模块化的真正意义是什么

对于模块化这样的话语，同学们都听过，但是你们时候真正了解，什么是模块化，为什么要模块化，以及如何模块化？相信同学们都有以上的疑问。对于模块化来说，他并不是一开始就存在的，而是随着 js 的能力浏览器的能力以及业务的发展，我们也需要对于 js 有一个更好的管理模式，接下来通过模块化发展来看一下，模块化是如何一步一步的成长到现在的。

模块化的发展历程以及各个阶段做的事情

当 Brendan Eich 设计 JavaScript 的第一个版本时，他可能不知道他的这个设计在过去二十年中将如何发展。目前已经有三种种主要版本的语言规范（ES3 ES5 ES6），其改进工作仍在继续。
说实话，JavaScript 从来就不是一个完美的编程语言。JS 的一个弱点是模块化的缺失。确实，将所有脚本语言仅用于页面上动画或表单验证，一切都可以在相同的全局范围内交互，代码的依赖关系又该怎么处理。
随着时间的推移，JavaScript 已经转变为通用语言，因为它开始用于在各种环境（浏览器，移动设备，服务器，物联网）中构建复杂的应用程序。程序组件通过全局范围进行交互的旧方法变得不可靠，因为越来越多的代码往往会使应用程序过于脆弱，能解决以上问题的关键在于模块化。

在早期的 Web 开发中，所有的嵌入到网页内的 JavaScript 对象都会使用全局的 window 对象来存放未使用 var 定义的变量。大概在上世纪末，JavaScript 多用于解决简单的任务，这也就意味着我们只需编写少量的 JavaScript 代码；不过随着代码库的线性增长，我们首先会碰到的就是所谓命名冲突（Name Collisions）困境

以下这种情况就存在命名冲突
这里面是两个或者多个 script 标签，代表若干个部分

```
<script>
        function showMsg(valuesArr) {
            console.log(valuesArr.join('-'))
```

```
        }
    </script>
    <script>
        function showMsg(msg) {
            console.log(msg)
        }

        showMsg('HELLO MODULE')
    </script>
```

或者：

```
    <script src="tools.js"></script>
    <script src="lib.js"></script>
```

我们在页面内同时引入这两个 JavaScript 脚本文件时，显而易见两个文件中定义的 showMsg 函数起了冲突，最后调用的函数取决于我们引入的先后顺序。此外在大型应用中，我们不可能将所有的代码写入到单个 JavaScript 文件中；我们也不可能手动地在 HTML 文件中引入全部的脚本文件，特别是此时还存在着模块间依赖的问题，相信很多开发者都会遇到 jQuery 尚未定义这样的问题。

命名空间模式（2002）

要解决名称冲突问题，您可以使用特殊代码约定。例如，您可以为所有变量和函数添加特定前缀 myApp_ myApp_address，myApp_validateUser（）。此外，您可以使用 JavaScript 中的函数是一等公民的事实，即您可以将它们分配给变量，对象的属性并从其他函数返回它们。因此，您可以使用与对象文档和窗口类似的属性创建对象（document.write（），window.alert（））。

如果命名空间的目的是避免冲突的话。下面这个系统，只要我们知道全局变量名前缀 myApp_ 是唯一的，可以像其他系统一样避免命名空间冲突。

```
// add uniquely named global properties
var myApp_sayHello = function() {
   alert('hello');
};
var myApp_sayGoodbye = function() {
   alert('goodbye');
};
```

```
// use the namespace properties
myApp_sayHello();
```

C 语言程序经常使用前缀命名空间。在 JavaScript 的世界中，你可能会碰见 Macromedia 的 MM_ 方法，例如 MM_showHideLayers。

认为前缀命名空间是 JavaScript 中最清楚明白的命名空间系统。（下面的对象命名空间策略在加入了 this 关键字后会导致困惑。）

前缀命名空间的确创建了很多全局对象。这对于前缀用来避免的命名空间冲突并不是什么问题。前缀命名空间的问题是，有些网页浏览器（例如 IE6）在有很多全局对象时表现很糟糕，就我所听说。我做了一些测试并且发现有一个 comp.lang.javascript 的小线程，不过我没有就这个话题研究彻底。

单对象命名空间

当下，最流行的 JavaScript 命名空间实践是使用一个全局变量来引用一个对象。这个被引用的对象引用你的『真正的业务』，并且因为你的全局对象的命名独一无二，你的代码和其他人的代码就可以一起嗨皮地运行。

如果你确定这个世界上没有任何人用了这个全局变量名 myApp，那么你可以有这样的代码：

```
// define the namespace object
var myApp = {};

// add properties to the namespace object
myApp.sayHello = function() {
   alert('hello');
};
myApp.sayGoodbye = function() {
   alert('goodbye');
};

// use the namespace properties
myApp.sayHello();
```

当上面代码的最后一行执行时，JavaScript 解释器首先找到 myApp 对象，然后找到并调用这个对象的 syaHello 属性。

对象命名空间的一个问题是它会导致与面向对象消息传递混淆。这两者之间并没有明显的句法差异：

```
// 1
namespace.prop();
```

```
// 2
receiver.message();
```

更仔细地研究这个混淆，我们得出下面的命名空间想法。假设我们有以下库。

```
var myApp = {};
myApp.message = 'hello';
myApp.sayHello = function() {
   alert(myApp.message);
};
```

