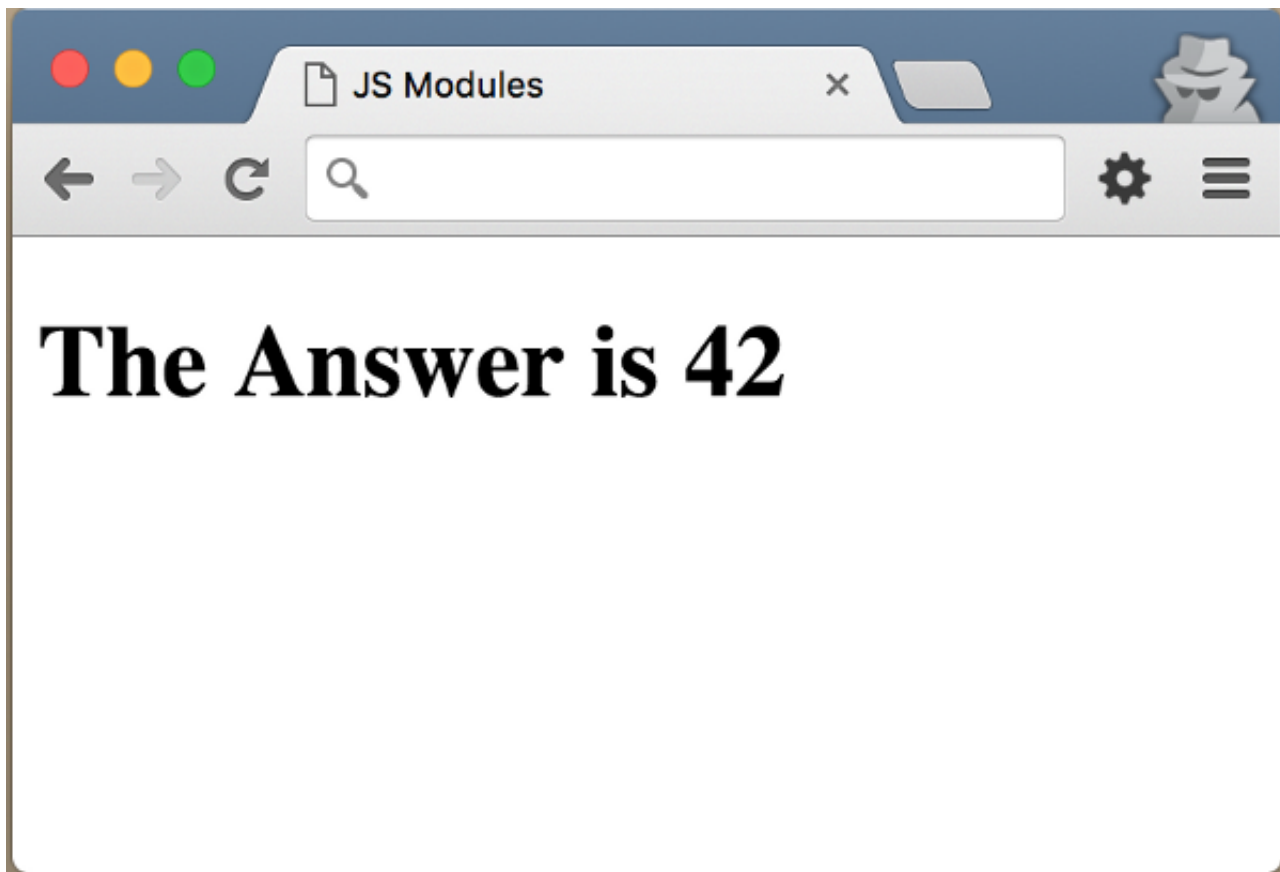


<https://medium.com/@sungyeol.choi/javascript-module-module-loader-module-bundler-es6-module-confused-yet-6343510e7bde#.u17wzyjbc>

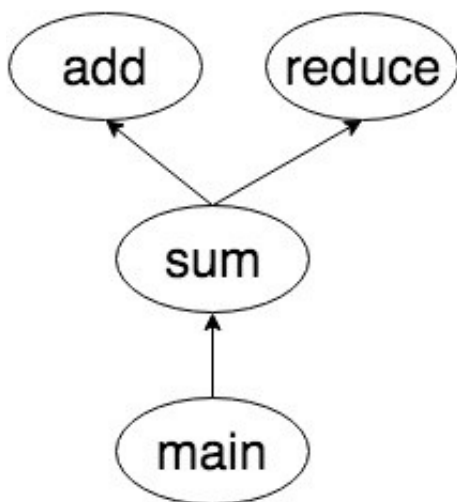
本文解释 JavaScript 模块是什么，它们要解决什么问题，以及如何解决问题。

一、示例应用程序

在本文中，会用一个简单的应用程序来阐述模块的概念。这个应用程序要在浏览器上显示数组的和，它由四个函数和一个 `index.html` 文件组成。



应用程序运行界面



函数的依赖示意图

main 函数计算数组中数字的和，然后把答案显示给 `span#answer`。sum 函数依赖于两个函数：add 和 reduce。add 函数做它名字所做的，把两个数相加。reduce 函数遍历数组，并且调用 iteratee 回调函数。

