==

[模块化编程 1](#_Toc27129)

[从[JS函数]到[JS模块化编程]---[模块化编程的思想与实现] 1](#_Toc28629)

[JS模块加载器加载原理 2](#_Toc9852)

[异步模块定义（Asynchronous Module Definition） 2](#_Toc21316)

[模块化编程的学习资料 2](#_Toc7779)

[Requirejs 3](#_Toc15758)

[requireJs的简单使用 3](#_Toc16297)

[require 的定义与加载 3](#_Toc14067)

[加载非规范的模块 4](#_Toc17140)

[例子一个 4](#_Toc18755)

[使用requireJS的shim参数，完成jquery插件的加载 4](#_Toc30606)

[使用requireJS加载不符合AMD规范的js文件：shim的使用方式和实现原理 5](#_Toc10354)

[前端模块管理器简介--阮一峰 2014年9月14日 6](#_Toc9257)

[sea.js 6](#_Toc207)

[基础使用[快速入门] 6](#_Toc19117)

[define() CMD模块定义规范 6](#_Toc8206)

[seajs.config() 7](#_Toc20396)

[seajs.use() 7](#_Toc6017)

[ID 和路径匹配原则 #930 7](#_Toc18983)

[运行原理 8](#_Toc10547)

[使用-加载方式[同步][异步] 9](#_Toc9930)

[实战 9](#_Toc7874)

[[ CMD | AMD ] [ Modules/Transport Modules/Wrappings ] 9](#_Toc25041)

[cmd规范与amd规范 9](#_Toc24623)

[CommonJS 的 Modules/Transport 和 Modules/Wrappings 规范有什么区别 10](#_Toc29991)

[关于 CMD的合并、 10](#_Toc4600)

[ID 和路径匹配原则 10](#_Toc17545)

[CMD 模块的构建[关于 id 与 dependencies ] 11](#_Toc32056)

[传统 js合并与 seajs文件合并 12](#_Toc24496)

## 模块化编程

|  |  |
| --- | --- |
| 从[JS函数]到[JS模块化编程]---[模块化编程的思想与实现] | |
| 原始写法 | function m1(){} function m2(){} |
|  | 函数m1()和m2()，组成一个模块。使用的时候，直接调用就行了。  这种做法的缺点很明显："污染"了全局变量，无法保证不与其他模块发生变量名冲突，而且模块成员之间看不出直接关系。 |
| 对象写法 | var module1 = new Object({  　　　　\_count : 0,  　　　　m1 : function (){ 　　　　　　//... 　　　　},  　　　　m2 : function (){ 　　　　　　//... 　　　　}  }); |
|  | 上面的函数m1()和m2(），都封装在module1对象里。使用的时候，就是调用这个对象的属性。  　　module1.m1();  但是，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。比如，外部代码可以直接改变内部计数器的值。  　　module1.\_count = 5; |
| 立即函数 | var module1 = (function(){  　　　　var \_count = 0;  　　　　var m1 = function(){ 　　　　　　//... 　　　　};  　　　　var m2 = function(){ 　　　　　　//... 　　　　};  　　　　return { 　　　　　　m1 : m1, 　　　　　　m2 : m2 　　　　};  　　})(); |
|  | 使用"[立即执行函数](http://benalman.com/news/2010/11/immediately-invoked-function-expression/" \t "http://www.ruanyifeng.com/blog/2012/10/_blank)"（Immediately-Invoked Function Expression，IIFE），可以达到不暴露私有成员的目的。  使用上面的写法，外部代码无法读取内部的\_count变量。  console.info(module1.\_count); //undefined  module1就是Javascript模块的基本写法。下面，再对这种写法进行加工。 |
| 放大模式 | var module1 = (function (mod){  　　　　mod.m3 = function () { 　　　　　　//... 　　　　};  　　　　return mod;  　})(module1); |
|  | 如果一个模块很大，必须分成几个部分，或者一个模块需要继承另一个模块，这时就有必要采用"放大模式"（augmentation）。 |
| 宽放大模式 | var module1 = ( function (mod){  　　　　//...  　　　　return mod;  })(window.module1 || {}); |
|  | 在浏览器环境中，模块的各个部分通常都是从网上获取的，有时无法知道哪个部分会先加载。如果采用上一节的写法，第一个执行的部分有可能加载一个不存在空对象，这时就要采用"宽放大模式"。  与"放大模式"相比，＂宽放大模式＂就是"立即执行函数"的参数可以是空对象。 |
| 输入全局变量 | var module1 = (function ($, YAHOO) {  　　　　//...  　})(jQuery, YAHOO); |
|  | 独立性是模块的重要特点，模块内部最好不与程序的其他部分直接交互。  为了在模块内部调用全局变量，必须显式地将其他变量输入模块。  上面的module1模块需要使用jQuery库和YUI库，就把这两个库（其实是两个模块）当作参数输入module1。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。这方面更多的讨论，参见Ben Cherry的著名文章[《JavaScript Module Pattern: In-Depth》](http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth" \t "http://www.ruanyifeng.com/blog/2012/10/_blank)。 |
| 模块的规范  [node.js]  [comment.js] | 因为有了模块，我们就可以更方便地使用别人的代码，想要什么功能，就加载什么模块。  但是，这样做有一个前提，那就是大家必须以同样的方式编写模块，否则你有你的写法，我有我的写法，岂不是乱了套！考虑到Javascript模块现在还没有官方规范，这一点就更重要了。  目前，通行的Javascript模块规范共有两种：[CommonJS](http://wiki.commonjs.org/wiki/Modules/1.1" \t "http://www.ruanyifeng.com/blog/2012/10/_blank)和[AMD](https://github.com/amdjs/amdjs-api/wiki/AMD" \t "http://www.ruanyifeng.com/blog/2012/10/_blank)。我主要介绍AMD，但是要先从CommonJS讲起。  2009年，美国程序员Ryan Dahl创造了[node.js](http://nodejs.org/" \t "http://www.ruanyifeng.com/blog/2012/10/_blank)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。  node.js的[模块系统](http://nodejs.org/docs/latest/api/modules.html" \t "http://www.ruanyifeng.com/blog/2012/10/_blank)，就是参照[CommonJS](http://wiki.commonjs.org/wiki/Modules/1.1" \t "http://www.ruanyifeng.com/blog/2012/10/_blank)规范实现的。在CommonJS中，有一个全局性方法require()，用于加载模块。假定有一个数学模块math.js，就可以像下面这样加载。  　　var math = require('math');  然后，就可以调用模块提供的方法：  　　var math = require('math');  　　math.add(2,3); // 5  因为这个系列主要针对浏览器编程，不涉及node.js，所以对CommonJS就不多做介绍了。我们在这里只要知道，require()用于加载模块就行了。  有了服务器端模块以后，很自然地，大家就想要客户端模块。而且最好两者能够兼容，一个模块不用修改，在服务器和浏览器都可以运行。  但是，由于一个重大的局限，使得CommonJS规范不适用于浏览器环境。还是上一节的代码，如果在浏览器中运行，会有一个很大的问题，你能看出来吗？  　　var math = require('math');  　　math.add(2, 3);  第二行math.add(2, 3)，在第一行require('math')之后运行，因此必须等math.js加载完成。也就是说，如果加载时间很长，整个应用就会停在那里等。  这对服务器端不是一个问题，因为所有的模块都存放在本地硬盘，可以同步加载完成，等待时间就是硬盘的读取时间。但是，对于浏览器，这却是一个大问题，因为模块都放在服务器端，等待时间取决于网速的快慢，可能要等很长时间，浏览器处于"假死"状态。  因此，浏览器端的模块，不能采用"同步加载"（synchronous），只能采用"异步加载"（asynchronous）。这就是AMD规范诞生的背景。 |
| 模块的规范  [AMD]  [] | Asynchronous Module Definition---异步模块定义  它采用异步方式加载模块，模块的加载不影响它后面语句的运行。所有依赖这个模块的语句，都定义在一个回调函数中，等到加载完成之后，这个回调函数才会运行。  AMD也采用require()语句加载模块，但是不同于CommonJS，它要求两个参数：  　　require([module], callback);  第一个参数[module]，是一个数组，里面的成员就是要加载的模块；第二个参数callback，则是加载成功之后的回调函数。如果将前面的代码改写成AMD形式，就是下面这样：  　　require(['math'], function (math) {  　　　　math.add(2, 3);  　　});  math.add()与math模块加载不是同步的，浏览器不会发生假死。所以很显然，AMD比较适合浏览器环境。  目前，主要有两个Javascript库实现了AMD规范：[require.js](http://requirejs.org/" \t "http://www.ruanyifeng.com/blog/2012/10/_blank)和[curl.js](https://github.com/cujojs/curl" \t "http://www.ruanyifeng.com/blog/2012/10/_blank)。 |
|  |  |
| JS模块加载器加载原理 | |
| www | 作者：JZeng 链接：https://www.zhihu.com/question/21157540/answer/33583597 |
|  | id即路径 原则。  通常我们的入口是这样的： require( [ 'a', 'b' ], callback ) 。这里的 'a'、'b' 都是 ModuleId。通过 id 和路径的对应原则，加载器才能知道需要加载的 js 的路径。在这个例子里，就是 baseUrl + 'a.js' 和 baseUrl + 'b.js'。  但 id 和 path 的对应关系并不是永远那么简单，比如在 AMD 规范里就可以通过配置 Paths 来给特定的 id 指配 path。  createElement('script') & appendChild  知道路径之后，就需要去请求。一般是通过 createElement('script') & appendChild 去请求。这个大家都知道，不多说。有时候有的加载器也会通过 AJAX 去请求脚本内容。  一般来说，需要给 <script> 设置一个属性用来标识模块 id, 作用后面会提到。  document.currentScript  a.js 里可能是 define( id, factory ) 或者是 define( factory )，后者被称为匿名模块。那么当 define(factory) 被执行的时候，我们怎么知道当前被定义的是哪个模块呢，具体地说，这个匿名模块的实际模块 id 是什么？ 答案是通过 document.currentScript 获取当前执行的<script>，然后通过上面给 script 设置的属性来得到模块 id。  需要注意的是，低级浏览器是不支持 currentScript 的，这里需要进行浏览器兼容。在高级浏览器里面，还可以通过 script.onload 来处理这个事情。  依赖分析  在继续讲之前，需要先简单介绍下模块的生命周期。模块在被 Define 之后并不是马上可以用了，在你执行它的 factory 方法来生产出最终的 export 之前，你需要保证它的依赖是可用的。那么首先就要先把依赖分析出来。  简单来说，就是通过 toString 这个方法得到 factory 的内容，然后用正则去匹配其中的 require( 'moduleId' )。当然也可以不用正则。  这就是为什么 require( var ); 这种带变量的语句是不被推荐的，因为它会影响依赖分析。如果一定要用变量，可以用 require( [ var ] ) 这种异步加载的方式。  递归加载  在分析出模块的依赖之后，我们需要递归去加载依赖模块。用伪代码来表达大概是这样的：  Module.prototype.load = function () {  var deps = this.getDeps();  for (var i = 0; i < deps.length; i++) {  var m = deps[i];  if (m.state < STATUS.LOADED) {  m.load();  }  }  this.state = STATUS.LOADED;  }  上面的代码只是表达一个意思，实际上 load 方法很可能是异步的，所以递归的返回要特殊处理下。  实现一个可用的加载器并没有那么简单，比如你要处理循环依赖，还有各种各样的牵一发动全身的细节。但要说原理，大概就是这么几条。个人觉得，比起照着规范实现一个加载器，更加吸引人的是 AMD 或者 CommonJS 这些规范的完善和背后的设计思路。 |
|  |  |