用这个库的代码可以随意进行写操作。

```
myApp.sayHello(); // works
var importedfn = myApp.sayHello;
importedfn(); // works
```

将这个和那个令人混淆的使用 this 的消息传递版本比较一下。

```
var myApp = {};
myApp.message = 'hello';

myApp.sayHello = function() {
   alert(this.message);
};
```

用这个库的代码可以随意进行写操作。

```
myApp.sayHello() // works because "this" refers to myApp object.

var importedfn = myApp.sayHello;

importedfn(); // error because "this" refers to global object.
```

这里面的要上的一课是，this 永远不能引用一个被作为命名空间的对象因为它肯能导致关于从命名空间引入标识符的混淆。这个问题是 this 在我的 JavaScript Warning Words 列表中的原因之一。

（这也表明了库的 API 属性应该指向用一个方法，这样这些方法可以被导入其他命名空间。懒惰方法定义可以在被隐藏在库中并且不是 API 的部分时安全使用。）

嵌套对象命名空间

嵌套对象命名空间是另一个普遍的实践，它扩展了对象命名空间的想法。

你可能见过类似如下代码：

```
YAHOO.util.Event.addListener(/*...*/)
```

解决上面的代码需要解释器首先找到全聚德 YAHOO 对象，然后它的 util 对象，然后它的 Event 对象，然后找到并调用它的 addListener 属性。这样的话每次事件处理器绑定到一个 DOM 元素上花的功夫太多了，因此导入的概念开始被采用。

```
(function() {
   var yue = YAHOO.util.Event;
   yue.addListener(/*...*/);
   yue.addListener(/*...*/);
})();
```

如果你清楚 YAHOO.util.Event.addListener 方法不会用 this 关键字并且永远引用同一个方法，那么导入可以变得更加简洁。

```
(function() {
   var yuea = YAHOO.util.Event.addEventListener;
   yuea(/*...*/);
   yuea(/*...*/);
})();
```

我觉得当目的只是避免标识符冲突时，嵌套对象命名空间的复杂是不必要的。难道 Yahoo! 还觉得这些全局标识符 YAHOO_util_Event 和 YAHOO_util_Event_addEventListener 不够独特吗？

我认为使用嵌套对象命名空间的动机是要看起来和 Java 包命名传统一样，这在 Java 中开销不大。例如，在 Java 中你可能看到如下：

```
package mytools.text;

class TextComponent {
   /* ... */
}
```

一个这个类的完全合格的引用应该是 mytools.text.TextComponent。

下面是 Niemeyer 和 Knudsen （写）的 Learning Java 中包命名的描述：

包名是按层级构成的，使用点分隔的命名传统。包名组成成分给编译器和

运行系统构成了独一无二的定位文件的路径。然而，它们并没在包之间创建其他的关系。并没有什么『subpackage』的说法，事实上，包命名空间是直接的，而非层级的。在包层级关系特定部分的包仅仅是因为习惯而有关联。比如，如果我们穿件了另一个叫做 mytools.text.poetry 的包（假设是为了跟诗有关的一些文字类），这些类并不是 mytools.text 包的一部分；它们没有包成员的访问权限。

嵌套命名空间的幻觉在 Perl 中也存在。在 Perl 中，嵌套包名由双冒号分隔开。你可以看到如下 Perl 代码：

```
package Red::Blue;
our $var = 'foo';
```

一个完全合格的上述变量引用应该是 $Red::Blue::var。
在 Perl 中，就像 Java，命名空间层级的主意只是方便程序员，而不是语言本身要求。Wall，Christiansen 和 Orwant 的 Programming Perl 解释道：双冒号可被用于链接在包名 $Red::Blue::var 中标识符。这意味着 $var 属于包 Red::Blue。包 Red::Blue 跟可能存在的 Red 包或 Blue 包一点关系都没有。只是说，Red::Blue 和 Red 或者 Blue 之间的关系可能对于写代码或者使用这个程序的人有什么意义，但跟 Perl 没关系。（好吧，除了在现在的实现中，符号表 Red::Blue 刚好存在符号表 Red 中。但是 Perl 语言并没有直接利用过它。）
上述引用中最后备注暗示了 Perl 可能有和在 JavaScript 中使用嵌套命名空间对象一样的标识符冲突开销。如果 Perl 的实现改变了，这个开销就会消失。在 JavaScript 中，我肯定嵌套对象命名空间的开销永远不会消失因为 JavaScript 使用延迟绑定。
我并不认为 JavaScript 中的嵌套对象命名空间提供了任何大好处，不过如果不使用导入的话在运行时可能会开销非常大。
一个折中方案
如果单纯地前缀命名空间在某些浏览器中真的很慢，而嵌套命名空间的概念帮助在开发者脑中保持各事务的有序，那我认为上述 Yahoo! 的例子也可以这样写：

```
YAHOO.util_Event_addListener
```

