异步加载，那么新增一个方案，能把现有模块转化为适合浏览器端的就行了，有点像“保皇派”。基于这个主张，制定了Modules/Transport（<http://wiki.commonjs.org/wiki/Modules/Transport>）规范，提出了先通过工具把现有模块转化为复合浏览器上使用的模块，然后再使用的方案。

browserify就是这样一个工具，可以把nodejs的模块编译成浏览器可用的模块。（Modules/Transport规范晦涩难懂，我也不确定browserify跟它是何关联，有知道的朋友可以讲一下）

目前的最新版是Modules/1.1.1（<http://wiki.commonjs.org/wiki/Modules/1.1.1>），增加了一些require的属性，以及模块内增加module变量来描述模块信息，变动不大。

**2. Modules/Async派**

这一波人有点像“革新派”，他们认为浏览器与服务器环境差别太大，不能沿用旧的模块标准。既然浏览器必须异步加载代码，那么模块在定义的时候就必须指明所依赖的模块，然后把本模块的代码写在回调函数里。模块的加载也是通过下载-回调这样的过程来进行，这个思想就是AMD的基础，由于“革新派”与“保皇派”的思想无法达成一致，最终从CommonJs中分裂了出去，独立制定了浏览器端的js模块化规范AMD（Asynchronous Module Definition）（<https://github.com/amdjs/amdjs-api/wiki/AMD>）

本文后续会继续讨论AMD规范的内容。

**3. Modules/2.0派**

这一波人有点像“中间派”，既不想丢掉旧的规范，也不想像AMD那样推到重来。他们认为，Modules/1.0固然不适合浏览器，但它里面的一些理念还是很好的，（如通过require来声明依赖），新的规范应该兼容这些，AMD规范也有它好的地方（例如模块的预先加载以及通过return可以暴漏任意类型的数据，而不是像commonjs那样exports只能为object），也应采纳。最终他们制定了一个Modules/Wrappings（<http://wiki.commonjs.org/wiki/Modules/Wrappings>）规范，此规范指出了一个模块应该如何“包装”，包含以下内容：

1. 全局有一个module变量，用来定义模块

2. 通过module.declare方法来定义一个模块

3. module.declare方法只接收一个参数，那就是模块的factory，次factory可以是函数也可以是对象，如果是对象，那么模块输出就是此对象。

4. 模块的factory函数传入三个参数：require,exports,module，用来引入其他依赖和导出本模块API

5. 如果factory函数最后明确写有return数据（js函数中不写return默认返回undefined），那么return的内容即为模块的输出。

使用该规范的例子看起来像这样：

[复制代码](javascript:void(0);)

//可以使用exprots来对外暴漏API

module.declare(function(require, exports, module)

{

exports.foo = "bar";

});

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

//也可以直接return来对外暴漏数据

module.declare(function(require)

{

return { foo: "bar" };

});

[复制代码](javascript:void(0);)

**AMD/RequireJs的崛起与妥协**

AMD的思想正如其名，异步加载所需的模块，然后在回调函数中执行主逻辑。这正是我们在浏览器端开发所习惯了的方式，其作者亲自实现了符合AMD规范的requirejs，AMD/RequireJs迅速被广大开发者所接受。

AMD规范包含以下内容：

1. 用全局函数define来定义模块，用法为：define(id?, dependencies?, factory);

2. id为模块标识，遵从CommonJS Module Identifiers规范

3. dependencies为依赖的模块数组，在factory中需传入形参与之一一对应

4. 如果dependencies的值中有"require"、"exports"或"module"，则与commonjs中的实现保持一致

5. 如果dependencies省略不写，则默认为["require", "exports", "module"]，factory中也会默认传入require,exports,module

6. 如果factory为函数，模块对外暴漏API的方法有三种：return任意类型的数据、exports.xxx=xxx、module.exports=xxx

7. 如果factory为对象，则该对象即为模块的返回值

基于以上几点基本规范，我们便可以用这样的方式来进行模块化组织代码了：

[复制代码](javascript:void(0);)

复制代码

//a.js

define(function(){

console.log('a.js执行');

return {

hello: function(){

console.log('hello, a.js');

}

}

});

复制代码

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

复制代码

//b.js

define(function(){

console.log('b.js执行');

return {

hello: function(){

console.log('hello, b.js');

}

}

});

复制代码

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

复制代码

//main.js

require(['a', 'b'], function(a, b){

console.log('main.js执行');

a.hello();

$('#b').click(function(){

b.hello();

});

})

复制代码

[复制代码](javascript:void(0);)

上面的main.js被执行的时候，会有如下的输出：

a.js执行  
b.js执行  
main.js执行  
hello, a.js

在点击按钮后，会输出：

hello, b.js

这结局，如你所愿吗？大体来看，是没什么问题的，因为你要的两个hello方法都正确的执行了。

但是如果细细来看，b.js被预先加载并且**预先执行**了，（第二行输出），b.hello这个方法是在点击了按钮之后才会执行，如果用户压根就没点，那么b.js中的代码应不应该执行呢？

