SeaJS 的模块系统遵照 CMD（Common Module Definition）标准。 不过我们还不需要细究那玩意。先从简单的开始吧。

首先，你得在页面上引入它：

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

跟 jQuery 的 $ 和 jQuery 全局对象一样，SeaJS 也通过全局对象与方法暴露出来。 它们分别是 seajs 和 define。

我们可以用 define 定义模块：

```
define('console', function(require, exports) {
    exports.log = function(msg) {
        if (window.console && console.log) {
            console.log(msg);
        }
        else {
            alert(msg);
        }
    };
});
```

用 seajs.use 来使用它：

```
seajs.use('console', function(console) {
    // now you can stop worrying about whether or not `console` is provided.
    // use it freely!
    console.log('hello world!');

    // well, we need to enhance the `console` module a bit.
    // for methods like console.warn, and method calls like console.log(msg1, msg2, msg3);
});
```

模块依赖别的模块时怎么办呢？我们可以把依赖的模块 require 进来：

```
define('jordan', function(require, exports) {
    // 可以内联
    exports.chamionship = function() {
        // 乔丹和皮蓬是对好基友
        return require('pippen').hasJoined();
    };

    // 也可以在头部引入
    var rodman = require('rodman');

    exports.next3 = function() {
        // 大虫
        return require('rodman').hasJoined();
    };
});
```

此外，如果模块的返回值只是纯对象，或者字符串，我们还可以直接写：

```
// 对象
define({ foo: 1 });

// 字符串
define('hello world');
```

可是我们的模块应该拆到单独文件里头去的！别急，且看下文。

SeaJS 之异步

加载基础 js；

根据业务需求不同，加载不同的 js 模块；

模块加载完毕，执行相应的业务逻辑。

那个基础 js，正是 SeaJS 了。

我们先加载：

```
<!-- the library and your app -->
<script src="sea.js"></script>
<script src="app.js"></script>
```

你还可以使用快捷方式，通过给 sea.js 的 script 标签加 @data-main 属性，来指定页面的初始脚本。

```
<!-- more compact way -->
<script src="sea.js" data-main="./app"></script>
```

./ 是指相对于当前页面路径。假定页面的 URL 是 http://foo.com/hello/world.html， 那么 SeaJS 会去加载 http://foo.com/hello/app.js.

./app 在 SeaJS 中被称作模块 ID。相对路径之外，你还可以用绝对路径。 呃，没那么绝对。

在此例中，如果你用 hello/app 来指定模块，SeaJS 将会使用 base 路径 + 模块 ID + .js 这一规则来拼模块的实际 js 地址。

等等，base 路径是神马？

如果模块 ID 不以 . 起始，SeaJS 使用 base 作为基本路径来拼模块的实际地址。 可以通过以下方式来配置 base。

```
seajs.config({
    base: '/'
});
```

如果没有配置，则默认为当前页面的上级目录，即 http://foo.com/hello/。

配置好了 base，hello/app 就可以被解析为 http://foo.com/hello/app.js 啦。

在模块中 require 其他模块的时候，规则也是相同的。相对路径将会相对与当前模块的 uri 来解析。

```
// http://foo.com/worker/carpenter.js
define(function(require, exports) {
    var hammer = require('../util/hammer');

    exports.nail = function() {
        hammer.smash();
    };
});
```

当前模块的 uri 是 http://foo.com/worker/carpenter.js，于是 ../util/hammer 就被解析为 http://foo.com/util/hammer.js。

与 Node.js 的模块机制唯一的不同，就是你的模块代码需要用 define 回调包起来：

```
define(id, dependencies, function(require, exports, module) {
    // module code.
});
```

id 与 dependencies 参数是可以省略的。

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。

模块依赖解析，靠的是三个重要的规则：

不能重命名 require

不能覆盖 require

require 的参数必须是字符串字面量，不可以 require(foo()) 或者 require(bar)， 也不可以是 require(should_be_a ? 'a' : 'b')。

前两点，把 requrie 当做 js 语言中的一个关键字，就容易理解了。我们不会这么做：

```
// 错误!
var func = function;
var function = 'aloha';
// 真的是错误!
```

第三点，则受限于 js 自身。如果我们需要用这样的功能：

```
// 是迈克尔·乔丹，还是迈克尔·杰克逊?
var name = prefer_singer() ? 'jackson' : 'jordan';
var celebrity = require(name);

// 我们要他们的签名!
celebrity.signature();
```

require 是满足不了需求的，可以从两个方面来理解：

如果动态去请求模块，那么 celebrity.signature() 这一步必须在模块请求完毕之后执行，然而动态请求天生就是异步的，没法直接串行执行；

SeaJS 的动态解析模块依赖并预加载依赖的机制在这里也行不通，因为 SeaJS 在解析 define 的回调， 即模块函数体的时候，无从知晓 name 的值，这也是为何 require 的参数必须是字符串字面量。