或者用更多的全局名称：

```
YAHOO_util_Event.addListener
```

哪个维度的命名空间？

Perl 的 CPAN 模块是基于他们所做的事情进行命名空间管理的。例如，我写了一个这个命名空间里的模块：

JavaScript::Minifier
如果别人用同样的名字写他自己的模块，并且他不自知地通过某些模块依赖通过同一个名字使用 CPAN 模块，那么就会有冲突。

Java 程序员采用最冗长但当然也是最安全的方法。（Java 程序员似乎都想着在大型系统上运行的代码。）在 Java 中，包经常是基于谁写的和做什么的来命名。（myFunc 风格的规范化。）『谁写的』部分甚至使用开发者自己的相对可以保证唯一性的名字。如果我写一个 Java 的 minifier，因为我有 michaux.ca 的域名，我可能用以下命名空间：

ca.michaux.javascript.minifier
在 JavaScript 中，经过这次讨论，可能这样写效率更高：

ca_michaux_javascript_minifier
因为 JavaScript 是以文本的形式服务的，这样的命名空间可能开销太大，因为增加了下载时间。Gzip 压缩会找到公共的字符串并用短字符串替换它们。如果 gzip 不可用的话那么就可以考虑使用导入了。

```
var ca_michaux_javascript_minifier = {};

(function() {
   var cmjm = ca_michaux_javascript_minifier;

   // refer to cmjm inside this module pattern
})();
```

我并不是说这些长的命名空间是绝对必须的，不过他们一定是避免命名空间冲突的最安全方法。
其他命名空间问题
标识符不仅在 JavaScript 资源中创建。一个表单的 name 属性也被加在 document.forms 对象上。像

这样命名是有意义的。
命名空间类名属性，比如 <div class="myCompany_story”>，可以在减少 CSS 命名空间冲突以及当 JavaScript 代码在通过类名搜索 DOM 元素时

很有价值。
利用这个模式的第一个重要项目是一个 ui 元素 Bindows 库。Bindows 是由我们在 2002 年熟悉的 Erik Arvidsson 创建的。他没有在函数和变量的名称中使用前缀，而是使用了一个全局对象，其属性包含库的数据和逻辑。这一事实大大减少了全球范围的污染。该代码组织的模式现在称为“命名空间”（命名空间模式）。
但是这种模式最大的问题在于**变量完全暴露，私有变量方法都没有办法进行设定。**

闭包模块化模式（2003）

命名空间为代码组织提供了某种顺序。但很明显，这还不够，因为还没有解决模块和数据的隔离问题。
解决这个问题的先驱是模块模式。它的主要思想是使用闭包封装数据和代码，并通过外部可访问的方法提供对它们的访问。以下是此类模式的基本示例：

```
var greeting = (function () {
    var module = {};

    var helloInLang = {
        en: 'Hello world!',
        es: '¡Hola mundo!',
        ru: 'Привет мир!'
    };

    module.getHello = function (lang) {
        return helloInLang[lang];
    };

    module.writeHello = function (lang) {
        document.write(module.getHello(lang))
    };

    return module;
}());
```

这里我们看到立即调用的函数，它返回一个模块对象，该模块对象又有一

个通过闭包 getHello 访问对象的方法 helloInLang。因此，helloInLang 从外部世界变得无法访问，我们得到一个代码片段，可以粘贴到任何其他脚本而不会发生名称冲突。接下来深入的说一些模块模式。

模块模式是一种常见的 JavaScript 编码模式。它通常被很好地理解，但有许多高级用途没有得到很多关注。在本文中，我将回顾基础知识并介绍一些非常出色的高级主题。

匿名闭包

这是使一切成为可能的基本结构，并且确实是 JavaScript 的唯一最佳功能。我们将简单地创建一个匿名函数，并立即执行它。在函数内部运行的所有代码都存在于闭包中，该闭包在我们的应用程序的整个生命周期中提供隐私和状态。

```
(function () {
    // ... all vars and functions are in this scope only
    // still maintains access to all globals
}());
```

注意()匿名函数。这是语言所必需的，因为以 function 开头的语句始终被视为函数声明。包括()创建一个函数表达式。立即执行函数。JavaScript 有一个称为隐含全局变量的功能。每当使用名称时，解释器向后遍历作用域链，寻找 var 该名称的语句。如果没有找到，则假定该变量是全局变量。如果它在赋值中使用，则创建全局（如果它尚不存在）。这意味着在匿名闭包中使用或创建全局变量很容易。不幸的是，这导致难以管理的代码，因为对于给定文件中哪些变量是全局的并不明显（对人类而言）。

幸运的是，我们的匿名功能提供了一个简单的选择。通过将 globals 作为参数传递给我们的匿名函数，我们将它们导入到我们的代码中，这比隐含的全局变量更清晰，更快。这是一个例子：

```
(function ($, YAHOO) {
    // now have access to globals jQuery (as $) and YAHOO in this code
}(jQuery, YAHOO));
```

有时候你不只是想使用全局变量，而是想要声明它们。我们可以通过使用匿名函数的返回值导出它们来轻松完成此操作。这样做将完成基本模块模式，所以这是一个完整的例子：

```
var MODULE = (function () {
    var my = {},
```

```
        privateVariable = 1;

    function privateMethod() {
        // ...
    }

    my.moduleProperty = 1;
    my.moduleMethod = function () {
        // ...
    };

    return my;
}());
```

请注意，我们已经声明了一个名为的全局模块 MODULE，它有两个公共属性：一个名为的方法 MODULE.moduleMethod 和一个名为的变量 MODULE.moduleProperty。此外，它使用匿名函数的闭包来维护私有内部状态。此外，我们可以使用上面学到的模式轻松导入所需的全局变量。