这其实也是AMD/RequireJs被吐槽的一点，预先下载没什么争议，由于浏览器的环境特点，被依赖的模块肯定要预先下载的。问题在于，是否需要预先执行？如果一个模块依赖了十个其他模块，那么在本模块的代码执行之前，要先把其他十个模块的代码都执行一遍，不管这些模块是不是马上会被用到。这个性能消耗是不容忽视的。

另一点被吐槽的是，在定义模块的时候，要把所有依赖模块都罗列一遍，而且还要在factory中作为形参传进去，要写两遍很大一串模块名称，像这样：

define(['a', 'b', 'c', 'd', 'e', 'f', 'g'], function(a, b, c, d, e, f, g){ ..... })

编码过程略有不爽。

好的一点是，AMD保留了commonjs中的require、exprots、module这三个功能（上面提到的第4条）。你也可以不把依赖罗列在dependencies数组中。而是在代码中用require来引入，如下：

[复制代码](javascript:void(0);)

复制代码

define(function(){

console.log('main2.js执行');

require(['a'], function(a){

a.hello();

});

$('#b').click(function(){

require(['b'], function(b){

b.hello();

});

});

});

复制代码

[复制代码](javascript:void(0);)

我们在define的参数中未写明依赖，那么main2.js在执行的时候，就不会预先加载a.js和b.js，只是执行到require语句的时候才会去加载，上述代码的输出如下：

main2.js执行  
a.js执行  
hello, a.js

可以看到b.js并未执行，从网络请求中看，b.js也并未被下载。只有在按钮被点击的时候b.js才会被下载执行，并且在回调函数中执行模块中的方法。这就是名副其实的“懒加载”了。

这样的懒加载无疑会大大减轻初始化时的损耗（下载和执行都被省去了），但是弊端也是显而易见的，在后续执行a.hello和b.hello时，必须得实时下载代码然后在回调中才能执行，这样的用户体验是不好的，用户的操作会有明显的延迟卡顿。

但这样的现实并非是无法接受的，毕竟是浏览器环境，我们已经习惯了操作网页时伴随的各种loading。。。

但是话说过来，有没有更好的方法来处理问题呢？资源的下载阶段还是预先进行，资源执行阶段后置，等到需要的时候再执行。这样一种折衷的方式，能够融合前面两种方式的优点，而又回避了缺点。

这就是Modules/Wrappings规范，还记得前面提到的“中间派”吗？

在AMD的阵营中，也有一部分人提出这样的观点，代码里写一堆回调实在是太恶心了，他们更喜欢这样来使用模块：

[复制代码](javascript:void(0);)

复制代码

var a = require('a');

a.hello();

$('#b').click(function(){

var b = require('b');

b.hello();

});

复制代码

[复制代码](javascript:void(0);)

于是，AMD也终于决定作妥协，兼容Modules/Wrappings的写法，但只是**部分兼容**，例如并没有使用module.declare来定义模块，而还是用define，模块的执行时机也没有改变，依旧是预先执行。因此，AMD将此兼容称为Simplified CommonJS wrapping，即并不是完整的实现Modules/Wrappings。

作了此兼容后，使用requirejs就可以这么写代码了：

[复制代码](javascript:void(0);)

复制代码

//d.js

define(function(require, exports, module){

console.log('d.js执行');

return {

helloA: function(){

var a = require('a');

a.hello();

},

run: function(){

$('#b').click(function(){

var b = require('b');

b.hello();

});

}

}

});

复制代码

[复制代码](javascript:void(0);)

注意定义模块时候的轻微差异，dependencies数组为空，但是factory函数的形参必须手工写上require,exports,module，（这不同于之前的dependencies和factory形参全不写），这样写即可使用Simplified CommonJS wrapping风格，与commonjs的格式一致了。

虽然使用上看起来简单，然而在理解上却给后人埋下了一个**大坑。**因为AMD只是支持了这样的语法，而并没有真正实现模块的延后执行。什么意思呢？上面的代码，正常来讲应该是预先下载a.js和b.js，然后在执行模块的helloA方法的时候开始执行a.js里面的代码，在点击按钮的时候开始执行b.js中的方法。实际却不是这样，只要此模块被别的模块引入，a.js和b.js中的代码还是被预先执行了。

我们把上面的代码命名为d.js，在别的地方使用它：

require(['d'], function(d){

});

上面的代码会输出

a.js执行  
b.js执行  
d.js执行

可以看出，尽管还未调用d模块的API，里面所依赖的a.js和b.js中的代码已经执行了。AMD的这种只实现语法却未真正实现功能的做法容易给人造成理解上的困难，被**强烈吐槽**。

（在requirejs2.0中，作者声明已经处理了此问题（<https://github.com/jrburke/requirejs/wiki/Upgrading-to-RequireJS-2.0#delayed>），但是我用2.1.20版测试的时候还是会预先执行，我有点不太明白原因，如果有懂的高手请指教）

