在2014年7月底，TC39 [1]又召开了一次会议，在此期间ECMAScript 6（ES6）模块语法的最后细节被最终确定。这篇博客文章概述了完整的 ES6 模块系统。

当前的 JavaScript 模块系统

愚人码头注：原文日期为 2014-09-07 ，请用当时的环境浏览本段内容

JavaScript 目前没有内置支持模块化，但社区创造了令人印象深刻的解决方案。两个最重要的（不幸的是互不兼容）标准是：

* CommonJS 模块：这个标准的主要实现在[Node.js](http://nodejs.org/api/modules.html)中（Node.js模块有一些超越 CommonJS 的功能）。特点：
  + 语法简洁
  + 为同步加载设计
  + 主要用途：服务器端
* 异步模块定义（AMD）：这个标准最流行的实现是 [RequireJS](http://requirejs.org/)。 特点：
  + 稍微复杂的语法，使AMD能够在没有 eval()（或编译步骤）的情况下工作。
  + 专为异步加载而设计
  + 主要用途：浏览器

以上只是对当前状况的大致的解释一下。如果你想要更深入的资料，可以看看Addy Osmani的文章 [使用AMD编写模块化JavaScript，CommonJS ＆ ES Harmony](http://addyosmani.com/writing-modular-js/)。

ECMAScript 6 模块

ECMAScript 6模块的目标是创建一个格式，使 CommonJS 和 AMD 的用户都满意：

* 与 CommonJS 类似，简洁的语法，倾向于单一的接口并且支持循环依赖。
* 与AMD类似，直接支持异步加载和可配置的模块加载。

内置语言允许ES6模块超越 CommonJS 和 AMD（细节将在后面解释）：

* 他们的语法比 CommonJS 更简洁。
* 他们的结构可以静态分析（用于静态检查，优化等）。
* 他们支持的循环依赖性优于 CommonJS。

ES6模块标准有两个部分：

* 声明语法（用于导入和导出）
* 编程式加载器（loader）API：配置如何加载模块以及有条件地加载模块

ES6模块语法概述

有两种导出方式：命名的导出（每个模块可以导出多个）和 默认的导出（每个模块仅导出一个）。

命名的导出（每个模块多个）

模块可以通过使用前缀关键词 export 声明来导出多个东西。这些导出由其名称进行区分，并称为命名的导出。

ECMAScript6 代码:

1. //------ lib.js ------
2. export const sqrt = Math.sqrt;
3. export function square(x) {
4. return x \* x;
5. }
6. export function diag(x, y) {
7. return sqrt(square(x) + square(y));
8. }
10. //------ main.js ------
11. import { square, diag } from 'lib';
12. console.log(square(11)); // 121
13. console.log(diag(4, 3)); // 5

还有其他方法来指定命名的导出（稍后解释），但我觉得这个方式很方便：如果没有外层环境，你可以只关注你的代码，然后以你想要的关键词来标识所有的东西。

如果你需要，您还可以 导入(import) 整个模块，并通过属性符号引用其命名的导出(export) ：

ECMAScript6 代码:

1. //------ main.js ------
2. import \* as lib from 'lib';
3. console.log(lib.square(11)); // 121
4. console.log(lib.diag(4, 3)); // 5

在 CommonJS 语法中的相同代码：有段时间我在 Node.js 下尝试了几种不错的策略，以减少我的模块导出的冗余代码。现在我喜欢以下简单但略微冗余的风格，让人联想到[暴露式模块模式](http://christianheilmann.com/2007/08/22/again-with-the-module-pattern-reveal-something-to-the-world/)：

ECMAScript6 代码:

1. //------ lib.js ------
2. var sqrt = Math.sqrt;
3. function square(x) {
4. return x \* x;
5. }
6. function diag(x, y) {
7. return sqrt(square(x) + square(y));
8. }
9. module.exports = {
10. sqrt: sqrt,
11. square: square,
12. diag: diag,
13. };
15. //------ main.js ------
16. var square = require('lib').square;
17. var diag = require('lib').diag;
18. console.log(square(11)); // 121
19. console.log(diag(4, 3)); // 5

默认的导出（每个模块一个）

在 Node.js 社区中，只导出单个值的模块非常受欢迎。但是它们在前端开发中也很常见，你经常用 构造函数 / 类 来创建模型，每个模块有一个模型。ECMAScript 6模块可以选择默认的导出方式，导出最主要的值。默认的导出方式特别容易导入。