### 异步模块定义（Asynchronous Module Definition）

|  |
| --- |
| 异步模块定义（Asynchronous Module Definition，通常称为AMD）已被设计为适合于浏览器的模块格式。  AMD的核心是define函数。调用define函数最常见的方式是传入三个参数——模块名（也就是说不再与文件名绑定）、该模块依赖的模块标识符数组、以及将会返回该模块定义的工厂函数。  define(**'cal'**,[**'adder'**],**function**(adder){  **return**{  add:**function**(c1,c2){  **return** adder.add(c1,c2);  }  } }); //实际调用的是adder(加法器)模块的add方法 //而且adder模块已经在['adder']中指明  由于此模块的定义包在define函数的调用中，因此这意味着可以欣然将多个模块都放在单个js文件中。此外，由于当调用模块工厂函数define时，模块加载器已拥有控制权，因此它可以自行安排时间去解决（模块间的）依赖关系——对于那些需要先异步下载的模块，真可谓得心应手。  为了与原先的CommonJS模块提案保持兼容已做出了巨大的努力。有些特殊行为是为了能在模块工厂函数中使用require及exports，这意味着，那些传统的CommonJS模块可直接拿来用。  看起来AMD正在成为颇受欢迎的组织客户端JavaScript应用程序的方式。无论是如RequireJS或curl.js等模块资源加载器，还是像Dojo等最近已支持AMD的JavaScript应用程序，情况都是如此。 |
| 模块化编程的学习资料 《了解模块化开发》 http://heeroluo.net/Article/Detail/111 很有意思。  Feeldesign上的专门讲模块化开发的系列文章：  JavaScript模块化开发（一）——基础知识  JavaScript模块化开发（二）——CommonJS规范  JavaScript模块化开发（三）——AMD规范  JavaScript模块化开发（四）——RequireJS  JavaScript模块化开发（五）——延伸阅读  阮一峰的网络日志上发表的三篇讲模块化开发的文章，相当经典：  Javascript模块化编程（一）：模块的写法  Javascript模块化编程（二）：AMD规范  Javascript模块化编程（三）：require.js的用法  附SeaJS，RequireJS： SeaJS 的介绍 http://cyj.me/why-seajs/zh/  RequireJS 网站上讲技术思想的两篇文章 http://cyj.me/why-seajs/requirejs/ |
| 前端模块化开发的价值:https://github.com/seajs/seajs/issues/547  [命名冲突][文件的依赖关系]---[sea.js来进行解决]  [实现原因] |