花点时间理解下面的代码。我会多次使用相同的函数。

0 - index.html

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8">
5     <title>JS Modules</title>
6   </head>
7   <body>
8     <h1>
9       The Answer is
10      <span id="answer"></span>
11    </h1>
12  </body>
13 </html>
```

1 - main.js

```
1 var values = [ 1, 2, 4, 5, 6, 7, 8, 9 ];
2 var answer = sum(values)
3 document.getElementById("answer").innerHTML = answer;
```

2 - sum.js

```
1 function sum(arr){
2   return reduce(arr, add);
3 }
```

3 - add.js

```
1 function add(a, b) {
2   return a + b;
3 }
```

4 - reduce.js

```
1 function reduce(arr, iteratee) {
2   var index = 0,
3     length = arr.length,
4     memo = arr[index];
5   for(index += 1; index < length; index += 1){
6     memo = iteratee(memo, arr[index])
7   }
8   return memo;
9 }
```

我们来看看如何将这些代码片段放在一起，来创建一个应用程序。

1. 使用内嵌脚本

内嵌脚本就是在<script></script>标记之间添加 JavaScript 代码。这是我开始学 JavaScript 时的做法。我相信大多数 JavaScript 开发者在其生命里至少这样做过一次。

这是开始的好方法。不需要操心外部文件或者依赖。但是这也导致了不可维护的代码，因为：

- 缺乏代码可重用性：如果需要添加另一个页面，并需要本页上的一些功能，我们就不得不复制粘贴代码。
- 缺乏依赖解析：你必须保证 main 函数之前就有 add、reduce 和 sum 函数。
- 命名空间污染：所有的函数与变量将都驻留在全局作用域。

```
1 <!-- index.html -->
2
3 <!DOCTYPE html>
4 <html>
5   <head>
6     <meta charset="UTF-8">
7     <title>JS Modules</title>
8   </head>
9   <body>
10    <h1>
11      The Answer is
12      <span id="answer"></span>
13    </h1>
14
15    <script type="text/javascript">
16      function add(a, b) {
17        return a + b;
18      }
19      function reduce(arr, iteratee) {
20        var index = 0,
21            length = arr.length,
22            memo = arr[index];
23        for(index += 1; index < length; index += 1){
24          memo = iteratee(memo, arr[index])
25        }
26        return memo;
27      }
28      function sum(arr){
29        return reduce(arr, add);
30      }
31      /* Main Function */
32      var values = [ 1, 2, 4, 5, 6, 7, 8, 9 ];
33      var answer = sum(values)
34      document.getElementById("answer").innerHTML = answer;
35    </script>
36  </body>
37 </html>
```

2. 使用Script 标记链接外部 JavaScript 文件

这是嵌入脚本的一个自然过渡。现在我们将大段的 JavaScript 分成更小的段，并用 <script src="..."></script> 标记加载它们。

通过将文件分离到多个 JavaScript 文件，就可以重用代码了。我们不再需要在不同的网页之间复制和粘贴代码，只需要将文件用 script 标记就可以了。这种方法虽然更好，但是依然有如下问题：

- 缺乏依赖解析：文件的顺序很重要。你要负责在 main.js 文件之前包含 add.js、reduce.js 和 sum.js

文件。

- 全局命令空间污染：所有的函数和变量依然在全局作用域中。

0 - index.html

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8">
5     <title>JS Modules</title>
6   </head>
7   <body>
8     <h1>
9       The Answer is
10     <span id="answer"></span>
11   </h1>
12
13   <script type="text/javascript" src="./add.js"></script>
14   <script type="text/javascript" src="./reduce.js"></script>
15   <script type="text/javascript" src="./sum.js"></script>
16   <script type="text/javascript" src="./main.js"></script>
17 </body>
```

1 - add.js

```
1 //add.js
2 function add(a, b) {
3   return a + b;
4 }
```

2 - reduce.js

```
1 //reduce.js
2 function reduce(arr, iteratee) {
3   var index = 0,
4     length = arr.length,
5     memo = arr[index];
6
7   index += 1;
8   for(; index < length; index += 1){
9     memo = iteratee(memo, arr[index])
10  }
11  return memo;
12 }
```

3 - sum.js

```
1 //sum.js
2 function sum(arr){
3   return reduce(arr, add);
4 }
```

4 - main.js

```
1 // main.js
2 var values = [ 1, 2, 4, 5, 6, 7, 8, 9 ];
3 var answer = sum(values)
4 document.getElementById("answer").innerHTML = answer;
```

