

SPM Handbook

sorrycc

Published
with GitBook



Table of Contents

1. [Introduction](#)
2. [新手入门](#)
3. [环境配置](#)
4. [package.json](#)
 - i. [buildArgs](#)
5. [组件开发](#)
 - i. [引入 CSS](#)
 - ii. [引入模板](#)
 - iii. [引入组件内部文件](#)
 - iv. [迁移到 SPM3](#)
6. [项目开发](#)
 - i. [应用 Bootstrap](#)
 - ii. [spm-server](#)
 - iii. [应对不同的项目类型](#)
7. [构建](#)
8. [命令行](#)
9. [源](#)
 - i. [私有源](#)

SPM Handbook

本书介绍如何用 [spm](#) 来开发组件和应用。

阅读

- [gitbook.io](#)

名词解释

- 源，详见 [源](#)
- 组件，指可以发布到源上的包，比如 [type 组件](#)

贡献

- [Bug reports](#)
- [Pull Requests](#)

新手入门

介绍

[spm](#) 是一套完整的浏览器端组件管理解决方案，包含对于 JavaScript、CSS 和模板的处理。



所有 [源](#) : [spmjs.io](#) 上的组件都是以 CommonJS 的方式组织，并可以通过 [seajs](#) 加载运行。同时，我们基于 [spm](#) 提供了一套完整的组件生命周期管理方案，包含以下特性：

- 初始化
- 依赖安装
- 本地开发
- 发布到 [spmjs.io](#)
- 单元测试
- 文档服务
- 构建

[spmjs.io](#) 是针对 spm 的组件管理服务。你可以在这里搜索需要的组件，也可以发布组件到这里。如果需要，你还可以搭建自己的私有源服务或镜像。

安装

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

如果遇到安装问题，请先确认 [环境配置](#) 是否正确，以及搜索下 [别人踩过的安装坑](#)。

使用

以下是组件开发的使用入门，你也可以直接跳到[项目开发](#)。

初始化组件：

```
$ mkdir example  
$ cd example  
$ spm init
```

安装依赖：

```
$ spm install jquery --save  
$ spm install moment@2.6.0 --save
```

发布到 [spmjs.io](#)

```
$ spm publish
```

你可能需要先执行 `spm login` 来获取权限，登录 spmjs.io 之后在 <http://spmjs.io/account> 可以找到 `authkey`。

如果组件包的尺寸过大，可以添加 `.spmignore` 来过滤掉不需要的文件，格式同 [.gitignore](#)。

贡献

欢迎通过以下方式来贡献 spm：

- [Bug reports](#)
- [Feature requests](#)
- [Pull requests](#)
- 推广 spm 到你的朋友圈

环境配置

环境准备

安装 node

作为一个前端，你需要 node 环境。

推荐使用 [nvm](#) 来安装：

```
$ curl https://raw.githubusercontent.com/creationix/nvm/v0.16.1/install.sh | bash
$ nvm install 0.10.24
$ nvm alias default 0.10.24
```

另外，也可以在[官方下载](#)或[通过包管理工具安装](#)。

去除 sudo

如果你是用 [nvm](#) 来安装的 **node**，请跳过步骤。

使用 npm 安装模块的时候经常要输入 sudo，还要输入密码，这点很让人烦躁。下面你可以简单粗暴的去除 sudo（看看作者的[软文](#)）

```
$ sudo chown -R $USER /usr/local
```

安装 git

版本管理工具也是必须的，可以先[了解 git 的相关内容](#)。

git 下载地址如下

- [git for mac](#)
- [git for windows](#)

安装 spm

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

windows 环境

推荐使用 windows 的包管理工具 [chocolatey](#)。

安装 nodejs 和 git：

```
c:\> cinst git.install
c:\> cinst nodejs.install
```

设置环境变量：

```
PATH = C:\Users\{{username}}\AppData\Roaming\npm  
NODE_PATH = C:\Users\{{username}}\AppData\Roaming\npm\node_modules
```

Package.json

spm 使用和 npm 基本一致的 package.json 来描述组件。除了包含一些特殊配置的 spm 域之外，其他都和 npm 保持一致。

Fields

Field	Description
name*	组件名，全部小写，以 - 或 . 分隔单词
version*	语义化的版本号，比如 1.0.0
description	a brief description of your package
keywords	an array contains keywords
homepage	url of your package's website
author	author of this package: Hsiaoming Yang <me@lepture.com> OR { "name": "Hsiaoming Yang", "email": "me@lepture.com" }
maintainers	an array of maintainers, just like author
repository	Specify the place where your code lives. { "type": "git", "url": "http://github.com/isaacs/npm.git" }
bugs	The url to your project's issue tracker and / or the email address to which issues should be reported.
license	license
spm*	
spm.main	组件的唯一入口，默认为 index.js，如果是一个 css 组件，可以设置为 index.css
spm.output	指定构建输出的文件，以数组表示，支持 glob 格式：src/**/*.js
spm.dependencies	指定依赖组件
spm.devDependencies	指定开发状态下的依赖组件
spm.tests	指定测试文件，支持 glob 格式：tests/*.spec.js
spm.buildArgs	配置 spm build 的命令行参数
spm.registry	指定组件发布的目标源，默认为 http://spmjs.io
spm.ignore	发布到源上时需要忽略的文件，以数组表示，功能同 .spmignore