但如果你真要这么用（因为这么用很爽），也是有办法的。

其一是把依赖的模块都在 define 头部手工声明，不再仰仗 SeaJS 的自动解析功能：

```
define(['jordan', 'jackson'], function(require, exports) {
    var celebrity = require(prefer_singer() ? 'jackson' : 'jordan');

    console.log(celebrity.height());
});
```

而另一种方法，则是使用 require.async：

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

```
    require.async(prefer_singer() ? 'jackson' : 'jordan', function(celebrity) {
        // 偷窥别人的年收入是不对的!
        console.log(celebrity.anual_income());
    });
});
```

具体使用哪一种，则完全视项目需求而定了。这两种的区别有两个：

require.async 方式加载的模块，不能打包工具找到，自然也不能被打包进上线的 js 中；而前一种方式可以。

如果需要在 require 模块之后串行执行代码，仰仗那个模块的返回值，require.async 就做不到了；而前一种可以。

那就专门用前一种？也未尽然，如果需要执行时动态加载的模块很大（比如大量 json 数据），则使用 require.async 才是好选择。 如果只是为了能够在执行时通过反射模式取得模块，并且这些模块都可能被反射到，则不如直接手工写入依赖，即前一种使用方式。

SeaJS 进阶

如何处理复杂代码结构

我们来写个复杂点的例子，写个 hello world 生成工具，会根据传入的编程语言，生成该语言的 hello world 代码片段。

首先先定义各个语言的 hello world：

lang/ruby.js

```
define('' +
    '#!ruby' +
    'puts "Hello, #{' + msg + '}"'
);
```

lang/js.js

```
define('' +
    '(function(console) {' +
    '    console.log("Hello", msg);' +
    '})(window.console || {' +
    '    log: function() {}' +
    '});'
);
```

lang/lisp.js

```
define('' +
    '(print "Hello, ' + string.escape(msg, ['"', '\\']) + '")'
);
```

```
<!doctype html>
<html>
<head></head>
<body>
    <pre id="output" data-lang="ruby"></pre>
    <script src="sea.js" data-main="./generator"></script>
</body>
</html>
```

生成工具需要能够按序执行如下任务：

从 pre 标签读取属性 @data-lang；
加载相应的模块，获取相应的 hello world 写法；
把结果塞到 pre 标签里头去。

```
seajs.use('./util/html', function(HTMLUtil) {
    var pre = document.getElementById('output'),
        lang = pre.getAttribute('data-lang');

    seajs.use('./lang/' + lang, function(lang) {
        pre.innerHTML = HTMLUtil.escape(lang.hello());
    });
});
```

到这里，这个简单的 demo 就完成了。现在回到页首我们提出的问题。以我们目前的 SeaJS 知识，问题已经可以解决了。 项目中的每个页面，都有且仅有一个 script 标签，只不过它们的 @data-main 属性都各不相同。

```
<!-- page a -->
<script src="sea.js" data-main="./page-a"></script>

<!-- page b -->
<script src="sea.js" data-main="./page-b"></script>
```

那些用 (function(){})(); 封装的模块们，现在都可以用 define 来封装到独立文件中去，形成 SeaJS 模块。 模块内部的依赖，使用 require 来搞定。于是页面甲乙丙丁的入口 js，会写成这样：

```
seajs.use(['mod1', 'mod2', 'mod3'], function() {
    // ah my awesome code.
});
```

如此，业务变更只需要改入口文件即可，模块之间的依赖神马的，再不足虑。

入门案例::::::

文件结构

- hello/
    - index.html
    - main.js
    - hello.js

index.html

- 在页面中引入 sea.js
- 通过 data-main 属性配置启动脚本，路径相对于当前页面

```
<!doctype html>
```

```
<html>
    <head>
        <meta charset="utf-8"/>
        <title>Hello, SeaJS!</title>
    </head>
    <body>
        <script src="http://seajs.org/dist/sea.js" data-main="./main"></script>
    </body>
</html>
```

main.js

main.js 作为启动脚本只干两件事：

- 通过 seajs.config(options) 进行配置。本例中 alias 用于配置模块的加载路径，稍后 hello.js 模块的 require(module) 会用到
- 通过 seajs.use(id, callback) 加载模块，use 的模块路径相对于当前页面

```
seajs.config({
    alias: {
        'jquery': 'http://modules.seajs.org/jquery/1.7.2/jquery.js'
    }
});

seajs.use(['./hello'], function(Hello) {
    new Hello()
});
```

hello.js

hello.js 是一个标准的模块文件，其中：

- define(factory) 定义一个模块
- require(module) 调用其他模块，本例中用到的 jquery 路径通过 seajs.config 的 alias 配置
- 通过 return 返回定义的模块

