HelloSea.js中文版

极客学院出版

前言

《Hello Sea.js》是一本Sea.js的入门指南，对Sea.js进行了全方位的介绍。通读本书，你能够了解Sea.js各个细节，甚至整个前端模块化的大框架。本书既是一本参考手册，可以随时查阅；也是对前端模块化的一次剖析，展望。

如果您觉得本书并不是如上一段描述的这样，欢迎讨论、意见，更期待您的贡献！

前言

后浪推前浪，在早几年前，前端界最火的莫过于jQuery，那是个插件纷飞的年代。得jQuery者得天下。而现在，CommonJS草案的提出，Node.js让JavaScript在服务端大展拳脚，前端界已经不是那个手持jQuery的小孩了。

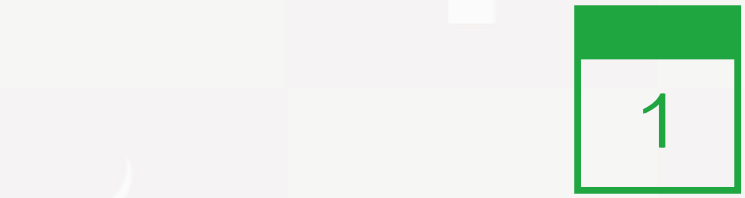
在这个新的浪潮中，JavaScript模块化开发开始流行起来。CommonJS标准制定后，Node.js兴起，RequireJS使得JavaScript模块化在客户端齐头并进，ES6模块标准呼之欲出，涌现出了很多模块化的方案，兼容ES6也好，不兼容也罢；国内外相关的项目如雨后春笋般涌现出来，谁都有可能引领标准。而本书正是关于Sea.js，关于前端模块化的小书。记录一些我在模块化方面的见闻、理解和思考。

> 本书基于Sea.js v2.1.1完成。
>
> 致谢
> 内容撰写：https://github.com/island205/HelloSea.js/

更新日期	更新内容
2015-05-26	Hello Sea.js 中文版

目录

前言 1
第 1 章　Sea.js是什么? 3
第 2 章　快速指南 9
第 3 章　使用指南 12
第 4 章　开发实战 25
第 5 章　Sea.js是如何工作的? 41
第 6 章　模块化JavaScript的未来 56
第 7 章　参考资料 58



1 Sea.js是什么?

起初被看作是一门玩具语言的JavaScript，最近已经发生了很大的变化。变化之一就是从HTML中的 <script> 标签转向了模块化。

模块化

模块就是一团黑乎乎的东西，有份文档会教你如何使用这团东西，你只知道它的接口，但不知道它内部是如何运作的，但这个模块能满足你的需求。

过程、函数、类都可以称作为模块，它们有一个共同的特点就是封装了功能，供外界调用。对于特定的语言，模块所指的东西各有不同。

在Python中，

> 模块基本上就是一个包含了所有你定义的函数和变量的文件。

我们来定义一个Python的模块：

```
#!/usr/bin/env python
# Filename: greet.py

def hello_python():
   print "Hello,Python"

def hello_javascript():
   print "Hello,JavaScript"
```

真的，就是这么简单，我们可以这样使用：

```
#!/usr/bin/env python
# Filename: use_greet.py

import greet

# call greet module's func
# print "Hello,Python"
greet.hello_python()
```

greet.py的模块中有两个方法，把它们import到use_greet.py中，我们就可以使用了。 Python还提供了另外一种引入模块的方法：

```
#!/usr/bin/env python
# Filename: use_greet.py
```

```
from greet import hello_python

 # call greet module's func
 # print "Hello,Python"
hello_python()
```

可以引入模块特定的API。

JavaScript的模块化

那JavaScript有模块化吗？我想说有，而且是与它一样的，看下面的例子：

```
// File: greet.js
function helloPython(){
    document.write("Hello,Python");
}
function helloJavaScript(){
    document.write("Hello,JavaScript");
}

// File:usegreet.js
helloJavaScript();
```

```
<!DOCTYPE html>
<!--index.html-->
<script src="./greet.js"></script>
<script src="./usegreet.js"></script>
```

在浏览器中打开index.html：

> Hello,JavaScript

可以看到，JavaScript这种通过全局共享的方式确实可以实现模块化，你只需要在HTML中引入需要使用的模块脚本即可。

但这样的模块化有两个很实在的问题：

1. 必须通过全局变量共享模块，有可能会出现命名冲突的问题；
2. 依赖的文件必须手动地使用标签引入到页面中。

Node.js的模块化

这些问题如何解决呢？我们要不再来看一下Node.js的模块。你应该知道Node.js，现在它是火得不行！

```
// File:greet.js
exports.helloPython = function() {
    console.log("Hello,Python");
}
exports.helloJavaScript = function() {
    console.log("Hello,JavaScript");
}

// File: usegreet.js
var greet = require("./greet");
greet.helloJavaScript();
```

运行 `node usegreet.js` ，控制台会打印：

```
Hello,JavaScript
```

Node.js 把 JavaScript 移植到了 Server 端的开发中，Node.js 通过 exports 和 require 来实现了代码的模块化组织。在一个 Node.js 的模块文件中，我们可以使用 exports 把对外的接口暴露出来，其他模块可以使用require函数加载其他文件，获得这些接口，从而使用模块提供出来的功能，而不关心其实现。在npmjs.org上已经有上万的Node.js开源模块了！

ECMA 标准草案

Node.js 模块化的组织方案是 Server 端的实现，并不能直接在浏览器中使用。JavaScript 原生并没有支持 `exports` 和 `require` 关键字。ECMAScript6 标准草案 harmony 已经考虑到了这种模块化的需求。举个例子：

```
// Define a module
module 'greet' {
    export function helloPython() {
        console.log("Hello,Python")
    }
    export function helloJavaScript() {
        console.log("Hello,JavaScript")
    }
}

// Use module
import {helloPython, helloJavaScript} from 'greet'
helloJavaScript()

// Or

module Greet from 'greet'
```

```
Greet.helloJavaScript()

// Or remote module
module Greet from 'http://bodule.org/greet.js'
Greet.helloJavaScript()
```

可以到这里查看更多的例子 (http://wiki.ecmascript.org/doku.php?id=harmony:modules_examples) 。

参考es6-module-loader (https://github.com/ModuleLoader/es6-module-loader) 这个项目。

不过该标准还处于草案阶段，没有主流的浏览器所支持，那我们该怎么办？恩，已经有一些先行者了。

LABjs

LABjs (https://github.com/getify/LABjs) 是一个动态的脚本加载类库，替代难看的，低性能的 <script> 标签。该类库可以并行地加载多个脚本，可按照需求顺序执行依赖的代码，这样在保证依赖的同时大大提高的脚本的加载速度。

LABjs 已经三岁了，其作者 getify 声称，由于社区里大家更喜欢使用 AMD 模式，随在 2012 年 7 月 25 号停止对该类库的更新。但 LABjs 绝对是 JavaScript 在浏览器端模块化的鼻祖，在脚本加载方面做了大量的工作。

requirejs

与 LABjs 不同的地方在于，RequireJS 是一个动态的模块加载器。其作者 James Burke 曾是 Dojo 核心库 loader 和 build system 的开发者。2009 年随着 JavaScript 代码加载之需要，在 Dojo XDloader 的开发经验基础之上，它开始了新项目RunJS。后更名为RequireJS，在AMD模块提案指定方面，他起到了重要的作用。James从XDloader到RunJS 再到RequireJS一直在思考着该如何实现一个module wrapper，让更多的js、更多的node模块等等可以在浏览器环境中无痛使用。

seajs

seajs 相对于前两者就比较年轻，2010 年玉伯发起了这个开源项目，SeaJS 遵循 CMD 规范，与 RequireJS 类似，同样做为模块加载器。那我们如何使用 seajs 来封装刚才的示例呢?

```
// File:greet.js
define(function (require, exports) {
   function helloPython() {
      document.write("Hello,Python");
   }
```

```
    function helloJavaScript() {
        document.write("Hello,JavaScript");
    }
    exports.helloPython = helloPython;
    exports.helloJavaScript = helloJavaScript;
});

// File:usegreet.js
sea.use("greet", function (Greet) {
    greet.helloJavaScript();
});
```