3. 模块对象和 IIFE（模块模式）

通过使用模块对象和立即运行的函数表达式（IIFE），我们可以减少全局命名空间污染。在本方法中，我们只暴露了一个对象给全局作用域，该对象包含了应用程序所需的所有方法和值。在本例中，我们只暴露了 myApp 对象给全局作用域。所有的函数将都被放到 myApp 对象中。

```
1 // 01 my-app.js
2 var myApp = {};
```

```
1 // 02 - add.js
2 (function(){
3   myApp.add = function(a, b) {
4     return a + b;
5   }
6 })();
```

```
1 // 03 - reduce.js
2 (function(){
3   myApp.reduce = function(arr, iteratee) {
4     var index = 0,
5         length = arr.length,
6         memo = arr[index];
7
8     index += 1;
9     for(; index < length; index += 1){
10      memo = iteratee(memo, arr[index])
11    }
12    return memo;
13  }
14 })();
```

```
1 // 04 - sum.js
2 (function(){
3   myApp.sum = function(arr){
4     return myApp.reduce(arr, myUtil.add);
5   }
6 })();
```

```
1 // 05 - main.js
2 (function(app){
3   var values = [ 1, 2, 4, 5, 6, 7, 8, 9 ];
4   var answer = app.sum(values)
5   document.getElementById("answer").innerHTML = answer;
6 })(myApp);
```

```
1 // 06 - index.html
2 <!DOCTYPE html>
3 <html>
4   <head>
5     <meta charset="UTF-8">
6     <title>JS Modules</title>
7   </head>
8   <body>
9     <h1>
10      The Answer is
11      <span id="answer"></span>
12    </h1>
13
```

```

14     <script type="text/javascript" src="./my-app.js"></script>
15     <script type="text/javascript" src="./add.js"></script>
16     <script type="text/javascript" src="./reduce.js"></script>
17     <script type="text/javascript" src="./sum.js"></script>
18     <script type="text/javascript" src="./main.js"></script>
19   </body>
20 </html>

```

注意除了 `my-app.js` 之外，其它每个文件都被封装成 IIFE 格式。

```
1 (function(){ /*... your code goes here ...*/ })();
```

通过将每个文件封装为 IIFE，所有的局部变量就都待在函数作用域内。所以，函数中的所有变量将都会待在函数作用域内，而不会污染全局作用域。

通过将 `add`、`reduce` 和 `sum` 函数附加在 `myApp` 对象上，从而对外暴露它们。并且我们可以像这样，通过引用 `myApp` 来访问这些函数：

```

1 myApp.add(1,2);
2 myApp.sum([1,2,3,4]);
3 myApp.reduce(add, value);

```

我们还可以通过 IIFE 参数，传递 `myApp` 全局对象，就像 `main.js` 文件中所示一样。通过将该对象作为参数传递给 IIFE，我们就可以为该对象选择一个较短的别名。而我们的代码会更简短点。

```

1 (function(obj){
2   // obj 是一个新的 veryLongNameOfGlobalObject
3 })(veryLongNameOfGlobalObject);

```

这与前例相比有很大的提升了。很多流行的 JavaScript 都会用这种模式。比如 jQuery，它暴露一个全局对象 `$`，所有 jQuery 函数都挂在 `$` 对象之下。

但是，这依然不算是完美的解决方案。这种方案依然会遇到上节相同的问题：

- 缺乏依赖解析：文件的顺序依然重要，`myApp.js` 必须出现在所有其它文件之前，`main.js` 必须处在所有其它库文件之后。
- 全局命令空间污染：现在全局变量的数量变成了 1，但是还不是 0。

二、CommonJS

在 2009 年，出现了关于将 JavaScript 带到服务器端的讨论，因而 ServerJS 诞生了。之后，ServerJS 更名为 CommonJS。

CommonJS 并非一个 JavaScript 库，而是一个标准化组织，像 ECMA 或者 W3C 一样。ECMA 定义了 JavaScript 语言规范。W3C 定义了 JavaScript Web API，比如 DOM 和 DOM 事件。CommonJS 的目标是为 Web 服务器、桌面和命令行应用程序定义一套通用的 API。

CommonJS 还定义了模块 API。因为在服务器应用程序中没有 HTML 页面，也没有 script 标记，所以就得

有一些清晰的模块 API。模块需要暴露（输出）给其它模块使用，并且是可访问的（导入）。它的输出模块语法像这样的：

```
1 // add.js
2 module.exports = function add(a, b){
3   return a+b;
4 }
```

上述代码定义和输出了一个模块。代码保存在 `add.js` 文件中。

要使用或者导入 `add` 模块，需要 `require` 函数用文件名或者模块名为参数。如下的语法描述如何导入一个模块到代码中：

```
1 var add = require('./add');
```

如果你曾经在 NodeJS 上写过代码，那么这种语法看起来会很熟悉。这是因为 NodeJS 实现了 CommonJS 风格的模块 API。

三、AMD（异步模块定义）

CommonJS 风格的模块定义的问题是，它不是异步的。当调用 `var add=require('add');` 时，系统会暂停，直到模块准备好了。这意味着在所有模块正在加载时，这行代码会冻结浏览器。所以这可能不是定义浏览器端模块的最佳方式。

为了把服务器端用的模块语法转换给浏览器端用，CommonJS 提出了几种模块格式。其中之一，即“Module/Transfer/C”，后来成为[异步模块定义（AMD）](#)。

AMD 的格式如下：

```
1 define(['add', 'reduce'], function(add, reduce){
2   return function(){...};
3 });
```

`define` 函数（或者关键字）用依赖列表和一个回调函数做参数。回调函数的参数与数组中的依赖有相同的次序。这等于导入模块。而回调函数会返回一个值，该值就是输出的值。