虽然上述内容足以满足许多用途，但我们可以进一步采用这种模式，并创建一些非常强大，可扩展的结构。让我们一个接一个地完成它们，继续我们的模块命名 MODULE。

到目前为止，模块模式的一个限制是整个模块必须在一个文件中。在大型代码库中工作的任何人都理解在多个文件之间拆分的价值。幸运的是，我们有一个很好的解决方案来增强模块。首先，我们导入模块，然后添加属性，然后导出它。这是一个例子，MODULE 从上面扩充我们：

```
var MODULE = (function (my) {
    my.anotherMethod = function () {
        // added method...
    };

    return my;
}(MODULE));
```

我们 var 再次使用关键字来保持一致性，即使它没有必要。运行此代码后，我们的模块将获得一个名为的新公共方法 MODULE.anotherMethod。此扩充文件还将维护其自己的私有内部状态和导入。

虽然我们上面的例子要求我们首先创建初始模块，然后将扩充设置为第二个，但这并不总是必要的。JavaScript 应用程序可以为性能做的最好的事情之一就是异步加载脚本。我们可以创建灵活的多部件模块，可以通过松散扩充以任何顺序加载自己。每个文件应具有以下结构：

```
var MODULE = (function (my) {
    // add capabilities...

    return my;
}(MODULE || {}));
```

在这种模式中，var 语句总是必要的。请注意，如果模块尚不存在，则导入将创建该模块。这意味着您可以使用像 LABjs 这样的工具并且并行加载所有模块文件，而无需阻止。

紧张增强

虽然松散增强很棒，但它确实会对您的模块造成一些限制。最重要的是，您无法安全地覆盖模块属性。初始化期间也不能使用其他文件中的模块属性（但可以在初始化后的运行时）。紧密增强意味着设置加载顺序，但允许覆盖。这是一个简单的例子（扩充我们的原创 MODULE）：

```
var MODULE = (function (my) {
    var old_moduleMethod = my.moduleMethod;

    my.moduleMethod = function () {
        // method override, has access to old through old_moduleMethod...
    };

    return my;
}(MODULE));
```

在这里，我们已经覆盖了 MODULE.moduleMethod，但如果需要，还要保留对原始方法的引用。

克隆和继承

```
var MODULE_TWO = (function (old) {
    var my = {},
        key;
```

```
    for (key in old) {
        if (old.hasOwnProperty(key)) {
            my[key] = old[key];
        }
    }

    var super_moduleMethod = old.moduleMethod;
    my.moduleMethod = function () {
        // override method on the clone, access to super through super_moduleMethod
    };

    return my;
}(MODULE));
```

这种模式可能是最不灵活的选择。它确实允许一些整洁的成分，但这是以牺牲灵活性为代价的。正如我所写的那样，作为对象或函数的属性不会重复，它们将作为一个具有两个引用的对象存在。改变一个将改变另一个。对于具有递归克隆过程的对象，这可能是固定的，但可能无法修复函数，除非可能使用 eval。尽管如此，我还是把它包括在内是为了完整性。

跨文件私有状态

在多个文件之间拆分模块的一个严重限制是每个文件都维护自己的私有状态，并且不能访问其他文件的私有状态。这可以修复。下面是一个松散扩充模块的示例，它将在所有扩充中维护私有状态：

```
var MODULE = (function (my) {
    var _private = my._private = my._private || {},
        _seal = my._seal = my._seal || function () {
            delete my._private;
            delete my._seal;
            delete my._unseal;
        },
        _unseal = my._unseal = my._unseal || function () {
            my._private = _private;
            my._seal = _seal;
            my._unseal = _unseal;
        };
```

```
    // permanent access to _private, _seal, and _unseal

    return my;
}(MODULE || {}));
```

任何文件都可以在其本地变量上设置属性_private，并且其他文件将立即可用。一旦此模块完全加载，应用程序应该调用 MODULE._seal()，这将阻止外部访问内部 _private。如果要再次扩充此模块，在应用程序的生命周期中，任何文件中的一个内部方法可以 _unseal()在加载新文件之前调用 _seal()，并在执行后再次调用。今天我在工作时发生了这种模式，我没有在其他地方见过这种模式。我认为这是一个非常有用的模式，并且本身值得一提。

子模块

我们最终的高级模式实际上是最简单的。创建子模块有很多好的案例。这就像创建常规模块一样：

```
MODULE.sub = (function () {
    var my = {};
    // ...

    return my;
}());
```

虽然这可能是显而易见的，但我认为值得包括。子模块具有普通模块的所有高级功能，包括扩充和私有状态。

模版依赖定义 (2006)

这时候开始流行后端模版语法，通过后端语法聚合 js 文件，从而实现依赖加载，说实话，现在 go 语言等模版语法也很流行这种方式，写后端代码的时候不觉得，回头看看，还是挂在可维护性上。

注释依赖定义 (2006)

和模版依赖定义同时出现，与 1999 年方案不同的，不仅仅是模块定义方

式，而是终于以文件为单位定义模块了，通过 lazyjs 加载文件，同时读取文件注释，继续递归加载剩下的文件。

外部依赖定义（2007）

这种定义方式在 cocos2d-js 开发中普遍使用，其核心思想是将依赖抽出单独文件定义，这种方式不利于项目管理，毕竟依赖抽到代码之外，我是不是得两头找呢？所以才有通过 webwpack 打包为一个文件的方式暴力替换为 commonjs 的方式出现。