2

快速指南

还记得jQuery如何使用的么？Sea.js也是如此。例子在这里 (https://github.com/island205/HelloSea.js) 可以找到，用anywhere (https://github.com/JacksonTian/anywhere) 起个静态服务来看看。

首先写个模块：

```
// File:js/module/greet.js
define(function (require, exports) {
    function helloPython() {
        document.write("Hello,Python");
    }
    function helloJavaScript() {
        document.write("Hello,JavaScript");
    }
    exports.helloPython = helloPython;
    exports.helloJavaScript = helloJavaScript;
});
```

如果你对Node.js非常熟悉，你可以把这个模块理解为Node.js的模块加上一个Wrapper。

在页面中引入Sea.js：

```
<!-- File:index.html -->
<!DOCTYPE HTML>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Getting start with Sea.js</title>
    <!-- 引入seajs-->
    <script src="/js/sea.js"></script>
</head>
<body>

</body>
</html>
```

加载模块文件！

```
<!DOCTYPE HTML>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <title>Getting start width Sea.js</title>
    <!-- 引入seajs-->
    <script src="/js/sea.js"></script>
    <script>
        seajs.use(['/js/module/greet'], function (Greet) {
            Greet.helloJavaScript()
        })
    </script>
</head>
<body>

</body>
</html>
```

看到页面上输出的 Hello,JavaScript 么，这确实太简单了！

3

使用指南

刚才的示例很简单？实际上Sea.js本身小巧而不失灵活，让我们再来深入地了解下如何使用Sea.js!

定义模块

Sea.js是[CMDhref="https://github.com/cmdjs/specification/blob/master/draft/module.md")这个模块系统的一个运行时，Sea.js可以加载的模块，就是CMD规范里所指明的。那我们该如何编写一个CMD模块呢?

Sea.js提供了一个全局方法—— `define` ，用来定义一个CMD模块。

`define(factory)`

```
define(function(require, exports, module) {
    // 模块代码
    // 使用require获取依赖模块的接口
    // 使用exports或者module来暴露该模块的对外接口
})
```

`factory` 是这样一个函数 `function (require?, exports?, module?) {}` ，如果模块本身既不依赖其他模块，也不提供接口， `require` 、 `exports` 和 `module` 都可以省略。但通常会是以下两种形式：

```
define(function(require, exports) {
    var Vango = require('vango')
    exports.drawCircle = function () {
        var vango = new Vango(document.body, 100, 100)
        vango.circle(50, 50, 50, {
            fill: true,
            styles:{
                fillStyle:"red"
            }
        })
    }
})
```

或者：

```
define(function(require, exports, module) {
    var Vango = require('vango');
    module.exports = {
        drawCircle: function () {
            var vango = new Vango(document.body, 100, 100);
            vango.circle(50, 50, 50, {
                fill: true,
                styles:{
                    fillStyle:"red"
                }
```

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

> 注意：必须保证参数的顺序，即需要用到require， exports不能省略；在模块中exports对象不可覆盖，如果需要覆盖请使用 module.exports 的形式（这与node的用法一致，在后面的原理介绍会有相关的解释）。你可以使用 module.exports 来export任意的对象（包括字符串、数字等等）。

`define(id?, dependencies?, factory)`

id：String 模块标识

dependencies：Array 模块依赖的模块标识

这种写法属于Modules/Transport/D (http://wiki.commonjs.org/wiki/Modules/Transport/D) 规范。

```
define('drawCircle', ['vango'], function(require, exports) {
  var Vango = require('vango');
  exports.drawCircle = function () {
    var vango = new Vango(document.body, 100, 100);
    vango.circle(50, 50, 50, {
      fill: true,
      styles:{
        fillStyle:"red"
      }
    });
  };
})
```

与CMD的 define 没有本质区别，我更情愿把它称作“具名模块”。Sea.js从用于生产的角度来说，必须支持具名模块，因为开发时模块拆得太小，生产环境必须把这些模块文件打包为一个文件，如果模块都是匿名的，那就傻逼了。（为什么会傻逼？（https://github.com/seajs/seajs/issues/930））

> 所以Sea.js支持具名模块也是无奈之举。

`define(anythingelse)`

除去以上两种形式，在CMD标准中，可以给define传入任意的字符串或者对象，表示接口就是对象或者字符串。不过这只是包含在标准中，在Sea.js并没有相关的实现。

配置Sea.js

Sea.js为了能够使用起来更灵活，提供了配置的接口。可配置的内容包括静态服务的位置，简化模块标识或路径。接下来我们来详细地了解下这些内容。

seajs.config(config)

config：Object，配置键值对。

Sea.js通过 .config API来进行配置。你甚至可以在多个地方调用seajs.config来配置。Sea.js会mix传入的多个config对象。

```
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']
})
```

上面两个配置会合并为：

```
{
  alias: {
    'jquery': 'path/to/jquery.js',
    'underscore': 'path/to/underscore.js',
    'a': 'path/to/biz/a.js'
  },
  preload: ['seajs-text', 'seajs-combo']

}
```

config 可以配置的键入下：

base

base：String，在解析绝对路径标识的模块时所使用的base路径。

默认地，在不配置base的情况下，base与sea.js的引用路径。如果引用路径为 http://example.com/assets/sea.js ，则base为 http://example.com/assets/ 。

> 在阅读Sea.js这份文档时看到：
> *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。*

> 即如果sea.js的引用路径为http://example.com/assets/1.0.0/sea.js，则base仍为http://example.com/assets/。这种方便性，我觉得过了点。

使用base配置，根本上可以分离静态文件的位置，比如使用CDN等等。

```
seajs.config({
  base: 'http://g.tbcdn.cn/tcc/'
})
```

> 如果我们有三个CDN域名，如何将静态资源散列到这三个域名上呢?

paths

paths：Object，如果目录太深，可以使用paths这个配置项来缩写，可以在require时少写些代码。

如果：

```
seajs.config({
  base: 'http://g.tbcdn.cn/tcc/',
  paths: {
    'index': 's/js/index'
  }
})
```

则：

```
define(function(require, exports, module) {
  // http://g.tbcdn.cn/tcc/s/js/index/switch.js
  var Switch = require('index/switch')
});
```

alias

alias：Object，本质上看不出和paths有什么区别，区别就在使用的概念上。

```
seajs.config({
  alias: {
    'jquery': 'jquery/jquery/1.10.1/jquery'
  }
})
```

然后：

```
define(function(require, exports, module) {
  // jquery/jquery/1.10.1/jquery
  var $ = require('jquery');
});
```

> 看出使用概念的区别了么?

preload

preload 配置项可以让你在加载普通模块之前提前加载一些模块。既然所有模块都是在use之后才加载的，preload有何意义? 然，看下面这段:

```
seajs.config({
  preload: [
    Function.prototype.bind ? '' : 'es5-safe',
    this.JSON ? '' : 'json'
  ]
});
```

preload比较适合用来加载一些核心模块，或者是shim模块。这是一个全局的配置，使用者无需关系核心模块或者是shim模块的加载，把注意力放在核心功能即可。

还有一些别的配置，比如 vars 、 map 等，可以参考配置 (https://github.com/seajs/seajs/issues/262) 。

使用模块

seajs.use(id)

Sea.js通过use方法来启动一个模块。

```
seajs.use('./main')
```

在这里， ./main 是main模块的id，Sea.js在main模块LOADED之后，执行这个模块。

Sea.js还有另外一种启动模块的方式:

seajs.use(ids, callbacks)

```
seajs.use('./main', function(main) {
  main.init()
})
```