以下 ECMAScript 6模块 **是** 单个函数：

ECMAScript6 代码:

1. //------ myFunc.js ------
2. export default function () { ... };
4. //------ main1.js ------
5. import myFunc from 'myFunc';
6. myFunc();

默认导出一个类的 ECMAScript 6 模块如下所示：

ECMAScript6 代码:

1. //------ MyClass.js ------
2. export default class { ... };
4. //------ main2.js ------
5. import MyClass from 'MyClass';
6. let inst = new MyClass();

注意：定义式导出声明的运算对象是一个表达式，往往不需要名字。相反，它将通过其模块的名称来标识。

在一个模块中同时具有命名的导出和默认的导出

以下模式在 JavaScript 中非常常见：一个库是单个函数，但是通过该函数的属性提供其他功能。例如 jQuery 和 Underscore.js 。 以下是 Underscore 作为 CommonJS 模块的大致写法：

JavaScript 代码:

1. //------ underscore.js ------
2. var \_ = function (obj) {
3. ...
4. };
5. var each = \_.each = \_.forEach =
6. function (obj, iterator, context) {
7. ...
8. };
9. module.exports = \_;
11. //------ main.js ------
12. var \_ = require('underscore');
13. var each = \_.each;
14. ...

使用 ES6 眼光去看，函数\_是默认的导出，而each和forEach均为命名的导出。事实证明，您实际上可以同时具有命名的导出和默认的导出。举个例子，例如之前的 CommonJS 模块，以 ES6 来重写模块，看起来像这样：

ECMAScript6 代码:

1. //------ underscore.js ------
2. export default function (obj) {
3. ...
4. };
5. export function each(obj, iterator, context) {
6. ...
7. }
8. export { each as forEach };
10. //------ main.js ------
11. import \_, { each } from 'underscore';
12. ...

注意 CommonJS 版本和 ECMAScript 6 版本只是大致相似。后者具有扁平式结构，而前者是嵌套式结构。喜欢哪一种风格有自己决定，但是扁平式结构具有静态可分析的优点（为什么这样好下面会解释）。CommonJS风格看起来为了满足对象的需要部分被作为命名空间，这种需求通常可以通过 ES6 模块和命名导出来实现。

需要注意的是 CommonJS 版本和 ECMAScript 6 版本仅仅是大致相同。后者为扁平式结构，而前者为嵌套式结构。喜欢哪一种风格有自己决定，不过扁平式结构具有做静态分析的优势（下面会提到优点）。CommonJS 风格看起来为了满足对象的需要部分被作为命名空间，可以通过 ES6 模块来实现这种需求并且导出实现。

默认导出只是另一个命名导出

默认导出实际上只是一个具有特殊名称default的命名导出。也就是说，以下两个语句是是等价的：

ECMAScript6 代码:

1. import { default as foo } from 'lib';
2. import foo from 'lib';

类似地，以下两个模块也是等价的默认导出：

ECMAScript6 代码:

1. //------ module1.js ------
2. export default 123;
4. //------ module2.js ------
5. const D = 123;
6. export { D as default };

为什么我们需要命名导出？

你可能想知道 – 如果我们可以简单地默认导出对象（如CommonJS），那为什么我们需要命名导出？答案是，你不能通过对象强制实施一个静态结构，并且会失去所有相关的优点（在下一节中描述）。

设计目标

如果你想了解 ECMAScript 6 模块设计理念，那么首先需要了解什么目标影响了他的设计。主要是以下几点：

* 默认导出是有利的
* 静态模块结构
* 同时支持同步和异步加载
* 支持模块之间的循环依赖性

以下小节解释这些目标。

默认导出是有利的

模块语法显示，默认导出可能导致模块看起来有点奇怪，但它是有道理的，你考虑一个主要的设计目标是使默认导出尽可能方便。引用[David Herman](http://esdiscuss.org/topic/moduleimport#content-0)的话说：

ECMAScript 6 支持单个/默认导出风格，并提供默认导入语法糖。导入命名的导出显得不太简洁。

静态模块结构

愚人码头注：关于 静态模块结构 可以看看这篇文章 [webpack 2中的Tree Shaking](http://www.css88.com/archives/6946)，有助于更好的理解。

