**前端模块化开发的价值**

随着互联网的飞速发展，前端开发越来越复杂。本文将从实际项目中遇到的问题出发，讲述模块化能解决哪些问题，以及如何使用 Sea.js 进行前端的模块化开发。

**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。

模块化开发并不是新鲜事物，但在 Web 领域，前端开发是新生岗位，一直处于比较原始的刀耕火种时代。直到最近两三年，随着 Dojo、YUI3、Node.js 等社区的推广和流行，前端的模块化开发理念才逐步深入人心。

前端的模块化构建可分为两大类。一类是以 Dojo、YUI3、国内的 KISSY 等类库为代表的大教堂模式。在大教堂模式下，所有组件都是颗粒化、模块化的，各组件之间层层分级、环环相扣。另一类是以 jQuery、RequireJS、国内的 Sea.js、OzJS 等类库为基础的集市模式。在集市模式下，所有组件彼此独立、职责单一，各组件通过组合松耦合在一起，协同完成开发。

这两类模块化构建方式各有应用场景。从长远来看，小而美更具备宽容性和竞争力，更能形成有活力的生态圈。

总之，模块化能给前端开发带来很多好处。如果你还没有尝试，不妨从试用 Sea.js 开始。

（完）  
特别感谢这篇文章： <http://chaoskeh.com/blog/why-seajs.html>  
参考了部分内容。