## Requirejs

|  |  |
| --- | --- |
| requireJs的简单使用 | |
| 阮一峰 | [第一部分](http://www.ruanyifeng.com/blog/2012/10/javascript_module.html" \t "http://blog.sina.com.cn/s/_blank) [第二部分](http://www.ruanyifeng.com/blog/2012/10/asynchronous_module_definition.html" \t "http://blog.sina.com.cn/s/_blank) [第三部分](http://www.ruanyifeng.com/blog/2012/11/require_js.html) |
| 说明与作用 | 依次加载多个js文件，这样的写法有很大的缺点。  首先，加载的时候，浏览器会停止网页渲染，加载文件越多，网页失去响应的时间就会越长；  其次，由于js文件之间存在依赖关系，因此必须严格保证加载顺序（比如上例的1.js要在2.js的前面），依赖性最大的模块一定要放到最后加载，当依赖关系很复杂的时候，代码的编写和维护都会变得困难。  require.js的诞生，就是为了解决这两个问题：  　　（1）实现js文件的异步加载，避免网页失去响应；   1. 管理模块之间的依赖性，便于代码的编写和维护。   有效地防止域名冲突  声明不同js文件之间的依赖  可以让我们的代码以模块化的方式组织，方便我们的复用，分开进行功能模块与引用模块的编程。 |
| 常用的方法 | require.config()  [为我们的模块指定别名，方便模块的引入]  require  [实现模块的引入，并通过它编写程序的主代码]  define  [利用他来编写模块，然后在相应的地方进行引入] |
| 简单的demo | <**script** src="js/require.js" data-main="js/main"></**script**> **<!--data-main定义入口文件，就是requirejs加载完毕后会立即调用的文件-->** |
|  | require.config({  **paths**:{  **"jquery"**:**"jquery.min"** } }); "jquery": "https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min" //引入我们定义好的模块 //参数1：数组:我们需要引入的模块名(jquery必须取名为这个，因为其已经在函数中国进行了adm设置) //参数2：回调函数,参数代表模块 require([**"jquery"**],**function**($){  $(**"body"**).css(**"background"**,**"red"**); }); |
| require 的定义与加载 | |
|  | require.js加载的模块，采用AMD规范。也就是说，模块必须按照AMD的规定来写。  具体来说，就是模块必须采用特定的define()函数来定义。如果一个模块不依赖其他模块，那么可以直接定义在define()函数之中。  假定现在有一个math.js文件，它定义了一个math模块。那么，math.js就要这样写：  　　// math.js  　　define(function (){  　　　　var add = function (x,y){  　　　　　　return x+y;  　　　　};  　　　　return {  　　　　　　add: add 　　　　};  });  加载方法如下：  　　// main.js  　　require(['math'], function (math){  　　　　alert(math.add(1,1));  });  如果这个模块还依赖其他模块，那么define()函数的第一个参数，必须是一个数组，指明该模块的依赖性。  　　define(['myLib'], function(myLib){  　　　　function foo(){  　　　　　　myLib.doSomething();  　　　　}  　　　　return {  　　　　　　foo : foo  　　　　};  　　});  当require()函数加载上面这个模块的时候，就会先加载myLib.js文件。 |
| -0-  无定义直接使用 | [使用写法1]  require([**"jquery"**],**function**($){  $(**"body"**).css(**"background"**,**"red"**);  **function** a(){}  **function** b(){}  **function** c(){} }); |
| 关于自定义模块的定义的使用  -1-  依赖于其他模块 | [自定义写法1]  //自定义一个模块：基于jquery **define**([**'jquery'**],**function**($){  **function** a(){}  **function** b(){}  **function** c(){} });  [自定义写法2]  //自定义一个模块：基于jquery **define**([**'jquery'**],**function**($){  **return**{  a:**function**(){},  b:**function**(){},  c:**function**(str1,str2){  **return** str1===str2;  }  } });  [使用写法1]---直接调用路径使用  require.config({  **paths**:{  **"jquery"**:**"module/jquery-1.8.3"** } });  require([**"jquery"**,**"module/myModule"**],**function**($,mm){  $(**"body"**).css(**"background"**,**"red"**);  alert(mm.c(1,2)) });  [使用写法2]---先配置别名、路径，再使用  require.config({  **paths**:{  **"jquery"**:**"module/jquery-1.8.3"**,  **"myModule"**:**"module/myModule"** } }); require([**"jquery"**,**"myModule"**],**function**($,mm){  $(**"body"**).css(**"background"**,**"red"**);  alert(mm.c(1,2)) }); |
| 关于自定义模块的定义的使用  -2-  不依赖与其他模块 | //main.js  require([**"module/math"**],**function**(math){  alert(math.add(1,2)) }); |
| 加载非规范的模块 | |
| 理论上，require.js加载的模块，必须是按照AMD规范、用define()函数定义的模块。但是实际上，虽然已经有一部分流行的函数库（比如jQuery）符合AMD规范，更多的库并不符合。那么，require.js是否能够加载非规范的模块呢？  回答是可以的。  这样的模块在用require()加载之前，要先用require.config()方法，定义它们的一些特征。  这时候就需要用到另一个功能：shim，shim解释起来也比较难理解，shim直接翻译为"垫"，其实也是有这层意思的，目前我主要用在两个地方 1. 非AMD模块输出，将非标准的AMD模块"垫"成可用的模块，例如：在老版本的jquery中，是没有继承AMD规范的，所以不能直接require["jquery"],这时候就需要shim，比如我要是用underscore类库，但是他并没有实现AMD规范，那我们可以这样配置  require.config({  shim: {  "underscore" : {  exports : "\_";  }  } })  这样配置后，我们就可以在其他模块中引用underscore模块：  require(["underscore"], function(\_){  \_.each([1,2,3], alert); })  2.插件形式的非AMD模块，我们经常会用到jquery插件，而且这些插件基本都不符合AMD规范，比如jquery.form插件，这时候就需要将form插件"垫"到jquery中：  require.config({  shim: {  "underscore" : {  exports : "\_";  },  "jquery.form" : {  deps : ["jquery"]  }  } })  也可以简写为：  require.config({  shim: {  "underscore" : {  exports : "\_";  },  "jquery.form" : ["jquery"]  } })  这样配置之后我们就可以使用加载插件后的jquery了  require.config(["jquery", "jquery.form"], function($){  $(function(){  $("#form").ajaxSubmit({...});  }) })  举例来说，underscore和backbone这两个库，都没有采用AMD规范编写。如果要加载它们的话，必须先定义它们的特征。  　　require.config({  　　　　shim: { 　　　　　　'underscore':{ 　　　　　　　　exports: '\_' 　　　　　　},  　　　　　　'backbone': { 　　　　　　　　deps: ['underscore', 'jquery'], 　　　　　　　　exports: 'Backbone' 　　　　　　}  　　　　}  　　});  require.config()接受一个配置对象，这个对象除了有前面说过的paths属性之外，还有一个shim属性，专门用来配置不兼容的模块。具体来说，每个模块要定义   1. exports值（输出的变量名），表明这个模块外部调用时的名称；   （2）deps数组，表明该模块的依赖性。  比如，jQuery的插件可以这样定义：  　　shim: {  　　　　'jquery.scroll': {  　　　　　　deps: ['jquery'],  　　　　　　exports: 'jQuery.fn.scroll'  　　　　}  　　} | |
| 例子一个 | |
|  | require.config({  shim:{"name":{deps:['依赖1','依赖2'],export:"名字"}}  paths:{"name":"path"},  }) |
|  |  |
| 使用requireJS的shim参数，完成jquery插件的加载 | |
| 说明 | jquery插件大多都是如下结构：  (function( $, undefined ) {  })( jQuery );  如果我们项目中使用了jquery插件，但是jquery框架是通过requireJS加载的（不会添加全局变量$），那怎么完成jquery插件的加载呢？使用传统的方，在HTML页面中通过<script>加载jquery插件，肯定是不行的。这个时候我们需要使用到  requireJS的shim参数，来完成jquery插件的加载。下面我们以加载jquery-ui的slider插件为例：  在path参数中，我们设置了模块名称(可以随意指定)和js文件路径的映射，然后在shim参数中，指定了模块名称和它的依赖数组，上面我们的jquery插件只依赖于jquery框架。通过这种方式，就可以使用requireJS完成jquery和其插件的加载，不会有全局变量污染问题。 |
|  | **requirejs**.config({  **shim**: {  **'jquery.ui.core'**: [**'jquery'**],  **'jquery.ui.widget'**: [**'jquery'**],  **'jquery.ui.mouse'**: [**'jquery'**],  **'jquery.ui.slider'**: [**'jquery'**]  },  **paths**: {  **jquery**: **'jquery-2.1.1/jquery'**,  **domReady**: **'require-2.1.11/domReady'**,  **'jquery.ui.core'**: **'jquery-ui-1.10.4/development-bundle/ui/jquery.ui.core'**,  **'jquery.ui.widget'**: **'jquery-ui-1.10.4/development-bundle/ui/jquery.ui.widget'**,  **'jquery.ui.mouse'**: **'jquery-ui-1.10.4/development-bundle/ui/jquery.ui.mouse'**,  **'jquery.ui.slider'**: **'jquery-ui-1.10.4/development-bundle/ui/jquery.ui.slider'** } });  **require**([**'jquery'**, **'domReady'**, **'jquery.ui.core'**, **'jquery.ui.widget'**, **'jquery.ui.mouse'**, **'jquery.ui.slider'**], **function** ($) {  $(**"#slider"**).slider({  **value**: 0, **min**: 0, **max**: 4, **step**: 1, slide: **function** (event, ui) {  }  }); }); |
| 使用requireJS加载不符合AMD规范的js文件：shim的使用方式和实现原理 | |
|  | 我们知道在[javascript](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=20&is_app=0&jk=51fc38b054991453&k=javascript&k0=javascript&kdi0=0&luki=2&mcpm=0&n=10&p=baidu&q=65035100_cpr&rb=0&rs=1&seller_id=1&sid=53149954b038fc51&ssp2=1&stid=9&t=tpclicked3_hc&td=1836545&tu=u1836545&u=http://www.bubuko.com/infodetail-671521.html&urlid=0" \t "http://www.bubuko.com/_blank)中定义全局变量有2种方式，本质上是等价的，都是向window对象注入属性或者方法。  // global.js  var g\_name = "aty";  window.g\_age = 25;  当global.js加载的时候，[浏览器](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=20&is_app=0&jk=51fc38b054991453&k=%E4%AF%C0%C0%C6%F7&k0=%E4%AF%C0%C0%C6%F7&kdi0=0&luki=4&mcpm=0&n=10&p=baidu&q=65035100_cpr&rb=0&rs=1&seller_id=1&sid=53149954b038fc51&ssp2=1&stid=9&t=tpclicked3_hc&td=1836545&tu=u1836545&u=http://www.bubuko.com/infodetail-671521.html&urlid=0" \t "http://www.bubuko.com/_blank)的全局对象window就会多出2个属性:g\_name和g\_age。  我们编写一个js工具类或者是js[框架](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=20&is_app=0&jk=51fc38b054991453&k=%BF%F2%BC%DC&k0=%BF%F2%BC%DC&kdi0=0&luki=1&mcpm=0&n=10&p=baidu&q=65035100_cpr&rb=0&rs=1&seller_id=1&sid=53149954b038fc51&ssp2=1&stid=9&t=tpclicked3_hc&td=1836545&tu=u1836545&u=http://www.bubuko.com/infodetail-671521.html&urlid=0" \t "http://www.bubuko.com/_blank)，通常有2种方式： 方式1：dateUtil.js  (function(window) {  var DateUtils = {};  DateUtils.toString = function(){  alert("toString");  };  // 全局变量  window.DateUtils = DateUtils;    })(window);  方式2：stringUtil.js  // 全局变量  var StringUtils = {};  StringUtils.toUpperCase = function(input){  alert("toUpperCase");  } |
| 普通使用方法 | ---index.html  <script data-main="main.js" src="./../requirejs-2.1.15.js"></script>  ---main.js  requirejs.config({  baseUrl: 'libs'  });  require(["dateUtil","stringUtil"], function(dateUtil,stringUtil) {  alert(dateUtil===undefined);//true  });  ---结果：  运行index.html，通过F12观察:  IMG_256  很明显dateUtil.js和stringUtil.js能够被requireJS正常加载，但是不能获取到这2个模块的返回值。我们修改下index.[html](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=20&is_app=0&jk=9c3e29c0d26eaf62&k=html&k0=html&kdi0=0&luki=5&mcpm=0&n=10&p=baidu&q=65035100_cpr&rb=0&rs=1&seller_id=1&sid=62af6ed2c0293e9c&ssp2=1&stid=9&t=tpclicked3_hc&td=1836545&tu=u1836545&u=http://www.bubuko.com/infodetail-671521.html&urlid=0" \t "http://www.bubuko.com/_blank)，给div注册事件处理函数，并在事件处理函数中调用stringUtil.js提供的方法test()，可以发现test()函数不会报错。也就是说，requireJS加载不符合AMD规范的js文件，跟我们直接在[html](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=20&is_app=0&jk=9c3e29c0d26eaf62&k=html&k0=html&kdi0=0&luki=5&mcpm=0&n=10&p=baidu&q=65035100_cpr&rb=0&rs=1&seller_id=1&sid=62af6ed2c0293e9c&ssp2=1&stid=9&t=tpclicked3_hc&td=1836545&tu=u1836545&u=http://www.bubuko.com/infodetail-671521.html&urlid=0" \t "http://www.bubuko.com/_blank)通过<script>[标签](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=20&is_app=0&jk=9c3e29c0d26eaf62&k=%B1%EA%C7%A9&k0=%B1%EA%C7%A9&kdi0=0&luki=6&mcpm=0&n=10&p=baidu&q=65035100_cpr&rb=0&rs=1&seller_id=1&sid=62af6ed2c0293e9c&ssp2=1&stid=9&t=tpclicked3_hc&td=1836545&tu=u1836545&u=http://www.bubuko.com/infodetail-671521.html&urlid=0" \t "http://www.bubuko.com/_blank)加载，没有太大的差别。js文件中引入的全局变量，依然会存在，依然能够正常使用。 |
| shim使用方法1  [无依赖关系] | requirejs.config({  baseUrl: 'libs',  shim:{  dateUtil:{  deps:[],  exports: 'DateUtils'  },  stringUtil:{  deps:[],  exports: 'StringUtils'  }  }  });  require(["dateUtil","stringUtil"], function(dateUtil,stringUtil) {  stringUtil.toUpperCase();  dateUtil.toString();  });  这段代码可以正常运行，可以看到：shim参数能够帮助我们以AMD模块的方式，使用那些不符合AMD规范的模块。下面接介绍下：deps和exports的含义。  exports很好理解，就是模块的返回值。main.js中exports的值，一定要与dateUtil.js和stringUtil.js中暴露出的全局变量名称一致。很显然dateUtil.js和stringUtil.js这2个模块的返回值，就是暴露出的全局变量window.DateUtils和window.StringUtils，requireJS[框架](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=20&is_app=0&jk=9c3e29c0d26eaf62&k=%BF%F2%BC%DC&k0=%BF%F2%BC%DC&kdi0=0&luki=1&mcpm=0&n=10&p=baidu&q=65035100_cpr&rb=0&rs=1&seller_id=1&sid=62af6ed2c0293e9c&ssp2=1&stid=9&t=tpclicked3_hc&td=1836545&tu=u1836545&u=http://www.bubuko.com/infodetail-671521.html&urlid=0" \t "http://www.bubuko.com/_blank)就是将这些全局变量的值返回，作为模块的返回结果。如果dateUtil.js或stringUtil.js中暴露了多个全局变量，那么exports可以指定其中任何的一个，作为模块的返回结果。不过一般的框架，都只会使用1个全局变量，这样冲突的可能性会减少，毕竟全局变量越少越好。 |
| shim使用方法2  [依赖] | //aplugin.js  (function(window,util) {  var a = {};  a.toString = function(){  alert("a="+util.add(1,2));  };  // 全局变量  window.a = a;  })(window,util);  //bplugin.js  var b = {};  b.toString = function(){  alert("b="+util.add(1,2));  }  //util.js  var util = {};  util.[add](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=20&is_app=0&jk=9c3e29c0d26eaf62&k=add&k0=add&kdi0=0&luki=3&mcpm=0&n=10&p=baidu&q=65035100_cpr&rb=0&rs=1&seller_id=1&sid=62af6ed2c0293e9c&ssp2=1&stid=9&t=tpclicked3_hc&td=1836545&tu=u1836545&u=http://www.bubuko.com/infodetail-671521.html&urlid=0" \t "http://www.bubuko.com/_blank) = function(v1,v2){  return v1+v2;  };  //main.js  requirejs.config({  baseUrl: 'libs',  shim:{  dateUtil:{  deps:[],  exports: 'DateUtils'  },  stringUtil:{  deps:[],  exports: 'StringUtils'  },  aplugin:{  deps:["util"],  exports: 'a'  },  bplugin:{  deps:["util"],  exports: 'b'  }  }  });  require(["stringUtil","dateUtil","aplugin","bplugin"], function(string,date) {  //string.toString();  //date.toString();  var aPl = require("aplugin");  var bPl = require("bplugin");  aPl.toString();  bPl.toString();  });  很显然util.js也不符合AMD规范，如果A模块依赖于B模块，A模块不符合AMD规范(使用的是全局变量)，那么B模块也必须是使用全局变量，否则会报错。即如果将util.js改成符合AMD规范的写法，那么aplugin.js和bplugin.js都会因找不到util对象而报错。  // 符合AMD规范的util.js  define(function(){  function [add](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=20&is_app=0&jk=9c3e29c0d26eaf62&k=add&k0=add&kdi0=0&luki=3&mcpm=0&n=10&p=baidu&q=65035100_cpr&rb=0&rs=1&seller_id=1&sid=62af6ed2c0293e9c&ssp2=1&stid=9&t=tpclicked3_hc&td=1836545&tu=u1836545&u=http://www.bubuko.com/infodetail-671521.html&urlid=0" \t "http://www.bubuko.com/_blank)(v1,v2) { return v1+v2; }  return {"add":add};  }); |
|  | 最后我们看下shim配置参数中init的作用。init可以指定一个函数主要就是用来避免类库之间的冲突。由于不符合AMD规范的js文件，会使用全局变量。所以当加载多个模块的时候存在名字冲突的可能。比如JQuery、UnderScore等[框架](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=20&is_app=0&jk=9c3e29c0d26eaf62&k=%BF%F2%BC%DC&k0=%BF%F2%BC%DC&kdi0=0&luki=1&mcpm=0&n=10&p=baidu&q=65035100_cpr&rb=0&rs=1&seller_id=1&sid=62af6ed2c0293e9c&ssp2=1&stid=9&t=tpclicked3_hc&td=1836545&tu=u1836545&u=http://www.bubuko.com/infodetail-671521.html&urlid=0" \t "http://www.bubuko.com/_blank)都会提供一个noConflict()函数来避免名字冲突，noConflict()的实现原理可以[参考这篇文章](http://blog.csdn.net/aitangyong/article/details/44200751" \t "http://www.bubuko.com/_blank)  我们编写一个不符合AMD规范的模块conflict.js，使用了全局变量$E，并提供noConflict方法。  //conflict.js  (function(window) {  // 保存之前数据  var \_$E = window.$E;  var myplugin = {"name":"aty"};  myplugin.noConflict = function(){  window.$E = \_$E;  return myplugin;  };    // 向全局对象注册$E  window.$E = myplugin;  })(window);  //index.html 将index.html修改如下，在requireJS加载之前，先定义一个全局变量$E。  <!doc[type](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=20&is_app=0&jk=9c3e29c0d26eaf62&k=type&k0=type&kdi0=0&luki=7&mcpm=0&n=10&p=baidu&q=65035100_cpr&rb=0&rs=1&seller_id=1&sid=62af6ed2c0293e9c&ssp2=1&stid=9&t=tpclicked3_hc&td=1836545&tu=u1836545&u=http://www.bubuko.com/infodetail-671521.html&urlid=0" \t "http://www.bubuko.com/_blank) [html](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=20&is_app=0&jk=9c3e29c0d26eaf62&k=html&k0=html&kdi0=0&luki=5&mcpm=0&n=10&p=baidu&q=65035100_cpr&rb=0&rs=1&seller_id=1&sid=62af6ed2c0293e9c&ssp2=1&stid=9&t=tpclicked3_hc&td=1836545&tu=u1836545&u=http://www.bubuko.com/infodetail-671521.html&urlid=0" \t "http://www.bubuko.com/_blank)>  <html>  <[head](http://cpro.baidu.com/cpro/ui/uijs.php?adclass=0&app_id=0&c=news&cf=1001&ch=0&di=128&fv=20&is_app=0&jk=9c3e29c0d26eaf62&k=head&k0=head&kdi0=0&luki=9&mcpm=0&n=10&p=baidu&q=65035100_cpr&rb=0&rs=1&seller_id=1&sid=62af6ed2c0293e9c&ssp2=1&stid=9&t=tpclicked3_hc&td=1836545&tu=u1836545&u=http://www.bubuko.com/infodetail-671521.html&urlid=0" \t "http://www.bubuko.com/_blank)>  <title>shim</title>  <meta charset="utf-8">  <script>  var $E = "before";  </script>  <script data-main="main.js" src="./../requirejs-2.1.15.js"></script>  </head>  </html>  //main.js  requirejs.config({  baseUrl: 'libs',  shim:{  conflict:{  deps:[],  exports: '$E',  init:function(){  return $E.noConflict();  }  }  }  });  require(["conflict"], function(mayConflict) {  alert(mayConflict.name);  alert(window.$E);//before  });  运行index.html，可以发现conflict.js能够与之前定义的全局变量$E共存，避免了冲突，这就是通过init实现的。如果没有定义init，可以看到alert(window.$E)打印的值是undefined。 |