CommonJS 和 AMD 解决了模块模式剩下的两个问题：[依赖解析](#)和[全局作用域污染](#)，现在我们只需要注意每个模块或者文件的依赖，并且不再有全局作用域污染。

四、RequireJS

AMD 可以把我们从浏览器应用程序中的 `script` 标记黑洞和全局污染中解救出来。那么，我们该如何使用它呢？这里 RequireJS 就可以帮助我们了。RequireJS 是一个 JavaScript 模块加载器。它可以帮助我们按需异步加载模块。

尽管 RequireJS 的名字中含有 `require`，但是它的目标却并非要去支持 CommonJS 的 `require` 语法。有了 RequireJS，我们就可以编写 AMD 风格的模块。

在编写自己的应用程序之前，你将不得不从 [RequireJS 网站](#) 下载 `require.js` 文件。如下代码是用

RequireJS 编写的示例应用程序。

AMD 风格的示例应用程序

0 - index.html

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8">
5     <title>JS Modules</title>
6   </head>
7   <body>
8     <h1>
9       The Answer is
10      <span id="answer"></span>
11    </h1>
12
13    <script data-main="main" src="require.js"></script>
14  </body>
15 </html>
```

1 - main.js

```
1 // main.js
2 define(['sum'], function(sum){
3   var values = [ 1, 2, 4, 5, 6, 7, 8, 9 ];
4   var answer = sum(values)
5   document.getElementById("answer").innerHTML = answer;
6 })
```

2 - sum.js

```
1 // sum.js
2 define(['add', 'reduce'], function(add, reduce){
3   var sum = function(arr){
4     return reduce(arr, add);
5   };
6
7   return sum;
8 })
```

3 - add.js

```
1 // add.js
2 define([], function(){
3   var add = function(a, b){
4     return a + b;
5   };
6
7   return add;
8 });
```














4 - reduce.js

```
1 // reduce.js
2 define([], function(){
3   var reduce = function(arr, iteratee) {
4     var index = 0,
5         length = arr.length,
6         memo = arr[index];
7
8     index += 1;
9     for(; index < length; index += 1){
10       memo = iteratee(memo, arr[index])
11     }
12     return memo;
13   }
14
15   return reduce;
16 })
```

注意在 `index.html` 中只有一个 `script` 标记。

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

这个标记加载 `require.js` 库到页面，`data-main` 属性告诉 RequireJS 应用程序的入口点在哪里。默认情况下，它假定所有文件都有 `.js` 扩展名，所以省略 `.js` 文件扩展名是可以的。在 RequireJS 加载了 `main.js` 文件之后，就会加载该文件的依赖，以及依赖的依赖，等等。Chrome 浏览器的开发者工具会显示所有文件被以如下顺序加载：

Name	Method	Status	Type	Initiator	Size	Time	Timeline
 index.html	GET	Finished	document	Other	0 B	18 ms	
 require.js	GET	Finished	script	index.html:13	0 B	2 ms	
 main.js	GET	Finished	script	require.js:1958	0 B	3 ms	
 sum.js	GET	Finished	script	require.js:1958	0 B	1 ms	
 add.js	GET	Finished	script	require.js:1958	0 B	1 ms	
 reduce.js	GET	Finished	script	require.js:1958	0 B	1 ms	

浏览器加载 `index.html`，`index.html` 又加载 `require.js`。剩下的文件及其依赖都是由 `require.js` 负责加载。

RequireJS 和 AMD 解决了我们以前所遇到的所有问题。但是，它带来了其它一些不怎么严重的问题：

- AMD 语法很古怪。因为所有东西都封装在 `define` 函数内，代码就有一些额外的缩进。对于小文件来说，这不是啥问题，但是对于大的代码库来说，就可能是精神上的疲惫。
- 数组中的依赖列表必须与函数的参数列表匹配。如果有很多依赖，就很难维护依赖的次序。如果模块中有几十个依赖，后来又要从中间删除一个，那么就很难找到匹配的模块和参数。
- 在当前浏览器下（HTTP 1.1），加载很多小文件会降低性能。

五、Browserify

因为以上原因，有些人就想用 CommonJS 语法来替换。但是，CommonJS 语法用于服务器和同步的，对

吧？这时 Browserify 就来解救我们了！有了 Browserify，我们就可以在浏览器应用程序中使用 CommonJS 模块。Browserify 是一个模块打包器，它遍历代码的依赖树，将依赖树中的所有模块打包成一个文件。

RequireJS 是一个 JS 库，但是 Browserify 是一个命令行工具，需要 NodeJS 和 NPM 来按住那个它。如果系统中安装了 NodeJS，就可以用如下命令来安装 Browserify：

```
1 npm install -g browserify
```

下面我们来看看用 CommonJS 语法写的示例应用程序。

0 - index.html

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8">
5     <title>JS Modules</title>
6   </head>
7   <body>
8     <h1>
9       The Answer is
10      <span id="answer"></span>
11    </h1>
12
13    <script src="bundle.js"></script>
14  </body>
15 </html>
```

