[介绍 2](#_Toc374950998)

[为什么使用SeaJs 2](#_Toc374950999)

[恼人的命名冲突 2](#_Toc374951000)

[烦琐的文件依赖 4](#_Toc374951001)

[使用 Sea.js 来解决 6](#_Toc374951002)

[小结 7](#_Toc374951003)

[下载及安装 8](#_Toc374951004)

[示例 8](#_Toc374951005)

[最新版本 12](#_Toc374951006)

[参考资料 13](#_Toc374951007)

# 介绍

SeaJS是一个遵循CommonJS规范的JavaScript模块加载框架，可以实现JavaScript的模块化开发及加载机制。与jQuery等JavaScript框架不同，SeaJS不会扩展封装语言特性，而只是实现JavaScript的模块化及按模块加载。SeaJS的主要目的是令JavaScript开发模块化并可以轻松愉悦进行加载，将前端工程师从繁重的JavaScript文件及对象依赖处理中解放出来，可以专注于代码本身的逻辑。SeaJS可以与jQuery这类框架完美集成。使用SeaJS可以提高JavaScript代码的可读性和清晰度，解决目前JavaScript编程中普遍存在的依赖关系混乱和代码纠缠等问题，方便代码的编写和维护。

SeaJS的作者是淘宝前端工程师玉伯。

SeaJS本身遵循KISS（Keep It Simple, Stupid）理念进行开发，其本身仅有个位数的API，因此学习起来毫无压力。在学习SeaJS的过程中，处处能感受到KISS原则的精髓——仅做一件事，做好一件事。

# 为什么使用SeaJs

## 恼人的命名冲突

我们从一个简单的习惯出发。我做项目时，常常会将一些通用的、底层的功能抽象出来，独立成一个个函数，比如

**function** each(arr) {

*// 实现代码*

}

**function** log(str) {

*// 实现代码*

}

并像模像样地把这些函数统一放在 util.js 里。需要用到时，引入该文件就行。这一切工作得很好，同事也很感激我提供了这么便利的工具包。

直到团队越来越大，开始有人抱怨。

小杨：我想定义一个 each 方法遍历对象，但页头的 util.js 里已经定义了一个，我的只能叫 eachObject 了，好无奈。

小高：我自定义了一个 log 方法，为什么小明写的代码就出问题了呢？谁来帮帮我。

抱怨越来越多。团队经过一番激烈的讨论，决定参照 Java 的方式，引入命名空间来解决。于是 util.js 里的代码变成了

**var** org **=** {};

org.CoolSite **=** {};

org.CoolSite.Utils **=** {};

org.CoolSite.Utils.each **=** **function** (arr) {

*// 实现代码*

};

org.CoolSite.Utils.log **=** **function** (str) {

*// 实现代码*

};

不要认为上面的代码是为了写这篇文章而故意捏造的。将命名空间的概念在前端中发扬光大，首推 Yahoo! 的 YUI2 项目。下面是一段真实代码，来自 Yahoo! 的一个开源项目。

**if** (org.cometd.Utils.isString(response)) {

**return** org.cometd.JSON.fromJSON(response);

}

**if** (org.cometd.Utils.isArray(response)) {

**return** response;

}

通过命名空间，的确能极大缓解冲突。但每每看到上面的代码，都忍不住充满同情。为了调用一个简单的方法，需要记住如此长的命名空间，这增加了记忆负担，同时剥夺了不少编码的乐趣。

作为前端业界的标杆，YUI 团队下定决心解决这一问题。在 YUI3 项目中，引入了一种新的命名空间机制。

YUI().use('node', **function** (Y) {

*// Node 模块已加载好*

*// 下面可以通过 Y 来调用*

**var** foo **=** Y.one('#foo');

});

YUI3 通过沙箱机制，很好的解决了命名空间过长的问题。然而，也带来了新问题。

YUI().use('a', 'b', **function** (Y) {

Y.foo();

*// foo 方法究竟是模块 a 还是 b 提供的？*

*// 如果模块 a 和 b 都提供 foo 方法，如何避免冲突？*

});

看似简单的命名冲突，实际解决起来并不简单。如何更优雅地解决？我们按下暂且不表，先来看另一个常见问题。

## 烦琐的文件依赖

继续上面的故事。基于 util.js，我开始开发 UI 层通用组件，这样项目组同事就不用重复造轮子了。

其中有一个最被大家喜欢的组件是 dialog.js，使用方式很简单。

<script src="util.js"></script>

<script src="dialog.js"></script>

<script>

org.CoolSite.Dialog.init({ */\* 传入配置 \*/* });

</script>

可是无论我怎么写文档，以及多么郑重地发邮件宣告，时不时总会有同事来询问为什么 dialog.js 有问题。通过一番排查，发现导致错误的原因经常是

<script src="dialog.js"></script>

<script>

org.CoolSite.Dialog.init({ */\* 传入配置 \*/* });