一个简单的例子


```
{
  "name": "arale-calendar",
  "version": "1.1.0",
  "description": "Calendar widget.",
  "keywords": [
    "widget",
    "month",
    "datepicker"
  ],
  "author": "Hsiaoming Yang <me@lepture.com>",
  "maintainers": [
    "hotoo <hotoo.cn@gmail.com>",
    "shengyan <shengyan1985@gmail.com>"
  ],
  "homepage": "http://aralejs.org/calendar/",
  "repository": {
    "type": "git",
    "url": "https://github.com/aralejs/calendar.git"
  },
  "bugs": {
    "url": "https://github.com/aralejs/calendar/issues"
  },
  "license": "MIT",
  "spm": {
    "main": "calendar.js",
    "dependencies": {
      "jquery": "1.7.2",
      "moment": "2.6.0",
      "arale-base": "1.1.0",
      "arale-position": "1.1.0",
      "arale-iframe-shim": "1.1.0",
      "handlebars": "1.3.0",
      "arale-widget": "1.2.0"
    },
    "devDependencies": {
      "expect.js": "0.3.1"
    },
    "tests": "tests/*-spec.js",
    "ignore": [
      "dist",
      "_site"
    ],
    "buildArgs": "--ignore jquery"
  }
}
```

buildArgs

参数列表

Field	Description
dest	输出目录，默认为 dist
include	指定打包策略，可选 relative、all、standalone、umd，默认为 relative
ignore	指定不进行 transport 的依赖
skip	指定不进行分析的依赖
idleading	模块名前缀，默认为 {{name}}/{{version}}
withDeps	同时打包所有依赖，默认为 false

include

Field	Description
relative	打包相对依赖
all	打包所有依赖
standalone	打包所有依赖并可自运行
umd	在 standalone 基础上再添加 umd wrap

举例说明

比如有一个组件 a/0.1.0/index.js：

```
require('./relative');
require('b');
```

include 为 relative 时，打包出来是：

```
define('a/0.1.0/index.js', ['a/0.1.0/relative', 'b/0.1.0/index.js'], function() {});
define('a/0.1.0/relative.js', function() {});
```

include 为 all 时，打包出来是：

```
define('a/0.1.0/index.js', ['a/0.1.0/relative', 'b/0.1.0/index.js'], function() {});
define('a/0.1.0/relative.js', function() {});
define('b/0.1.0/index.js', function() {});
```

include 为 standalone 时，打包出来是：

```
(function(){
  var a_0_1_0_index_js, a_0_1_0_relative_js, b_0_1_0_index_js;
  b_0_1_0_index_js = (function() {} )();
  a_0_1_0_relative_js = (function() {} )();
  a_0_1_0_index_js = (function() {} )();
})();
```

include 为 umd 时，打包出来是：

```
(function(){
var a_010_index_js, a_010_relative_js, b_010_index_js;
b_010_index_js = (function() {}){};
a_010_relative_js = (function() {}){};
a_010_index_js = (function() {}){};

// umd wrap
if (typeof exports == "object") {
  module.exports = a_010_index_js;
} else if (typeof define == "function" && define.amd) {
  define([], function(){ return a_010_index_js });
} else {
  this["a"] = a_100_indexjs;
}
})();
```

ignore

还是上面的例子，如果 include 为 relative 并且 ignore 为 b，打包出来是：

```
define('a/0.1.0/index.js', ['a/0.1.0/relative', 'b'], function() {});
define('a/0.1.0/relative.js', function() {});
```

组件开发

下面会一步步教你如何通过 [spm](#) 来开发一个组件。

```
$ spm -v
3.x.x
```

开始前，请确保安装了 spm@3.x。

init

```
$ mkdir now
$ cd now
$ spm init
```

```
Creating a spm package:
[?] Package name: (now)
[?] Version: (1.0.0)
[?] Description:
[?] Author: afc163 <afc163@gmail.com>

Initialize a spm package Succceccfully!
```

这样，你就有了一个叫 `now` 的组件。

安装依赖

先安装默认依赖：

```
$ spm install
```

然后我们需要 [moment](#) 依赖，[这里](#) 有他的详细信息。

```
$ spm install moment --save
```

开发

编辑 `index.js`，和在 `nodejs` 里一样：

```
var moment = require('moment');
var now = moment().format('MMMM Do YYYY, h:mm:ss a');

module.exports = now;
```

然后编辑 `examples/index.md`：

```
# Demo

---

## Normal usage

```javascript
seajs.use('index', function(now) {
 alert(now); // add this
});
```
```

本地调试

执行 `spm doc watch` 开启一个文档服务 `127.0.0.1:8000` ：

```
$ spm doc watch
```

然后在浏览器里打开 <http://127.0.0.1:8000/examples/> 即可看到结果。

你可以在 Markdown 文件里用 3 个 ` 来引用代码，也可以用 4 个 `。

这是一条特殊规则，引用的代码首先会高亮显示，然后还会被插入一个 `script` 标签来同步执行。这一点非常有用，在调试 demo 的同时，还可以写出优雅的文档。

如果想在 demo 中插入 `iframe`，需声明代码为 `frame` 类型：

```
```frame:600
I am in a iframe of 600px high
```
```

如果不想用 `spm doc watch` 来调试代码，你还可以试试 [seajs-wrap](#) 或 [spm-server](#) 来调试开发模式下的 `CommonJS` 代码。