Sea.js执行ids中的所有模块，然后传递给callback使用。

插件

Sea.js官方提供了7个插件，对Sea.js的功能进行了补充。

- seajs-text：用来加载HTML或者模板文件;
- seajs-style：提供了 importStyle ，动态地向页面中插入css;

- seajs-combo：该插件提供了依赖combo的功能，能把多个依赖的模块uri combo，减少HTTP请求；
- seajs-flush：该插件是对seajs-combo的补充，或者是大杀器，可以先hold住前面的模块请求，最后将请求的模块combo成一个url，一次加载hold住的模块；
- seajs-debug：Fiddler用过么？这个插件基本就是提供了这样一种功能，可以通过修改config，将线上文件proxy到本地服务器，便于线上开发调试和排错；
- seajs-log：提供一个seajs.log API，私觉得比较鸡肋；
- seajs-health：目标功能是，分析当前网页的模块健康情况。

由此可见，Sea.js的插件主要是解决一些附加问题，或者是给Sea.js添加一些额外的功能。私觉得有些功能并不合适让Sea.js来处理。

插件机制

总结一下，插件机制大概就是两种：

- 使用Sea.js在加载过程中的事件，注入一些插件代码，修改Sea.js的运行流程，实现插件的功能；
- 给seajs加入一些方法，提供一些额外的功能。

> 私还是觉得Sea.js应该保持纯洁；为了实现插件，在Sea.js中加入的代码，感觉有点不值；combo这种事情，更希望采取别的方式来实现。 Sea.js应该做好运行时。

构建与部署

很多时候，某个工具或者类库，玩玩可以，但是一用到生产环境，就感觉力不从心了。就拿Sea.js来说，开发的时候根据业务将逻辑拆分到很多小模块，逻辑清晰，开发方便。但是上线后，模块太多，HTTP请求太多，就会拖慢页面速度。

所以我们必须对模块进行打包压缩。这也是SPM的初衷。

SPM是什么?

使用者认为SPM是Sea.js Package Manager，但是实际上代表的是Static Package Manager，及静态包管理工具。如果大家有用过npm，你可以认为SPM是一个针对前端模块的包管理工具。当然它不仅仅如此。

SPM包括：

- 源服务：类似于npm源服务器的源服务；
- 包管理工具：相当于npm的命令行，安装、发布模块，解决模块依赖；
- 构建工具：模块CMD化、合并模块、压缩等；都是针对我们一开始提到的问题；

- 配置管理：管理配置；
- 辅助功能：比较像Yeoman，以插件提供一些便于平时开发的组件。

> SPM心很大，SPM囊括yo、bower和grunt这三个工具。

spm

> spm is a package manager, it is not build tools.

这句话来自github上spm2 (https://github.com/spmjs/spm2) 的README文件。`spm是一个包管理工具，不是构建工具！`，它与npm非常相似。

spm的包规范

一个spm的模块至少包含：

```
-- dist
   -- overlay.js
   -- overlay.min.js
-- package.json
```

package.json

在模块中必须提供一个package.json，该文件遵循Common Module Definition (https://github.com/cmdjs/specification) 模块标准。与node的 `package.json` 兼容。在此基础上添加了两个key。

- family，即是包发布者在spmjs.org上的用户名；
- spm，针对spm的配置。

一个典型的 `package.json` 文件：

```
{
   "family": "arale",
   "name": "base",
   "version": "1.0.0",
   "description": "base is ....",
   "homepage": "http://aralejs.org/base/",
   "repository": {
      "type": "git",
      "url": "https://github.com/aralejs/base.git"
   },
   "keywords": ["class"],

   "spm": {
      "source": "src",
```

```
    "output": ["base.js", "i18n/*"],
    "alias": {
      "class": "arale/class/1.0.0/class",
      "events": "arale/events/1.0.0/events"
    }
  }
}
```

dist

`dist` 目录包含了模块必要的模块代码；可能是使用spm-build打包的，当然只要满足两个条件，就是一个spm的包。

安装

```
$ npm install spm -g
```

安装好了spm，那该如何使用spm呢？让我们从help命令开始：

help

我们可以运行 `spm help` 查看 `spm` 所包含的功能：

```
$ spm help

  Static Package Manager

  Usage: spm <command> [options]

  Options:

    -h, --help     output usage information
    -V, --version  output the version number

  System Commands:

    plugin        plugin system for spm
    config        configuration for spm
    help          show help information

  Package Commands:

    tree          show dependencies tree
    info          information of a module
    login         login your account
    search        search modules
    install       install a module
    publish       publish a module
    unpublish     unpublish a module
```

```
Plugin Commands:

  init          init a template
  build         Build a standar cmd module.
```

spm 包含三种命令，**系统命令**，即与 spm 本身相关（配置、插件和帮助），**包命令**，与包管理相关，**插件命令**，插件并不属于 spm 的核心内容，目前有两个插件 init 和 build 。

也可以使用 help 来查看单个命令的用法：

```
$ spm help install

  Usage: spm-install [options] family/name[@version]

  Options:

    -h, --help              output usage information
    -s, --source [name]     the source repo name
    -d, --destination [dir]  the destination, default: sea-modules
    -g, --global            install the package to ~/.spm/sea-modules
    -f, --force             force to download a unstable module
    -v, --verbose           show more logs
    -q, --quiet             show less logs
    --parallel [number]     parallel installation
    --no-color              disable colorful print


  Examples:

    $ spm install jquery
    $ spm install jquery/jquery arale/class
    $ spm install jquery/jquery@1.8.2
```

config

我们可以使用 config 来配置用户信息、安装方式以及源。

```
; Config username
$ spm config user.name island205

; Or, config default source
$ spm config source.default.url http://spmjs.org
```

search

spm 是一个包管理工具，与 npm 类似，有自己的源服务器。我们可以使用 search 命令来查看源提供的包。

> 由于 spm 在包规范中加入了 family 的概念，常常想运行 spm install backbone ，发现并没有backbone这个包。原因就是 backbone 是放在 gallery 这族下的。

```
$ spm search backbone

 1 result

 gallery/backbone
 keys: model view controller router server client browser
 desc: Give your JS App some Backbone with Models, Views, Collections, and Events.
```

install

然后我们就可以使用 install 来安装了，注意我们必须使用包的全名，即 族名/包名 。

```
$ spm install gallery/backbone

      install: gallery/backbone@stable
        fetch: gallery/backbone@stable
     download: repository/gallery/backbone/1.0.0/backbone-1.0.0.tar.gz
         save: c:\Users\zhi.cun\.spm\cache\gallery\backbone\1.0.0\backbone-1.0.0.tar.gz
      extract: c:\Users\zhi.cun\.spm\cache\gallery\backbone\1.0.0\backbone-1.0.0.tar.gz
        found: dist in the package
    installed: sea-modules\gallery\backbone\1.0.0
     depends: gallery/underscore@1.4.4

      install: gallery/underscore@1.4.4
        fetch: gallery/underscore@1.4.4
     download: repository/gallery/underscore/1.4.4/underscore-1.4.4.tar.gz
         save: c:\Users\zhi.cun\.spm\cache\gallery\underscore\1.4.4\underscore-1.4.4.tar.gz
      extract: c:\Users\zhi.cun\.spm\cache\gallery\underscore\1.4.4\underscore-1.4.4.tar.gz
        found: dist in the package
    installed: sea-modules\gallery\underscore\1.4.4
```

spm 将模块安装在了 sea_modules 中，并且在 ~/.spm/cache 中做了缓存。

```
`~sea-modules/
 `~gallery/
  |~backbone/
  | `~1.0.0/
  |   |-backbone-debug.js
  |   |-backbone.js
  |   `-package.json
  `~underscore/
    `~1.4.4/
      |-package.json
```

```
    |-underscore-debug.js
    `-underscore.js
```

spm 还加载了 backbone 的依赖 underscore 。

当然，Sea.js也是一个模块，你可以通过下面的命令来安装：

```
$ spm install seajs/seajs
```

seajs 的安装路径为 sea_modules/seajs/seajs/2.1.1/sea.js ，看到这里，结合seajs顶级模块定位的方式，对于seajs在计算base路径的时，去掉了 seajs/seajs/2.1.1/ 的原因。

build

spm 并不是以构建工具为目标，它本身是一个包管理器。所以 spm 将构建的功能以插件的功能提供出来。我们可以通过plugin命令来安装 build ：

```
$ spm plugin install build
```

安装好之后，如果你使用的是标准的 spm 包模式，就可以直接运行 spm build 来进行标准的打包。

SPM2的功能和命令就介绍到这里，更多的命令在之后的实践中介绍。

spm与spm2

spm与spm2

其实之前介绍的spm是其第二个版本spm2 (https://github.com/spmjs/spm2) 。spm的第一个版本可以在这里 (https://github.com/spmjs/spm) 找到。

spm与spm2同样都是包管理工具，那它们之间有什么不同呢?