</script>

在 dialog.js 前没有引入 util.js，因此 dialog.js 无法正常工作。同样不要以为我上面的故事是虚构的，在我待过的公司里，至今依旧有类似的脚本报错，特别是在各种快速制作的营销页面中。

上面的文件依赖还在可控范围内。当项目越来越复杂，众多文件之间的依赖经常会让人抓狂。下面这些问题，我相信每天都在真实地发生着。

1. 通用组更新了前端基础类库，却很难推动全站升级。
2. 业务组想用某个新的通用组件，但发现无法简单通过几行代码搞定。
3. 一个老产品要上新功能，最后评估只能基于老的类库继续开发。
4. 公司整合业务，某两个产品线要合并。结果发现前端代码冲突。
5. ……

以上很多问题都是因为文件依赖没有很好的管理起来。在前端页面里，大部分脚本的依赖目前依旧是通过人肉的方式保证。当团队比较小时，这不会有什么问题。当团队越来越大，公司业务越来越复杂后，依赖问题如果不解决，就会成为大问题。

文件的依赖，目前在绝大部分类库框架里，比如国外的 YUI3 框架、国内的 KISSY 等类库，目前是通过配置的方式来解决。

YUI.add('my-module', **function** (Y) {

*// ...*

}, '0.0.1', {

requires**:** ['node', 'event']

});

上面的代码，通过 requires 等方式来指定当前模块的依赖。这很大程度上可以解决依赖问题，但不够优雅。当模块很多，依赖很复杂时，烦琐的配置会带来不少隐患。

命名冲突和文件依赖，是前端开发过程中的两个经典问题。下来我们看如何通过模块化开发来解决。为了方便描述，我们使用 Sea.js 来作为模块化开发框架。

## 使用 Sea.js 来解决

Sea.js 是一个成熟的开源项目，核心目标是给前端开发提供简单、极致的模块化开发体验。这里不多做介绍，有兴趣的可以访问 [seajs.org](http://seajs.org/) 查看官方文档。

使用 Sea.js，在书写文件时，需要遵守 CMD （Common Module Definition）模块定义规范。一个文件就是一个模块。前面例子中的 util.js 变成

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

exports.each **=** **function** (arr) {

*// 实现代码*

};

exports.log **=** **function** (str) {

*// 实现代码*

};

});

通过 exports 就可以向外提供接口。这样，dialog.js 的代码变成

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

**var** util **=** require('./util.js');

exports.init **=** **function**() {

*// 实现代码*

};

});

关键部分到了！我们通过 require('./util.js') 就可以拿到 util.js 中通过 exports 暴露的接口。这里的 **require** 可以认为是 Sea.js 给 JavaScript 语言增加的一个 **语法关键字**，通过require 可以获取其他模块提供的接口。

这其实一点也不神奇。作为前端工程师，对 CSS 代码一定也不陌生。

**@import** url**(**"base.css"**)**;

**#id** { **...** }

**.class** { **...** }

Sea.js 增加的 require 语法关键字，就如 CSS 文件中的 @import 一样，给我们的源码赋予了依赖引入功能。

如果你是后端开发工程师，更不会陌生。Java、Python、C# 等等语言，都有include、import 等功能。JavaScript 语言本身也有类似功能，但目前还处于草案阶段，需要等到 ES6 标准得到主流浏览器支持后才能使用。

这样，在页面中使用 dialog.js 将变得非常简单。

<script src="sea.js"></script>

<script>

seajs.use('dialog', **function**(Dialog) {

Dialog.init(*/\* 传入配置 \*/*);

});

</script>

首先要在页面中引入 sea.js 文件，这一般通过页头全局把控，也方便更新维护。想在页面中使用某个组件时，只要通过 seajs.use 方法调用。

好好琢磨以上代码，我相信你已经看到了 Sea.js 带来的两大好处：

1. **通过 exports 暴露接口**。这意味着不需要命名空间了，更不需要全局变量。这是一种彻底的命名冲突解决方案。
2. **通过 require 引入依赖**。这可以让依赖内置，开发者只需关心当前模块的依赖，其他事情 Sea.js 都会自动处理好。对模块开发者来说，这是一种很好的 **关注度分离**，能让程序员更多地享受编码的乐趣。

## 小结

除了解决命名冲突和依赖管理，使用 Sea.js 进行模块化开发还可以带来很多好处：

1. **模块的版本管理**。通过别名等配置，配合构建工具，可以比较轻松地实现模块的版本管理。
2. **提高可维护性**。模块化可以让每个文件的职责单一，非常有利于代码的维护。Sea.js 还提供了 nocache、debug 等插件，拥有在线调试等功能，能比较明显地提升效率。
3. **前端性能优化**。Sea.js 通过异步加载模块，这对页面性能非常有益。Sea.js 还提供了 combo、flush 等插件，配合服务端，可以很好地对页面性能进行调优。
4. **跨环境共享模块**。CMD 模块定义规范与 Node.js 的模块规范非常相近。通过 Sea.js 的 Node.js 版本，可以很方便实现模块的跨服务器和浏览器共享。