### 前端模块管理器简介--[阮一峰](http://www.ruanyifeng.com/) [2014年9月14日](http://www.ruanyifeng.com/blog/2014/09/)

|  |  |
| --- | --- |
|  | <http://www.ruanyifeng.com/blog/2014/09/package-management.html>  浏览器本身并不提供模块管理的机制，为了调用各个模块，有时不得不在网页中，加入一大堆script标签。这样就使得网页体积臃肿，难以维护，还产生大量的HTTP请求，拖慢显示速度，影响用户体验。  为了解决这个问题，前端的模块管理器（package management）应运而生。它可以轻松管理各种JavaScript脚本的依赖关系，自动加载各个模块，使得网页结构清晰合理。不夸张地说，将来所有的前端JavaScript项目，应该都会采用这种方式开发。  最早也是最有名的前端模块管理器，非[RequireJS](http://requirejs.org/" \t "http://www.ruanyifeng.com/blog/2014/09/_blank)莫属。它采用[AMD格式](http://requirejs.org/docs/whyamd.html" \t "http://www.ruanyifeng.com/blog/2014/09/_blank)，异步加载各种模块。具体的用法，可以参考我写的[教程](http://www.ruanyifeng.com/blog/2012/11/require_js.html" \t "http://www.ruanyifeng.com/blog/2014/09/_blank)。Require.js的问题在于各种参数设置过于繁琐，不容易学习，很难完全掌握。而且，实际应用中，往往还需要在服务器端，将所有模块合并后，再统一加载，这多出了很多工作量。  今天，我介绍另外四种前端模块管理器：[Bower](http://bower.io/" \t "http://www.ruanyifeng.com/blog/2014/09/_blank)，[Browserify](http://browserify.org/" \t "http://www.ruanyifeng.com/blog/2014/09/_blank)，[Component](https://github.com/componentjs/component" \t "http://www.ruanyifeng.com/blog/2014/09/_blank)和[Duo](http://duojs.org/" \t "http://www.ruanyifeng.com/blog/2014/09/_blank)。它们各自都有鲜明的特点，很好地弥补了Require.js的缺陷，是前端开发的利器. |