依赖注入（2009）

2004 年，Martin Fowler 介绍了“ 依赖注入 ”（DI）的概念，用于描述 Java 中组件的新通信机制。要点是所有依赖关系都来自组件外部，因此组件不负责初始化它只使用它们的依赖关系。
五年后，MiškoHevery 成为 Sun 和 Adobe 的前雇员（他主要从事 Java 开发），他开始为他的创业公司设计一个新的 JavaScript 框架，它使用依赖注入作为组件之间通信的关键机制。商业理念尚未证明其有效性，框架的源代码在他的创业公司 getangular.com 的领域被打开并引入世界。我们都知道接下来发生了什么。谷歌已经采取了它的翼 Miško 和他的项目，现在 Angular 是最着名的 JavaScript 框架之一。
Angular 中的模块通过依赖注入机制实现。顺便说一句，模块化不是 DI 的主要目的，Miško 在相应问题的答案中也明确表示。

CommonJS（2009）

为什么会出现 CommonJS 规范？
因为 JavaScript 本身并没有模块的概念，不支持封闭的作用域和依赖管理，传统的文件引入方式又会污染变量，甚至文件引入的先后顺序都会影响整个项目的运行。同时也没有一个相对标准的文件引入规范和包管理系统，这个时候 CommonJS 规范就出现了
CommonJS API 定义很多普通应用程序（主要指非浏览器的应用）使用的 API，从而填补了这个空白。它的终极目标是提供一个类似 Python，Ruby

和 Java 标准库。这样的话，开发者可以使用 CommonJS API 编写应用程序，然后这些应用可以运行在不同的 JavaScript 解释器和不同的主机环境中。

在兼容 CommonJS 的系统中，你可以使用 JavaScript 开发以下程序：

服务器端 JavaScript 应用程序
命令行工具
图形界面应用程序
混合应用程序（如，Titanium 或 Adobe AIR）

2009 年，美国程序员 Ryan Dahl 创造了 node.js 项目，将 javascript 语言用于服务器端编程。这标志”Javascript 模块化编程”正式诞生。因为老实说，在浏览器环境下，没有模块也不是特别大的问题，毕竟网页程序的复杂性有限；但是在服务器端，一定要有模块，与操作系统和其他应用程序互动，否则根本没法编程。NodeJS 是 CommonJS 规范的实现，webpack 也是以 CommonJS 的形式来书写。

原理

浏览器不兼容 CommonJS 的根本原因，在于缺少四个 Node.js 环境的变量。

module
exports
require
global

只要能够提供这四个变量，浏览器就能加载 CommonJS 模块。下面是一个简单的示例。

```
var module = {
  exports: {}
};
(function(module, exports) {
  exports.multiply = function (n) { return n * 1000 };
}(module, module.exports))
var f = module.exports.multiply;
f(5) // 5000
```

上面代码向一个立即执行函数提供 module 和 exports 两个外部变量，模块就放在这个立即执行函数里面。模块的输出值放在 module.exports 之中，这样就实现了模块的加载。

根据 CommonJS 规范，一个单独的文件就是一个模块。每一个模块都是一个单独的作用域，也就是说，在一个文件定义的变量（还包括函数和类），都是私有的，对其他文件是不可见的。

```
var x = 5;
var addX = function(value) {
  return value + x;
};
```

上面代码中，变量 x 和函数 addX，是当前文件私有的，其他文件不可见。

如果想在多个文件分享变量，必须定义为 global 对象的属性。\

上面代码的 waining 变量，可以被所有文件读取。当然，这样写法是不推荐的。

CommonJS 规定，每个文件的对外接口是 module.exports 对象。这个对象的所有属性和方法，都可以被其他文件导入。

```
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 对象，定义对外接口，输出变量 x 和函数 addX。module.exports 对象是可以被其他文件导入的，它其实就是文件内部与外部通信的桥梁。

require 方法用于在其他文件加载这个接口，具体用法参见《Require 命令》的部分。

```
var example = require('./example.js');
console.log(example.x); // 5
console.log(addX(1)); // 6
```

CommonJS 模块的特点如下。

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。

module 对象

每个模块内部，都有一个 module 对象，代表当前模块。它有以下属性。

module.id 模块的识别符，通常是带有绝对路径的模块文件名。

module.filename 模块的文件名，带有绝对路径。

module.loaded 返回一个布尔值，表示模块是否已经完成加载。
module.parent 返回一个对象，表示调用该模块的模块。
module.children 返回一个数组，表示该模块要用到的其他模块。
下面是一个示例文件，最后一行输出 module 变量。

```
// example.js
var jquery = require('jquery');
exports.$ = jquery;
console.log(module);
```

执行这个文件，命令行会输出如下信息。

```
{ id: '.',
  exports: { '$': [Function] },
  parent: null,
  filename: '/path/to/example.js',
  loaded: false,
  children:
   [ { id: '/path/to/node_modules/jquery/dist/jquery.js',
       exports: [Function],
       parent: [Circular],
       filename: '/path/to/node_modules/jquery/dist/jquery.js',
       loaded: true,
       children: [],
       paths: [Object] } ],
  paths:
   [ '/home/user/deleted/node_modules',
     '/home/user/node_modules',
     '/home/node_modules',
     '/node_modules' ]
}
```

如果在命令行下调用某个模块，比如 node something.js，那么 module.parent 就是 undefined。如果是在脚本之中调用，比如 require('./something.js')，那么 module.parent 就是调用它的模块。利用这一点，可以判断当前模块是否为入口脚本。

