[基本操作 3](#_Toc374957527)

[define Function 3](#_Toc374957528)

[define define(factory) 3](#_Toc374957529)

[define define(id?, deps?, factory) 4](#_Toc374957530)

[define.cmd Object 4](#_Toc374957531)

[require Function 4](#_Toc374957532)

[require require(id) 4](#_Toc374957533)

[require.async require.async(id, callback?) 5](#_Toc374957534)

[require.resolve require.resolve(id) 5](#_Toc374957535)

[exports Object 6](#_Toc374957536)

[module Object 7](#_Toc374957537)

[module.id String 7](#_Toc374957538)

[module.uri String 8](#_Toc374957539)

[module.dependencies Array 8](#_Toc374957540)

[module.exports Object 8](#_Toc374957541)

[小结 9](#_Toc374957542)

[模块加载 9](#_Toc374957543)

[同步加载 10](#_Toc374957544)

[异步加载 10](#_Toc374957545)

[预加载 10](#_Toc374957546)

[载入API 11](#_Toc374957547)

[seajs.use 11](#_Toc374957548)

[与 DOM ready 的关系 13](#_Toc374957549)

[require 13](#_Toc374957550)

[require.async(id, callback?) 14](#_Toc374957551)

[路径 15](#_Toc374957552)

[模块标识 15](#_Toc374957553)

[相对标识 15](#_Toc374957554)

[顶级标识 16](#_Toc374957555)

[普通路径 17](#_Toc374957556)

[文件后缀的自动添加规则 18](#_Toc374957557)

[全局配置 18](#_Toc374957558)

[seajs.config seajs.config(options) 18](#_Toc374957559)

[alias Object 20](#_Toc374957560)

[paths Object 20](#_Toc374957561)

[vars Object 21](#_Toc374957562)

[map Array 21](#_Toc374957563)

[preload Array 22](#_Toc374957564)

[debug Boolean 22](#_Toc374957565)

[base String 23](#_Toc374957566)

[charset String | Function 23](#_Toc374957567)

[提示 24](#_Toc374957568)

[多次配置自动合并 24](#_Toc374957569)

[插件的配置 25](#_Toc374957570)

[配置文件 25](#_Toc374957571)

[直接使用非CMD的模块，如jQuery、我的类库等 26](#_Toc374957572)

[与Node.js结合使用，实现模块的跨服务器和浏览器共享 26](#_Toc374957573)

[让 Sea.js 的模块跑在 Node 上 26](#_Toc374957574)

[让 Node 的模块跑在浏览器端 27](#_Toc374957575)

[通用的前提条件 27](#_Toc374957576)

[附录 28](#_Toc374957577)

# 基本操作

在 Sea.js 中，所有 JavaScript 模块都遵循 CMD（[Common Module Definition](https://github.com/cmdjs/specification/blob/master/draft/module.md)） 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。代码的书写格式如下：

define(factory);

## define Function

define 是一个全局函数，用来定义模块。

### define define(factory)

define 接受 factory 参数，factory 可以是一个函数，也可以是一个对象或字符串。

factory 为对象、字符串时，表示模块的接口就是该对象、字符串。比如可以如下定义一个 JSON 数据模块：

define({ "foo"**:** "bar" });

也可以通过字符串定义模板模块：

define('I am a template. My name is {{name}}.');

factory 为函数时，表示是模块的构造方法。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) {

*// 模块代码*

});

### define define(id?, deps?, factory)

define 也可以接受两个以上参数。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) {

*// 模块代码*

});

id 和 deps 参数可以省略。省略时，可以通过构建工具自动生成。

id 用来显式指定模块 ID。当你的项目上线，所有的模块都合并到了一个文件中，如果不显示指定， SeaJS 就无从知道哪个模块是哪个了。在开发的时候，一般用不到它。

dependencies 也是如此。它列出了当前模块所依赖的模块，在开发的时候是不需要写明的。 SeaJS 会检查你的模块回调函数，找到所有的 require 语句，从而得到你的模块的所有依赖。 在真正 require 当前模块时，会先去请求这个模块的依赖，加载完毕，再去初始化当前的模块。