添加测试用例

编辑测试文件 `tests/now-spec.js`，我们默认引用了一个断言方案 [expect.js](#)。

```
var expect = require('expect.js');
var now = require('./index');

describe('now', function() {

  it('normal usage', function() {
    expect(now).to.be.a('string'); // add this
  });

});
```

看看测试结果：

```
$ spm test
```

你也可以在浏览器里打开 <http://127.0.0.1:8000/tests/runner.html> 查看结果。

发布

现在你已拥有一个包含完整功能和完善测试用例的组件里，尝试把他发布到 spmjs.io 吧。

```
$ spm publish
```

你应该要先执行 `spm login` 来获取权限，否则会提示验证失败。

```
$ spm login
```

`username` 是你的 github 账号，而 `authkey` 可以在你登陆后在 <http://spmjs.io/account> 找到。

当然，组件 `now` 是我发布的。你应该先改个名字然后重新试一次。

文档

spmjs.io 可以托管组件文档。你只需要编辑 `README.md` 和 `examples` 目录，通过 `spm doc watch` 预览，然后发布到 spmjs.io。

```
$ spm doc publish
```

最新版文档的 url 是 `http://spmjs.io/docs/{{name}}/`，也可以通过 `http://spmjs.io/docs/{{name}}/{{version}}/` 访问到所有版本。

比如：<http://spmjs.io/docs/now/>。

构建

```
$ spm build
```

`spm build` 命令会构建 `spm.main` 和 `spm.output` 声明的文件到 `dist` 目录。`spm.buildArgs` 会作为参数传入。

默认的构建结果是可以被部署到 cdn 上的包。然后通过 [Sea.js](#) 来引用他。比如：[seajs@2.2.1](#)。

```
<script src="http://cdn.example.com/path/to/sea.js"></script>
<script src="http://cdn.example.com/path/to/seajs-combo.js"></script><!-- If your need that -->
<script>
  seajs.config({
    base: 'http://cdn.example.com/',
    alias: {
      now: 'now/1.0.0/index.js'
    }
  });
  // load http://cdn.example.com/??now/1.0.0/index.js,moment/2.6.0/moment.js
  seajs.use(['now'], function(now) {
    console.log(now);
  });
</script>
```

你也可以添加参数来构建 `standalone` 的包。

```
$ spm build --include standalone
```

```
<!-- use it without loader -->
<script src="http://cdn.example.com/path/to/standalone.js"></script>
```

Congratulation

至此，你应该已学会如何用 spm 开发一个组件，欢迎发布组件到这里！继续阅读：

- [引入 CSS](#)
- [引入模板](#)
- [引入组件内部文件](#)

引入 CSS

spmjs.io 上有不少 CSS 组件，比如 [normalize.css](#) 和 [alice](#) 系列的 UI 组件。

和 JS 一样，你可以通过 `@import` 引入源上的 CSS 组件或本地的 CSS 文件。

```
@import 'normalize.css';
@import './layout/layout.css';

body {
  color: teal;
  background: url('./background-image.jpg');
}
```

此外，spm 还提供了在 JS 里引入 CSS 的功能。

```
@require 'alice-box';
@require './theme/basic.css';
```

通过这种方式引入 CSS 之后，构建时候会自动安装 [import-style](#)，来加载 CSS。

引入模板

SPM 支持在 JS 文件里直接引入两种模板：

1. tpl
2. handlebars

以 handlebars 为例，使用步骤为：

1. 新建 `a.handlebars`，编辑模板内容
2. 在 JS 文件里 `require("./a.handlebars");`
3. 调试(spm doc watch 或 spm-server) 和构建时会预编译 `a.handlebars`
4. 构建时如果发现有 `require handlebars` 文件，则会自动安装 `handlebars-runtime` 并保存到 `spm/dependencies`

引入组件内部文件

在 spm 里，一个组件只允许有一个单一出口，这在某些情况下并不能完全满足需求。

比如 `bootstrap` 组件，入口是 `js/bootstrap.js`，而如果要引入 CSS 文件，则需要：

```
@import 'bootstrap/css/bootstrap.css';
```

或者你仅需要他的 tooltip 功能：

```
require('bootstrap/js/tooltip');
```

比如 CSS 组件 `anima-ui`，由很多模块组成，如果用户需要单独引入某一个模块：

```
@import 'anima-ui/build/util/flexbox.css';
```

