七天学会 NodeJS #2

有经验的 C 程序员在编写一个新程序时首先从 make 文件写起。同样的，使用 NodeJS 编写程序前，为了有个良好的开端，首先需要准备好代码的目录结构和部署方式，就如同修房子要先搭脚手架。本章将介绍与之相关的各种知识。

模块路径解析规则

我们已经知道，require 函数支持斜杠（/）或盘符（C:）开头的绝对路径，也支持./开头的相对路径。但这两种路径在模块之间建立了强耦合关系，一旦某个模块文件的存放位置需要变更，使用该模块的其它模块的代码也需要跟着调整，变得牵一发动全身。因此，require 函数支持第三种形式的路径，写法类似于 foo/bar，并依次按照以下规则解析路径，直到找到模块位置。

内置模块

如果传递给 require 函数的是 NodeJS 内置模块名称，不做路径解析，直接返回内部模块的导出对象，例如 require('fs')。

node_modules 目录

NodeJS 定义了一个特殊的 node_modules 目录用于存放模块。例如某个模块的绝对路径是 /home/user/hello.js，在该模块中使用 require('foo/bar')方式加载模块时，则 NodeJS 依次尝试使用以下路径。

```
/home/user/node_modules/foo/bar

 /home/node_modules/foo/bar

 /node_modules/foo/bar
```

NODE_PATH 环境变量

与 PATH 环境变量类似，NodeJS 允许通过 NODE_PATH 环境变量来指定额外的模块搜索路径。NODE_PATH 环境变量中包含一到多个目录路径，路径之间在*nix 下使用:分隔，在 Windows 下使用;分隔。例如定义了以下 NODE_PATH 环境变量：

```
NODE_PATH=/home/user/lib:/home/lib
```

当使用 require('foo/bar')的方式加载模块时，则 NodeJS 依次尝试以下路径。

```
/home/user/lib/foo/bar

 /home/lib/foo/bar
```

包（package）

我们已经知道了JS模块的基本单位是单个JS文件，但复杂些的模块往往由多个子模块组成。为了便于管理和使用，我们可以把由多个子模块组成的大模块称做包，并把所有子模块放在同一个目录里。

在组成一个包的所有子模块中，需要有一个入口模块，入口模块的导出对象被作为包的导出对象。例如有以下目录结构。

```
- /home/user/lib/
    - cat/
        head.js
        body.js
        main.js
```

其中 cat 目录定义了一个包，其中包含了3个子模块。main.js 作为入口模块，其内容如下：

```
var head = require('./head');
var body = require('./body');

exports.create = function (name) {
    return {
        name: name,
        head: head.create(),
        body: body.create()
    };
};
```

在其它模块里使用包的时候，需要加载包的入口模块。接着上例，使用 require('/home/user/lib/cat/main')能达到目的，但是入口模块名称出现在路径里看上去不是个好主意。因此我们需要做点额外的工作，让包使用起来更像是单个模块。

index.js

当模块的文件名是 index.js，加载模块时可以使用模块所在目录的路径代替模块文件路径，因此接着上例，以下两条语句等价。

```
var cat = require('/home/user/lib/cat');

var cat = require('/home/user/lib/cat/index');
```

这样处理后，就只需要把包目录路径传递给 `require` 函数，感觉上整个目录被当作单个模块使用，更有整体感。

package.json

如果想自定义入口模块的文件名和存放位置，就需要在包目录下包含一个 `package.json` 文件，并在其中指定入口模块的路径。上例中的 `cat` 模块可以重构如下。

```
- /home/user/lib/

    - cat/

        + doc/

        - lib/

            head.js

            body.js

            main.js

        + tests/

        package.json
```

其中 `package.json` 内容如下。

```
{

    "name": "cat",

    "main": "./lib/main.js"

}
```

如此一来，就同样可以使用 `require('/home/user/lib/cat')`的方式加载模块。**NodeJS** 会根据包目录下的 `package.json` 找到入口模块所在位置。

命令行程序

使用 **NodeJS** 编写的东西，要么是一个包，要么是一个命令行程序，而前者最终也会用于开发后者。因此我们在部署代码时需要一些技巧，让用户觉得自己是在使用一个命令行程序。

例如我们用 NodeJS 写了个程序，可以把命令行参数原样打印出来。该程序很简单，在主模块内实现了所有功能。并且写好后，我们把该程序部署在/home/user/bin/node-echo.js 这个位置。为了在任何目录下都能运行该程序，我们需要使用以下终端命令。

```
$ node /home/user/bin/node-echo.js Hello World

Hello World
```