而到了线上，代码压缩、合并之后，则会提供此参数，以省却模块解析的时间，提高效率。

**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。

### define.cmd Object

一个空对象，可用来判定当前页面是否有 CMD 模块加载器：

**if** (**typeof** define **===** "function" **&&** define.cmd) {

*// 有 Sea.js 等 CMD 模块加载器存在*

}

## require Function

require 是 factory 函数的第一个参数。

### require require(id)

require 是一个方法，接受 [模块标识](https://github.com/seajs/seajs/issues/258) 作为唯一参数，用来获取其他模块提供的接口。

define(**function**(require, exports) {

*// 获取模块 a 的接口*

**var** a **=** require('./a');

*// 调用模块 a 的方法*

a.doSomething();

});

**注意**：在开发时，require 的书写需要遵循一些 [简单约定](https://github.com/seajs/seajs/issues/259)。

### require.async require.async(id, callback?)

require.async 方法用来在模块内部异步加载模块，并在加载完成后执行指定回调。callback 参数可选。

define(**function**(require, exports, module) {

*// 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();

});

});

**注意**：require 是同步往下执行，require.async 则是异步回调执行。require.async 一般用来加载可延迟异步加载的模块。

### require.resolve 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。

define(**function**(require, exports) {

console.log(require.resolve('./b'));

*// ==> http://example.com/path/to/b.js*

});

这可以用来获取模块路径，一般用在插件环境或需动态拼接模块路径的场景下。

## exports Object

exports 是一个对象，用来向外提供模块接口。

define(**function**(require, exports) {

*// 对外提供 foo 属性*

exports.foo **=** 'bar';

*// 对外提供 doSomething 方法*

exports.doSomething **=** **function**() {};

});

除了给 exports 对象增加成员，还可以使用 return 直接向外提供接口。

define(**function**(require) {

*// 通过 return 直接提供接口*

**return** {

foo**:** 'bar',

doSomething**:** **function**() {}

};

});

如果 return 语句是模块中的唯一代码，还可简化为：

define({

foo**:** 'bar',

doSomething**:** **function**() {}

});

上面这种格式特别适合定义 JSONP 模块。

**特别注意**：下面这种写法是错误的！

define(**function**(require, exports) {

*// 错误用法！！!*

exports **=** {

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**() {}

};

});

**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。

## module Object

module 是一个对象，上面存储了与当前模块相关联的一些属性和方法。

### module.id String

模块的唯一标识。

define('id', [], **function**(require, exports, module) {

*// 模块代码*

});

上面代码中，define 的第一个参数就是模块标识。

### 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。

define(**function**(require, exports, module) {

console.log(module.uri);

*// ==> http://example.com/path/to/this/file.js*

});

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

dependencies 是一个数组，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*

});

## 小结

这就是 CMD 模块定义规范的所有内容。经常使用的 API 只有 define, require,require.async, exports, module.exports 这五个。其他 API 有个印象就好，在需要时再来查文档，不用刻意去记。

# 模块加载

## 同步加载

SeaJS会在html页面打开时通过静态分析一次性记载所有需要的js文件（use和require加载的模块）。

这里简要介绍一下SeaJS的自动加载机制。使用SeaJS后html只要包含sea.js即可，那么其它js文件是如何加载进来的呢？SeaJS会首先下载入口模块，然后顺着入口模块使用正则表达式匹配代码中所有的require，再根据require中的文件路径标识下载相应的js文件，对下载来的js文件再迭代进行类似操作。整个过程类似图的遍历操作（因为可能存在交叉循环依赖所以整个依赖数据结构是一个图而不是树）。