```
if (!module.parent) {
    // ran with `node something.js`
```

```
    app.listen(8088, function() {
        console.log('app listening on port 8088');
    })
} else {
    // used with `require('/.something.js')`
    module.exports = app;
}
```

module.exports 属性

module.exports 属性表示当前模块对外输出的接口，其他文件加载该模块，实际上就是读取 module.exports 变量。

```
var EventEmitter = require('events').EventEmitter;
module.exports = new EventEmitter();

setTimeout(function() {
  module.exports.emit('ready');
}, 1000);
```

上面模块会在加载后 1 秒后，发出 ready 事件。其他文件监听该事件，可以写成下面这样。

```
var a = require('./a');
a.on('ready', function() {
  console.log('module a is ready');
});
```

exports 变量

为了方便，Node 为每个模块提供一个 exports 变量，指向 module.exports。这等同在每个模块头部，有一行这样的命令。

```
var exports = module.exports;
```

造成的结果是，在对外输出模块接口时，可以向 exports 对象添加方法。

```
exports.area = function (r) {
  return Math.PI * r * r;
};

exports.circumference = function (r) {
  return 2 * Math.PI * r;
```

};
注意，不能直接将 exports 变量指向一个值，因为这样等于切断了 exports 与 module.exports 的联系。

exports = function(x) {console.log(x)};
上面这样的写法是无效的，因为 exports 不再指向 module.exports 了。

AMD 规范(2009)

介绍了同步方案，我们当然也有异步方案。在浏览器端，我们更常用 AMD 来实现模块化开发。AMD 是 Asynchronous Module Definition 的简称，即“异步模块定义”。
我们看一下 AMD 模块的使用方式：
在这里，我们使用了 define 函数，并且传入了两个参数。
第一个参数是一个数组，数组中有两个字符串也就是需要依赖的模块名称。AMD 会以一种非阻塞的方式，通过 appendChild 将这两个模块插入到 DOM 中。在两个模块都加载成功之后，define 会调用第二个参数中的回调函数，一般是函数主体。
第二个参数也就是回调函数，函数接受了两个参数，正好跟前一个数组里面的两个模块名一一对应。因为这里只是一种参数注入，所以我们使用自己喜欢的名称也是完全没问题的。
同时，define 既是一种引用模块的方式，也是定义模块的方式。
所以我们可以看到，AMD 优先照顾浏览器的模块加载场景，使用了异步加载和回调的方式，这跟 CommonJS 是截然不同的。
接下来说说 AMD 规范的 API
define()函数
本规范只定义了一个函数"define"，它是全局变量。函数的描述为:

define(id?, dependencies?, factory)id

第一个参数, id 是个字符串。它指的是定义模块的名字, 这个参数是可选的。如果没有提供该参数, 模块的名字应该默认为模块加载器请求的指定脚本的名字。如果提供了该参数, 模块名必须是"顶级" 的和绝对的(不允许相对名字)。
模块名的格式
模块名用来唯一标识定义中模块， 它们同样在依赖数组中使用。AMD 的模块名规范是 CommonJS 模块名规范的超集。引用如下:

模块名是由一个或多个单词以正斜杠为分隔符拼接成的字符串
单词须为驼峰形式，或者".", ".."
模块名不允许文件扩展名的形式，如".js"
模块名可以为"相对的"或"顶级的"。如果首字符为"." 或 ".."则为"相对的"模块名
顶级的模块名从根命名空间的概念模块解析
相对的模块名从"require"书写和调用的模块解析

上文引用的 CommonJS 模块 id 属性常被用于 JavaScript 模块。
相对模块名解析示例:

如果模块"a/b/c"请求"../d"，则解析为"a/d"
如果模块"a/b/c"请求"./e"，则解析为"a/b/e"

如果 AMD 的实现加载器插件(Loader-Plugins)，则"!"符号用于分隔加载器插件模块名和插件资源名。由于插件资源名可以非常自由地命名，大多数字符都允许在插件资源名使用。
(译注: 关于 Loader-Plugins)
依赖

第二个参数, dependddencies 是个定义中模块所依赖模块的数组。依赖模块必须根据模块的工厂方法优先级执行，并且执行的结果应该按照依赖数组中的位置顺序以参数的形式传入(定义中模块的)工厂方法中。依赖的模块名如果是相对的,应该解析为相对定义中的模块。换句话来说，相对名解析为相对于模块的名字，并非相对于寻找该模块的名字的路径。

本规范定义了三种特殊的依赖关键字。如果"require", "exports", 或"module"出现在依赖列表中，参数应该按照 CommonJS 模块规范自由变量去解析。

依赖参数是可选的，如果忽略此参数，它应该默认为["require", "exports", "module"]。然而，如果工厂方法的形参个数小于 3，加载器会选择以函数指定的参数个数调用工厂方法。

工厂方法
第三个参数, factory，为模块初始化要执行的函数或对象。如果为函数，它应该只被执行一次。如果是对象，此对象应该为模块的输出值。

如果工厂方法返回一个值(对象，函数，或任意强制类型转换为 true 的值)，应该为设置为模块的输出值。
简单的 CommonJS 转换

如果依赖性参数被忽略,模块加载器可以选择扫描工厂方法中的 require 语句获得依赖性(字面量形参为 require("module-id"))。 第一个参数必须字面量为 require 从而使此机制正常工作。
在某些情况下，因为脚本大小的限制或函数不支持 toString 方法(Opera Mobile 是已知的不支持函数的 toString 方法)，模块加载器可以选择扫描不扫描依赖性。
如果有依赖参数，模块加载器不应该在工厂方法中扫描依赖性。

define.amd 属性

为了清晰的标识全局函数(为浏览器加载 script 必须的)遵从 AMD 编程接口，任何全局函数应该有一个"amd"的属性，它的值为一个对象。这样可以防止与现有的定义了 define 函数但不遵从 AMD 编程接口的代码相冲突。

当前，define.amd 对象的属性没有包含在本规范中。实现本规范的作者，可以用它通知超出本规范编程接口基本实现的额外能力。

define.amd 的存在表明函数遵循本规范。如果有另外一个版本的编程接口，那么应该定义另外一个属性，如 define.amd2，表明实现只遵循该版本的编程接口。

一个如果定义同一个环境中允许多次加载同一个版本的模块的实现:

```
define.amd = {
    multiversion: true
}
```

最简单的定义:

```
define.amd = {}
```

一次输出多个模块
在一个脚本中可以使用多次 define 调用。这些 define 调用的顺序不应该是重要的。早一些的模块定义中所指定的依赖，可以在同一脚本中晚一些定义。模块加载器负责延迟加载未接解决的依赖，直到全部脚本加载完毕，防止没必要的请求。
例子
使用 require 和 exports

创建一个名为"alpha"的模块，使用了 require, exports 和名为"beta"的模块:

```
define("alpha", ["require", "exports", "module"], function (require, exports, beta) {
    exports.verb = function () {
        return beta.verb()
        // Or:
        return requre("beta").verb()
    }
})
```

一个返回对象的匿名模块:

```
define(["alpha"], function (alpha) {
    return {
        verb: function () {
            return alpha.verb() + 2
        }
    }
})
```

一个没有依赖性的模块可以直接定义对象:

```
define({
    add: function (x, y) {
        return x + y
    }
})
```

一个使用了简单 CommonJS 转换的模块定义:

```
define(function(require, exports, module) {
    var a = require("a")
        b = require("b")

    exports.action = function () {}
})
```

全局变量

本规范保留全局变量"define"以用来实现本规范。包额外信息异步定义编程接口是未将来的 CommonJS API 保留的。模块加载器不应在此函数添加额外的方法或属性。

本规范保留全局变量"require"被模块加载器使用。模块加载器可以在合适的情况下自由地使用该全局变量。它可以使用这个变量或添加任何属性以完成模块加载器的特定功能。它同样也可以选择完全不使用"require"。

UMD（2011）

模块格式的明显对抗甚至在 AMD 与 CommonJS 模块分离之前就开始了。当时 AMD 阵营已经有很多开发人员喜欢使用模块化代码的最小入门门槛。由于 Node.JS 的日益普及和 Browserify 的出现，CommonJS 模块的支持者数量也迅速增长。

所以有两种格式，彼此无法相处。在没有代码修改的情况下，AMD 模块不能用于实现 CommonJS 模块规范的环境中。CommonJS 模块不能与使用 AMD 作为主要格式（require.js，curl.js）的加载器一起使用。整个 JavaScript 生态系统都是一个糟糕的情况。

已经开发了 UMD 格式来解决该问题。UMD 代表通用模块定义，因此这种格式允许您使用与 AMD 工具相同的模块以及 CommonJS 环境。

ECMA2015 模块（2015）

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，旨在成为浏览器和服务器通用的模块解决方案。其模块功能主要由两个命令构成：export 和 import。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。其实 ES6 还提供了 export default 命令，为模块指定默认输出，对应的 import 语句不需要使用大括号。这也更趋近于 ADM 的引用写法。