这种使用方式看起来不怎么像是一个命令行程序，下边的才是我们期望的方式。

```
$ node-echo Hello World
```

*nix

在*nix 系统下，我们可以把 JS 文件当作 shell 脚本来运行，从而达到上述目的，具体步骤如下：

在 shell 脚本中，可以通过#!注释来指定当前脚本使用的解析器。所以我们首先在 node-echo.js 文件顶部增加以下一行注释，表明当前脚本使用 NodeJS 解析。

```
#! /usr/bin/env node
```

NodeJS 会忽略掉位于 JS 模块首行的#!注释，不必担心这行注释是非法语句。

然后，我们使用以下命令赋予 node-echo.js 文件执行权限。

```
$ chmod +x /home/user/bin/node-echo.js
```

最后，我们在 PATH 环境变量中指定的某个目录下，例如在/usr/local/bin 下边创建一个软链文件，文件名与我们希望使用的终端命令同名，命令如下：

```
$ sudo ln -s /home/user/bin/node-echo.js /usr/local/bin/node-echo
```

这样处理后，我们就可以在任何目录下使用 node-echo 命令了。

Windows

在 Windows 系统下的做法完全不同，我们得靠.cmd 文件来解决问题。假设 node-echo.js 存放在 C:\Users\user\bin 目录，并且该目录已经添加到 PATH 环境变量里了。接下来需要在该目录下新建一个名为 node-echo.cmd 的文件，文件内容如下：

```
@node "C:\User\user\bin\node-echo.js" %*
```

这样处理后，我们就可以在任何目录下使用 node-echo 命令了。

工程目录

了解了以上知识后，现在我们可以来完整地规划一个工程目录了。以编写一个命令行程序为例，一般 我们会同时提供命令行模式和 API 模式两种使用方式，并且我们会借助三方包来

编写代码。除了代码外，一个完整的程序也应该有自己的文档和测试用例。因此， 一个标准的工程目录都看起来像下边这样。

```
- /home/user/workspace/node-echo/    # 工程目录
    - bin/                           # 存放命令行相关代码
        node-echo
    + doc/                           # 存放文档
    - lib/                           # 存放 API 相关代码
        echo.js
    - node_modules/                  # 存放三方包
        + argv/
    + tests/                         # 存放测试用例
    package.json                     # 元数据文件
    README.md                        # 说明文件
```

其中部分文件内容如下：

```
/* bin/node-echo */
var argv = require('argv'),
    echo = require('../lib/echo');
console.log(echo(argv.join(' ')));


/* lib/echo.js */
module.exports = function (message) {
    return message;
};
```

```
/* package.json */

{

    "name": "node-echo",

    "main": "./lib/echo.js"

}
```

以上例子中分类存放了不同类型的文件，并通过 `node_moudles` 目录直接使用三方包名加载模块。此外，定义了 `package.json` 之后， `node-echo` 目录也可被当作一个包来使用。

NPM

NPM 是随同 **NodeJS** 一起安装的包管理工具，能解决 **NodeJS** 代码部署上的很多问题，常见的使用场景有以下几种：

允许用户从 **NPM** 服务器下载别人编写的三方包到本地使用。

允许用户从 **NPM** 服务器下载并安装别人编写的命令行程序到本地使用。

允许用户将自己编写的包或命令行程序上传到 **NPM** 服务器供别人使用。

可以看到， **NPM** 建立了一个 **NodeJS** 生态圈， **NodeJS** 开发者和用户可以在里边互通有无。以下分别介绍这三种场景下怎样使用 **NPM**。

下载三方包

需要使用三方包时，首先得知道有哪些包可用。虽然 **npmjs.org** 提供了个搜索框可以根据包名来搜索，但如果连想使用的三方包的名字都不确定的话，就请百度一下吧。知道了包名后，比如上边例子中的 `argv`，就可以在工程目录下打开终端，使用以下命令来下载三方包。

```
$ npm install argv

...

argv@0.0.2 node_modules\argv
```

下载好之后， `argv` 包就放在了工程目录下的 `node_modules` 目录中，因此在代码中只需要通过 `require('argv')`的方式就好，无需指定三方包路径。

以上命令默认下载最新版三方包，如果想要下载指定版本的话，可以在包名后边加上 `@<version>`，例如通过以下命令可下载**0.0.1**版的 `argv`。

```
$ npm install argv@0.0.1

...
```

```
argv@0.0.1 node_modules\argv
```

如果使用到的三方包比较多，在终端下一个包一条命令地安装未免太人肉了。因此 NPM 对 package.json 的字段做了扩展，允许在其中申明三方包依赖。因此，上边例子中的 package.json 可以改写如下：

```
{
    "name": "node-echo",
    "main": "./lib/echo.js",
    "dependencies": {
        "argv": "0.0.2"
    }
}
```