1 - main.js

```
1 //main.js
2 var sum = require('./sum');
3 var values = [ 1, 2, 4, 5, 6, 7, 8, 9 ];
4 var answer = sum(values)
5
6 document.getElementById("answer").innerHTML = answer;
```

2 - sum.js

```
1 //sum.js
2 var reduce = require('./reduce');
3 var add = require('./add');
4
5 module.exports = function(arr){
6   return reduce(arr, add);
7 };
```

3 - add.js

```
1 //add.js
2 module.exports = function add(a,b){
3   return a + b;
```

```
4 };
```

4 - reduce.js

```
1 //reduce.js
2 module.exports = function reduce(arr, iteratee) {
3   var index = 0,
4       length = arr.length,
5       memo = arr[index];
6
7   index += 1;
8   for(; index < length; index += 1){
9     memo = iteratee(memo, arr[index])
10  }
11  return memo;
12 };
```

你可能注意到，在 index.html 文件中，script 标记加载 bundle.js，但是 bundle.js 文件在哪里？一旦我们执行了如下命令，Browserify 就会为我们生成这个文件：

```
1 $ browserify main.js -o bundle.js
```

Browserify 解析 main.js 中的 require() 函数调用，并遍历项目中的依赖树。然后将依赖树打包到一个文件中。

如下是Browserify 生成的 bundle.js 文件的代码：

```
1 function e(t,n,r){function s(o,u){if(!n[o]){if(!t[o]){var a=typeof
  require=="function"&&require;if(!u&&a)return a(o,!0);if(i)return i(o,!0);var f=new
  Error("Cannot find module '"+o+"'");throw f.code="MODULE_NOT_FOUND",f}var l=n[o]=
  {exports:{}};t[o][0].call(l.exports,function(e){var n=t[o][1][e];return s(n?
  n:e)},l,l.exports,e,t,n,r)}return n[o].exports}var i=typeof
  require=="function"&&require;for(var o=0;o<r.length;o++)s(r[o]);return s}({1:
  [function(require,module,exports){
2 module.exports = function add(a,b){
3   return a + b;
4 }];
5
6 },{}],2:[function(require,module,exports){
7 var sum = require('./sum');
8 var values = [ 1, 2, 4, 5, 6, 7, 8, 9 ];
9 var answer = sum(values)
10
11 document.getElementById("answer").innerHTML = answer;
12
13 },{"./sum":4}],3:[function(require,module,exports){
14 module.exports = function reduce(arr, iteratee) {
15   var index = 0,
16       length = arr.length,
17       memo = arr[index];
18
19   index += 1;
20   for(; index < length; index += 1){
21     memo = iteratee(memo, arr[index])
22   }
23 }];
24 };
```

```

23   return memo;
24 };
25
26 },{}],4:[function(require,module,exports){
27 var reduce = require('./reduce');
28 var add = require('./add');
29
30 module.exports = function(arr){
31   return reduce(arr, add);
32 };
33
34 },{"../add":1,"./reduce":3}],{},{},[2]);

```

我们不必一行一行理解这个打包文件，只是要注意，所有熟悉的代码、main 文件及其所有依赖，都包含在这个文件中。

六、UMD — 只会让你更糊涂

现在我们已经学习了全局模块对象、CommonJS 和 AMD 风格的模块，并且有很多库可以帮助我们要么用 CommonJS，要么用 AMD。但是，如果我们正编写一个模块，并要把它部署到互联网上该怎么办？我们到底需该用哪种风格写代码呢？

用三种不同的模块类型，即全局模块对象、CommonJS 和 AMD，都是可以的。但是我们就不得不维护三种不同的文件，用户就不得不识别他们正在下载的模块的类型。

通用模块定义 UMD (Universal Module Definition) 是用来解决这个特殊问题的。本质上，UMD 是一套用来识别当前环境支持的模块风格的 if/else 语句。如下是用 UMD 风格编写的 sum 模块：

```

1 //sum.umd.js
2 (function (root, factory) {
3   if (typeof define === 'function' && define.amd) {
4     // AMD
5     define(['add', 'reduce'], factory);
6   } else if (typeof exports === 'object') {
7     // Node, CommonJS-like
8     module.exports = factory(require('add'), require('reduce'));
9   } else {
10    // Browser globals (root is window)
11    root.sum = factory(root.add, root.reduce);
12  }
13 }(this, function (add, reduce) {
14   // private methods
15
16   // exposed public methods
17   return function(arr) {
18     return reduce(arr, add);
19   }
20 }));

```

七、ES6 模块语法

JavaScript 全局模块对象、CommonJS、AMD 和 UMD，太多选择了。现在也许你会问，下一个项目我该用哪一个呢？答案是一个都不用。

JavaScript 语言中并没有内置模块系统。这正是我们有如此多输入和输出模块的不同方式的原因。但是这种

情况最近得到改变了。在 ES6 规范中，模块已经成为 JavaScript 的一部分。所以这个问题的答案是，如果想让项目不会过时，就得用 ES6 模块语法。

ES6 用 `import` 和 `export` 关键字来输入和输出模块。如下是用 ES6 模块语法编写的示例应用程序。

01 - main.js

```
1 // main.js
2 import sum from './sum';
3
4 var values = [ 1, 2, 4, 5, 6, 7, 8, 9 ];
5 var answer = sum(values);
6
7 document.getElementById("answer").innerHTML = answer
```