迁移 到 spm3

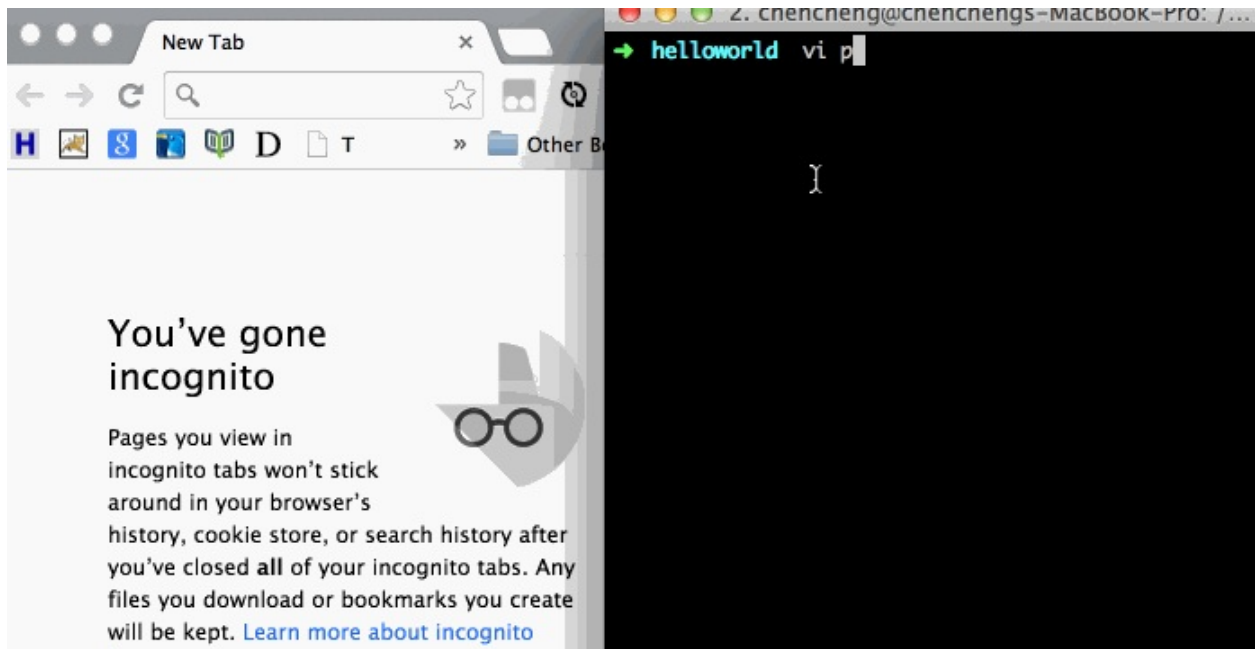
请根据组件类型进行选择：

- [迁移 spm2 到 spm3](#)
- [迁移业界优秀模块到 spmjs.io 并反推的简易教程](#)

项目开发

由于目前 spm 更侧重于解决组件方面的问题，项目开发的工具（比如 [spm-server](#)、部署、项目的脚手架等）并没有内置到 spm 中，需额外安装。

演示



基本流程

1. 建 package.json
2. 开发和用 `spm install` 安装依赖
3. 用 `spm-server` 调试
4. 部署和发布

目录结构

```
helloworld
├─ dist/          (构建生成的目录，用于发布和部署)
├─ spm_modules/  (依赖组件安装后所在目录)
├─ index.js
├─ index.css
├─ index.html
└─ package.json (配置文件，详见 package.json/README.html)
```

开发

编辑 package.json :

```
{
  "name": "helloworld",
  "version": "0.1.0",
  "spm": {
    "output": [
      "index.js",
      "index.css"
    ],
    "buildArgs": "--include standalone"
  }
}
```

output 用于声明哪些文件需要构建，buildArgs 用于配置构建选项，详见 [package.json](#)。

编辑 index.js：

```
var $ = require('jquery');
$('body').append('<div>Hello World!</div>');
```

编辑 index.css：

```
@import 'normalize.css';
div {
  color: red;
}
```

编辑 index.html：

```
<link rel="stylesheet" href="index.css" />
<script src="index.js"></script>
```

安装依赖

在项目目录中执行命令：

```
$ spm install jquery normalize.css --save
```

spm install 命令会从 源：[spmjs.io](#) 上下载依赖的组件，并保存到 `spm_modules` 下，结构如下：

```
spm_modules
├─ jquery/2.1.1/
├─ normalize.css/3.0.1/
```

本地调试

`spm-server` 是 spm 的本地调试工具之一，封装了不少常用的调试功能。比如 `livereload`，`https`，`less` 和 `coffee`，详见 [spm-server 文档](#)。

```
$ npm install spm-server -g
$ spm-server
  server: listen on 8000
```

启动成功后，在浏览器里打开 <http://localhost:8000> 即可看到效果。

通过 `spm-server --livereload` 可开启 `livereload` 模式 (监听文件改动并自动刷新)

构建

在项目目录中执行命令：

```
$ spm build
```

该命令会分析 `package.json`，对 `output` 中指定的文件进行 CMD 转换、压缩等处理，然后输出到 `dist` 目录。

构建后 `dist` 的目录结构如下：

```
dist
├─ helloworld/0.1.0/
│  ├─ index-debug.css
│  ├─ index-debug.js
│  ├─ index.css
│  └─ index.js
```