- 从定位上，spm2更加强调该工具是一个CMD包管理工具；
- 从提供的用户接口（cmd命令）spm2比起spm更加规范，作为包管理工具，在使用方式和命令都更趋同于npm；
- 在spm2中，构建命令以插件的方式独立出来，并且分层清晰；Transport和Concat封装成了grunt，便于自定义build方式；基于基础的grunt，构建了一个标准的spm-build工具，用于构建标准的CMD模块；
- 与此类似，deploy和init的功能都是以插件的形式提供的；
- 修改了package.json规范。

为什么作者对spm进行了大量的重构?

之所以进行大量的重构，就是为了保持spm作为包管理工具的特征。如npm一般，只指定最少的规范（package.json），提供包管理的命令，但是这个包如何构建，代码如何压缩并不是spm关心的事情。

只有规则简单合理，只定义接口，不关心具体实现，才有更广的实用性。

> spm本身是从业务需求成长起来的一个包管理工具，spm1更多的是一些需求功能的堆砌，而spm2就是对这些功能的提炼，形成一套适用于业界的工具。

apm

apm的全称是：

> Alipay package manager

即支付宝的包管理工具。

apm是基于spm的一套专门为支付宝开发的工具集。我们可以这么看，spm2和apm是spm升级后的两个产物，spm2更加专注于包管理和普适性，而apm更加专注于支付宝业务。由于业务细节和规模的不同，apm可能并不适合其他公司所用，所以需要spm2，而又因为支付宝业务的特殊性和基因，必须apm。

谢谢 @lepture 的指正：

> 不一定要用 apm，只是 apm 把所有要用到的插件都打包好了，同时相应的默认配置也为支付宝做了处理，方便内部员工使用，不用再配置了。

4

开发实战

venus-in-cmd

Venus是一个javascript类库，是一个canvas的wrapper，为了学习spm，我们使用cmd的模式来重构这个类库。

安装spm-init

spm提供了初始cmd模块的脚手架，我们可以使用下面的命令来安装这个脚手架：

```
$ spm plugin install init
```

初始化一个cmd项目

运行：

```
$ spm init
```

就可以初始化一个cmd模块的项目，回答一些spm的问题，就能在当前目录生成必要的文件和文件夹：

```
|~examples/
| `-index.md
|~src/
| `-venus.js
|~tests/
| `-venus-spec.js
|-LICENSE
|-Makefile
|-package.json
`-README.md
```

我们在 `src` 中添加 `venus` 的代码。

编写cmd模块

或者将现有的模块转化为cmd模块。

本例中的Venus (https://github.com/island205/Venus) 本来就已经存在，那我们如何将其转成cmd模块呢?

在Venus的源码中我惊喜地发现这段代码：

```
// File: vango.js

/*
 * wrapper for browser,nodejs or AMD loader evn
 */
(function(root, factory) {
    if (typeof exports === "object") {
        // Node
        module.exports = factory();
    // AMD loader
    } else if (typeof define === "function" && define.amd) {
        define(factory);
    } else {
        // Browser
        root.Vango = factory();
    }
})(this, function() {
    // Factory for build Vango
})
```

这段代码可以令 vango.js 支持浏览器（通过script直接引入）、node环境以及AMD加载器。

于是事情就简单了，因为我们可以很简单地将一个Node模块转成CMD模块，添加如下的wrapper即可：

```
// File: vango.js
define(function (require, exports, module) {
    (function(root, factory) {
        if (typeof exports === "object") {
            // Node
            module.exports = factory();
        // AMD loader
        } else if (typeof define === "function" && define.amd) {
            define(factory);
        } else {
            // Browser
            root.Vango = factory();
        }
    })(this, function() {
        // Factory for build Vango
        // return Vango
    })
})
```

UMD

上面那段有点黑魔法的代码还有一个更复杂的形式，即Universal Module Definition (https://github.com/umdjs/umd)：

```
(function (root, factory) {
  if (typeof define === 'function' && define.amd) {
    // AMD
    define('backbone', ['jquery', 'underscore'], function (jQuery, _) {
      return factory(jQuery, _);
    });
  } else if (typeof exports === 'object') {
  // Node.js
    module.exports = factory(require('jquery'), require('underscore'));
  } else {
    // Browser globals
    root.Backbone = factory(root.jQuery, root._);
  }
}(this, function (jQuery, _) {
  var Backbone = {};
  // Backbone code that depends on jQuery and _
  return Backbone;

}));
```

当然并不是所有的CMD模块都得这么写，你可以按照自己的方式，使用 require 、 exports 和 module 这三个关键字，遵循CMD的规范即可。

最后 src 有两个文件， venus.js 就很简单了：

```
define(function (require, exports, module) {
    var Vango = require('./vango');
    exports.Vango = Vango;
})
```

我们的venus的cmd版本搞定了，vango.js作为vango具体实现，而venus.js只是这些将这些画家暴露出来。

构建

作为标准的cmd模块，我们可以使用 spm-build 来构建，别忘了之前提到的，你可以使用 spm plugin install build 来安装。

在项目的根目录下运行 spm build ：

```
$ spm build
        Task: "clean:build" (clean) task

        Task: "spm-install" task
```

```
     Task: "transport:src" (transport) task
transport: 2 files

     Task: "concat:css" (concat) task
 concated: 0 files

     Task: "transport:css" (transport) task
transport: 0 files

     Task: "concat:js" (concat) task
 concated: 2 files

     Task: "copy:build" (copy) task

     Task: "cssmin:css" (cssmin) task

     Task: "uglify:js" (uglify) task
     file: ".build/dist/venus.js" created.

     Task: "clean:dist" (clean) task

     Task: "copy:dist" (copy) task
   copied: 2 files

     Task: "clean:build" (clean) task
 cleaning: ".build"...

     Task: "spm-newline" task
   create: dist/venus-debug.js
   create: dist/venus.js

     Done: without errors.
```

从构建的log中可以看出，`spm`完全就是使用grunt来构建的，涉及到多个grunt task。你完全可以自己编写Gruntfile.js来实现自定义的构建过程。

venus就被构建好了，`spm`在目录中生成了一个`dist`文件夹：

```
|~dist/
 |-venus-debug.js
 `-venus.js
```

`venus-debug.js`中的内容为：

```
define("island205/venus/1.0.0/venus-debug", [ "./vango-debug" ], function(require, exports, module) {
  var Vango = require("./vango-debug");
```

```
    exports.Vango = Vango;
});

define("island205/venus/1.0.0/vango-debug", [], function(require, exports, module) {
    // Vango's code
})
```

venus.js 的内容与之一样，只是经过了压缩，去掉了模块名最后的 -debug 。

spm 将 src 中的vango.js和venus.js根据依赖打到了一起。作为包的主模块，venus.js被放到了最前面。

> 这是Sea.js的默认约定，打包后的模块文件其中的一个define即为该包的主模块（ ID 和路径相匹配的那一个），也就是说，你通过 require('island205/venus/1.0.0/venus') ，虽然Sea.js加载的是整个打包的模块，但是会把的一个factory的exports作为venus暴露的接口。