在当前的JavaScript模块系统中，你必须执行代码，来找出什么是 导入 和 什么是 导出。这是 ECMAScript 6 与这些模块系统（愚人码头注：指 CMD，AMD）决裂的主要原因： 通过将模块系统构建到JavaScript语言中，您可以在语法上强制执行静态模块结构。让我们先来看看这意味着什么，带来什么好处。

模块的静态结构，意味着您可以在编译时确定导入和导出（静态） – 你只需要看看源代码，你不必执行它。下面是两个 CommonJS 模块的例子，告诉你为什么 CommonJS 模块在编译时确定导入和导出是不可能的。在第一示例中，你必须运行代码才可以找出它导入的是什么：

JavaScript 代码:

1. var mylib;
2. if (Math.random()) {
3. mylib = require('foo');
4. } else {
5. mylib = require('bar');
6. }

在第二个示例中，您必须运行代码才可以找出它导出的内容：

JavaScript 代码:

1. if (Math.random()) {
2. exports.baz = ...;
3. }

ECMAScript 6 模块的灵活性不如 CommonJS 模块，强迫使用静态结构。但却使你得到几个好处（参考引用 David Herman 的[“Static module resolution”](http://calculist.org/blog/2012/06/29/static-module-resolution/)），下面描述。

好处1：更快的查找

如果你在 CommonJS 中 require 一个库，你会得到一个对象：

JavaScript 代码:

1. var lib = require('lib');
2. lib.someFunc(); // 属性查找

因此，通过lib.someFunc 访问命名导出意味着您必须进行属性查找，这是很慢，因为它是动态的。

相反，如果您在 ES6 中导入一个库，您可以静态地了解其内容并可以优化访问：

ECMAScript6 代码:

1. import \* as lib from 'lib';
2. lib.someFunc(); // 静态解析

好处2：变量检查

利用静态模块结构，你总是静态地知道哪些变量在模块内的任何位置是可见的：

* 全局变量：越来越多，唯一完全的全局变量将将来自适当的语言。一切都将来自模块（包括来自标准库和浏览器的功能）。也就是说，你静态地知道所有的全局变量。
* 模块导入：你也能静态地知道。
* 模块局部变量：可以通过静态检查模块来确定。

这有助于检查给定的标识符是否拼写正确。这种检查是程序检测器中一个受欢迎的特性，如JSLint和JSHint; 而在 ECMAScript 6 中，大多数可以由 JavaScript 引擎执行。

此外，还可以静态检查命名导入（例如lib.foo）的任何访问。

好处3：为宏命令做准备

宏命令仍然是JavaScript未来的未来。如果JavaScript引擎支持宏命令，你可以通过一个库添加新的语法。[Sweet.js](http://sweetjs.org/)是JavaScript一个实验性的宏系统。下面是Sweet.js网站的一个例子：一个类的宏。

ECMAScript6 代码:

1. macro class {
2. rule {
3. $className {
4. constructor $cparams $cbody
5. $($mname $mparams $mbody) ...
6. }
7. } => {
8. function $className $cparams $cbody
9. $($className.prototype.$mname
10. = function $mname $mparams $mbody; ) ...
11. }
12. }
14. // 使用宏
15. class Person {
16. constructor(name) {
17. this.name = name;
18. }
19. say(msg) {
20. console.log(this.name + " says: " + msg);
21. }
22. }
23. var bob = new Person("Bob");
24. bob.say("Macros are sweet!");

对于宏来说，JavaScript引擎在编译之前执行预处理步骤：如果由解析器产生的token流中的token序列与宏的模式部分匹配，它被由宏的主体生成的token替换。只有当您能够静态地找到宏定义时，预处理步骤才有效。 因此，如果你想通过模块导入宏，那么它们必须有一个静态结构。

好处4：为类型做准备

静态类型检查强加类似于宏的约束：它只能在可以静态找到类型定义时才能完成。同样，只有当模块具有静态结构时，才能从模块导入类型。

类型是吸引人的，因为它们支持静态类型的JavaScript的快速dialect，其中可以编写性能关键代码。一种这样的dialect是[低级JavaScript](http://lljs.org/)（LLJS）。它目前编译为[asm.js](http://www.2ality.com/2013/02/asm-js.html).