## 异步加载

可以使用require.async(id, callback?)，在执行到这句时才下载，即可实现按需加载。

## 预加载

使用 preload 配置项，可以在普通模块加载前，提前加载并初始化好指定模块。

*// 在老浏览器中，提前加载好 ES5 和 json 模块*

seajs.config({

preload**:** [

Function.prototype.bind **?** '' **:** 'es5-safe', // “es5-safe”为js文件名，省略了“.js”

**this**.JSON **?** '' **:** 'json'

]

});

preload 中的空字符串会被忽略掉。

**注意**：preload 中的配置，需要等到 use 时才加载。比如：

seajs.config({

preload**:** 'a'

});

*// 在加载 b 之前，会确保模块 a 已经加载并执行好*

seajs.use('./b');

preload 配置不能放在模块文件里面：

seajs.config({

preload**:** 'a'

});

define(**function**(require, exports) {

*// 此处执行时，不能保证模块 a 已经加载并执行好*

});

## 载入API

根据应用场景的不同，SeaJS提供了三个载入模块的API，分别是seajs.use，require和require.async，下面分别介绍。

### seajs.use

seajs.use主要用于载入入口模块。入口模块相当于C程序的main函数，同时也是整个模块依赖树的根。上面在TinyApp小例子中，init就是入口模块。seajs.use用法如下：

1. //单一模式
2. seajs.use('./a');
4. //回调模式
5. seajs.use('./a', function(a) {
6. a.run();
7. });
9. //多模块模式
10. seajs.use(['./a', './b'], function(a, b) {
11. a.run();
12. b.run();
13. });

一般seajs.use只用在页面载入入口模块，SeaJS会顺着入口模块解析所有依赖模块并将它们加载。如果入口模块只有一个，也可以通过给引入sea.js的script标签加入”data-main”属性来省略seajs.use，例如，上面TinyApp的index.html也可以改为如下写法：

1. <!DOCTYPE HTML>
2. <html lang="zh-CN">
3. <head>
4. <meta charset="UTF-8">
5. <title>TinyApp</title>
6. </head>
7. <body>
8. <p class="content"></p>
9. <script src="./sea.js" data-main="./init"></script>
10. </body>
11. </html>

这种写法会令html更加简洁。

#### 与 DOM ready 的关系

**注意**：seajs.use 与 DOM ready 事件没有任何关系。如果某些操作要确保在 DOM ready 后执行，需要使用 jquery 等类库来保证，比如：

seajs.use(['jquery', './main'], **function**($, main) {

$(document).ready(**function**() {

main.init();

});

});

### require

require是SeaJS主要的模块加载方法，当在一个模块中需要用到其它模块时一般用require加载：

1. var m = require('/path/to/module/file');

传给require的路径标识必须是字符串字面量，不能是表达式，如下面使用require的方法是错误的：

1. require('module' + '1');
3. require('Module'.toLowerCase());

这都会造成SeaJS无法进行正确的正则匹配以下载相应的js文件。

### require.async(id, callback?)

上文说过SeaJS会在html页面打开时通过静态分析一次性记载所有需要的js文件，如果想要某个js文件在用到时才下载，可以使用require.async：

1. require.async('/path/to/module/file', function(m) {
2. //code of callback...
3. });

这样只有在用到这个模块时，对应的js文件才会被下载，也就实现了JavaScript代码的按需加载。

define(**function**(require, exports, module) {

*// 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();

});

});

# 路径

## 模块标识

模块标识是一个字符串，用来标识模块。在 require、 require.async 等加载函数中，第一个参数都是模块标识。

Sea.js 中的模块标识是 [CommonJS 模块标识](http://wiki.commonjs.org/wiki/Modules/1.1.1) 的超集:

1. 一个模块标识由斜线（/）分隔的多项组成。
2. 每一项必须是小驼峰字符串、 . 或 .. 。
3. 模块标识可以不包含文件后缀名，比如 .js 。
4. 模块标识可以是 **相对** 或 **顶级** 标识。如果第一项是 . 或 ..，则该模块标识是相对标识。
5. 顶级标识根据模块系统的基础路径来解析。
6. 相对标识相对 require 所在模块的路径来解析。

注意，符合上述规范的标识肯定是 Sea.js 的模块标识，但 Sea.js 能识别的模块标识不需要完全符合以上规范。 比如，除了大小写字母组成的小驼峰字符串，Sea.js 的模块标识字符串还可以包含下划线（\_）和连字符（-）， 甚至可以是 http://、https://、file:/// 等协议开头的绝对路径。

## 相对标识

相对标识以 . 开头，只出现在模块环境中（define 的 factory 方法里面）。相对标识永远相对当前模块的 URI 来解析：

*// 在 http://example.com/js/a.js 的 factory 中：*

require.resolve('./b');

*// => http://example.com/js/b.js*

*// 在 http://example.com/js/a.js 的 factory 中：*

require.resolve('../c');

*// => http://example.com/c.js*

## 顶级标识

顶级标识不以点（.）或斜线（/）开始， 会相对模块系统的基础路径（即 Sea.js 的 base 路径）来解析：

*// 假设 base 路径是：http://example.com/assets/*

*// 在模块代码里：*

require.resolve('gallery/jquery/1.9.1/jquery');

*// => http://example.com/assets/gallery/jquery/1.9.1/jquery.js*

模块系统的基础路径即 base 的默认值，与 sea.js 的访问路径相关：

如果 sea.js 的访问路径是：

http://example.com/assets/sea.js

则 base 路径为：

http://example.com/assets/

当 sea.js 的访问路径中含有版本号时，base 不会包含 seajs/x.y.z 字串。 当 sea.js有多个版本时，这样会很方便。

如果 sea.js 的路径是：

http://example.com/assets/seajs/1.0.0/sea.js

则 base 路径是：

http://example.com/assets/

当然，也可以手工配置 base 路径：

seajs.config({

base**:** 'http://code.jquery.com/'

});

*// 在模块代码里：*

require.resolve('jquery');

*// => http://code.jquery.com/jquery.js*

## 普通路径

除了相对和顶级标识之外的标识都是普通路径。普通路径的解析规则，和 HTML 代码中的<script src="..."></script> 一样，会相对当前页面解析。

*// 假设当前页面是 http://example.com/path/to/page/index.html*

*// 绝对路径是普通路径：*

require.resolve('http://cdn.com/js/a');

*// => http://cdn.com/js/a.js*

*// 根路径是普通路径：*

require.resolve('/js/b');

*// => http://example.com/js/b.js*

*// use 中的相对路径始终是普通路径：*

seajs.use('./c');

*// => 加载的是 http://example.com/path/to/page/c.js*

seajs.use('../d');

*// => 加载的是 http://example.com/path/to/d.js*

**提示**：

1. 顶级标识始终相对 base 基础路径解析。
2. 绝对路径和根路径始终相对当前页面解析。
3. require 和 require.async 中的相对路径相对当前模块路径来解析。
4. seajs.use 中的相对路径始终相对当前页面来解析。

## 文件后缀的自动添加规则

Sea.js 在解析模块标识时， 除非在路径中有问号（?）或最后一个字符是井号（#），否则都会自动添加 JS 扩展名（.js）。如果不想自动添加扩展名，可以在路径末尾加上井号（#）。

*// ".js" 后缀可以省略：*

require.resolve('http://example.com/js/a');

require.resolve('http://example.com/js/a.js');

*// => http://example.com/js/a.js*

*// ".css" 后缀不可省略：*

require.resolve('http://example.com/css/a.css');

*// => http://example.com/css/a.css*

*// 当路径中有问号（"?"）时，不会自动添加后缀：*

require.resolve('http://example.com/js/a.json?callback=define');