发布

如果你用过npm，那你对spm的发布功能应该不会陌生了。spm也像npm一样，有一个公共仓库，我们可以通过 spm plublish 将venus发布到仓库中，与大家共享。

```
$ spm publish
      publish: island205/venus@1.0.0
        found: readme in markdown.
      tarfile: venus-1.0.0.tar.gz
      execute: git rev-parse HEAD
         yuan: Authorization required.
         yuan: `spm login` first
```

如果你碰到上面这种情况，你需要登录下。

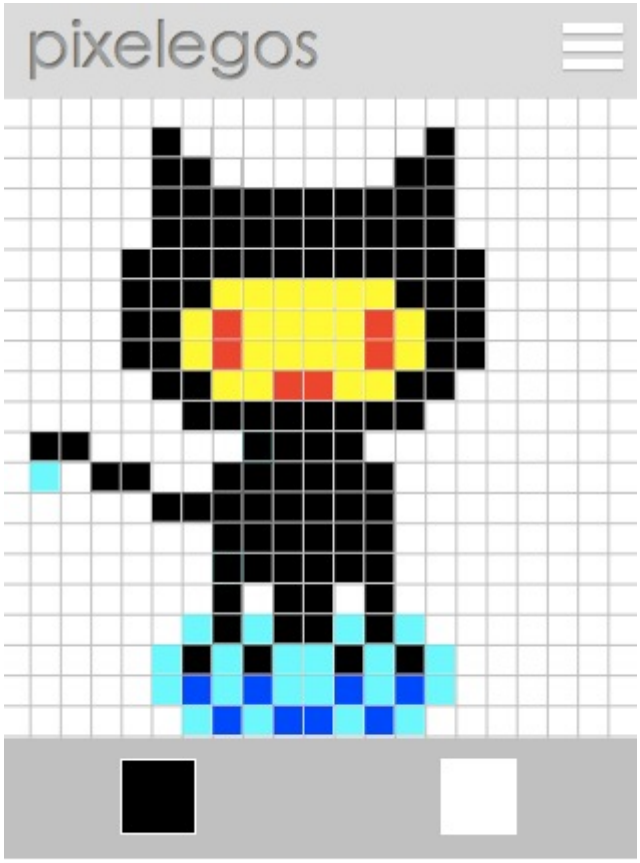
```
$ spm publish
      publish: island205/venus@1.0.0
        found: readme in markdown.
      tarfile: venus-1.0.0.tar.gz
      execute: git rev-parse HEAD
    published: island205/venus@1.0.0
```

接下来我们使用venus编写一个名为pixelegos的网页程序，你可以使用这个程序来生成一些头像的位图。例如，spmjs的头像(这是github为spmjs生成的随机头像)：

图片 4.1 spmjs

pixelegos

pixelegos完成后的样子：



图片 4.2 pixelegos

准备

创建一个名为 pixelegos 的文件夹，初始化一个npm项目：

```
$ mkdir pixelegos && cd pixelegos && npm init
```

在目录中多了一个packege.json文件，在这个文件中包含了一些pixelegos的信息，之后还会保存一些node module和spm的配置。

安装依赖

本项目中需要依赖的cmd模块包括 backbone 、 seajs 、 venus 、 zepto 。我们运行下面的命令安装这些依赖:

```
$ spm install seajs/seajs gallery/backbone zepto/zepto island205/venus
```

在 pixelegos 目录下增加了一个 sea-modules 目录，上面的cmd依赖都安装在这个目录中，由于backone依赖于underscore，spm自动安装了依赖。

```
├── gallery
│   ├── backbone
│   │   └── 1.0.0
│   │       ├── backbone-debug.js
│   │       ├── backbone.js
│   │       └── package.json
│   └── underscore
│       └── 1.4.4
│           ├── package.json
│           ├── underscore-debug.js
│           └── underscore.js
├── island205
│   └── venus
│       └── 1.0.0
│           ├── package.json
│           ├── venus-debug.js
│           └── venus.js
├── seajs
│   └── seajs
│       └── 2.1.1
│           ├── package.json
│           ├── sea-debug.js
│           ├── sea.js
│           └── sea.js.map
└── zepto
    └── zepto
        └── 1.0.0
            ├── package.json
            ├── zepto-debug.js
            └── zepto.js
```

开始

新建一些html、css、js文件，结构如下：

```
├── index.css
├── index.html
├── js
│   ├── canvas.js
│   ├── config.js
│   ├── menu.js
```

```
|   ├── pixelegos.js
|   └── tool.js
├── package.json
└── sea-modules
    ├── gallery
    ├── island205
    ├── seajs
    └── zepto
```

给index.html添加如下内容：

```
<!DOCTYPE HTML>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
    <meta content='width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=0' name='viewport' />
    <meta name="viewport" content="width=device-width" />
    <link rel="stylesheet" href="/index.css" />
    <script type="text/javascript" src="/sea-modules/seajs/seajs/2.1.1/sea-debug.js"></script>
    <script type="text/javascript" src="/config.js"></script>
    <script type="text/javascript">
        seajs.use('/js/pixelegos')
    </script>
</head>
<body>
</body>
</html>
```

其中，config.js在开发时用来配置alias，pixelegos作为整个程序的启动模块。

```
// config.js
seajs.config({
    alias: {
        '$': 'zepto/zepto/1.0.0/zepto',
        "backbone": "gallery/backbone/1.0.0/backbone",
        "venus": "island205/venus/1.0.0/venus"
    }
})
```

```
// pixelegos.js
define(function (require, exports, module) {
    var Menu = require('./menu')
    var Tool = require('./tool')
    var Canvas = require('./canvas')
    var $ = require('$')
```

```
    $(function() {
        var menu = new Menu()
        var tool = new Tool()
        var canvas = new Canvas()
        tool.on('select', function(color) {
            canvas.color = color
        })
        tool.on('erase', function() {
            canvas.color = 'white'
        })
    })
})
```

在其他js文件中分别基于backbone实现一些pixelegos的组件。例如：

```
// menu.js
define(function (require, exports, module) {
    var Backbone = require('backbone')
    var $ = require('$')

    var Menu = Backbone.View.extend({
        el: $('header'),
        show: false,
        events: {
            'click .menu-trigger': 'toogle'
        },
        initialize: function() {
            this.menu = this.$el.next()
            this.render()
        },
        toogle: function(e) {
            e.preventDefault()
            this.show = ! this.show
            this.render()
        },
        render: function() {
            if (this.show) {
                this.menu.css('height', 172)
            } else {
                this.menu.css('height', 0)
            }
        }
    })
```

```
    module.exports = Menu
})
```

menu.js依赖于backbone和$（在config.js将zepto alias为了$），实现了顶部的菜单。

当当当当

当当当当，巴拉巴拉，我们敲敲打打完成了pixelegos得功能，我们已经可以画出那只Octocat了！

构建发布

终于来到了我们的重点，关于cmd模块的构建。

> 有童靴觉得spm提供出来的构建工具很难用，搞不懂。我用下来确实些奇怪的地方，等我慢慢到来吧。

spm为自定义构建提供了两个工具:

- grunt-cmd-transport：将cmd模块转换成具名模块，即将 `define(function (require, exports, module) {})` 转换为 `define(id, deps, function(require, exports, module) {})` ，可基于package.json中的spm配置来替换被alias掉的路径等等。本身还可以将css或者html文件转换为cmd包。
- grunt-cmd-concat：根据依赖树，将多个具名的cmd模块打包到一起。

接下来就是用这些工具将我们零散的js打包成一个名为pixelegos.js的文件。

grunt

grunt是目前JavaScript最炙手可热的构建工具，我们先来安装下：

```
" 在全部安装grunt的命令行接口
$ npm install grunt-cli -g

" 安装需要用的grunt task
$ npm install grunt grunt-cmd-concat grunt-cmd-transport grunt-contrib-concat grunt-contrib-jshint grunt-contrib-ug
```