好处5：支持其他语言

如果你想支持编译语言的宏和静态类型的JavaScript，JavaScript的模块应该有一个静态结构，因为前两节提到的原因。

同时支持同步和异步加载

ECMAScript 6 模块必须能独立于引擎加载模块，不论是否同步地（例如在服务器上）或异步地（例如在浏览器中）。它的语法非常适合于同步加载，通过其静态结构启用异步加载：因为你可以静态确定所有导入，您可以在评估模块的主体之前加载它们（这种让人联想到AMD模块的方式）。

支持模块之间的循环依赖性

如果两个模块 A 和 B ，A（可能间接）导入 B，并且 B 导入 A ，那么模块 A 和 B 相互依赖。如果可能，应避免循环依赖，因为这样会导致 A 和 B 紧密耦合 – 它们只能一起使用和改进。

为什么支持循环依赖？

循环依赖不是天生就是邪恶的。特别是对于对象来说，你有时甚至想要这种依赖。例如，在一些树（例如DOM文档）中，父元素引用子元素，并且子元素引用回父元素。在库中，通常可以通过仔细设计避免循环依赖。但在大型系统中，它们可能发生，特别是在重构过程中。然后，如果模块系统支持它们是非常有用的，因为当你重构时，系统不会中断。

Node.js文档承认循环依赖的重要性（[查看 Node.js API 文档中的“Modules: Cycles”](https://nodejs.org/api/modules.html#modules_cycles)），并且Rob Sayre提供了额外的证据：

数据点：我曾经为Firefox实现了一个类似[ECMAScript 6 modules]的系统。我被[要求](https://bugzilla.mozilla.org/show_bug.cgi?id=384168#c7)循环依赖支持3周后发布。

Alex Fritze 发明的系统，我工作起来不完美，并且语法不是很漂亮。但[它仍然被使用](https://developer.mozilla.org/en-US/docs/Mozilla/JavaScript_code_modules/Using)7年后，所以它必须得到解决。

让我们看看 CommonJS 和 ECMAScript 6 如何处理循环依赖。

CommonJS中的循环依赖

在CommonJS中，如果模块 B require 主体当前正在被评估的模块 A，它会回到其当前状态下模块 A的出口对象（以下示例中的行#1）。这使得 B 能够引用该对象内部导出（行#2）的属性。在 B 的评估完成后填充属性，此时 B 的导出能工作正常。

ECMAScript6 代码:

1. //------ a.js ------
2. var b = require('b');
3. exports.foo = function () { ... };
5. //------ b.js ------
6. var a = require('a'); // (1)
7. // Can’t use a.foo in module body,
8. // but it will be filled in later
9. exports.bar = function () {
10. a.foo(); // OK (2)
11. };
13. //------ main.js ------
14. var a = require('a');

作为基本规则，请记住，使用循环依赖关系，您无法访问模块主体中的导入。这是现象固有的，并且不随 ECMAScript 6 模块而改变。

CommonJS方法的局限性是：

* Node.js风格的单值导出不能工作。 在Node.js中，您可以导出单个值而不是对象，如下所示：  
  module.exports = function（）{...}  
  如果你在模块A中这么做了，你将无法使用模块B中的导出函数，因为B中的变量 a 仍然引用A的原始导出对象。
* 您不能直接使用命名导出。 也就是说，模块B不能像这样导入a.foo：  
  var foo = require('a').foo;  
  foo 将简单地未定义。 换句话说，你别无选择，只能通过导出对象 a 引用 foo 。

CommonJS有一个独特的功能：您可以在导入之前导出。这样的导出保证可以在导入模块的主体中访问。也就是说，如果A这样做，他们可以在B的主体中访问。但是，在导入之前导出很少有用。

ECMAScript 6中的循环依赖

为了消除上述两个限制，ECMAScript 6 模块导出绑定，而不是值。也就是说，在模块体内声明的变量是保持活动的。这可以通过以下代码演示。

JavaScript 代码:

1. //------ lib.js ------
2. export let counter = 0;
3. export function inc() {
4. counter++;
5. }
7. //------ main.js ------
8. import { inc, counter } from 'lib';
9. console.log(counter); // 0
10. inc();
11. console.log(counter); // 1