02 - sum.js

```
1 // sum.js
2 import add from './add';
3 import reduce from './reduce';
4
5 export default function sum(arr){
6   return reduce(arr, add);
7 }
```

03 - add.js

```
1 // add.js
2 export default function add(a,b){
3   return a + b;
4 }
```

04 - reduce.js

```
1 //reduce.js
2 export default function reduce(arr, iteratee) {
3   let index = 0,
4   length = arr.length,
5   memo = arr[index];
6
7   index += 1;
8   for(; index < length; index += 1){
9     memo = iteratee(memo, arr[index]);
10  }
11  return memo;
12 }
```

关于 ES6 模块有很多广告语：ES6 模块语法是简洁的；ES6 模块将统治 JavaScript 世界；ES6 模块是未来。。。但是不幸的是，有个问题，浏览器还没有为这个新语法做好准备。在本文编写时，只有 Chrome 浏览器支持 `import` 语句。即使到了大多数浏览器都支持 `import` 和 `export` 的时候，如果应用程序必须支持老的浏览器，我们就还会遇到一个问题。

幸运的是，现在已经有很多工具可以用了，这些工具让我们现在就可以用 ES6 模块语法。

八、Webpack

Webpack 是一个模块打包器。就像 Browserify 一样，它会遍历依赖树，然后将其打包到一到多个文件。如果

Webpack 与 Browserify 完全相同，那么我们为什么依然需要另一个模块打包器呢？Webpack 可以处理 CommonJS、AMD 和 ES6 模块，并且它带来了更大的灵活性和很酷的一些功能，比如：

- 代码分离：如果有多个 app 共享相同的模块。Webpack 可以将代码打包到两到多个文件。例如，如果有两个 app：app1 和 app2，二者都共用很多模块。如果用 Browserify 的话，就有 app1.js 和 app2.js，两个文件都要包含所有依赖模块。但是如果是用 Webpack 的话，我们就可以创建 app1.js、app2.js 和 shared-lib.js。是的，我们必须从 html 页面中加载两个文件。但是有了哈希文件名、浏览器缓存和 CDN，就可以降低初始加载时间。
- 加载器：用自定义加载器，可以加载任何文件到源文件中。用 `require()` 语法，不仅仅可以加载 JavaScript 文件，还可以加载 CSS、CoffeeScript、Sass、Less、HTML 模板、图像，等等。
- 插件：Webpack 插件可以在打包写入到文件之前对它进行操作。有很多社区创建的插件。例如，给打包代码添加注释，添加 Source map，将打包文件分离成块等等。
- WebpackDevServer 是一个开发服务器，它可以在源代码改变被检测到时自动打包源代码，并刷新浏览器。通过提供代码的即时反馈，从而加快开发过程。

下面我们来看看如何用 Webpack 创建示例应用程序。Webpack 需要一点引导工作和配置。

因为 Webpack 是 JavaScript 命令行工具，所以需要先安装上 NodeJS 和 NPM。装好 NPM 后，执行如下命令初始化项目：

```
1 $ mkdir project; cd project
2 $ npm init -y
3 $ npm install -D webpack webpack-dev-server
```

你需要为 webpack 写一个配置文件 `webpack.config.js`。文件中至少需要 `entry` 和 `output` 两个字段。

```
1 module.exports = {
2   entry: './app/main.js',
3   output: {
4     filename: 'bundle.js'
5   }
6 }
```

打开 'package.json' 文件，在 'script' 字段后添加如下行：

```
1 "scripts": {
2   "start": "webpack-dev-server --progress --colors",
3   "build": "webpack"
4 },
```

现在在 'project/app' 目录下添加所有 JavaScript 模块，在 'project' 目录下添加 index.html。

01 - index.html

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8">
5     <title>JS Modules</title>
6   </head>
```

```

7   <body>
8     <h1>
9       The Answer is
10      <span id="answer"></span>
11    </h1>
12
13    <script src="bundle.js"></script>
14  </body>
15 </html>

```

02 - webpack.config.js

```

1 module.exports = {
2   entry: './app/main.js',
3   output: {
4     path: './dist',
5     filename: 'bundle.js'
6   }
7 }

```

03 - package.json

```

1 {
2   "name": "jsmodules",
3   "version": "1.0.0",
4   "description": "",
5   "main": "main.js",
6   "scripts": {
7     "start": "webpack-dev-server --progress --colors",
8     "build": "webpack"
9   },
10  "keywords": [],
11  "author": "",
12  "license": "ISC",
13  "devDependencies": {
14    "webpack": "^1.12.14",
15    "webpack-dev-server": "^1.14.1"
16  }
17 }
18

```

04 - app-add.js

```

1 // app/add.js
2 module.exports = function add(a,b){
3   return a + b;
4 };

```

05 - app-reduce.js