整个打包的流程为：

1. 将业务代码transport成cmd的具名模块
2. concat所有的文件到一个文件pixelegos.js
3. uglify

编写Gruntfile.js文件

第一步先把js文件夹中的业务js转换成具名模块：

```
transport : {
    options: {
        idleading: '/dist/',
        alias: '<%= pkg.spm.alias %>',
        debug: false
    },
    app:{
        files:[{
            cwd: 'js/',
            src: '**/*',
            dest: '.build'
        }]
    }
}
```

这是一些transport的配置，即将js/中的js transport到.build中间文件夹中。

接下来，将.build中的文件合并到一起（包含sea-modules中的依赖项。）：

```
concat : {
    options : {
        include : 'all'
    },
    app: {
        files: [
            {
                expand: true,
                cwd: '.build/',
                src: ['pixelegos.js'],
                dest: 'dist/',
                ext: '.js'
            }
        ]
    }
}
```

这里我们只对pixelegos.js进行concat，因为它是app的入口文件，将 include 配置成 all ，只需要concat这个文件，就能将所有的依赖项打包到一起。 include 还可以配置成其他值：

- self，相当于不做concat，只是copy该文件
- relative，只concat通过想对路径依赖的模块

既然我们已经transport和concat好了文件，那我们直接使用整个文件就行了，于是我们的发布页面可写成：

```
<!DOCTYPE HTML>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
    <meta content='width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=0' name='viewport' />
    <meta name="viewport" content="width=device-width" />
    <link rel="stylesheet" href="/index.css" />
    <script type="text/javascript" src="/sea-modules/seajs/seajs/2.1.1/sea.js"></script>
    <script type="text/javascript">
        seajs.use('/dist/pixelegos')
    </script>
</head>
<body>
...
</body>
</html>
```

当我运行index-product.html时我遇到了坑。在backbone包中并没有指明$依赖的具体包，导致打包后的js无法找到$.js文件。原本以为backbone中的$会被业务级的配置所替换，但是事实并非如此。如何解决?

我们必须使用seajs.config接口提供一个dom的engine，在js/中创建engine.js文件：

```
// engine.js
seajs.config({
    alias: {
        '$': 'zepto/zepto/1.0.0/zepto'
    }
})
```

接下来把这个文件和pixelegos.js concat在一起：

```
normalconcat: {
    app: {
        src: ['js/engine.js', 'dist/pixelegos.js'],
        dest: 'dist/pixelegos.js'
    }
}
```

由于grunt-contrib-concat和grunt-cmd-concat产生了task name的冲突，可以通过grunt.renameTask来修改task名。

下一步，uglify!

```
uglify : {
    app : {
```

```
      files: [
        {
          expand: true,
          cwd: 'dist/',
          src: ['**/*.js', '!**/*-debug.js'],
          dest: 'dist/',
          ext: '.js'
        }
      ]
    }
}
```

大功告成，完整的Gruntfile.js如下:

```
module.exports = function (grunt) {
    grunt.initConfig({
        pkg : grunt.file.readJSON("package.json"),
        transport : {
            options: {
                idleading: '/dist/',
                alias: '<%= pkg.spm.alias %>',
                debug: false
            },
            app:{
                files:[{
                    cwd: 'js/',
                    src: '**/*',
                    dest: '.build'
                }]
            }
        },
        concat : {
            options : {
                include : 'all'
            },
            app: {
                files: [
                    {
                        expand: true,
                        cwd: '.build/',
                        src: ['pixelegos.js'],
                        dest: 'dist/',
                        ext: '.js'
                    }
                ]
            }
```

```
    },
    normalconcat: {
      app: {
        src: ['js/engine.js', 'dist/pixelegos.js'],
        dest: 'dist/pixelegos.js'
      }
    },
    uglify : {
      app : {
        files: [
          {
            expand: true,
            cwd: 'dist/',
            src: ['**/*.js', '!**/*-debug.js'],
            dest: 'dist/',
            ext: '.js'
          }
        ]
      }
    },
    clean:{
      app:['.build', 'dist']
    }
  })

  grunt.loadNpmTasks('grunt-cmd-transport')
  grunt.loadNpmTasks('grunt-contrib-concat')
  grunt.renameTask('concat', 'normalconcat')
  grunt.loadNpmTasks('grunt-cmd-concat')
  grunt.loadNpmTasks('grunt-contrib-uglify')
  grunt.loadNpmTasks('grunt-contrib-clean')

  grunt.registerTask('build', ['clean', 'transport:app', 'concat:app', 'normalconcat:app', 'uglify:app'])
  grunt.registerTask('default', ['build'])
}
```

总结

我们使用spm将一个非cmd模块venus转成了标准的cmd模块venus-in-cmd，然后我们用它结合多个cmd模块构建了一个简单的网页程序。很有成就，有没有！接下来我们要进入hard模式了，我们来看看，Sea.js是如何实现的？只有了解了它的内部是如何运作的，在使用它的过程才能游刃有余！

5

Sea.js是如何工作的?

> 蒙惠者虽知其然，而未必知其所以然也。

写了这么多，必须证明一下本书并不是一份乏味的使用文档，我们来深入看看Sea.js，搞清楚它时如何工作的吧！

CMD规范

要想了解Sea.js的运作机制，就不得不先了解其CMD规范。

Sea.js采用了和Node相似的CMD规范，我觉得它们应该是一样的。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：

1. 模块的管理
2. 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。加载期需要将执行期所有用到的模块从服务端同步过来，在再执行期按照代码的逻辑顺序解析执行模块。本身执行期与node的运行期没什么区别。

所以Sea.js需要三个接口：

1. define用来wrapper模块，指明依赖，同步依赖；
2. use用来启动加载期；
3. require关键字，实际上是执行期的桥梁。

> 并不太喜欢Sea.js的use API，因为其回调函数并没有使用与Define一样的参数列表。

模块标识（id）

模块id的标准参考Module Identifiers (http://wiki.commonjs.org/wiki/Modules/1.1.1#Module_Identifiers)，简单说来就是作为一个模块的唯一标识。

出于学习的目的，我将它们翻译引用在这里：

1. 模块标识由数个被斜杠（/）隔开的词项组成；
2. 每次词项必须是小写的标识、“.”或“..”；
3. 模块标识并不是必须有像“.js”这样的文件扩展名；
4. 模块标识不是相对的，就是顶级的。相对的模块标识开头要么是“.”，要么是“..”；
5. 顶级标识根据模块系统的基础路径来解析；

6. 相对的模块标识被解释为相对于某模块的标识，“require”语句是写在这个模块中，并在这个模块中调用的。

模块（factory）

顾名思义，factory就是工厂，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。

```
function (require, exports, module) {
  // here is module body
}
```

在整个运行时中只有模块，即只有factory。

依赖（dependencies）

依赖就是一个id的数组，即模块所依赖模块的标识。

依赖加载的原理

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?

node的方式-同步的 require

想要解释这个问题，我们还是从Node模块说起，node于Ruby类似，用我们之前使用过的一个模块作为例子：

```
// File: usegreet.js
var greet = require("./greet");
greet.helloJavaScript();
```

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。例如，`helloJavaScript` 这个函数。详细细节可以参看node源码中的module.js (https://github.com/joyent/node/blob/master/lib/module.js) 。