*// => http://example.com/js/a.json?callback=define*

*// 当路径以井号（"#"）结尾时，不会自动添加后缀，且在解析时，会自动去掉井号：*

require.resolve('http://example.com/js/a.json#');

*// => http://example.com/js/a.json*

# 全局配置

可以对 Sea.js 进行配置，让模块编写、开发调试更方便。

## seajs.config seajs.config(options)

用来进行配置的方法。

seajs.config({

*// 别名配置*

alias**:** {

'es5-safe'**:** 'gallery/es5-safe/0.9.3/es5-safe',

'json'**:** 'gallery/json/1.0.2/json',

'jquery'**:** 'jquery/jquery/1.10.1/jquery'

},

*// 路径配置*

paths**:** {

'gallery'**:** 'https://a.alipayobjects.com/gallery'

},

*// 变量配置*

vars**:** {

'locale'**:** 'zh-cn'

},

*// 映射配置*

map**:** [

['http://example.com/js/app/', 'http://localhost/js/app/']

],

*// 预加载项*

preload**:** [

Function.prototype.bind **?** '' **:** 'es5-safe',

**this**.JSON **?** '' **:** 'json'

],

*// 调试模式*

debug**:** **true**,

*// Sea.js 的基础路径*

base**:** 'http://example.com/path/to/base/',

*// 文件编码*

charset**:** 'utf-8'

});

支持以下配置选项：

### alias Object

当模块标识很长时，可以使用 alias 来简化。

seajs.config({

alias**:** {

'jquery'**:** 'jquery/jquery/1.10.1/jquery',

'app/biz'**:** 'http://path/to/app/biz.js',

}

});

define(**function**(require, exports, module) {

**var** $ **=** require('jquery');

*//=> 加载的是 http://path/to/base/jquery/jquery/1.10.1/jquery.js*

**var** biz **=** require('app/biz');

*//=> 加载的是 http://path/to/app/biz.js*

});

使用 alias，可以让文件的真实路径与调用标识分开，有利于统一维护。

### paths Object

当目录比较深，或需要跨目录调用模块时，可以使用 paths 来简化书写。

seajs.config({

paths**:** {

'gallery'**:** 'https://a.alipayobjects.com/gallery',

'app'**:** 'path/to/app',

}

});

define(**function**(require, exports, module) {

**var** underscore **=** require('gallery/underscore');

*//=> 加载的是 https://a.alipayobjects.com/gallery/underscore.js*

**var** biz **=** require('app/biz');

*//=> 加载的是 path/to/app/biz.js*

});

paths 配置可以结合 alias 配置一起使用，让模块引用非常方便。

### vars Object

有些场景下，模块路径在运行时才能确定，这时可以使用 vars 变量来配置。

seajs.config({

vars**:** {

'locale'**:** 'zh-cn'

}

});

define(**function**(require, exports, module) {

**var** lang **=** require('./i18n/{locale}.js');

*//=> 加载的是 path/to/i18n/zh-cn.js*

});

vars 配置的是模块标识中的变量值，在模块标识中用 {key} 来表示变量。

### map Array

该配置可对模块路径进行映射修改，可用于路径转换、在线调试等。

seajs.config({

map**:** [

[ '.js', '-debug.js' ]

]

});

define(**function**(require, exports, module) {

**var** a **=** require('./a');

*//=> 加载的是 path/to/a-debug.js*

});

更多用法可参考：[调试实践](https://github.com/seajs/seajs/issues/263)

### preload Array

使用 preload 配置项，可以在普通模块加载前，提前加载并初始化好指定模块。

*// 在老浏览器中，提前加载好 ES5 和 json 模块*

seajs.config({

preload**:** [

Function.prototype.bind **?** '' **:** 'es5-safe',

**this**.JSON **?** '' **:** 'json'

]

});

preload 中的空字符串会被忽略掉。

**注意**：preload 中的配置，需要等到 use 时才加载。比如：

seajs.config({

preload**:** 'a'