## sea.js

|  |  |
| --- | --- |
| 基础使用[快速入门] | |
| www | <http://www.zhangxinxu.com/sp/seajs/>  <http://seajs.org/docs/#docs>  <https://www.zhihu.com/question/21308647>  <http://island205.github.io/HelloSea.js/reference.html>  <http://natumsol.github.io/2015/12/21/a-mirco-cmd-loader/> |
| 简要介绍 | 在 SeaJS 的世界里，一个文件就是一个模块。所有模块都遵循 [CMD](https://github.com/seajs/seajs/issues/242" \t "http://www.zhangxinxu.com/sp/seajs/_blank) 书写规范：   |  |  | | --- | --- | | hello.js | define(function(require, exports, module) {  var $ = require('jquery');  exports.sayHello = function() {  $('#hello').toggle('slow');  };  }); | | jquery.js | --- |   使用seajs进行加载：   |  |  | | --- | --- | |  | seajs.config({  alias: {  'jquery': 'http://modules.seajs.org/jquery/1.7.2/jquery.js'  }  });  seajs.use(['./hello', 'jquery'], function(hello, $) {  $('#beautiful-sea').click(hello.sayHello);  }); |   SeaJS 具有以下核心特性：  简单一致的模块格式  依赖的自动管理  脚本的异步并行加载  丰富的插件  友好的调试 |
| define() CMD模块定义规范 | |
| 基本使用方式 | www：<https://github.com/seajs/seajs/issues/242>  define [require require.async exports module.exports]  在sea.js中，所有的js模块都遵循 CMD模块定义规范，该规范明确了模块的基本书写格式和基本交互规则。  在CMD规范中，一个模块就是一个文件，其书写格式为： define( factory );  define 是一个全局函数，用来定义模块；  factory 参数可以是字符串、对象、函数；  factory为字符串、对象时，表示模块的接口就是该字符串、对象：  define('I am a template. My name is {{name}}.'); //定义一个模板模块  define({ "foo": "bar" }); //定义一个对象  factory 为对象时，表示是模块的构造方法。通过执行该构造方法，可以得到模块向外提供的接口。factory方法在执行时，默认会传入三个参数：require、exports 和 module：  define(function(require, exports, module) { === }); |
| define 也可以接受两个以上参数：define(id?, deps?, factory)  字符串 id 表示模块标识，数组 deps 是模块依赖。  比如：  define('hello', ['jquery'], function(require, exports, module) {===});  id 和 deps 参数可以省略。  省略时，可以通过构建工具自动生成。  注意：带 id 和 deps 参数的 define 用法不属于 CMD 规范，而属于 [Modules/Transport](https://github.com/cmdjs/specification/blob/master/draft/transport.md) 规范。  判定当前页面是否有 CMD 模块加载器方法：  if (typeof define === "function" && define.cmd) { //有Sea.js 等CMD模块加载器存在 } |
| require | 1. require(id):   是一个方法，接受 [模块标识](https://github.com/seajs/seajs/issues/258) 作为唯一参数，用来获取指定模块的接口;参数需为字符串;   1. require.async(id,callback):   模块内部异步加载一个或多个模块，并在加载完成后执行指定回调;callback 参数可选.  define(function(require) {  var a = require('./a'); // 获取模块 a 的接口  a.doSomething();// 调用模块 a 的方法  // 异步加载一个模块，在加载完成时，执行回调  require.async('./b', function(b) {  b.doSomething();  });  // 异步加载多个模块，在加载完成时，执行回调  require.async(['./c', './d'], function(c, d) {  c.doSomething();  d.doSomething();  });  console.log(require.resolve('./b')); // ==> http://example.com/path/to/b.js  exports.foo = 'bar';// 对外提供 foo 属性  exports.doSomething = function() {}; // 对外提供 doSomething 方法  // 对外提供接口;与 exports 类似，用来在模块内部对外提供接口：  module.exports = {  name: 'a',  doSomething: function() {};  };  console.log(module.uri); // ==> http://example.com/path/to/this/file.js  });   1. require.resolve(id)   使用模块系统内部的路径解析机制来解析并返回模块路径。该函数不会加载模块，只返回解析后的绝对路径。 |
| exports | 1. exports   一个对象，用来向外提供模块接口   |  |  |  | | --- | --- | --- | | A | **B** | C | | define(function(require, exports) {  // 对外提供 foo 属性  exports.foo = 'bar';  // 对外提供 doSomething 方法  exports.doSomething = function() {};  });  给 exports 对象增加成员 | define(function(require) {  // 通过 return 直接提供接口  return {  foo: 'bar',  doSomething: function() {}  };  });  使用 return 直接向外提供接口;当return 语句是模块中的唯一代码，可以简化代码如下： | define(function(require, exports) {  // 错误用法！！!  exports = {  foo: 'bar',  doSomething: function() {}  };  }); | | 提示：exports 仅仅是 module.exports 的一个引用。在 factory 内部给 exports 重新赋值时，并不会改变 module.exports 的值。因此给 exports 赋值是无效的，不能用来更改模块接口。 | define({  foo: 'bar',  doSomething: function() {}  }); | 正确的写法是用 return 或者给 module.exports 赋值：  define(function(require, exports, module) {  // 正确写法  module.exports = {  foo: 'bar',  doSomething: function() {}  };  }); | |
| module | 1. module存储了与当前模块相关联 属性和方法 的 一个对象；  |  |  |  | | --- | --- | --- | | 属性： |  |  | | module.id | String | 模块的唯一标识 | | module.uri | String | 根据模块系统的路径解析规则得到的模块绝对路径 | | 一般情况下（没有在 define 中手写 id 参数时），module.id 的值就是 module.uri，两者完全相同 | | | | module.dependencies | Array | 表示当前模块的依赖 | | module.exports | Object | 当前模块对外提供的接口 | | 传给 factory 构造方法的 exports 参数是 module.exports 对象的一个引用。只通过 exports 参数来提供接口，有时无法满足开发者的所有需求。 比如当模块的接口是某个类的实例时，需要通过module.exports 来实现：  define(**function** (require, exports, module) {  // exports 是 module.exports 的一个引用  **console**.log(module.**exports** === exports); // true  // 重新给 module.exports 赋值  module.**exports** = **new** SomeClass();  // exports 不再等于 module.exports  **console**.log(module.**exports** === exports); // false }); //注意：对 module.exports 的赋值需要同步执行，不能放在回调函数里。下面这样是不行的： // x.js define(**function** (require, exports, module) {  // 错误用法  setTimeout(**function** () {  module.**exports** = {**a**: **"hello"**};  }, 0); }); //在 y.js 里有调用到上面的 x.js: // y.js define(**function** (require, exports, module) {  **var x** = require(**'./x'**);  //无法立刻得到模块 x 的属性 a  **console**.log(**x**.a); // undefined }); | | | |
| seajs.config() | |
| seajs.config(object) | **seajs**.config({  //设置路径，以便在别名alias中调用  **paths**: {  **'gallery'**: **'https://a.alipayobjects.com/gallery'** },   // 当模块标识很长时，可以使用 alias 来简化。设置别名，方便调用。  **alias**: {  **'underscore'**: **'gallery/underscore'** } }); |
| seajs.use() | |
| seajs.use() | 用来在页面中加载一个或多个模块 // 加载一个模块seajs.use('./a'); // 加载一个模块，在加载完成时，执行回调 **seajs**.use(**'./a'**, **function** (a) {  a.doSomething(); }); // 加载多个模块，在加载完成时，执行回调 **seajs**.use([**'./a'**, **'./b'**], **function** (a, b) {  a.doSomething();  b.doSomething(); }); |
| ID 和路径匹配原则 #930 | |
|  | 所谓 ID 和路径匹配原则 是指，使用 seajs.use 或 require 进行引用的文件，如果是具名模块（即定义了 ID 的模块），会把 ID 和 seajs.use 的路径名进行匹配，如果一致，则正确执行模块返回结果。反之，则返回 null。  例如：  seajs.use('lib/jquery', function($) {  // use $  });  或者在模块中 require ：  define(function(require, exports, module) {  var $ = require('lib/jquery');  // use $  }); |
| 当 jQuery 文件定义方式是下面的情况时，上述的变量 $ 能拿到正确的返回结果：  // 文件路径是 lib/jquery.js// ID 和实际路径匹配了（.js 后缀会自动补上）  define('lib/jquery', function(require, exports, module) {  // jquery code  }); |
| 当 jQuery 文件定义方式是下面的情况时，上述的变量 $ 返回 null：  // 文件路径是 lib/jquery.js  // 但是 ID 是 lib/jquery.min.js// ID 和路径不匹配  define('lib/jquery.min', function(require, exports, module) {  // jquery code  }); |
| 而匿名模块始终能正确返回结果：  // lib/jquery.js  // 匿名模块，不需要进行匹配  // 但是文件中只能有一个 define 块！！！  define(function(require, exports, module) {  // jquery code  });  注意这里用于匹配的 ID 都是经过 alias 和 path 解析并且补完后缀之后的。 |
| 运行原理 | |
| www | <http://www.mamicode.com/info-detail-860853.html> |
| 基础概念 | Sea.js采用了和Node相似的CMD规范，使用require、exports和module来组织模块。但Sea.js比起Node的不同点在于，前者的运行环境是在浏览器中，这就导致A依赖的B模块不能同步地读取过来，所以Sea.js比起Node， 除了运行之外，还提供了两个额外的东西：     a.模块的管理     b.模块从服务端的同步  即Sea.js必须分为模块加载期和执行期。   1. 加载期需要将执行期所有用到的模块从服务端同步过来 2. 执行期按照代码的逻辑顺序解析执行模块，与node的运行期没什么区别。 |
| Sea.js需要三个接口：  define 用来wapper模块，指明依赖，同步依赖;  use 用来启动加载期;  require 实际上是加载期到执行期的桥梁。 |
| 模块标识：模块id的标准参考[Module Identifiers](http://wiki.commonjs.org/wiki/Modules/1.1.1" \l "Module_Identifiers)，简单说来就是作为一个模块的唯一标识。 |
| Factory:一个可以产生模块的工厂。node中的工厂就是新的运行时，而在Sea.js中（Tea.js中也同样），factory就是一个函数。这个函数接受三个参数。 |
| 依赖（Dependencies）：依赖就是一个id的数组，即模块所依赖模块的标识 |
| 依赖加载 | 有很多语言都有模块化的结构，比如c/c++的#include语句，Ruby的require语句等等。模块的执行，必然需要其依赖的模块准备就绪才能顺利执行。  c/c++是编译语言，在预编译时，替换#include语句，将依赖的文件内容包含进来，在编译后的执行期，所有的模块才会开始执行；  而Ruby是解释型语言，在模块执行前，并不知道它依赖什么模块，待到执行到require语句时，执行将暂停，从外部读取并执行依赖，然后再回来继续执行当前模块。  JavaScript作为一门解释型语言，在复杂的浏览器环境中，Sea.js是如何处理CMD模块间的依赖的呢？ |
| a.Node的依赖加载原理  node于Ruby类似，当我们使用node usegreet.js来运行这个模块时，实际上node会构建一个运行的上下文，在这个上下文中运行这个模块。运行到 require(‘./greet‘)这句话时，会通过注入的API，在新的上下文中解析greet.js这个模块，然后通过注入的exports或 module这两个关键字获取该模块的接口，将接口暴露出来给usegreet.js使用，即通过greet这个对象来引用这些接口。  node的模块方案的特点如下：     （1）使用require、exports和module作为模块化组织的关键字；     （2）每个模块只加载一次，作为单例存在于内存中，每次require时使用的是它的接口；      （3）require是同步的，通俗地讲，就是node运行A模块，发现需要B模块，会停止运行A模块，把B模块加载好，获取的B的接口，才继续运行A模 块。如果B模块已经加载到内存中了，当然require B可以直接使用B的接口，否则会通过fs模块化同步地将B文件内存，开启新的上下文解析B模块，获取B的API。        注意：实际上node如果通过fs异步的读取文件的话，require也可以是异步的，所以曾经node中有require.async这个API。  b.Sea.js加载原理  由于在浏览器端，采用与node同样的依赖加载方式是不可行的，因为依赖只有在执行期才能知道，但是此时在浏览器端，我们无法像node一样直接同步地读取一个依赖文件并执行！我们只能采用异步的方式。于是Sea.js的做法是，分成两个时期——加载期和执行期:  加载期：即在执行一个模块之前，将其直接或间接依赖的模块从服务器端同步到浏览器端；  执行期：在确认该模块直接或间接依赖的模块都加载完毕之后，执行该模块。 |