```
/** export default **/
//定义输出
export default { basicNum, add };
//引入
import math from './math';
function test(ele) {
    ele.textContent = math.add(99 + math.basicNum);
}
```

ES6 的模块不是对象，import 命令会被 JavaScript 引擎静态分析，在编译时就引入模块代码，而不是在代码运行时加载，所以无法实现条件加载。也正因为这个，使得静态分析成为可能。

ES6 模块与 CommonJS 模块的差异

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。CommonJS 模块输出的是值的拷贝，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。

ES6 模块的运行机制与 CommonJS 不一样。JS 引擎对脚本静态分析的时候，遇到模块加载命令 import，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。换句话说，ES6 的 import 有点像 Unix 系统的“符号连接”，原始值变了，import 加载的值也会跟着变。因此，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。

2. CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。

运行时加载: CommonJS 模块就是对象；即在输入时是先加载整个模块，生成一个对象，然后再从这个对象上面读取方法，这种加载称为“运行时加载”。

编译时加载: ES6 模块不是对象，而是通过 export 命令显式指定输出的代码，import 时采用静态命令的形式。即在 import 时可以指定加载某个输出值，而不是加载整个模块，这种加载称为“编译时加载”。

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。

模块化的扩展

从语言层面到文件层面的模块化。
从 1999 年开始，模块化探索都是基于语言层面的优化，真正的革命从 2009 年 CommonJS 的引入开始，前端开始大量使用预编译。

模块化历史的方案都是逻辑模块化，从 CommonJS 方案开始前端把服务端的解决方案搬过来之后，算是看到标准物理与逻辑统一的模块化。但之后前端工程不得不引入模块化构建这一步。正是这一步给前端开发无疑带来了诸多的不便，尤其是现在我们开发过程中经常为了优化这个工具带了很多额外的成本。

为什么模块化方案这么晚才成型，可能早期应用的复杂度都在后端，前端都是非常简单逻辑。后来 Ajax 火了之后，web app 概念的开始流行，前端的复杂度也呈指数级上涨，到今天几乎和后端接近一个量级。工程发展到一定阶段，要出现的必然会出现。

前端三剑客的模块化展望
从 js 模块化发展史，我们还看到了 css html 模块化方面的严重落后，如今依赖编译工具的模块化增强在未来会被标准所替代。

原生支持的模块化，解决 html 与 css 模块化问题正是以后的方向。

再回到 JS 模块化这个主题，开头也说到是为了构建 scope，实则提供了业务规范标准的输入输出的方式。但文章中的 JS 的模块化还不等于前端工程的模块化，Web 界面是由 HTML、CSS 和 JS 三种语言实现，不论是 CommonJS 还是 AMD 包括之后的方案都无法解决 CSS 与 HTML 模块化的问题。

对于 CSS 本身它就是 global scope，因此开发样式可以说是喜忧参半。近几年也涌现把 HTML、CSS 和 JS 合并作模块化的方案，其中 react/css-modules 和 vue 都为人熟知。当然，这一点还是非常依赖于 webpack/rollup 等构建工具，让我们意识到在 browser 端还有很多本质的问题需要推进。

对于 css 模块化，目前不依赖预编译的方式是 styled-component，通过 js

动态创建 class。而目前 css 也引入了与 js 通信的机制 与 原生变量支持。未来 css 模块化也很可能是运行时的，所以目前比较看好 styled-component 的方向。

对于 html 模块化，小尤最近爆出与 chrome 小组调研 html Modules，如果 html 得到了浏览器，编辑器的模块化支持，未来可能会取代 jsx 成为最强大的模块化、模板语言。

对于 js 模块化，最近出现的

<script type="module">
模块化加载方式，虽然还没有得到浏览器原生支持，但也是我比较看好的未来趋势，这样就连 webpack 的拆包都不需要了，直接把源代码传到服务器，配合 http2.0 完美抛开预编译的枷锁。

上述三中方案都不依赖预编译，分别实现了 html、css、js 模块化，相信这就是未来。

模块化标准推进速度仍然缓慢
2015 年提出的标准，在 17 年依然没有得到实现，即便在 nodejs 端。

这几年 TC39 对语言终于重视起来了，慢慢有动作了，但针对模块标准制定的速度，与落实都非常缓慢，与 javascript 越来越流行的趋势逐渐脱节。nodejs 至今也没有实现 ES2015 模块化规范，所有 jser 都处在构建工具的阴影下。

Http 2.0 对 js 模块化的推动
js 模块化定义的再美好，浏览器端的支持粒度永远是瓶颈，http 2.0 正是考虑到了这个因素，大力支持了 ES 2015 模块化规范。

幸运的是，模块化构建将来可能不再需要。随着 HTTP/2 流行起来，请求和响应可以并行，一次连接允许多个请求，对于前端来说宣告不再需要在开发和上线时再做编译这个动作。

几年前，模块化几乎是每个流行库必造的轮子（YUI、Dojo、Angular），大牛们自己爽的同时其实造成了社区的分裂，很难积累。有了 ES2015 Modules 之后，JS 开发者终于可以像 Java 开始者十年前一样使用一致的

方式愉快的互相引用模块。

不过 ES2015 Modules 也只是解决了开发的问题，由于浏览器的特殊性，还是要经过繁琐打包的过程，等 Import，Export 和 HTTP 2.0 被主流浏览器支持，那时候才是彻底的模块化。

Http 2.0 后就不需要构建工具了吗？
看到大家基本都提到了 HTTP/2，对这项技术解决前端模块化及资源打包等工程问题抱有非常大的期待。很多人也认为 HTTP/2 普及后，基本就没有 Webpack 什么事情了。

不过 Webpack 作者 @sokra 在他的文章 webpack & HTTP/2 里提到了一个新的 Webpack 插件 AggressiveSplittingPlugin。简单的说，这款插件就是为了充分利用 HTTP/2 的文件缓存能力，将你的业务代码自动拆分成若干个数十 KB 的小文件。后续若其中任意一个文件发生变化，可以保证其他的小 chunck 不需要重新下载。

可见，即使不断的有新技术出现，也依然需要配套的工具来将前端工程问题解决方案推向极致。

模块化是大型项目的银弹吗？
只要遵循了最新模块化规范，就可以使项目具有最好的可维护性吗？ Js 模块化的目的是支持前端日益上升的复杂度，但绝不是唯一的解决方案。

分析下 JavaScript 为什么没有模块化，为什么又需要模块化：这个 95 年被设计出来的时候，语言的开发者根本没有想到它会如此的大放异彩，也没有将它设计成一种模块化语言。按照文中的说法，99 年也就是 4 年后开始出现了模块化的需求。如果只有几行代码用模块化是扯，初始的 web 开发业务逻辑都写在 server 端，js 的作用小之又小。而现在 spa 都出现了，几乎所有的渲染逻辑都在前端，如果还是没有模块化的组织，开发过程会越来越难，维护也是更痛苦。

文中已经详细说明了模块化的发展和优劣，这里不准备做过多的讨论。我想说的是，在模块化之后还有一个模块间耦合的问题，如果模块间耦合度大也会降低代码的可重用性或者说复用性。所以也出现了降低耦合的观察者模式或者发布/订阅模式。这对于提升代码重用，复用性和避免单点故障等都很重要。