});

*// 在加载 b 之前，会确保模块 a 已经加载并执行好*

seajs.use('./b');

preload 配置不能放在模块文件里面：

seajs.config({

preload**:** 'a'

});

define(**function**(require, exports) {

*// 此处执行时，不能保证模块 a 已经加载并执行好*

});

### debug Boolean

值为 true 时，加载器不会删除动态插入的 script 标签。插件也可以根据 debug 配置，来决策 log 等信息的输出。

### base String

Sea.js 在解析顶级标识时，会相对 base 路径来解析。

模块系统的基础路径即 base 的默认值，与 sea.js 的访问路径相关：

如果 sea.js 的访问路径是：

http://example.com/assets/sea.js

则 base 路径为：

http://example.com/assets/

当 sea.js 的访问路径中含有版本号时，base 不会包含 seajs/x.y.z 字串。 当 sea.js有多个版本时，这样会很方便。

如果 sea.js 的路径是：

http://example.com/assets/seajs/1.0.0/sea.js

则 base 路径是：

http://example.com/assets/

当然，也可以手工配置 base 路径：

seajs.config({

base**:** 'http://code.jquery.com/'

});

*// 在模块代码里：*

require.resolve('jquery');

*// => http://code.jquery.com/jquery.js*

**注意**：一般请不要配置 base 路径，把 sea.js 放在合适的路径往往更简单一致。

### charset String | Function

获取模块文件时，<script> 或 <link> 标签的 charset 属性。 默认是 utf-8

charset 还可以是一个函数：

seajs.config({

charset**:** **function**(url) {

*// xxx 目录下的文件用 gbk 编码加载*

**if** (url.indexOf('http://example.com/js/xxx') **===** 0) {

**return** 'gbk';

}

*// 其他文件用 utf-8 编码*

**return** 'utf-8';

}

});

## 提示

### 多次配置自动合并

seajs.config 可以多次运行，每次运行时，会对配置项进行合并操作：

seajs.config({

alias**:** {

'jquery'**:** 'path/to/jquery.js',

'a'**:** 'path/to/a.js'

},

preload**:** ['seajs-text']

});

seajs.config({

alias**:** {

'underscore'**:** 'path/to/underscore.js',

'a'**:** 'path/to/biz/a.js'

},

preload**:** ['seajs-combo']

});

上面两处 config 运行的结果是：

alias **=** {

'jquery'**:** 'path/to/jquery.js',

'underscore'**:** 'path/to/underscore.js',

'a'**:** 'path/to/biz/a.js'

};

preload **=** ['seajs-text', 'seajs-combo'];

即：config 会自动合并不存在的项，对存在的项则进行覆盖。

### 插件的配置

插件可以给 Sea.js 添加配置项，请查看具体插件了解相关配置。

### 配置文件

配置可以直接写在 html 页面上，也可以独立出来成为一个文件。

config.js

seajs.config({

...

});

独立成一个文件时，一般通过 script 标签在页面中同步引入。

常用的配置项是 alias、paths、base，其他配置项有需要时，来查查文档就会用了。

# 直接使用非CMD的模块，如jQuery、我的类库等

待研究

参考资料：

[seajs进阶](http://blog.csdn.net/it_man/article/details/8482633)

# 与Node.js结合使用，实现模块的跨服务器和浏览器共享

Sea.js 的模块遵循 [CMD](https://github.com/seajs/seajs/issues/242) 规范，与 Node.js 的模块规范非常相近，两者的模块可以很容易互相迁移。

## 让 Sea.js 的模块跑在 Node 上

非常简单。首先需要安装 seajs 的 Node 模块：

$ npm install seajs -g

安装好后，在需要调用 Sea.js 模块的入口文件里，require 下 seajs ：

a.js

define(**function**(require, exports) {

exports.name **=** 'A';

});

main.js

require('seajs');