```

1 // app/reduce.js
2 module.exports = function reduce(arr, iteratee) {
3   var index = 0,
4       length = arr.length,
5       memo = arr[index];
6
7   index += 1;
8   for(; index < length; index += 1){
9     memo = iteratee(memo, arr[index])
10  }

```

```
11   return memo;
12 };
```

06 - app-sum.js

```
1 // app/sum.js
2 define(['./reduce', './add'], function(reduce, add){
3   sum = function(arr){
4     return reduce(arr, add);
5   }
6
7   return sum;
8 });
```

07 - app-main.js

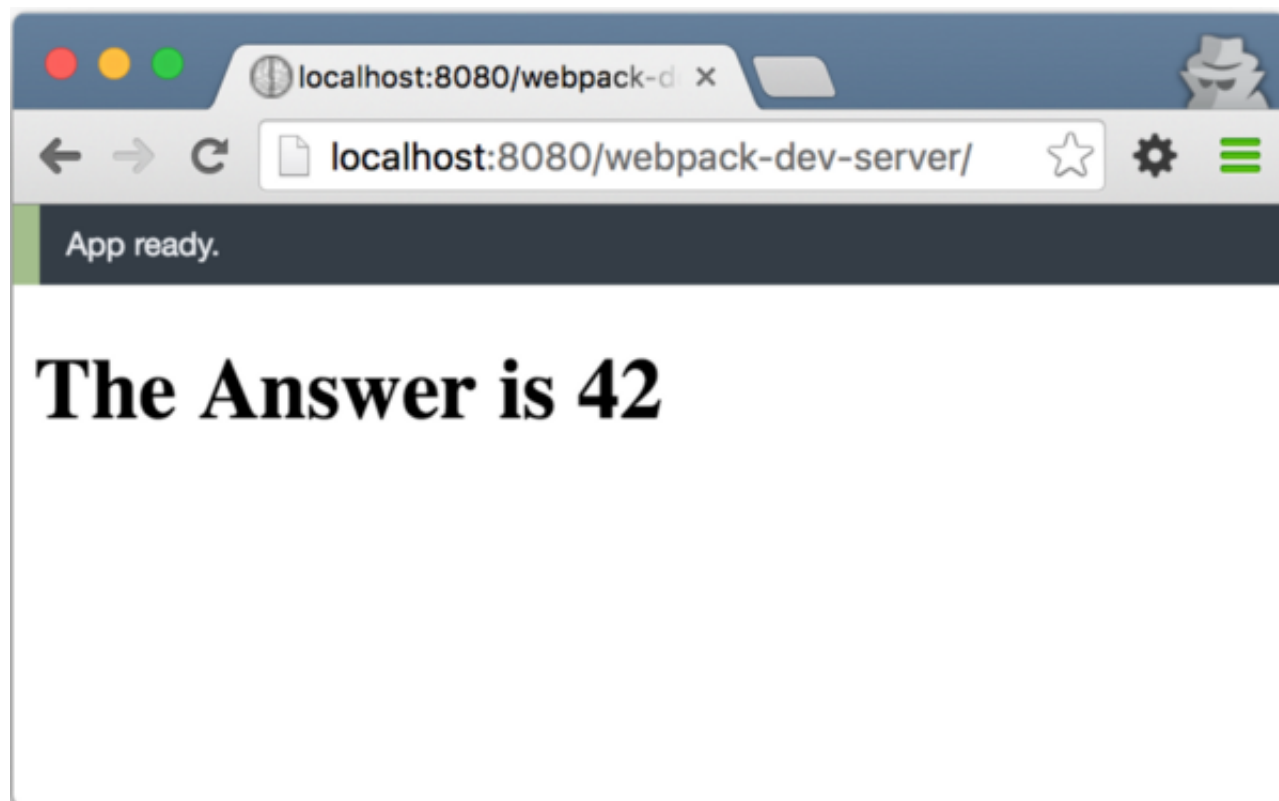
```
1 // app/main.js
2 var sum = require('./sum');
3 var values = [ 1, 2, 4, 5, 6, 7, 8, 9 ];
4 var answer = sum(values)
5
6 document.getElementById("answer").innerHTML = answer;
```

注意 `add.js` 和 `reduce.js` 是用 CommonJS 风格写的，而 `sum.js` 是用 AMD 风格写的。Webpack 默认是可以处理 CommonJS 和 AMD。如果你用 ES6 模块，就需要安装和配置 'babel loader'。

所有文件准备好，就可以运行你的应用程序了。

```
1 $ npm start
```

打开浏览器，把 URL 指向 <http://localhost:8080/webpack-dev-server/>。



webpack dev server in action

此时，你可以打开你喜欢的编辑器编辑代码。保存文件时，浏览器会自动刷新以显示修改后的结果。

这里你可能会注意到一件事情，就是找不到 `dist/bundle.js` 文件。这是因为 Webpack Dev Server 会创建打包文件，但是不会写入到文件系统中，而是放在内存中。

如果要部署，就得创建打包文件。可以键入如下命令，创建 `bundle.js` 文件：