`dist` 目录路径中会包含版本号，通过版本化的非覆盖式发布可解决静态资源缓存更新、灰度发布、快速回滚、保留历史版本等问题。

部署和发布

`dist` 下的文件是用于发布的，把里面的文件传到服务器上即可。

如果需要 `zip` 包，可执行 `spm build --zip` 命令，在构建之后自动压缩 `dist` 目录。

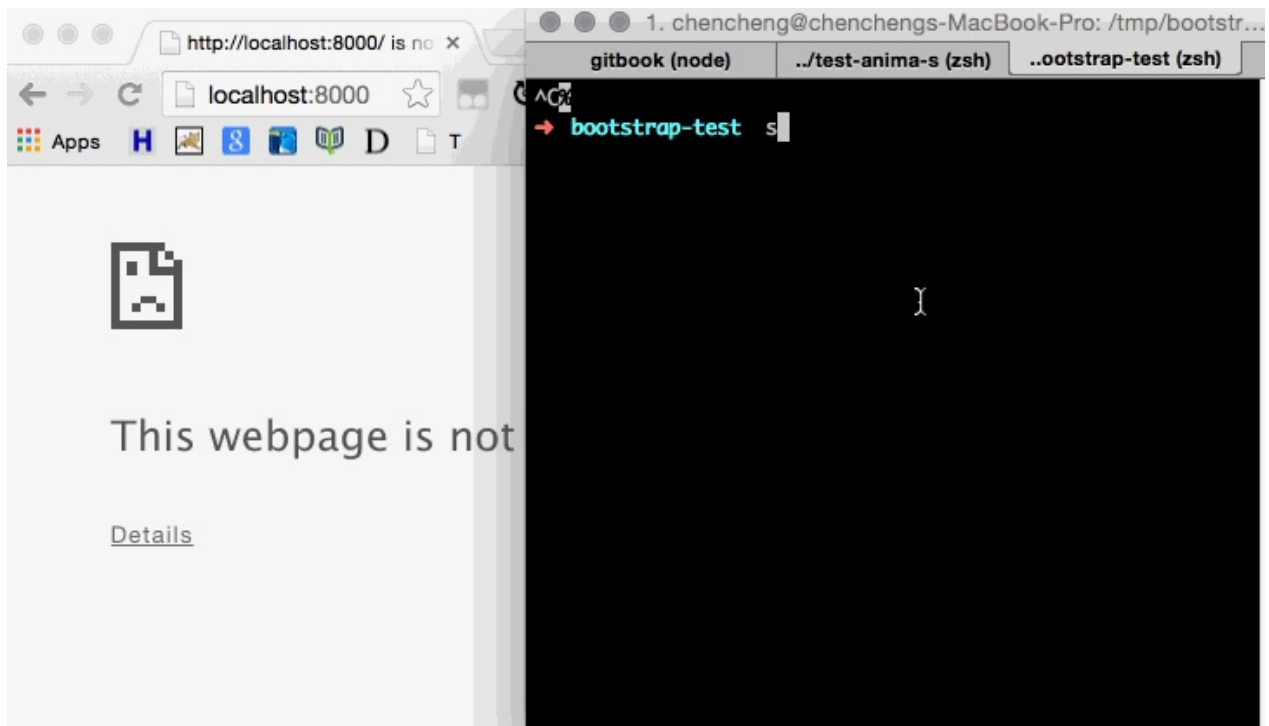
Congratulation

至此，你应该已学会如何用 `spm` 构建一个简单的项目了吧。继续阅读：

- [应用 bootstrap](#)
- [spm-server 使用文档](#)

应用 Bootstrap

演示



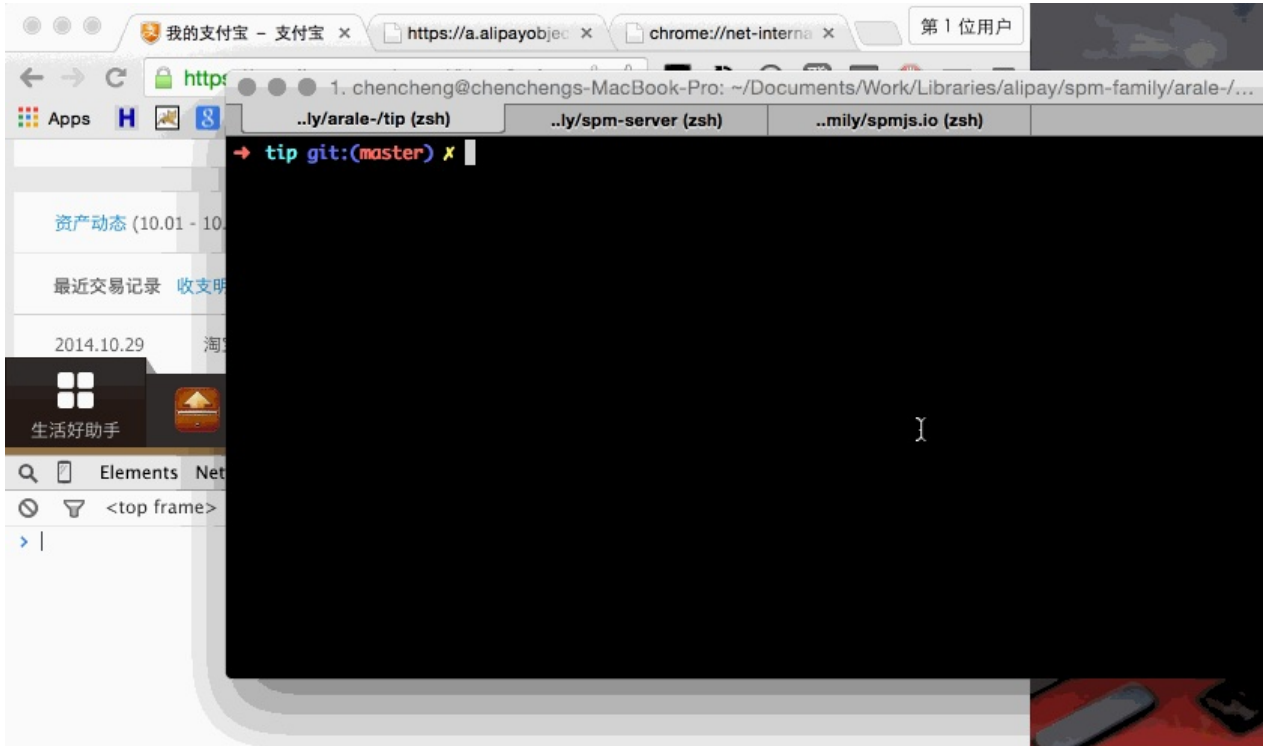
spm-server (source)