**var** a **=** require('./a');

console.log(a.name);

这样就可以在 Node 环境中运行 Sea.js 的模块了：

$ node main.js

A

## 让 Node 的模块跑在浏览器端

这个也很简单，将 Node 的模块封装成 CMD 模块即可：

a.js

exports.name **=** 'A';

封装成

define(**function**(require, exports) {

exports.name **=** 'A';

});

这样在浏览器端就可以通过 Sea.js 来加载使用了：

seajs.use('./a', **function**(a) {

console.log(a.name);

});

## 通用的前提条件

通过上面的例子可以看出，只要遵循 CMD 规范来书写模块代码，就可以非常方便的同时运行在浏览器端和服务器端。这是 CMD 中 C 字母的含义：Common（通用）。

这比 RequireJS 的 AMD 规范好很多。

但是，无论是 CMD 还是 AMD 规范，或者是未来的某个 Modules/Cool 规范，一个模块要想同时在不同环境下执行，就得遵循以下前提条件：

1. **不能调用浏览器端的私有特性。**比如 attachEvent 之类的，除非你在 Node 端提前模拟好。
2. **不能调用服务器端的私有特性。**比如 process 对象，除非你在浏览器端自己实现一个类似的 process 对象。
3. **只用不同规范中的交集。**比如 CMD 中有 module.uri 属性，但 Node 中没有，要通用，就不能去调用这些有差异的接口。类似的，Node 端的 \_\_filename 在浏览器端不存在，要通用的话，也不能调用。

其实上面这些严格要求，是 **非常自然的**。这就如浏览器兼容一样，要写出所有浏览器下都能跑的代码，最好的方式是只用那些所有浏览器都支持的特性，不然你就得用 if ... else ... 去搞啦。

## 附录

Node.js 与 Sea.js 在模块接口上的主要差异如下：

1. Node.js 里，模块文件里的 this === module.exports；Sea.js 里，this === window。
2. Node.js 里，模块文件里的 return xx 会被忽略；Sea.js 里，return xx 等价module.exports = xx。
3. Node.js 里，require 是懒加载 + 懒执行的。在 Sea.js 里是提前加载好 + 懒执行。
4. Sea.js 里，require(id) 中的 id 必须是字符串直接量。Node.js 里没这个限制。

模块加载器有一大堆。有CommonJS规范、RequireJS库和node.js require方法，没有一个可以很好地协同使用。如果你不用模块加载器，就想在客户端和服务器之间共享代码(这是服务器上JS的一大优点)，那么你可以使用这个有点变通的模式：

1. (**function**(exports) {
3. **var** MyClass = **function**() { /\* ... \*/ };
4. **var** myObject = {};
6. exports.MyClass = MyClass;
7. exports.myObject = MyObject;
9. })(**typeof** global === "undefined" ? window : exports);

这个变通方法靠的是这一点：node.js定义了global(全局)对象，而浏览器没有定义。有了这个变通方法，node.js require()会很高兴，你还可以在<script>标签中加入文件，不会污染你的名称空间，当然假设没有其他JS以window.global对象污染你的名称空间！

遗憾的是，这个方法只适用于一个共享模块。一旦你有了多个彼此依赖的模块（通过node-land中的require方法和browser-land中的global对象），节点的名称空间与浏览器的加入之间的差异会变得异常明显，需要更多的变通方法。

另一个方法是使用browserify，捆绑所有JS，在浏览器里面模拟require。这种方法依赖node.js来提供生成的JS，这并不理想，因为静态文件应该由专门为该用途优化的web服务器来提供。不过，node.js+ browserify可以进行配置，以便编译可以静态提供的JS，不必依赖节点来提供。这个方法带来了一些开销：

1. 多出了构建这个步骤，以便部署。

2. 无论browserify使用什么机制来支持require()调用，都需要性能开销。

总的来说，这个方法在我看来比较好，我希望在将来编写的osmus版本中试用一下。