|  | 加载期：不难想见，模块间的依赖就像一棵树。启动模块作为根节点，依赖模块作为叶子节点。下面是pixelegos(一个开源项目)的依赖树：  [loadingperiod](http://images0.cnblogs.com/blog/762181/201506/131641082549838.png)  如上图，在页面中通过seajs.use(‘/js/pixelegos‘)调用，目的是执行pixelegos这个模 块。Sea.js并不知道pixelegos还依赖于其他什么模块，只是到服务端加载pixelegos.js，将其加载到浏览器端之后，通过分析发现它 还依赖于其他的模块，于是Sea.js又去加载其他的模块。随着更多的模块同步到浏览器端后，一棵依赖树才慢慢地通过递归显现出来。  执行期：  在执行期，执行也是从根节点开始，本质上是按照代码的顺序结构，对整棵树进行了遍历。有的模块可能已经EXECUTED，而有的还需要执行获取其exports。由于在执行期时，所有依赖的模块都加载好了，所以与node执行过程有点类似。  pixelegos通过同步的require函数获取tool、canvas和menu，后三者同样通过require来执行各自的依赖模块，于是通过这样一个递归的过程，pixelegos就执行完毕了。  打包模块的加载过程：  打包的方式有三种，self,relative和all。  self，只是自己做了transport  relative，将多有相对路径的模块transport，concat  all，包括相对路径模块和库模块（即在seajs-modules文件夹中的），transport，concat  加载方式（以压缩的[pixelegos.js](http://127.0.0.1:81/dist/pixelegos.js%E5%8A%A0%E8%BD%BD%E6%89%A7%E8%A1%8C%EF%BC%8C%E6%89%80%E6%9C%89%E6%89%93%E5%8C%85%E5%9C%A8%E9%87%8C%E9%9D%A2%E7%9A%84%E6%A8%A1%E5%9D%97%E8%A2%ABdefine%EF%BC%9B)为例）    （1）在use时，定义一个匿名的use\_模块，依赖于/dist/pixelegos模块，匿名的use\_模块load依赖，开始加载[pixelegos.js](http://127.0.0.1:81/dist/pixelegos.js)模块；    （2）[pixelegos.js加载执行，所有打包在里面的模块被define；](http://127.0.0.1:81/dist/pixelegos.js%E5%8A%A0%E8%BD%BD%E6%89%A7%E8%A1%8C%EF%BC%8C%E6%89%80%E6%9C%89%E6%89%93%E5%8C%85%E5%9C%A8%E9%87%8C%E9%9D%A2%E7%9A%84%E6%A8%A1%E5%9D%97%E8%A2%ABdefine%EF%BC%9B)    （3）[pixelegos.js的onload回调执行，调用/dist/pixelegos模块的load，加载其依赖模块，但依赖的模块都加载好了；](http://127.0.0.1:81/dist/pixelegos.js%E7%9A%84onload%E5%9B%9E%E8%B0%83%E6%89%A7%E8%A1%8C%EF%BC%8C%E8%B0%83%E7%94%A8/dist/pixelegos%E6%A8%A1%E5%9D%97%E7%9A%84load%EF%BC%8C%E5%8A%A0%E8%BD%BD%E5%85%B6%E4%BE%9D%E8%B5%96%E6%A8%A1%E5%9D%97%EF%BC%8C%E4%BD%86%E4%BE%9D%E8%B5%96%E7%9A%84%E6%A8%A1%E5%9D%97%E9%83%BD%E5%8A%A0%E8%BD%BD%E5%A5%BD%E4%BA%86%EF%BC%9B)    （4）通知匿名的use\_加载完成，开始执行期。 |
| Sea.js的实现方式  核心源码分析 | [module.js](https://github.com/seajs/seajs/blob/master/src/module.js)是Sea.js的核心,Sea.js中为模块定义了六种状态:  FETCHING：开始从服务端加载模块  SAVED：**模块加载**完成  LOADING：**加载依赖模块**中  LOADED：依赖模块加载完成  EXECUTING：**模块执行**中  EXECUTED：模块执行完成   |  | | --- | | Sea.use调用Module.use构造一个没有factory的模块，该模块即为这个运行期的根节点 | | 模块构造完成，则调用mod.load()来同步其子模块；直接跳过fetching这一步；mod.callback也是Sea.js不纯粹的一点，在模块加载完成后，会调用这个callback。  在load方法中，获取子模块，加载子模块，在子模块加载完成后，会触发mod.onload()； | | 模块的状态是最关键的，模块状态的流转决定了加载的行为；  是否触发onload是由模块的\_remian属性来确定，在load和子模块的onload函数中都对\_remain进行了计算，如果为0，则表示模块加载完成，调用onload： | | 模块的\_remain和\_waitings是两个非常关键的属性，子模块通过\_waitings获得父模块，通过\_remain来判断模块是否加载完成。  当这个没有factory的根模块触发onload之后，会调用其方法callback，callback是这样的： | | 这预示着加载期结束，开始执行期；  而执行期相对比较无脑，首先是直接调用根模块依赖模块的exec方法获取其exports，用它们来调用use传经来的callback。而子模块在执行时，都是按照标准的模块解析方式执行的：  注意：var exports = isFunction(factory) ? factory(require, mod.exports = {}, mod) : factory 真的，整个Sea.js就是为了这行代码能够完美运行 | |  | |  | |  | |
| 资源定位 | 资源定位与模块标识相关，而在Sea.js中有三种模块标识：  普通路径：普通路径与网页中超链接一样，相对于当前页面解析。  相对标识：在define的factory中的相对路径（.. .）是相对标识，相对标识相对当前的URI来解析  顶级标识：不以.或者‘/‘开头的模块标识是顶级标识。  获取真实路径：在Sea.js中，使用data.cwd来代表当前页面的目录；使用data.base来代表sea.js的加载地址。 |
| factory依赖分析 | 在Sea.js的API中，define(factory)，并没有指明模块的依赖项，那Sea.js是如何获得的呢。   |  | | --- | | /\*\*  \* util-deps.js - The parser for dependencies  \* ref: tests/research/parse-dependencies/test.html  \*/  var REQUIRE\_RE =  /"(?:\\"|[^"])\*"|‘(?:\\‘|[^‘])\*‘|\/\\*[\S\s]\*?\\*\/|\/(?:\\\/|[^\/\r\n])+\/(?=[^\/])|\/\/.\*|\.\s\*require|(?:^|[^$])\brequire\s\*\(\s\*(["‘])(.+?)\1\s\*\)/g  var SLASH\_RE = /\\\\/g  function parseDependencies(code) {  var ret = []  code.replace(SLASH\_RE, "")  .replace(REQUIRE\_RE, function(m, m1, m2) {  if (m2) {  ret.push(m2)  }  })  return ret  } |   Sea.js就是使用REQUIRE\_RE在factory的源码中匹配出该模块的依赖项。从REQUIRE\_RE这么长的正则来看，这里坑很多；在CommonJS的wrapper方案中可以使用JS语法分析器来获取依赖会更准确。 |
| 使用-加载方式[同步][异步] | |
| www | <http://www.zhangxinxu.com/sp/seajs/docs/zh-cn/bootstrapping.html> |
| 同步加载 | 加载 sea.js 最直接的方式是通过 script 标签来引入。  <script src="http://example.com/libs/seajs/1.1.0/sea.js"></script>  可以将 sea.js 与其他常见类库合并，例如 jquery.js：  <script src="http://example.com/libs/sea-jquery.js"></script>  推荐使用 [nginx concat module](http://code.taobao.org/svn/nginx_concat_module/trunk/README" \t "http://www.zhangxinxu.com/sp/seajs/docs/zh-cn/_blank) 进行自动合并。  <script src="http://example.com/libs/??seajs/1.1.0/sea.js,jquery/1.7.1/jquery.js"></script>  \*\* 注意：通过服务器自动 combo 时，通常需要手动配置 base 路径：  <script src="http://example.com/libs/??seajs/1.1.0/sea.js,jquery/1.7.1/jquery.js"></script>  <script>  seajs.config({ 'base': 'http://example.com/libs/' });  seajs.use('underscore/1.1.6/underscore');  // => http://example.com/libs/underscore/1.1.6/underscore.js  </script> |
| 异步加载 | 可以在页面中内嵌以下代码，来让 sea.js 自身也能通过异步的方式加载：  <script>  // 其他代码  // 下面是 seajs 的异步载入代码：  ;(function(m, o, d, u, l, a, r) {  if(m[d]) return;  function f(n, t) { return function() { r.push(n, arguments); return t; } }  m[d] = a = { args: (r = []), config: f(0, a), use: f(1, a) };  m.define = f(2);  u = o.createElement('script');  u.id = d + 'node';  u.src = 'http://example.com/libs/seajs/1.0.2/sea.js';  l = o.getElementsByTagName('head')[0];  l.insertBefore(u, l.firstChild);  })(window, document, 'seajs');  // 下面立刻就可以调用 seajs 的方法了：  seajs.config({  'base': 'http://example.com/libs/'  });  seajs.use('./init', function(init) {  // do some cool things.  });  </script> |
| use | use  通过 use 方法，可以在页面中加载任意模块：  <script src="http://example.comlibs/seajs/1.1.0/sea.js"></script>  <script>  seajs.use('path/to/init', function(init) {  init.somethingA(/\* You can pass config as parameters here. \*/);  init.somethingB(/\* You can pass config as parameters here. \*/);  });  </script>  callback 参数是可选的：  <script src="http://example.com/libs/seajs/1.1.0/sea.js"></script>  <script>  seajs.use('path/to/init');  </script>  上面的代码还可以通过 data-main 来简化：  <script src="http://example.com/libs/seajs/1.1.0/sea.js"  data-main="path/to/init" >  </script> |
|  |  |
| 实战 | |
| www | sea.js最佳实践一基础框架搭建:  <http://my.oschina.net/centerLife/blog/138251> |
|  | 由于一直在使用，所以了解了下seajs的源代码。这里是我对下面几个问题的理解：  1、seajs的require(XXX)的方法是怎样实现模块加载的？  2、为什么需要预加载？  3、为什么需要构建工具？  4、构建前后的代码究竟有些什么区别，为什么要这么做？  问题1： seajs的require(XXX)的方法是怎样实现模块加载的？  代码逻辑比较绕，对源代码的理解放在文章的末尾，这里先简单梳理下模块加载的逻辑：  1、从seajs.use方法入口，开始加载use到的模块。  2、use到的模块这时mod缓存当中一定是不存在的。seajs创建一个新的mod,赋予一些初始的状态。  3、执行mod.load方法  4、一堆逻辑之后走到seajs.request方法，请求模块文件。模块加载完成之后，执行define方法。  5、define方法分析提取模块的依赖模块，保存起来。缓存factory但不执行。  6、模块的依赖模块再被加载，如果继续有依赖模块，则继续加载。直至所有被依赖的模块都加载完毕。  7、所有的模块加载完毕之后，执行use方法的callback.  8、模块内部逻辑从callback开始执行。require方法在这个过程当中才被执行。  问题2：为什么需要预加载？  我们看到seajs.use方法实际上是在所有依赖模块都加载完了之后才执行callback。可以理解成在业务逻辑代码在执行之前，必须先预加载所有被依赖的模块代码。那么为什么是一个这样必须先做预加载的逻辑？  答案在于逻辑代码里面引用其他模块方法的这个require方法的执行方法：  var mod = require(id);  这个语法决定了mod的取得是个同步执行的过程，如果模块代码在此之前没有被预加载的话，就只能采用异步加载回调的方法来实现了，那么整个seajs的执行逻辑将完全会是另一个样子。因为异步你会搞不懂模块的执行顺序，逻辑会变的难以掌控。  问题3：为什么需要构建工具？  可以看到没有构建前各个依赖模块都是单独加载的。这会产生过多的模块请求，对于页面的加载性能是不利的。构建工具本质上就是为了解决模块合并加载的问题。  问题4：构建前后的代码究竟有些什么区别，为什么要这么做？  构建工具究竟做了些什么。我们说它本质上是为了解决代码合并加载的问题，那么它所做的只是简单的将各个模块文件合并成一个文件？  当然不是。测试一下，你如果只是简单把几个模块文件合并到一个文件以后，会发现这个文件根本没有办法正常执行。  原因在于define方法的实现。  seajs是推崇定义模块的时候只在define方法传入factory参数的。回顾define方法内部，当没有传入id（姑且等同于模块的url）时，会通过getCurrentScript()方法去取得当前正在执行的这个模块文件的url路径，然后把这个路径作为键值与模块本身一起缓存到cachedMods。这里很关键的一点是，整个seajs内部的这个模块缓存机制其实是依赖每个模块的url来做缓存的键值。require(id)方法，归根结底也是通过url键值到。require(id)方法，归根结底也是通过url键值到cachedMods里面去找相应的模块。这个键值不能重复不能出错，不然模块的对应关系就混乱了。如果把a、b、c几个模块文件简单合并到一个目标文件x之后，getCurrentScript()只能获取到x的路径，三个模块的键值就没法做出区别了，执行肯定出错。  所以如果要把几个模块文件合并在一起，就必须为每个模块明确uri。也就是define方法必须都传入id参数。当id传入的时候，seajs会将这个id转换为url用作缓存的键值。  如果只传id和factory，也就是 define(id, factory)，那么deps = undefined，define方法就会去执行parseDependencies(factory.toString())方法提取factory里面的依赖模块，后续会走到解析模块路径，线上单独加载各个模块的逻辑里面去，这个时候就失去了合并加载的意义了。  所以合并加载，define方法必须正确的传入id，deps,factory三个参数才能正确执行。  seajs 所谓CMD的模块定义方法，是提倡大家写模块的阶段都只传factory一个参数的，其他两个参数在后期代码构建的阶段来生成。上面解释了为什么这两参数在构建后是必须的。  至于为什么提倡定义模块的时候只传factory，我看主要是因为手工传入的id和deps参数，极易出错，不便维护。工具可以提高效率并保证参数的正确。  附： 对seajs 主要代码逻辑的理解。  说明：源代码版本是Sea.js 2.3.0  1、先看看define方法做了些什么  Module.define = function (id, deps, factory)  define方法的时候，支持三个参数。其中id，deps是选填的。factory必须。代码里面通过以下逻辑来控制：  IMG_256  但其实deps是必须的，因为seajs必须知道每个模块依赖了哪些模块，不然无法执行加载。  所以，当factory是函数，并且deps没有被主动传入的时候，就需要使用parseDependencies方法来分析出factory当中的依赖模块了。  IMG_257  parseDependencies方法做的事情主要就是用一个正则表达式把函数体里面所有require(XXX)里面的XXX提取出来，这也就是这个函数依赖到的所有模块了。  IMG_258  方法本身不复杂，但是这个正则表达式不简单：  分析完deps之后,将模块定义存入缓存：  IMG_259  注意，我们会发现define方法纯粹只是分析模块、存储模块，并没有执行模块。  2、真正执行模块，是在require方法里面。我们接下来看require。  IMG_260  IMG_261  简而言之require方法就是根据id在define定义存储的模块缓存中找到相应的模块，并执行它，获得模块定义返回的方法：  IMG_262  整个这个大步骤中，有一个很关键的步骤，有必要细说：  Module.get(require.resolve(id))。  require一个模块的时候，首先要找到这个模块。 Module.get方法就起这个作用。  IMG_263  cachedMods里面没有的话，就创建一个新的Module并缓存到cachedMods里面：  IMG_264  define和rquire方法这样看来不算复杂。seajs主要还是模块加载的逻辑有点复杂。  3、seajs真正执行的入口，是use方法：  IMG_265  通过use方法，从这里的ids开始触发模块的加载和执行。  IMG_266  可以看到加载的关键点在mod.load方法。  load方法代码有点长，其中的主要逻辑是：判断mod的当前状态是否为已加载或者加载中。  IMG_267  在Module的舒适化函数中，我们可以看到status默认值是0.  IMG_268  所以没有加载过的新模块，到这里都是： mod.status = STATUS.LOADING 状态设置为加载中，并执行后续加载逻辑。  接来下是获取模块的依赖urls  IMG_269  mod.resolve方法：  IMG_270  Module.resolve方法本质上就是把相对路径、配置的path、别名等转换成一个绝对路径。不贴代码了。  更新模块加载状态。  [IMG_271](http://files.jb51.net/file_images/article/201504/2015041409341658.png" \t "http://www.jb51.net/article/_blank)  加载模块的逻辑：  IMG_272  主要是m.fetch方法，里面其他逻辑这里略过。  IMG_273  可以看到 seajs.request最终会去执行模块文件的加载:  IMG_274  当所有依赖模块加载完了之后，执行mod的onload方法  IMG_275  这里是 mod.onload()方法  IMG_276  到此，seajs的核心逻辑就差不多都看到了。供参考，有理解不到位或者表达不准确的地方，欢迎一起探讨。  以上所述就是本文的全部内容了，希望大家能够喜欢。 |