如果不喜欢本地启 server 的调试方法，可以试下 [seajs-wrap](#)。

基于 [serve-spm](#) 的 SPM3 本地调试工具，同时也内置了一些路径映射，可以很方便地进行线上调试。

DEMO

在 我的支付宝 页面调试 `arale-tip` 组件：



安装

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

特性

- 支持 `??` 协议的 `combo` 解析和代理

比如请求 `/?a.js,b.js`，本地只有 `a.js`，没有 `b.js`，则会从 CDN 上获取 `b.js`，和 `a.js` 合并后输出。

- 支持 `livereload`

通过添加参数 `--livereload` 开启，需和 [Livereload Chrome 插件](#) 配合使用。

- 支持 `https`

通过添加参数 `--https` 开启。线上 assets 服务器是 `https` 协议，且需要调试线上问题时使用。

- 支持 `less` 和 `coffeescript`

比如，请求 `/a.css` 时会检查 `/a.less` 是否存在，然后实时编译返回。

用法

```
$ spm-server [OPTIONS]
```

选项

- `-p, --port <port>` , 端口号, 默认为 : 8000
- `-b, --base <path>` , base path to access package in production
- `--idleading <idleading>` , 模块名前缀, 默认为 : `{{name}}/{{version}}`
- `--https` , 同时开启 https proxy
- `--livereload` , 开启 livereload, 需和 [livereload Chrome 插件](#) 配合使用

使用场景

- `relative/all`
 - [内链引入 seajs](#), 通过 [seajs.use](#) 载入所有模块
 - [内链 combo 引入 seajs](#) 和依赖模块, 通过 [seajs.use](#) 载入口模块
 - [内链 combo 引入 seajs](#) 和所有模块, 通过 [seajs.use](#) 初始化入口模块
- `standalone`
 - [单独引入入口模块](#)
 - [combo 引入入口模块和其他 JS 文件](#)
 - [base 不是根目录](#), 运行 `spm-server` 时指定 `base`

路径规则

详见：[spmjs/serve-spm#2](#)

```
# 目录结尾, 自动查找 index.htm 和 index.html
- /$ -> index.htm, index.html

# 当前 pkg 的 dist 目录下的文件, 代理到源码
- ^/dist/curr_name/curr_version/a.js -> /a.js

# 依赖模块的请求, 先找 dist 下的(基于性能上的考虑), 再照 spm_modules 下的
- ^/name/version/a.js -> /dist/name/version/a.js
- ^/name/version/a.js -> /spm_modules/name/version/a.js

# handlebars 处理
- ^handlebars-runtime.js, ^/dist/cjs/handlebars.runtime.js -> hanelebars.runtime.js

# js require css 和 tpl 文件的处理
- a.css.js^ -> a.css, a.tpl.js -> a.tpl, ...

# 预编译语言, 目前支持 less 和 coffeescript
- a.js^ -> a.coffee
- a.css^ -> a.less
```

FAQ

- 部分文件我不想加 `cmd wrap` 怎么办?

url 添加 `nowrap` 参数, 比如 `/a.js?nowrap`。

- 和 `spm doc watch` 有什么区别?

`doc watch` 面向组件开发。基于 `nico`, 所以支持 markdown 解析, 启动时会先生成 `_site` 目录, 然后基于 `_site` 目录启动

服务器。spm-server 面向应用开发。不生成额外的目录，支持一些常用的应用开发功能，比如 livereload，https server，cdn combo proxy 等。

应对不同的项目类型

由于业务类型的不同，以及前端技术的发展，我们现在要面对各种项目的类型。有些是单页应用，有些是多页应用，多页应用可能还要考虑页面之间的缓存共用，等等。

而针对这些问题，SPM 提供了 `include` 和 `ignore` 这两个构建参数(详见：[buildArgs](#))。通过他们的组合，理论上可以满足各种项目类型的需要。

单页应用和不太在意性能的多页应用

推荐用 `--include standalone`，这种方式会把所有的依赖都打包到一起，并且可以自运行且无需 loader。

HTML 里的使用方法：

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

在意性能的多页应用

上面这种方式处理多页应用时会有一定量的冗余。

比如有两个页面，每个页面都依赖 jQuery，用上面的方式会把 jQuery 打包到各自的输出文件里，这样 jQuery 就不能在两个页面间公用缓存了。

这时推荐用 `--include all --ignore jquery`，构建时会打包除 jquery 外的所有依赖为 CMD 模块，通过 seajs 载入运行。ignore 的 jquery 可以通过 seajs 的 alias 配置指定具体的版本。

HTML 里的使用方法：

```
<script src="seajs/2.2.0/seajs.js,jquery/1.7.2/jquery.js"></script>
<script>
seajs.config({
  alias: {
    'jquery': 'jquery/1.7.2/jquery.js'
  }
});
seajs.use('entry.js');
</script>
```

其他

- 还可以选择 `--include relative` 的方式，构建时只会包含相对目录的本地文件，不会包含依赖。所以所有的依赖组件都需要被发布到 cdn 上。

延伸阅读

- [standalone 支持 umd 方案整理](#)

构建

如果你不喜欢 cmd，还可以自行写构建工具实现基于模块规范的输出

\$ spm build

目前内置的 `spm build` 命令适用于组件和简单项目，功能包含：

- 转换 commonjs 为 cmd 模块 ([gulp-transport](#))
- [spm-standalonify](#)
- [uglifyjs](#)
- [cssmin](#)
- ...