```
1 $ npm run build
```

如果有兴趣学习更多的 Webpack 技巧，请参考 [Webpack 文档页](#)。

九、Rollup (2015 年 5 月)

将一个大的 JavaScript 库包含进来，只是为了用它函数中的少数几个，你是否有这样的经历？Rollup 是另一个 JavaScript ES6 模块打包器。与 Browserify 和 Webpack 不同，rollup 只包含在项目中用到的代码。如果有大模块，带有很多函数，但是你只是用到少数几个，rollup 只会将需要的函数包含到打包文件中，从而显著减少打包文件大小。

Rollup 可以被用作命令行工具。如果有 NodeJS 和 NPM，那么就可以用如下命令安装 rollup：

```
1 $ npm install -g rollup
```

Rollup 可以与任何类型的模块风格一起工作。但是，推荐使用 ES6 模块风格，这样就可以利用 tree-shaking 功能。如下是用 ES6 编写的示例应用程序代码：

01 - add.js

```
1 let add = (a,b) => a + b;
2 let sub = (a,b) => a - b;
3
4 export { add, sub };
```

02 - reduce.js

```
1 // reduce.js
2 export default (arr, iteratee) => {
3   let index = 0,
4   length = arr.length,
5   memo = arr[index];
6
7   index += 1;
8   for(; index < length; index += 1){
9     memo = iteratee(memo, arr[index]);
10  }
11  return memo;
12 }
```

03 - sum.js

```
1 // sum.js
2 import { add } from './add';
3 import reduce from './reduce';
4
```

```
5 export default (arr) => reduce(arr, add);
```

04 - main.js

```
1 // main.js
2 import sum from "./sum";
3
4 var values = [ 1, 2, 4, 5, 6, 7, 8, 9 ];
5 var answer = sum(values);
6
7 document.getElementById("answer").innerHTML = answer;
```

注意，在 `add` 模块中，我引入了另一个函数 `sub()`。但是该函数在应用程序中并没有用到。

现在我们用 `rollup` 将这些代码打包：

```
1 $ rollup main.js -o bundle.js
```

这会生成像如下的 `bundle.js` 文件：

```
1 let add = (a,b) => a + b;
2
3 var reduce = (arr, iteratee) => {
4   let index = 0,
5   length = arr.length,
6   memo = arr[index];
7
8   index += 1;
9   for(; index < length; index += 1){
10     memo = iteratee(memo, arr[index]);
11   }
12   return memo;
13 }
14
15 var sum = (arr) => reduce(arr, add);
16
17 var values = [ 1, 2, 4, 5, 6, 7, 8, 9 ];
18 var answer = sum(values);
19
20 document.getElementById("answer").innerHTML = answer;
```

这里我们可以看到 `sub()` 函数并没有包含在这个打包文件中。

十、SystemJS

SystemJS 是一个通用的模块加载器，它能在浏览器或者 NodeJS 上动态加载模块，并且支持 CommonJS、AMD、全局模块对象和 ES6 模块。通过使用插件，它不仅可以加载 JavaScript，还可以加载 CoffeeScript 和 TypeScript。

SystemJS 的另一个优点是，它建立在 ES6 模块加载器之上，所以它的语法和 API 在将来很可能是语言的一部分，这会让我们代码更不会过时。

要异步输入一个模块，可以用如下语法：

```
1 System.import('module-name');
```

然后我们可以用配置 API 来配置 SystemJS 的行为：

```
1 System.config({  
2   transpiler: 'babel',  
3   baseUrl: '/app'  
4 });
```

上面的配置会让 SystemJS 使用 babel 作为 ES6 模块的编译器，并且从 /app 目录加载模块。

随着现代 JavaScript 应用程序变得越来越大，越来越复杂，开发工作流也是如此。所以我们不仅仅模块加载器，还得去寻找开发服务器、生产的模块打包器以及第三方模块的包管理器。

十一、JSPM

JSPM 是 JavaScript 开发工具的瑞士军刀，它是既是包管理器，又是模块加载器，又是模块打包器。

现代 JavaScript 开发很少只是需要自己的模块，绝大部分时候，我们还需要第三方模块。使用 JSPM，我们可以使用如下的命令，从 NPM 或者 Github 安装第三方模块：

```
1 jspm install npm:package-name or github:package/name
```

上述命令会从 npm 或者 github 下载包，并将包安装到 jspm_packages 目录。

在开发模式下，我们可以使用 jspm-server。像 Webpack Dev Server 一样，它会检测代码改变，重新加载浏览器来显示改变。与 Webpack Dev Server 不同的是，jspm-server 用的是 SystemJS 模块加载器。所以，每次它检测了文件的改变时，不会将所有文件读取来打包，而是只加载页面所需要的模块。

在部署时，肯定要打包代码。JSPM 带有打包器，可以用如下命令对代码打包：

```
1 jspm bundle main.js bundle.js
```

在幕后，JSPM 用 rollup 作为它的打包器。

总结

我希望本文给了足够的信息来理解 JavaScript 模块的词汇。现在你也许会问，下一个项目我应该用什么呢？不幸的是，我回答不了这个问题。现在你有能力开始自己的探索。希望本文能让你更容易理解我提到的有关工具的文档和文章。

本文所有的代码示例都可以在 [这个 Github 仓库](#) 中找到。如有任何疑问，请在下面留言。