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。

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。

Sea.js的方式-加载期与执行期

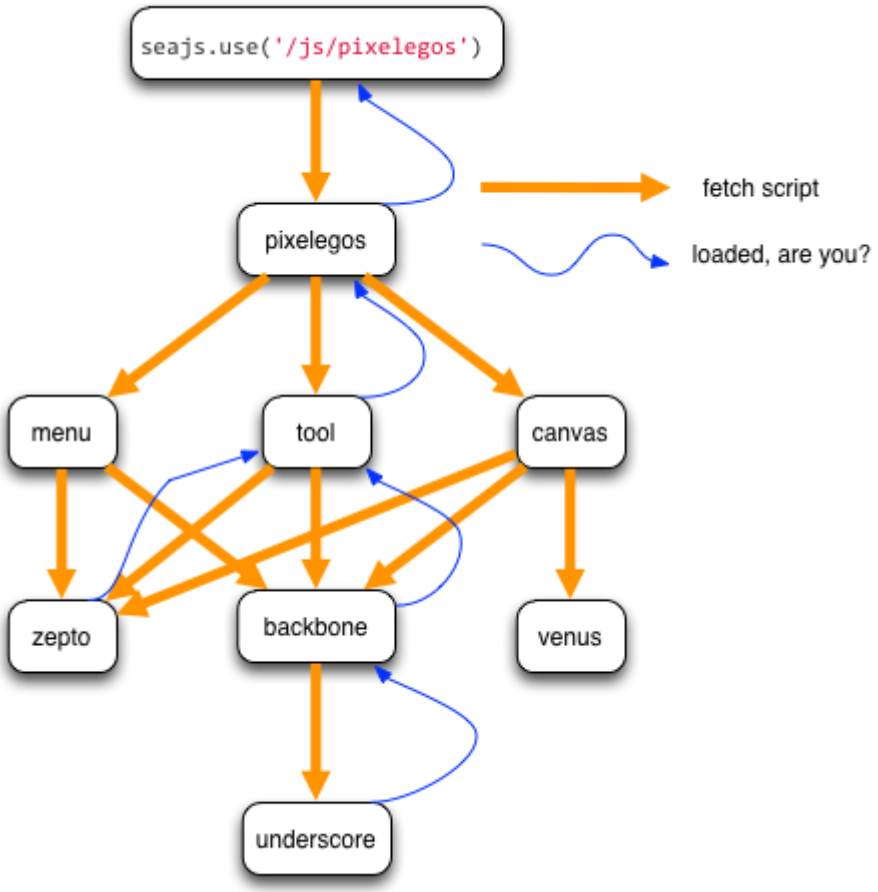
由于在浏览器端，采用与node同样的依赖加载方式是不可行的，因为依赖只有在执行期才能知道，但是此时在浏览器端，我们无法像node一样直接同步地读取一个依赖文件并执行！我们只能采用异步的方式。于是Sea.js的做法是，分成两个时期——加载期和执行期；

> 的确，我们可以使用同步的XHR从服务端加载依赖，但是本身就是单进程的JavaScript还需要等待文件的加载，那性能将大打折扣。

- **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的依赖树：



图片 5.1 loadingperiod

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。

> 那Sea.js如何确定pixelegos所有依赖的模块都加载好了呢?

从依赖树中可以看出，如果pixelegos.js所依赖的直接子节点加载好了（此种加载好，即为节点和其依赖的子节点都加载好），那就表示它就加载好了，于是就可以启动该模块。明显，这种加载完成的过程也是一个递归的过程。

从最底层的叶子节点开始（例如undercore），由于没有再依赖于其他模块，所以它从服务端同步过来之后，就加载好了。然后开始询问其父节点backbone是否已经加载好了，即询问backbone所依赖的所有节点都加载好了。同理对于pixelegos模块，其子节点menu、tool、canvas都会询问pixelegos其子节点加载好了没有。

如果三个依赖都已loading完毕，则pixelgos也加载完成，即其整棵依赖树都加载好了，然后就可以启动pixelegos这个模块了。

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

在Sea.js中，为了支持模块的combo，存在一个js文件包含多个模块的情况。根据依赖情况，使用grunt-cmd-concat可以将一个模块以及其依赖的子模块打包成一个js文件。

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

例如，我们 `seajs.use('/dist/pixelegos')` ，解析为需要加载 `http://127.0.0.1:81/dist/pixelegos.js` 这个文件，且这个文件是all全打包的。其加载过程如下：

加载方式

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 `http://127.0.0.1:81/dist/pixelegos.js` 模块；
2. `http://127.0.0.1:81/dist/pixelegos.js` 加载执行，所有打包在里面的模块被 `define` ；
3. `http://127.0.0.1:81/dist/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；
4. 通知匿名的 `use_` 加载完成，开始执行期。

针对每一次执行期，对应的加载依赖树与整个模块依赖树是有区别的，因为子模块已经加载好了的模块，并不在加载树中。

Sea.js的实现

模块的状态

由于浏览器端与Node的环境差异，模块存在加载期和执行期，所以Sea.js中为模块定义了六种状态。

```
var STATUS = Module.STATUS = {
  // 1 - The `module.uri` is being fetched
  FETCHING: 1,
  // 2 - The meta data has been saved to cachedMods
  SAVED: 2,
  // 3 - The `module.dependencies` are being loaded
  LOADING: 3,
  // 4 - The module are ready to execute
  LOADED: 4,
  // 5 - The module is being executed
  EXECUTING: 5,
  // 6 - The `module.exports` is available
  EXECUTED: 6
}
```

分别为：

- 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

module.js (https://github.com/seajs/seajs/blob/master/src/module.js) 是Sea.js的核心，我们来看一下，module.js是如何控制模块加载过程的。

如何确定整个依赖树加载好了呢?

1. 定义A模块，如果有模块依赖于A，把该模块加入到等待A的模块队列中；
2. 加载A模块，状态变为FETCHING

3. A加载完成，获取A模块依赖的BCDEFG模块，发现B模块没有定义，而C加载中，D自己已加载好，E加载子模块中，F加载完成，运行中，G已经解析好，SAVED；
4. 由于FG本身以及子模块都已加载好，因此A模块要确定已经加载好了，必须等待BCDE加载好；开始加载必须的子模块，LOADING；
5. 针对B重复步骤1；
6. 将A加入到CDE的等待队列中；
7. BCDE加载好之后都会从自己的等待队列中取出等待自己加载好的模块，通知A自己已经加载好了；
8. A每次收到子模块加载好的通知，都看一遍自己依赖的模块是否状态都变成了加载完成，如果加载完成，则A加载完成，A通知其等待队列中的模块自己已加载完成，LOADED；

加载过程

- 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。

```
// Use function is equal to load a anonymous module
Module.use = function (ids, callback, uri) {
  var mod = Module.get(uri, isArray(ids) ? ids: [ids])

  mod.callback = function () {
    var exports = []
    var uris = mod.resolve()

    for (var i = 0, len = uris.length; i < len; i++) {
      exports[i] = cachedMods[uris[i]].exec()
    }

    if (callback) {
      callback.apply(global, exports)
    }

    delete mod.callback
  }

  mod.load()
}
```

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()：

```
// Load module.dependencies and fire onload when all done
Module.prototype.load = function () {
  var mod = this

  // If the module is being loaded, just wait it onload call
  if (mod.status >= STATUS.LOADING) {
    return
  }

  mod.status = STATUS.LOADING

  // Emit `load` event for plugins such as combo plugin
  var uris = mod.resolve()
  emit("load", uris)

  var len = mod._remain = uris.length
  var m

  // Initialize modules and register waitings
  for (var i = 0; i < len; i++) {
    m = Module.get(uris[i])

    if (m.status < STATUS.LOADED) {
      // Maybe duplicate
      m._waitings[mod.uri] = (m._waitings[mod.uri] || 0) + 1
    }
    else {
      mod._remain--
    }
  }

  if (mod._remain === 0) {
    mod.onload()
    return
  }

  // Begin parallel loading
  var requestCache = {}

  for (i = 0; i < len; i++) {
    m = cachedMods[uris[i]]

    if (m.status < STATUS.FETCHING) {
      m.fetch(requestCache)
    }
```

```
        else if (m.status === STATUS.SAVED) {
            m.load()
        }
    }

    // Send all requests at last to avoid cache bug in IE6-9. Issues#808
    for (var requestUri in requestCache) {
        if (requestCache.hasOwnProperty(requestUri)) {
            requestCache[requestUri]()
        }
    }
}
```