[ CMD | AMD ] [ Modules/Transport Modules/Wrappings ]

|  |  |  |
| --- | --- | --- |
| cmd规范与amd规范 | | |
| www | | 知乎：玉伯  <https://www.zhihu.com/question/20351507/answer/14859415> |
| 模块化规范 | | AMD 是 RequireJS 在推广过程中对模块定义的规范化产出。  CMD 是 SeaJS 在推广过程中对模块定义的规范化产出:[https://github.com/seajs/seajs/issues/242]  类似的还有 CommonJS Modules/2.0 规范，是 BravoJS 在推广过程中对模块定义的规范化产出。  还有不少⋯⋯  这些规范的目的都是为了 JavaScript 的模块化开发，特别是在浏览器端的。目前这些规范的实现都能达成浏览器端模块化开发的目的。 |
| 书写区别 | | 1. 对于依赖的模块，AMD 是提前执行，CMD 是延迟执行。   不过 RequireJS 从 2.0 开始，也改成可以延迟执行（根据写法不同，处理方式不同）。CMD 推崇 as lazy as possible.  2. CMD 推崇依赖就近，AMD 推崇依赖前置。看代码：   |  | | --- | | // CMD **define**(**function** (require, exports, module) {  **var** a = require('./a')  a.doSomething()  // 此处略去 100 行  **var** b = require('./b') // 依赖可以就近书写  b.doSomething()  // ...  }) | | // AMD 默认推荐的是 **define**(['./a', './b'], **function** (a, b) { // 依赖必须一开始就写好  a.doSomething()  // 此处略去 100 行  b.doSomething()  // ... }) |   虽然 AMD 也支持 CMD 的写法，同时还支持将 require 作为依赖项传递，但 RequireJS 的作者默认是最喜欢上面的写法，也是官方文档里默认的模块定义写法。  3. AMD 的 API 默认是一个当多个用，CMD 的 API 严格区分，推崇职责单一。比如 AMD 里，require 分全局 require 和局部 require，都叫 require。CMD 里，没有全局 require，而是根据模块系统的完备性，提供 seajs.use 来实现模块系统的加载启动。CMD 里，每个 API 都简单纯粹。  4. SeaJS 和 RequireJS 的具体差异：[https://github.com/seajs/seajs/issues/277](//link.zhihu.com/?target=https://github.com/seajs/seajs/issues/277" \t "_blank) |
| AMD/CMD的区别以及我的看法 | | CMD推崇依赖就近，可以把依赖写进你的代码中的任意一行，例：  define(function(require, exports, module) {  var a = require('./a')  a.doSomething()  var b = require('./b')  b.doSomething()}  ) 代码在运行时，首先是不知道依赖的，需要遍历所有的require关键字，找出后面的依赖。具体做法是将function toString后，用正则匹配出require关键字后面的依赖。显然，这是一种牺牲性能来换取更多开发便利的方法。  而AMD是依赖前置的，换句话说，在解析和执行当前模块之前，模块作者必须指明当前模块所依赖的模块，表现在require函数的调用结构上为：  define(['./a','./b'],function(a,b){  a.doSomething()  b.doSomething()}  )  代码在一旦运行到此处，能立即知晓依赖。而无需遍历整个函数体找到它的依赖，因此性能有所提升，缺点就是开发者必须显式得指明依赖——这会使得开发工作量变大，比如：当你写到函数体内部几百上千行的时候，忽然发现需要增加一个依赖，你不得不回到函数顶端来将这个依赖添加进数组。 |
| 依赖 | | 细心的读者可能发现，到目前位置我讨论的AMD和CMD的思想的关于依赖的部分，都只讨论的“硬依赖”，也就是执行前肯定需要的依赖，但是这不是全部的情况。有的时候情况是这样的：  // 函数体内：  if(status){  a.doSomething()  } 在这个函数体内，可能依赖a，也可能不依赖a，我把这种可能的依赖成为“软依赖”。对于软依赖当然可以直接当硬依赖处理，但是这样不经济，因为依赖是不一定的，有可能加载了此处的依赖而实际上没有用上。 对于软依赖的处理，我推荐依赖前置+回调函数的实现形式。上面的例子简单表述如下：  // 函数体内：if(status){  async(['a'],function(a){  a.doSomething()  })} 至此可以对由commonJS衍生出来的方案做出总结了。在浏览器端来设计模块加载机制，需要考虑依赖的问题。 我们先把依赖分为两种，[强依赖/肯定需要] 和[弱依赖/可能需要]。 对于强依赖，如果要性能优先，则考虑参照依赖前置的思想设计你的模块加载器，我个人也更推崇这个方案一些；如果考虑开发成本优先，则考虑按照依赖就近的思想设计你的模块加载器。 对于弱依赖，只需要将弱依赖的部分改写到回调函数内即可。 如果现在我要实现一个模块加载器，我会将强依赖前置，弱依赖采用异步回调函数的形式，其它的方法我认为都只是语法糖而已，仅此就够了。 |
| 扩展阅读 | | [JavaSript模块规范 - AMD规范与CMD规范介绍](http://blog.chinaunix.net/uid-26672038-id-4112229.html):  <http://blog.chinaunix.net/uid-26672038-id-4112229.html>  SeaJS与RequireJS最大的区别:  <https://www.douban.com/note/283566440/>  <http://zccst.iteye.com/blog/2084545>  <http://www.tuicool.com/articles/FvYVzqB>  <https://github.com/seajs/seajs/issues/277>  <http://blog.csdn.net/kongjiea/article/details/48316049> |
| CommonJS 的 Modules/Transport 和 Modules/Wrappings 规范有什么区别 | | |
| www | 作者：玉伯 链接：http://www.zhihu.com/question/20789867/answer/16187950 | |
| 定义 | Modules/Wrappings 一般是指：  define(function(require, exports, module) {  //  })  define 可以换成 module.declare 等等词汇。表示模块书写格式，这些写就好。但书写格式直接上线存在问题，比如不能直接压缩、不能合并等，因此在压缩、合并前，需要先做一些处理，这些处理就是 transport 操作，transport 后的代码格式称之为 Modules/Transport 规范，这一般是通过构建工具自动生成。典型的：  define(id, deps, factory) | |
| 以上是 CommonJS 社区里一部分人的理解。还有一部分觉得，模块在写的时候，就应该带上 id 、deps 等信息，比如：  define("a", ...)  define(["a", "b"], function(a, b) { ... }) // 这就是 AMD 的标准写法  在这部分人心中，不应该存在 Transport 规范，Transport 规范也应该是 Wrappings 规范的一种。 还有一批人的想法是，以上格式都是 Transport 格式，都应该由工具生成。真正的模块书写格式应该是：  var a = require("a")  exports.foo = ... 这部分代表喜欢用自动构建的方式来解决在浏览器上的运行问题，类似 coffee 这种方式。 | |
| 书写与使用 | SeaJS 里，推崇的 Modules/Wrappings 规范是 CMD 规范：  define(function(require, exports, module) {  var a = require("a")  exports.foo = ...  }) 以上直接是由开发者手写的，写完后，可直接不经过任何构建工具就在浏览器上加载运行。 但 CMD 模块在正式上线前，依旧需要通过构建工具先转换为 Modules/Transport 格式：  define("id", ["dep-1", "dep-2"], function(require, exports, module) {  // source code  }) 转换成 Transport 格式后，才能进一步压缩、合并等。 | |
|  | 究竟哪种写法是哪个规范，不同社区有不同看法，CommonJS 社区至今也没统一意见，主要流派就上面三种。 | |