这样处理后，在工程目录下就可以使用 npm install 命令批量安装三方包了。更重要的是，当以后 node-echo 也上传到了 NPM 服务器，别人下载这个包时，NPM 会根据包中申明的三方包依赖自动下载进一步依赖的三方包。例如，使用 npm install node-echo 命令时，NPM 会自动创建以下目录结构。

```
- project/
    - node_modules/
        - node-echo/
            - node_modules/
                + argv/
            ...
    ...
```

如此一来，用户只需关心自己直接使用的三方包，不需要自己去解决所有包的依赖关系。

安装命令行程序

从 NPM 服务上下载安装一个命令行程序的方法与三方包类似。例如上例中的 node-echo 提供了命令行使用方式，只要 node-echo 自己配置好了相关的 package.json 字段，对于用户而言，只需要使用以下命令安装程序。

```
$ npm install node-echo -g
```

参数中的-g表示全局安装，因此node-echo会默认安装到以下位置，并且NPM会自动创建好*nix系统下需要的软链文件或Windows系统下需要的.cmd文件。

```
- /usr/local/                    # *nix 系统下
    - lib/node_modules/
        + node-echo/
        ...
    - bin/
        node-echo
        ...
    ...

- %APPDATA%\npm\                 # Windows 系统下
    - node_modules\
        + node-echo\
        ...
    node-echo.cmd
    ...
```

发布代码

第一次使用NPM发布代码前需要注册一个账号。终端下运行npm adduser，之后按照提示做即可。账号搞定后，接着我们需要编辑package.json文件，加入NPM必需的字段。接着上边node-echo的例子，package.json里必要的字段如下。

```
{
    "name": "node-echo",            # 包名，在 NPM 服务器上须要保持唯一
    "version": "1.0.0",             # 当前版本号
    "dependencies": {               # 三方包依赖，需要指定包名和版本号
```

```
    "argv": "0.0.2"
  },
  "main": "./lib/echo.js",          # 入口模块位置
  "bin" : {
    "node-echo": "./bin/node-echo"      # 命令行程序名和主模块位置
  }
}
```

之后，我们就可以在 `package.json` 所在目录下运行 `npm publish` 发布代码了。

版本号

使用 NPM 下载和发布代码时都会接触到版本号。NPM 使用语义版本号来管理代码，这里简单介绍一下。

语义版本号分为 `X.Y.Z` 三位，分别代表主版本号、次版本号和补丁版本号。当代码变更时，版本号按以下原则更新。

+ 如果只是修复 bug，需要更新 Z 位。

+ 如果是新增了功能，但是向下兼容，需要更新 Y 位。

+ 如果有大变动，向下不兼容，需要更新 X 位。

版本号有了这个保证后，在申明三方包依赖时，除了可依赖于一个固定版本号外，还可依赖于某个范围的版本号。例如`"argv": "0.0.x"`表示依赖于`0.0.x`系列的最新版 `argv`。NPM 支持的所有版本号范围指定方式可以查看官方文档。

灵机一点

除了本章介绍的部分外，NPM 还提供了很多功能，`package.json` 里也有很多其它有用的字段。除了可以在 `npmjs.org/doc/`查看官方文档外，这里再介绍一些 NPM 常用命令。

NPM 提供了很多命令，例如 `install` 和 `publish`，使用 `npm help` 可查看所有命令。

使用 `npm help <command>`可查看某条命令的详细帮助，例如 `npm help install`。

在 `package.json` 所在目录下使用 `npm install . -g` 可先在本地安装当前命令行程序，可用于发布前的本地测试。

使用 `npm update <package>`可以把当前目录下 `node_modules` 子目录里边的对应模块更新至最新版本。

使用 `npm update <package> -g` 可以把全局安装的对应命令行程序更新至最新版。

使用 `npm cache clear` 可以清空 **NPM** 本地缓存，用于对付使用相同版本号发布新版本代码的人。

使用 `npm unpublish <package>@<version>`可以撤销发布自己发布过的某个版本代码。

小结

本章介绍了使用 **NodeJS** 编写代码前需要做的准备工作，总结起来有以下几点：

编写代码前先规划好目录结构，才能做到有条不紊。

捎大些的程序可以将代码拆分为多个模块管理，更大些的程序可以使用包来组织模块。

合理使用 `node_modules` 和 `NODE_PATH` 来解耦包的使用方式和物理路径。

使用 **NPM** 加入 **NodeJS** 生态圈互通有无。

想到了心仪的包名时请提前在 **NPM** 上抢注。