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：

```
// Call this method when module is loaded
Module.prototype.onload = function () {
    var mod = this
    mod.status = STATUS.LOADED

    if (mod.callback) {
        mod.callback()
    }

    // Notify waiting modules to fire onload
    var waitings = mod._waitings
    var uri, m

    for (uri in waitings) {
        if (waitings.hasOwnProperty(uri)) {
            m = cachedMods[uri]
            m._remain -= waitings[uri]
            if (m._remain === 0) {
                m.onload()
            }
        }
    }

    // Reduce memory taken
    delete mod._waitings
    delete mod._remain
}
```

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：

```
mod.callback = function () {
    var exports = []
    var uris = mod.resolve()

    for (var i = 0, len = uris.length; i < len; i++) {
        exports[i] = cachedMods[uris[i]].exec()
    }

    if (callback) {
        callback.apply(global, exports)
    }

    delete mod.callback
}
```

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：

```
// Execute a module
Module.prototype.exec = function () {
    var mod = this

    // When module is executed, DO NOT execute it again. When module
    // is being executed, just return `module.exports` too, for avoiding
    // circularly calling
    if (mod.status >= STATUS.EXECUTING) {
        return mod.exports
    }

    mod.status = STATUS.EXECUTING

    // Create require
    var uri = mod.uri

    function require(id) {
        return Module.get(require.resolve(id)).exec()
    }
```

```
  require.resolve = function (id) {
    return Module.resolve(id, uri)
  }

  require.async = function (ids, callback) {
    Module.use(ids, callback, uri + "_async_" + cid())
    return require
  }

  // Exec factory
  var factory = mod.factory

  var exports = isFunction(factory) ? factory(require, mod.exports = {},
  mod) : factory

  if (exports === undefined) {
    exports = mod.exports
  }

  // Emit `error` event
  if (exports === null && ! IS_CSS_RE.test(uri)) {
    emit("error", mod)
  }

  // Reduce memory leak
  delete mod.factory

  mod.exports = exports
  mod.status = STATUS.EXECUTED

  // Emit `exec` event
  emit("exec", mod)

  return exports
}
```

> 看到这一行代码了么？ `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

资源定位是Sea.js，或者说模块加载器中非常关键部分。那什么是资源定位呢?

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，在Sea.js中，普通路径包有以下几种：

```
// 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
```

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。

```
// File http://example.com/js/b.js
define(function(require) {
    var a = require('./a');
    a.doSomething();
});
// => 加载的是http://example.com/js/a.js
```

这与node模块中相对路径的解析一致。

顶级标识

不以 `.` 或者'/'开头的模块标识是顶级标识，相对于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
```

> 在node中即是在paths中搜索模块（node_modules文件夹中）。

模块定位小演

使用seajs.use启动模块，如果不是顶级标识或者是绝对路径，就是相对于页面定位；如果是顶级标识，就从Sea.js的模块系统中加载（即base）；如果是绝对路径，直接加载； 之后的模块加载都是在define的factory

中，如果是相对路径，就是相对标识，相对当前模块路径加载；如果是绝对路径，直接加载； 由此可见，在Sea.js中，模块的配置被分割成2+x个地方：

- 与页面放在一起；
- 与Sea.js放在一起；
- 通过绝对路径添加更多的模块源。

> 由此可见，Sea.js确实海纳百川。

获取真实的加载路径

1.在Sea.js中，使用data.cwd来代表当前页面的目录，如果当前页面地址为 `http://www.dianping.com/promo/195800` ，则cwd为 `http://www.dianping.com/promo/` ；使用data.base来代表sea.js的加载地址，如果sea.js的路径为 `http://i1.dpfile.com/lib/1.0.0/sea.js` ，则base为 `http://i1.dpfile.com/lib/` 。

> “当 sea.js 的访问路径中含有版本号或其他东西时，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”（https://github.com/seajs/seajs/issues/258）。看到这一句，我凌乱了，这Sea.js是多么的人性化！但是我觉得这似乎没有必要。

2.seajs.use是，除了绝对路径，其他都是相对于cwd定位，即如果模块标识为：

- './a'，则真实加载路径为http://www.dianping.com/promo/a.js；
- '/a'，则为http://www.dianping.com/a.js；
- '../a'，则为http://www.dianping.com/a.js；

> 从需求上看，相对页面地址定位在现实生活中并不太适用，如果页面地址或者静态文件的路径稍微变化下，就跪了。

如果模块标识为绝对路径：

- 'https://a.alipayobjects.com/ar/a'，则加载路径就是'https://a.alipayobjects.com/ar/a.js'。

如果模块标识是顶级标识，就基于base来加载：

- 'jquery'，则加载路径为'http://i1.dpfile.com/lib/jquery.js'。

3.除此之外，就是factory中的模块标识了：

- 'https://a.alipayobjects.com/ar/b'，加载路径为'https://a.alipayobjects.com/ar/b.js'
- '/c'，加载路径为'http://www.dianping.com/c.js'；

- './d'，如果父模块的加载路径是'http://i1.dpfile.com/lib/e.js'，则加载路径为'http://i1.dpfile.com/lib/d.js'

> 在模块系统中使用'/c'绝对路径是什么意思？Sea.js会将其解析为相对页面的模块，有点牛马不相及的感觉。

factory的依赖分析

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？

这段是Sea.js的源码：

```
/**
 * 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*\(
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
}
```

`REQUIRE_RE` 这个硕大无比的正则就是关键。推荐使用regexper (http://www.regexper.com/) 来看看这个正则表达式。非native的函数factory我们可以通过的toString()方法获取源码，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。

6

模块化JavaScript的未来

JavaScript未来的模块化会是什么样子？这很难讲。如前所说，ES6已经开始起草这一块的标准，而AMD，CommonJS已经流行起来。通常，标准的制定，都是在有了实现的前提之下。不管怎样，我们先来展望一二吧。

7

参考资料

- seajs (https://github.com/)
- 实例解析 SeaJS 内部执行过程 – 从 use 说起 (https://github.com/seajs/seajs/issues/308)
- SeaJS v1.2 中文注释版 (https://github.com/seajs/seajs/issues/305)
- hello seajs (http://mrzhang.me/blog/hello-seajs.html)
- seajs.org/docs (http://seajs.org/docs/)
- 使用SeaJS实现模块化JavaScript开发 (http://cnodejs.org/topic/4f16442ccae1f4aa270010d9)
- use.js (http://documentup.com/tbranyen/use.js)
- harmony:modules (http://wiki.ecmascript.org/doku.php?id=harmony:modules)
- harmony:module_loaders (http://wiki.ecmascript.org/doku.php?id=harmony:module_loaders)
- AMD规范 (https://github.com/amdjs/amdjs-api/wiki/AMD)
- CMD规范 (https://github.com/seajs/seajs/issues/242)
- AMD 和 CMD 的区别有哪些？ (http://www.zhihu.com/question/20351507/answer/14859415)
- 与 RequireJS 的异同 (https://github.com/seajs/seajs/issues/277)
- 基于CommonJS Modules/2.0的实现：BravoJS (http://www.cnblogs.com/snandy/archive/2012/06/10/2543893.html)
- Dynamic Script Request (DSR) API (http://tagneto.org/how/reference/js/DynamicScriptRequest.html)
- Achieving A Runtime CPAN With Dojo's XD Loader (http://tagneto.org/talks/AjaxExperienceXDomain/)
- jQueryRequireJS (http://www.tagneto.org/talks/jQueryRequireJS/jQueryRequireJS.html)
- labjs (http://www.slideshare.net/itchina110/labjs)
- jsi (http://code.google.com/p/jsi/wiki/History)
- 从零开始编写自己的JavaScript框架（一）(http://www.ituring.com.cn/article/48461)
- ECMAScript 6 Modules: What Are They and How to Use Them Today (http://www.infoq.com/news/2013/08/es6-modules)
- aralejs.org (http://aralejs.org/)

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/hello-seajs/