使用

```
$ spm build
```

扩展

对于有自定义需求的复杂项目，或者使用了预编译语言比如 less, coffee 的项目，则需自行基于 spm 提供的基础库实现构建流程。

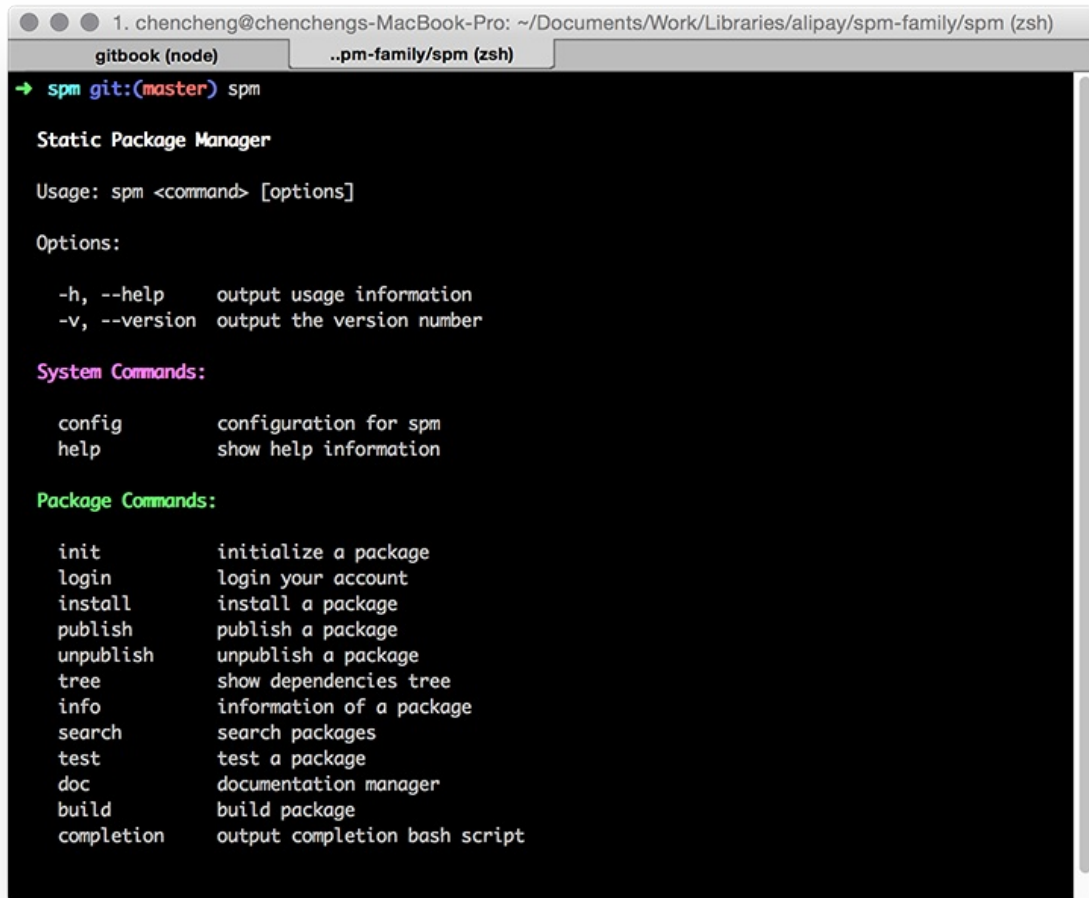
附：一个构建的例子，功能包含：

- less, coffee 的预编译支持
- pathmap 自定义输出路径
- 自定义任务和流
- autoprefixer
- ...

命令行

spm 有一系列的命令来管理组件的生命周期。

下面是简单的列表，你可以执行 `spm help [command]` 查看详细信息。

A terminal window screenshot showing the output of the 'spm' command. The window title is '1. chencheng@chenchengs-MacBook-Pro: ~/Documents/Work/Libraries/alipay/spm-family/spm (zsh)'. The prompt is 'spm git:(master) spm'. The output is as follows:

```
Static Package Manager

Usage: spm <command> [options]

Options:

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

System Commands:

  config          configuration for spm
  help            show help information

Package Commands:

  init            initialize a package
  login           login your account
  install         install a package
  publish         publish a package
  unpublish       unpublish a package
  tree            show dependencies tree
  info            information of a package
  search          search packages
  test            test a package
  doc             documentation manager
  build           build package
  completion      output completion bash script
```

spm init

通过模板生产组件的脚手架。

spm login

登录，获取组件发布权限。

spm install [name[@version]]

安装依赖到本地目录。

spm publish

发布组件。

spm unpublish [name[@version]]

撤销组件发布。

spm tree

显示组件的依赖树。

spm info [name[@version]]

通过名字查询组件信息。

spm search [query]

搜索组件。

spm test

通过 phantomjs 跑测试用例。

spm doc [build|watch|publish]

文档管理工具集。

- spm doc build

构建文档到 `_site` 文件夹。

- spm doc watch

构建文档，并启动监听服务，地址是：<http://127.0.0.1:8000>。

- spm doc publish

发布 `_site` 文件夹到 spmjs.io，Demo url 是 `http://spmjs.io/docs/{{package-name}}`。

spm build

构建组件。

- `-O [dir]` output directory, default: dist
- `--include [include]` determine which files will be included, optional: relative, all, standalone