# 下载及安装

要在项目中使用SeaJS，你所有需要做的准备工作就是下载sea.js然后放到你项目的某个位置。

下载页面：<http://seajs.org/docs/#downloads>

下载完成后放到项目的相应位置，然后在页面中通过<script>标签引入，你就可以使用SeaJS了。

# 示例

首先是index.html：

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"></script>
10. <script>
11. seajs.use('./init', function(init) {
12. init.initPage();
13. });
14. </script>
15. </body>
16. </html>

可以看到html页面不再需要引入所有依赖的js文件，而只是引入一个sea.js，sea.js会处理所有依赖，加载相应的js文件，加载策略可以选择在渲染页面时一次性加载所有js文件，也可以按需加载（用到时才加载响应js），具体加载策略使用方法下文讨论。

index.html加载了init模块，并使用此模块的initPage方法初始化页面数据，这里先不讨论代码细节。

下面看一下模块化后JavaScript的写法：

1. //jquery.js
2. define(function(require, exports, module) = {
4. //原jquery.js代码...
6. module.exports = $.noConflict(true);
7. });
9. //init.js
10. define(function(require, exports, module) = {
11. var $ = require('jquery');
12. var m1 = require('module1');
14. exports.initPage = function() {
15. $('.content').html(m1.run());
16. }
17. });
19. //module1.js
20. define(function(require, exports, module) = {
21. var $ = require('jquery');
22. var m2 = require('module2');
23. var m3 = require('module3');
25. exports.run = function() {
26. return $.merge(['module1'], $.merge(m2.run(), m3.run()));
27. }
28. });
30. //module2.js
31. define(function(require, exports, module) = {
32. exports.run = function() {
33. return ['module2'];
34. }
35. });
37. //module3.js
38. define(function(require, exports, module) = {
39. var $ = require('jquery');
40. var m4 = require('module4');
42. exports.run = function() {
43. return $.merge(['module3'], m4.run());
44. }
45. });
47. //module4.js
48. define(function(require, exports, module) = {
49. exports.run = function() {
50. return ['module4'];
51. }
52. });

乍看之下代码似乎变多变复杂了，这是因为这个例子太简单，如果是大型项目，SeaJS代码的优势就会显现出来。不过从这里我们还是能窥探到一些SeaJS的特性：

一是html页面不用再维护冗长的script标签列表，只要引入一个sea.js即可。

二是js代码以模块进行组织，各个模块通过require引入自己依赖的模块，代码清晰明了。

通过这个例子朋友们应该对SeaJS有了一个直观的印象，下面本文具体讨论SeaJS的使用。

# 最新版本

与 1.3 相比，2.0 在以下几点做了很大优化：

1. **API 精简**。2.0 的 API 非常简约，常用的只有 7 个： [API 快速参考](https://github.com/seajs/seajs/issues/266)
2. **配置更友好**。让 alias、map 的含义更明确，同时增加了 vars、plugins、data-config 等配置，详见：[配置](https://github.com/seajs/seajs/issues/262)、[模块的加载启动](https://github.com/seajs/seajs/issues/260)
3. **插件优化**。从 2.0 开始，SeaJS 内置了小巧的事件机制，所有插件都基于事件来实现，更简单易懂。详情：[常用插件](https://github.com/seajs/seajs/issues/265)、[插件开发指南](https://github.com/seajs/seajs/issues/264)
4. **对开发调试更友好**。sea.js 已支持 SourceMap，发生错误时，可以快速定位到源码处。另外，增加了 nocache 插件，优化了 debug 插件，详见：[SeaJS 的调试接口](https://github.com/seajs/seajs/issues/263)、[调试插件](https://github.com/seajs/seajs/issues/270)
5. **对 Node.js 的支持更完美**。目前 SeaJS 在 Node.js 的版本也是基于事件扩展，这样能保证与浏览器版本的功能一致性。详见源码，以及 [与 Node.js 兼容](https://github.com/seajs/seajs/issues/275)
6. **文档大量优化**。[seajs.org](http://seajs.org/) 已经全部梳理过一遍，每一篇文档都花了很大心思，推荐无论是新用户还是老用户，都仔细阅读一遍。
7. **测试用例优化**。目前测试用例已达 400 多个，涵盖了绝大部分使用场景。这可以保证 SeaJS 的大胆前行。测试用例在线地址：[tests/runner.html](http://seajs.org/tests/)
8. **对循环依赖的支持更好**。解决了 1.3 中循环依赖检测时的性能问题，同时提示信息更精准。
9. **可以直接使用 jQuery 等非 CMD 模块**。通过 shim 插件，可以直接使用普通 JavaScript 文件了，详见：[常用插件](https://github.com/seajs/seajs/issues/265)

# 参考资料

[Sea.js官方文档](http://seajs.org/docs/#docs)

[**使用SeaJS实现模块化JavaScript开发**](http://blog.codinglabs.org/articles/modularized-javascript-with-seajs.html)