**兼容并包的CMD/seajs**

既然requirejs有上述种种不甚优雅的地方，所以必然会有新东西来完善它，这就是后起之秀seajs，seajs的作者是国内大牛淘宝前端步道者玉伯。seajs全面拥抱Modules/Wrappings规范，不用requirejs那样回调的方式来编写模块。而它也不是完全按照Modules/Wrappings规范，seajs并没有使用declare来定义模块，而是使用和requirejs一样的define，或许作者本人更喜欢这个名字吧。（然而这或多或少又会给人们造成理解上的混淆），用seajs定义模块的写法如下：

[复制代码](javascript:void(0);)

复制代码

//a.js

define(function(require, exports, module){

console.log('a.js执行');

return {

hello: function(){

console.log('hello, a.js');

}

}

});

复制代码

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

复制代码

//b.js

define(function(require, exports, module){

console.log('b.js执行');

return {

hello: function(){

console.log('hello, b.js');

}

}

});

复制代码

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

复制代码

//main.js

define(function(require, exports, module){

console.log('main.js执行');

var a = require('a');

a.hello();

$('#b').click(function(){

var b = require('b');

b.hello();

});

});

复制代码

[复制代码](javascript:void(0);)

定义模块时无需罗列依赖数组，在factory函数中需传入形参require,exports,module，然后它会调用factory函数的toString方法，对函数的内容进行正则匹配，通过匹配到的require语句来分析依赖，这样就真正实现了commonjs风格的代码。

上面的main.js执行会输出如下：

main.js执行  
a.js执行  
hello, a.js

a.js和b.js都会预先下载，但是b.js中的代码却没有执行，因为还没有点击按钮。当点击按钮的时候，会输出如下：

b.js执行  
hello, b.js

可以看到b.js中的代码此时才执行。这样就真正实现了“就近书写，延迟执行“，不可谓不优雅。

如果你一定要挑出一点不爽的话，那就是b.js的预先下载了。你可能不太想一开始就下载好所有的资源，希望像requirejs那样，等点击按钮的时候再开始下载b.js。本着兼容并包的思想，seajs也实现了这一功能，提供require.async API，在点击按钮的时候，只需这样写：

var b = require.async('b');

b.hello();

b.js就不会在一开始的时候就加载了。这个API可以说是简单漂亮。

关于模块对外暴漏API的方式，seajs也是融合了各家之长，支持commonjs的exports.xxx = xxx和module.exports = xxx的写法，也支持AMD的return写法，暴露的API可以是任意类型。

你可能会觉得seajs无非就是一个抄，把别人家的优点都抄过来组合了一下。其实不然，seajs是commonjs规范在浏览器端的践行者，对于requirejs的优点也加以吸收。看人家的名字，就是海纳百川之意。（再论起名的重要性~），既然它的思想是海纳百川，讨论是不是抄就没意义了。

鉴于seajs融合了太多的东西，已经无法说它遵循哪个规范了，所以玉伯干脆就自立门户，起名曰CMD（Common Module Definition）规范，有了纲领，就不会再存在非议了。

**面向未来的ES6模块标准**

既然模块化开发的呼声这么高，作为官方的ECMA必然要有所行动，js模块很早就列入草案，终于在2015年6月份发布了ES6正式版。然而，可能由于所涉及的技术还未成熟，ES6移除了关于模块如何加载/执行的内容，只保留了定义、引入模块的语法。所以说现在的ES6 Module还只是个雏形，半成品都算不上。但是这并不妨碍我们先窥探一下ES6模块标准。

定义一个模块不需要专门的工作，因为一个模块的作用就是对外提供API，所以只需用exoprt导出就可以了：

//方式一, a.js

export var a = 1;

export var obj = {name: 'abc', age: 20};

export function run(){....}

[复制代码](javascript:void(0);)

//方式二, b.js

var a = 1;

var obj = {name: 'abc', age: 20};

function run(){....}

export {a, obj, run}

[复制代码](javascript:void(0);)

使用模块的时候用import关键字，如：

import {run as go} from 'a'

run()

如果想要使用模块中的全部API，也可以不必把每个都列一遍，使用module关键字可以全部引入，用法：

module foo from 'a'

console.log(foo.obj);

a.run();

在花括号中指明需使用的API，并且可以用as指定别名。

ES6 Module的基本用法就是这样，可以看到确实是有些薄弱，而且目前还没有浏览器能支持，只能说它是面向未来了。

目前我们可以使用一些第三方模块来对ES6进行编译，转化为可以使用的ES5代码，或者是符合AMD规范的模块，例如ES6 module transpiler。另外有一个项目也提供了加载ES6模块的方法，es6-module-loader(<https://github.com/ModuleLoader/es6-module-loader>)，不过这都是一些临时的方案，或许明年ES7一发布，模块的加载有了标准，浏览器给与了实现，这些工具也就没有用武之地了。

未来还是很值得期待的，从语言的标准上支持模块化，js就可以更加自信的走进大规模企业级开发。