- relative default

Only contain relative dependencies. Absolute dependencies should also be deployed so that it can run on Sea.js.

```
// would load abc, and abc's dependencies separately.
seajs.use('abc');
```

- all

Contain relative and absolute dependencies.

```
// only need to load abc.  
seajs.use('abc');
```

- standalone

Build a standalone package that could be used in script tag way without any loader.

```
<script src="path/to/abc.js"></script>
```

- `--ignore [ignore]` determine which id will not be transported
- `--idleading [idleading]` prefix of module name, default: `{{name}}/{{version}}`

spm completion

spm 命令自动完成脚本，可通过 `npm completion >> ~/.bashrc (or ~/.zshrc)` 安装。

源

什么是源？

这里的源指的是 spm 的服务端，即：<http://spmjs.io>。所有源上的包都是以 commonjs 或 umd 的形式存在。

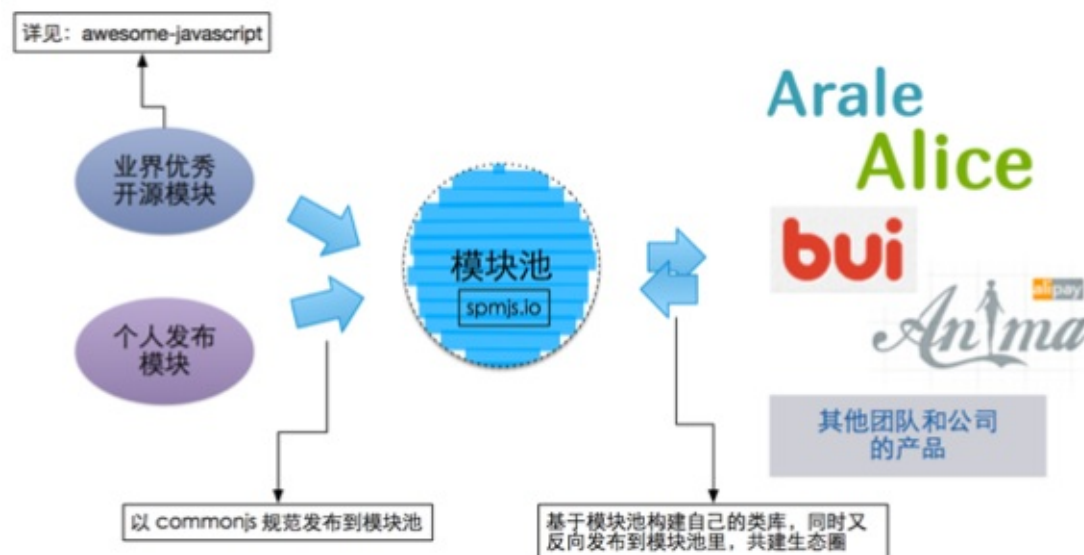
SPM 为什么选择自搭源？

这里有具体的讨论和分析：<https://github.com/spmjs/spm/issues/718>

另外，个人觉得自搭源最重要的优势是 稳定性，尤其是对于大公司来说。很难想象研发流程的链路上需要依赖一个外部的不稳定服务，比如随时可能被墙的 github，有速度问题的 npm。

出于这个考虑，SPM 的源提供了 同步机制，自己搭建的私有源可以从 spmjs.io 上定时同步。比如在支付宝，就有搭建自己的私有源，从而保证研发流程的稳定。

组件池构成



如上图。目前源上的组件池主要包含以下 3 部分：

- 业界优秀开源模块 (持续推广和迁移中，详见：[awesome-javascript](#))
- 个人发布模块
- 团队和公司的产品 (通常带前缀，比如 arale-dialog)

欢迎共建组件池。

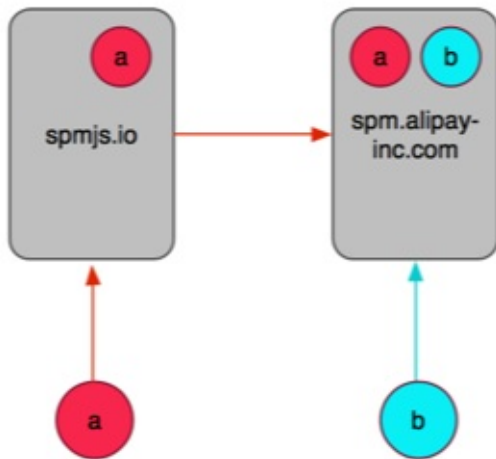
附：<https://github.com/spmjs/spm/wiki/迁移业界优秀模块到-spmjs.io-并反推的简易教程>

私有源

自搭私有源通常有两个目的：

- 保障研发流程的稳定性
- 保证业务组件的私有性

单向同步



私有源可开启来自公有源 (spmjs.io) 的单向同步。

搭建

详见：<https://github.com/spmjs/spmjs.io>。然后可以修改 `config/base.yaml` 里的 `sync` 为 `on` 可开启来自 spmjs.io 的单向同步。

使用

spm 里所有和源操作相关的命令，都可以添加 `--registry` 参数来指定源路径，比如：

```
$ spm publish --registry http://path/to/your-private-registry
```

另外，如果不想每次都多输入 `--registry` 参数，也可以把配置到 `spmrc` 里：

```
$ spm config registry http://path/to/your-private-registry
```