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

    var $ = require('jquery')

    function Hello(){
```

```
        this.render()
    }

    Hello.prototype.render = function(){
        $('<h1 style="display:none;">Hello
SeaJS !</h1>').appendTo('body').fadeIn(2000)
    }

    return Hello

});
```

seajs 模块化 jQuery 与 jQuery 插件

发表于 2013 年 11 月 18 日 由 admin

把 **jQuery** 修改成 **SeaJs** 的模块代码非常简单，就是用下面这段语句将 **jQuery** 源代码包裹起来：

```
define('jquery',[],function(require, exports, module){
        //这里放 jQuery 源代码
        module.exports = jQuery;
});
```

也可以加一个判断，如果 define 已经被定义，就把 jQuery 模块化，如果 define 没有被定义，正常执行 jQuery 代码：

```
/*
 * http://julabs.com
 */
(function(factory) {
        if (typeof define === 'function') {
                define('/jquery', [], factory);
        }
        else {
                factory();
        }

})(function(require) {
        //这里放 jQuery 源代码
```

```
        if (require) return $.noConflict(true);


});
```

如果你用的是 **jQuery1.7** 版本以上的，那就更方便了。可以看下 **jQuery** 源码的最后几行，你会发现类似下方的代码：

```
if ( typeof define === "function" && define.amd && define.amd.jQuery )
{
        define( "jquery", [], function () { return jQuery; } );
}
```

如果判断语句为真，那么 jQuery 就会自动模块化。所以改下判断语句，只留 `typeof define === "function"`，**jQuery** 便可以自动模块化：

```
if ( typeof define === "function") {
        define( "jquery", [], function () { return jQuery; } );
}
```

模块化后的调用代码如下：

```
<script src="/script/sea.js"></script>
<script>
/*
 * http://julabs.com
 */
seajs.config({
        'base':'/script',
        'alias':{
                'jquery':'jquery.sea.js'
        }
});


seajs.use(['jquery'],function($){
        console.log($);
});
</script>
```

jQuery 插件的模块化：

以一个简单的插件为例：

```
/*!
 * http://julabs.com
 */
(function($){
        $.sayHello = function(){
                console.log("Hello");
        };
})(jQuery)
```

一般模块化代码像下面这样：

```
/*!
 * http://julabs.com
 */
(function (factory) {
        if (typeof define === 'function') {
                // 如果 define 已被定义，模块化代码
                define('jquerySayHello',['jquery'], function(){
                        // 返回构造函数
                        return factory
                });
        } else {
                // 如果 define 没有被定义，正常执行插件代码
                factory(jQuery);
        }
}(function ($) {
        // 这里才是插件真正的构造函数
        console.log('init'); // 注意这行代码
        $.sayHello = function(){
                console.log("Hello");
```

```
        };
}));
```

使用插件的代码如下：

```
seajs.config({
        'base':'/script',
        'alias':{
                'jquery':'jquery.sea.js',
                'jquerySayHello':'jquery.sayHello.sea.js'
        }
});

seajs.use(['jquery','jquerySayHello'],function($,jquerySayHello){
        jquerySayHello($); // 初始化插件
        $.sayHello();
});

seajs.use(['jquery','jquerySayHello'],function($,jquerySayHello){
        jquerySayHello($); // 初始化插件
        $.sayHello();
});
```

注意我在插件构造函数里面写的 `console.log('init');`这段代码，可以看到，如果我请求两次插件，插件就要初始化两次。这个虽然可以让代码只在使用时才执行，减少了资源消耗，但如果一个页面中多处需要这个插件的话，就要执行很多次。如果改成下面这种，直接在本模块里就执行：

```
/*!
 * http://julabs.com
 */
(function (factory) {
        if (typeof define === 'function') {
        // 如果 define 已被定义，模块化代码
```

```
        define('jquerySayHello',['jquery'],
function(require,exports,moudles){
                factory(require('jquery')); // 初始化插件
                return jQuery; // 返回 jQuery
        });
} else {
        // 如果 define 没有被定义，正常执行 jQuery
        factory(jQuery);
}
}(function ($) {
        console.log('init');
        $.sayHello = function(){
                console.log("Hello");
        };
}));
```

使用插件的代码如下：

```
/*
 * http://julabs.com
 */
seajs.config({
        'base':'/script',
        'alias':{
                'jquery':'jquery.sea.js',
                'jquerySayHello':'jquery.sayHello.sea.js'
        }
});

seajs.use(['jquery','jquerySayHello'],function($){
        $.sayHello();
});
```

```
seajs.use(['jquery','jquerySayHello'],function($){
        $.sayHello();
});
```