因此，面对循环依赖，无论是直接访问命名导出还是通过其模块访问命名导出：在这2种情况下，只要有一个间接引用，它总是能正常工作。

更多关于导入和导出

导入

ECMAScript 6 提供了以下的导入方式（参见 ECMAScript 6 规范中的 [Imports](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-imports)）：

ECMAScript6 代码:

1. // 默认导出和命名导出
2. import theDefault, { named1, named2 } from 'src/mylib';
3. import theDefault from 'src/mylib';
4. import { named1, named2 } from 'src/mylib';
6. // 重命名: 导入 named1 作为 myNamed1
7. import { named1 as myNamed1, named2 } from 'src/mylib';
9. // 导入模块作为一个对象
10. // (每个命名导出都作为一个属性)
11. import \* as mylib from 'src/mylib';
13. // 只加载模块，不导入任何东西
14. import 'src/mylib';

导出

有两种方法可以导出当前模块中的内容（参见 ECMAScript 6 规范中的 [Exports](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-exports)）。 第一种是，您可以使用关键字 export 来声明。

ECMAScript6 代码:

1. export var myVar1 = ...;
2. export let myVar2 = ...;
3. export const MY\_CONST = ...;
5. export function myFunc() {
6. ...
7. }
8. export function\* myGeneratorFunc() {
9. ...
10. }
11. export class MyClass {
12. ...
13. }

默认导出（愚人码头注：通过关键字default声明）的运算对象是一个表达式（包括函数表达式和类表达式）。 例如：

ECMAScript6 代码:

1. export default 123;
2. export default function (x) {
3. return x
4. };
5. export default x => x;
6. export default class {
7. constructor(x, y) {
8. this.x = x;
9. this.y = y;
10. }
11. };

第二种是，您可以在模块的末尾列出要导出的所有内容（风格上与模块模式类似）。

ECMAScript6 代码:

1. const MY\_CONST = ...;
2. function myFunc() {
3. ...
4. }
6. export { MY\_CONST, myFunc };

您也可以使用不同的名称导出：

ECMAScript6 代码:

1. export { MY\_CONST as THE\_CONST, myFunc as theFunc };

请注意，您不能使用[保留字](http://speakingjs.com/es5/ch07.html#identifiers)（如default和new）作为变量名称，但您可以将其用作导出的名称（在 ECMAScript 5 中，您也可以将它们用作属性名称）。如果要直接导入此类命名的导出，那么你必须将它们重命名为正确的变量名称。

重新导出

重新导出意味着将另一个模块的导出添加到当前模块的导出。 你可以添加所有其他模块的导出：

ECMAScript6 代码:

1. export \* from 'src/other\_module';

或者你可以有更多选择性（随意地重命名）：

ECMAScript6 代码:

1. export { foo, bar } from 'src/other\_module';
3. // 导出其他模块的 foo 作为 myFoo
4. export { foo as myFoo, bar } from 'src/other\_module';

eval() 和 模块

eval() 不支持模块语法。它根据脚本语法规则解析其参数,而脚本不支持模块语法（原因稍后解释）。如果要评估模块代码， 您可以使用模块加载器API（如下所述）。

ECMAScript 6 模块加载器 API

除了使用模块的声明性的语法外，还有一个[编程式的API](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-loader-objects)。 它允许您：

* 以编程方式使用模块和脚本
* 配置模块加载

加载器处理解析 *模块说明符*（在 import...from 后面的字符串 ID）加载模块，等。他们的构造函数是Reflect.Loader。每个平台在全局变量 System 中保留自定义实例（*系统加载器*），实现其平台特定的模块加载方式。

导入模块并加载脚本

您可以通过基于 [ES6 promises](http://www.html5rocks.com/en/tutorials/es6/promises/)的 API 以编程方式导入模块：

ECMAScript6 代码:

1. System.import('some\_module')
2. .then(some\_module => {
3. // Use some\_module
4. })
5. .catch(error => {
6. ...
7. });

System.import() 使你可以：

* 在<script>元素中使用模块（不支持模块语法，有关详细信息，请参阅“[更多信息](http://www.2ality.com/2014/09/es6-modules-final.html#further_information)”部分）。
* 有条件地加载模块。

System.import() 检索单个模块，您可以使用Promise.all() 来导入多个模块：

ECMAScript6 代码:

1. Promise.all(
2. ['module1', 'module2', 'module3']
3. .map(x => System.import(x)))
4. .then(([module1, module2, module3]) => {
5. // Use module1, module2, module3
6. });

更多加载器方法：

* [System.module(source, options?)](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-reflect.loader.prototype.module) 计算source中的JavaScript代码到模块（通过promise异步传递）。
* [System.set(name, module)](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-reflect.loader.prototype.set) 用于注册一个模块（例如，您通过 System.module() 创建的一个模块）。
* [System.define(name, source, options?)](https://people.mozilla.org/~jorendorff/es6-draft.html#sec-reflect.loader.prototype.define) 都会评估source中的模块代码并注册结果。

配置模块加载

模块加载器 API 具有用于配置的各种 hook(钩子) 。目前它仍在发展中。用于浏览器的第一系统加载器正在实施和测试。目标是找出如何最好地配置模块加载。

加载器API将允许在很大程度上定制加载过程。例如：

* Lint模块导入（例如通过JSLint或JSHint）。
* 在导入时自动转译模块（它们可能包含 CoffeeScript 或 TypeScript 代码）。
* 使用旧版模块（ AMD，Node.js ）。

可配置模块加载是 Node.js 和 CommonJS 受限的一个领域。

更多信息

以下内容回答与 ECMAScript 6 模块相关的两个重要问题：现在我如何使用他们？如何将它们嵌入到HTML中？

愚人码头注：

由于作者写这篇文章的时间较早，有些观点和资源现在已经不适用了。

比如，现在流行的 ES6 转换器有 [Babel](https://babeljs.io/) ，[Buble](https://gitlab.com/Rich-Harris/buble) , [Rollup.js](http://rollupjs.org/) 等等。

现在很多框架已经支持 ES6 语法，项目一般都通过打包工具（例如：[webpack](https://webpack.js.org/) ）打包发布。

所以现在使用 ES6 已经非常普遍简单了，不需要有太多顾虑。

* “[今天开始使用 ECMAScript 6](http://www.2ality.com/2014/08/es6-today.html)” 概述了 ECMAScript 6 的使用，并解释如何将其编译为 ECMAScript 5 。如果你对后者感兴趣，请先阅读[Sect.2](http://www.2ality.com/2014/08/es6-today.html#using_ecmascript_6_today)。 一个简便使用的解决方案是使用[ES6模块转换器](https://github.com/esnext/es6-module-transpiler)（愚人码头注：该项目已经废弃）将 ES6 模块语法添加到 ES5 并将其编译为 AMD 或 CommonJS 。
* **将ES6模块嵌入HTML中**：<script>元素中的代码不支持模块语法，因为元素的同步性质与模块的异步性不兼容。相反，您需要使用新的<module>元素。博客文章“[ECMAScript 6 模块在未来的浏览器](http://www.2ality.com/2013/11/es6-modules-browsers.html)”解释了<module>是如何工作的。它与比<script>相比有几个显着的优点，可以在其替代版本<script type="module">中进行 polyfill。
* **CommonJS vs. ES6**：“[JavaScript模块](http://jsmodules.io/)”（作者[Yehuda Katz](https://github.com/wycats/jsmodules)）是 ECMAScript 6 模块的快速介绍。特别有趣的是[第二页](http://jsmodules.io/cjs.html)，CommonJ S模块与 ECMAScript 6 版本并排显示。

ECMAScript 6模块的优点

乍一看，ECMAScript 6 中内置的模块看起来可能是一个无聊的功能 – 毕竟我们已经有了几个好的模块系统。但 ECMAScript 6 模块具有无法通过库添加的功能，例如非常紧凑的语法和静态模块结构（这有助于优化，静态检查等）。他们还有望结束当前主流标准 CommonJS 和 AMD 之间的分裂。

具有独立的原生标准对于模块来说意味着：

* 无需更多UMD（[通用模块定义](https://github.com/umdjs/umd)）：UMD是模式的名称，即使得相同的文件能够被多个模块系统（例如 CommonJS 和 AMD ）使用。一旦ES6是唯一模块成为标准，UMD已过时。
* 新的浏览器API成为模块，代替全局变量或引导属性。
* 没有更多的对象命名空间：对象如Math和JSON作为ECMAScript 5中函数的命名空间。在将来，这样的功能可以由模块提供。