关于 CMD的合并、

|  |  |
| --- | --- |
| ID 和路径匹配原则 | |
| www | https://github.com/seajs/seajs/issues/930 |
|  | 首先声明这是个万人坑，不要怨恨苦恼，不要垂头丧气，一定要相信希望就在前方，未来一片光明！ |
| 不同情况 | 所谓 ID 和路径匹配原则 是指，使用 seajs.use 或 require 进行引用的文件，如果是具名模块（即定义了 ID 的模块），会把 ID 和 seajs.use 的路径名进行匹配，如果一致，则正确执行模块返回结果。反之，则返回 null。例如：  seajs.use('lib/jquery', **function**($) {  // use $ });  或者在模块中 require ：  define(**function**(require, exports, module) {  **var** $ = require('lib/jquery');  // use $ });  当 jQuery 文件是下面的情况时，上述的变量 $ 能拿到正确的返回结果。  // 文件路径是 lib/jquery.js  // ID 和实际路径匹配了（.js 后缀会自动补上） **define**('lib/jquery', **function** (require, exports, module) {  // jquery code });  下面的代码则返回 null：  // 文件路径是 lib/jquery.js // 但是 ID 是 lib/jquery.min.js // ID 和路径不匹配 **define**('lib/jquery.min', **function**(require, exports, module) {  // jquery code });  而匿名模块始终能正确返回结果：  // lib/jquery.js // 匿名模块，不需要进行匹配 // 但是文件中只能有一个 define 块 **define**(**function** (require, exports, module) {  // jquery code });  注意:这里用于匹配的 ID 都是经过 alias 和 path 解析并且补完后缀之后的。 |
| 为什么要有这个原则 | 回答这个问题前，请先阅读这篇文章：[#426](https://github.com/seajs/seajs/issues/426) 。  首先，Sea.js 的模块启动接口秉承的是路径即 ID 的设计原则。seajs.use 的方法的第一个参数被规定为文件路径（而不是 ID），这样的设计减轻了记忆模块 ID 的负担，无论是匿名模块还是具名模块，开发者只需要知道文件放在哪儿就行了。  进一步的，之所以有这个 ID 和路径匹配原则，是因为在 CMD 的书写规范中，一个文件对应一个模块，所有的模块都是匿名模块（即 define(factory) 的形式）。那么当 seajs.use 某模块时，这个模块对应的文件里的唯一的 define 方法理所当然的是这个模块的执行代码，这时可以正确返回结果。  但是在生产环境下，静态文件不可避免地需要进行合并打包或者进行 combo，以优化请求数提高页面性能。这时，一个 js 文件可能有很多 define() 方法。  define(funtion(require, exports, module) {  // module a  });  define(funtion(require, exports, module) {  // module b  });  define(funtion(require, exports, module) {  // module c  });  那么请问，当 seajs.use 这个文件时，应该返回哪个模块？  所以这时候 ID 就派上了用场，我们可以这样写：  // path/a.js  define('path/a', funtion(require, exports, module) {  // module a  });  define('path/b', funtion(require, exports, module) {  // module b  });  define('path/c', funtion(require, exports, module) {  // module c  });  我们定义好每个模块的 id ，在 Sea.js 里，那个和文件路径匹配的 ID 的模块就是这个文件的主模块。此时模块使用方式：  seajs.use('path/a', function(a) {  // got a, not b or c  });  这个原则保证了我们能够自由合并模块来优化性能，[seajs-combo](https://github.com/seajs/seajs-combo) 和 [spm-build](https://github.com/spmjs/spm-build) 的构建机制都是基于此原则。  在 RequireJS 中，也有类似的原则：[http://requirejs.org/docs/errors.html#mismatch](http://requirejs.org/docs/errors.html" \l "mismatch) |
| 其他解决方式 | 可能有人要问为啥一定要把 ID 定为文件路径，Sea.js 不是可以自定义 ID 吗，像下面这样：  define('module-id', funtion(require, exports, module) {  // module id  });  // 然后就可以  seajs.use('module-id', function(Module) {  // Module  });  上面的代码当然可以运行。但是有一点，任何一个模块的运行都涉及到两个步骤：模块定义 和 模块执行，上面的代码两个步骤都包括在内。  而使用了 Sea.js ，我们不希望用户去手动写 script 标签引用模块。希望只需要 seajs.use 模块的文件路径即可（入口唯一）：  seajs.use('path/to/module', function(Module) {  // Module  });  Sea.js 会自动插入 script 标签，完成定义步骤，然后执行模块，拿到模块的输出。所以当一个文件里有多个 define 时，只能用 ID 是否匹配 use 中的路径来判断是否主模块。  当然可以回避掉这个原则，你只需要自己负责模块的定义部分，再自己 seajs.use 之前定义好的模块 ID 就行。  **<!-- 各种模块的定义 define define define -->** <**script** src="http://example.com/modules.js"></**script**> <**script**>  // 这时 use 的第一个参数就可以不必是文件路径了，因为已经有定义好的模块 ID 了  seajs.use('jquery', **function** ($) {  // $  }); </**script**>  或者通过 alias 来帮助 ID 匹配上最终的路径，这样就和 RequireJS 的方案基本一致了。  // lib/jquery-1.7.2.js 的内容如下  define('$', funtion(require, exports, module) {  // jQuery  });  这样就不需要自己去引用上面的文件，可以直接通过 seajs.use 调用。  seajs.config({  alias: {  $: 'lib/jquery-1.7.2.js'  }  });  seajs.use('$', function() {  // Got $ !  }); |
| 其他 | 我们推荐使用配套的[构建工具](https://github.com/seajs/seajs/issues/538)来打包模块:使用 spm-build 和 grunt 进行打包  在 spm-build 中，所有的匿名模块通过[标准的 transport 流程](http://docs.spmjs.org/doc/build-task" \l "js-%E6%96%87%E4%BB%B6)，会打包成具有实际 ID 的具名模块，而主模块（在 package.json 中指定输出的文件）的 ID 和实际路径是匹配的，符合ID 和路径匹配原则。  如果没有使用官方工具，你需要在自己的打包和部署过程中保证这个原则。 |
| ID 和路径匹配原则 是 Sea.js 实现中的一个约定，这个约定帮助我们减少了对 ID 的记忆负担，同时增加了构建的复杂度。  同样的，这也是一把双刃剑，目前还没有『完美』的处理方案，都会在某些地方存在取舍和权衡。 |
| CMD 模块的构建[关于 id 与 dependencies ] | |
| www | https://github.com/seajs/seajs/issues/426 |
| 基础及原理 | CMD 模块在构建时，有两个基本操作：   1. 提取操作   提取模块的标识 id 和依赖 dependencies。  假设模块代码为：  //a.js **define**(**function**(require, exports) {  **var** b = require('./b'); }); 经过提取操作后，a.js 的源码会转换成临时文件; **define**('xxx/1.0.0/a', ['./b'], **function**(require, exports) {  **var** b = require('./b'); })   1. 压缩操作   经过上面的提取操作后，构建工具就可以调用任何 JS 压缩工具来进行压缩了，require 参数也可以被压缩成任意字符。  可以看出，和普通压缩工具相比，CMD 模块的构建过程中增加了 id 和 dependencies 的提取操作。下面说明为什么需要预先提取这两个信息。 |
| 关于id | |  |  | | --- | --- | | 默认情况下，书写 CMD 模块时，不需要手写 id | 上面两个模块，如果直接合并，会变成： | | a.js define(function(require, exports) {  ... }); | a+b.js define(function(require, exports) {  ... }); define(function(require, exports) {  ... }); | | b.js define(function(require, exports) {  ... }); | |  | | | 这会导致无法区分 define 对应哪个模块。因此在合并前，我们需要通过工具将 id 提取出来:  a+b.js: define('a', function(require, exports) {  ... }); define('b', function(require, exports) {  ... });  此外，即便不合并，保持一个文件一个模块，如果压缩时不提取 id，那么在 IE6-9 下也有可能会出现问题。这是实现上的困难，具体请看源码。如果要确保上线后在 IE 下没问题，请务必要手写或通过工具提取 id。 | | | 在开发时，Sea.js 是如何知道一个模块的具体依赖呢:  a.js  define(function(require, exports) {  var b = require('./b');  var c = require('./c');  });  Sea.js 在运行 define 时，接受 factory 参数，可以通过 factory.toString() 拿到源码，再通过正则匹配 require 的方式来得到依赖信息。依赖信息是一个数组，比如上面 a.js 的依赖数组是：['./b', './c']  由于 Sea.js 的这个实现原理，使得书写 CMD 模块代码时，必须遵守 [require 书写约定](https://github.com/seajs/seajs/issues/259)，否则获取不到依赖数组，Sea.js 也就无法正确运行。  而且正则匹配取依赖的实现方案并非百分百可靠，除了 require 关键字被压缩的问题以外，对于一些极端情况无法保证正确性，特别对于压缩后的代码。有兴趣的可以看看社区里[关于 require 正则提取依赖的有奖挑战](https://github.com/seajs/seajs/issues/478)。 | | |
| 关于依赖数组  dependencies | 为了保证压缩工具可以随意压缩代码，构建工具在提取 id 字符串时，同时也会提取 dependencies 数组。提取过后的代码变成：  define('xxx/1.0.0/a', ['./b', './c'], function(require, exports) {  var b = require('./b');  var c = require('./c');  });  这样，Sea.js 就不需要通过 factory.toString() 和正则匹配的方式来获取依赖，直接从第二个参数中就可以拿到依赖数组。这意味着，提取过 id 和 dependencies 的模块代码，就可以用任何压缩工具压缩了。  注意，一旦设置了 define 的第二个参数 dependencies，Sea.js 将不会用正则匹配的方式来获取依赖，而直接将 dependencies 作为所有的依赖。 |
| 用普通压缩工具如何压缩 CMD 模块 | 由于各种原因，暂时无法使用 Sea.js 配套的构建工具来压缩时，需要注意以下几点：  如果项目需要支持 IE，请手写 id，即定义模块时，需要人肉写上第一个参数，比如：  define('a', function(require, exports) {  ...  });  如果项目对性能有要求，上线后需要合并文件，也请确保手工写上 id 参数。  压缩时，不要压缩 require 参数，目前 UglifyJS 支持通过参数来指定保留名字：  $ uglifyjs --reserved 'require' -o test-min.js test.js  或者自己写工具来保证 id 和 dependencies 的预先提取。 |
| 传统 js合并与 seajs文件合并 | |
| 传统的js文件合并 | 如果采用 <script> 标签的话，JS 合并非常简单，比如：   1. 合并前：  |  |  | | --- | --- | | 源码 | // util.js  function add (a,b) {  return a + b;  }  // index.js  var c = add(1, 2);  alert(c); | | 使用 | <script src="/js/util.js"></script>  <script src="/js/index.js"></script> |  1. 合并后：   按照 html 上 JS 文件的引入顺序，将相应的文件拼合即可：   |  |  | | --- | --- | | 源码 | // 合并后的util\_index.js  function add(a,b) {  return a + b;  }  var c = add(1,2);  alert(c); | | 使用 | <script src="/js/util\_index.js"></script> | |
| seajs的合并1 | 1. 合并前   <script src="/js/sea.js"></script>  <script>seajs.use("/js/index");</script>  // util.js  define(function (require, exports) {  exports.add = function (a, b) {  return a + b;  };  });  // index.js  define(function (require) {  var util = require('./util');  var c = util.add(1, 2);  alert(c);  });  这时候如果要做 JS 合并的话，该怎么弄呢？ 很多人会觉得这有什么难搞的，就跟传统的方式一样呗，而且更简单了，因为我们都不用手工指定该怎么合并，只要通过对 index.js 的内容进行分析就可以了，有 require 关键词嘛   1. 合并胡   // 合并后的 JS，替换原来的 index.js  define(function (require, exports) {  exports.add = function (a, b) {  return a + b;  };  });  define(function (require) {  var util = require('./util');  var c = util.add(1, 2);  alert(c);  });  现在请问，上面这个合并后的 index.js 如果通过 seajs.use 加载进来的话能正常执行吗？ 答案是，在 seajs 1.3.1 版本下，可以正常执行，但是如果抓包的话，会发现浏览器在加载了这个合并后的 index.js 之后，还是会再加载同目录下的 util.js  所以如果你把这个新的 index.js 换个目录存放并且相应修改 seajs.use 的模块路径，那么会发现这个页面没能蹦出预期中的 “3”，因为那个新的目录下没有 util.js  也就是说，这个合并的策略并不能奏效 |
| seajs的合并1 | 1. [CMD 规范](https://github.com/seajs/seajs/issues/242)   这是为啥呢？其实也很简单，翻开 SeaJS 的 [CMD 规范](https://github.com/seajs/seajs/issues/242) ，开头就说了：一个模块就是一个文件。换句话说，一个文件里面只能定义一个 CMD 模块，而刚才那个文件里面定义了两个，所以出现异常也不奇怪了。 再来分析一下刚才的例子，我们发现，当一个文件出现了多个 CMD 模块时，其实只有最后那个被 SeaJS 识别了，所以执行时依然需要去再加载 util.js 这个文件。 如果继续深入 SeaJS 源码的话，就知道，CMD 模块其实是“匿名”模块，也就是说开发者没有显式地指定该模块的 id，对于匿名的模块，SeaJS 会用这个 JS 文件的 URL 作为它的 id ，并缓存 id 与 模块之间的关系（你可以理解为“识别”）。所以只有最后一个定义的 CMD 模块会被识别，因为前面定义的模块都被它覆盖了。 |
| 1. Transport 格式   如果 SeaJS 只支持 CMD 模块的话，我们就没法实现 JS 文件的合并了，所以其实 SeaJS 还支持一种 Transport 格式:[[CommonJS 的 Modules/Transport 和 Modules/Wrappings 规范有什么区别？](http://www.zhihu.com/question/20789867/answer/16187950)]  SeaJS 里，推崇的 Modules/Wrappings 规范是 CMD 规范：define(function(){ })；直接是由开发者手写的，写完后，可直接不经过任何构建工具就在浏览器上加载运行。 但 CMD 模块在正式上线前，依旧需要通过构建工具先转换为 Modules/Transport 格式：  define("id", ["dep-1", "dep-2"], function(require, exports, module) {  // source code  })  转换成 Transport 格式后，才能进一步压缩、合并等。可以看到，Transport 格式其实就是加上了名字的 CMD 模块，SeaJS 在遇到这种模块时就直接通过定义的 id 来缓存模块了。 |
| 看到这里，你可能会想，这步转换也没啥难的嘛，我们给文件里面两个模块分别加上 id 就 OK 了啊，比如分别叫做 util 和 index  define('util', [], function (require, exports) {  exports.add = function (a, b) {  return a + b;  };  });  define('index', [], function (require) {  var util = require('./util');  var c = util.add(1, 2);  alert(c);  });  实验一下你会发现，浏览器不会再发起对 util.js 的请求了，这又是为啥呢？  当调用 seajs.use('/js/index') 时，如果对应的 JS 文件中有两个 Transport 格式的模块，哪一个模块才是调用者想要的（哪个才是 js/index）？ 答案相信大家都能想到，根据模块的 id 呗！SeaJS 也正是这么做的，它会比较模块的 id 与 use() 方法的参数（其实是相应 JS 文件的 URL），选用匹配的那个  可是这个“匹配”的规则该怎么定义呢？  比如设想这样一个稍微复杂的情况：util.js 文件中的两个模块 id 分别叫做 text/util 和 util ，当调用 seajs.use('/js/util') 时，究竟哪个模块才是我们需要的呢？我想这时候大家都会想到这个万无一失的方案：把模块 ID 转换为完整的 URL 再匹配！没错，这也是 SeaJS 的做法： 将所有的模块 id 都转为完整的 URL ，然后选取与当前这个 JS 文件的 URL 完全匹配的那个模块；转换的规则就不细说了，不过看到这里我们大概知道上一步为啥有问题了，因为上一步的 id 只是取了个名字，完全没有考虑 URL，略作修改  // http://localhost/js/index.js 的内容  define('http://localhost/js/util', [], function (require, exports) {  exports.add = function (a, b) {  return a + b;  };  });  define('http://localhost/js/index', [], function (require) {  var util = require('./util');  var c = util.add(1, 2);  alert(c);  });  id 直接用完整的 URL ，这样都不需要转换，这时候执行一下，总算 OK 了 |
| firstModuleInPackage  那么上一个例子里面是什么情况呢？这涉及到 SeaJS 中的 firstModuleInPackage 策略： 简单来说，上面例子中，所有模块的 id 都与当前 JS 的 URL 不匹配，这时候 SeaJS 会使用文件中第一个模块，所以实际上页面中只是执行了那个 add 方法的定义；  关于 firstModuleInPackage 可以看看[这篇讨论](https://groups.google.com/forum/" \l "!msg/seajs/sMjsUuz4JyM/4qcryy1Muz4J)，这个特性是 1.2.1 引入的，但是玉伯又决定在 2.0 里面[去掉这个特性](https://github.com/seajs/seajs/issues/438)，改为如果没有匹配时不执行任何模块  结语  看到这里，相信大家应该大致了解 SeaJS 文件该如何合并了，它相比传统方式的 JS 合并要复杂许多，原因也不难理解：  一： SeaJS 引入了额外的复杂度，原来简单的文件合并方式不会奏效  二： SeaJS 相比 RequireJS 简化了模块书写，导致合并时需要做模块格式的转换，比如自动加上 id  SeaJS 虽好，但是不可能没有缺点，当你在一个方面获得巨大的好处时，通常会在其他方面付出代价，所以我们在做选择时一定要做好权衡。  另外，希望这篇文章能让大家更理解 SeaJS 与 Spm，事情没有想象中的那么简单，少一些抱怨，多一些建设性意见！ |
| 其他 | 如何使用 Spm2 压缩合并业务模块：  <http://chaoskeh.com/blog/how-to-build-seajs-business-module-by-spm2.html> |