

目 录

致谢

阅前必读

简介

开发 npm 模块

开发node-validator

编写测试用例

管理npm模块

开发命令行工具

调用接口

美化输出

完善工具

Express 站点开发

使用 Express

开发 Express 应用

服务器部署

附录

Node.js 基础

npm 基础

参考

更新说明

致谢

当前文档《Node.js 实战》由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-03-17。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：<http://www.bookstack.cn/books/node-in-action>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

阅前必读

- [Node.js 实战](#)

Node.js 实战

这是一本关于 Node.js 实战的小书（更新说明可以[点此查看](#)），旨在让 Node.js 初学者能够快速创建自己的 Node.js 项目并享受其中的开发和探索的乐趣。

本书尽量通过通俗易懂的文字、并通过作者的三个 Node.js 项目的开发过程，帮助读者掌握一些基本的 Node.js 知识和开发技巧：

- [node-validator](#)
- [node-translator](#)
- [Riki](#)

注意：如果还不了解 *Node.js* 与 *npm* 的有关内容，可以先浏览一下附录中的[Node.js 基础](#)与[npm 基础](#)

对本书的内容有任何疑问或建议，都可以在本书的 Github 主页以 [issues](#)的方式提出，或者写[邮件](#)给作者反馈。

最后，欢迎给本书点上一个 [Star](#) 或通过支付宝[打赏作者](#)一杯咖啡。

本作品采用 [CC-BY-NC 4.0协议](#)进行许可。



简介

- [Node.js 实战](#)

Node.js 实战

这是一本关于 Node.js 实战的小书（更新说明可以[点此查看](#)），旨在让 Node.js 初学者能够快速创建自己的 Node.js 项目并享受其中的开发和探索的乐趣。

本书假设读者有一定的 JavaScript 语言基础，并尽量通过通俗易懂的文字和三个 Node.js 项目的开发实践过程，帮助读者掌握一些基本的 Node.js 知识和开发技巧：

- [node-validator](#)
- [node-translator](#)
- [Riki](#)

注意：如果还不了解 *Node.js* 与 *npm* 的有关内容，可以先浏览一下附录中的[Node.js 基础](#)与[npm 基础](#)

对本书的内容有任何疑问或建议，都可以在本书的 Github 主页以[issues](#)的方式提出，或者写[邮件](#)给作者反馈。

最后，欢迎给本书点上一个 [Star](#) 或通过支付宝[打赏作者](#)一杯咖啡。

本作品采用[CC-BY-NC 4.0](#)协议进行许可。



开发 npm 模块

- [开发npm模块](#)

开发npm模块

在第一章中，将通过 `node-validator` 的开发过程来介绍如何从头开发、测试、维护一个npm的模块，加入 npm 的大家庭。

在学习开发你的第一个 npm 模块之前，让我们一起来看看将要实现的是一个什么样功能的包。

`node-validator` 是一个用来校验字符串的 Node.js 模块，我们可以通过

```
1. npm install is-valid --save
```

将其作为依赖安装到你的项目中。

接下来，可以通过简单的代码来看看这个即将完成的模块实现的功能，当然你也可以使用Node.js的REPL：

```
1. var validator = require('is-valid');  
2.  
3. validator.isEmail('foo@bar.net'); // true
```

是的，我们的第一个模块很简单，但是学到的内容却很重要。

开发node-validator

- [开发node-validator](#)

开发node-validator

“node-validator”，顾名思义，我们要开发的是一个验证字符串合法性的npm模块。

首先我们需要建立一个包的目录：

```
1. node-validator
2.   |- lib/
3.   |- test/
4.   |- package.json
5.   |- index.js
6.   |- README.md
```

由于项目比较简单，可以把所有的代码放在根目录下的 `index.js` 中。

不过为了项目的可扩展性，我们会把所有实现代码放在 `lib` 文件夹内，这样一来 `index.js` 就可以只作为一个入口文件存在：

```
1. module.exports = require('./lib');
```

接下来在 `lib` 文件夹中创建一个 `index.js` 文件，用以编写我们的模块的「内容」：

```
1. module.exports = function () {
2.   console.log('Hello npm!');
3. };
```

是的，这就是npm版本的 “Hello, World!”。

这时候可以创建另外一个 JavaScript 文件，然后 `require('./lib')` 来测试、运行一下刚刚编写的代码。

在 CommonJS 的模块系统中，`module.exports` 可以输出一个函数，也可以输出一个对象。所以我们可以这样编写 `lib` 中的 `index.js`：

```
1. module.exports = {
2.   isEmail: function () {},
3.   isAllEnglish: function () {}
4. };
```

接下来，就可以开始编写函数体内的正则代码了：

```
1. module.exports = {
2.   isEmail: function (str) {
3.     return /^[a-zA-Z0-9.!#$%&'*/+=?^_`{|}~-]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?
       (?:\.[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)*$/i.test(str);
4.   },
5.   isAllEnglish: function (str) {
6.     return /^[a-zA-Z]+$/.test(str);
7.   }
8. }
```

那么到了这里，第一个版本的 `node-validator` 就已经完工了。

通过 `npm link` 可以将当前的npm包链接到存放系统中 `npm` 模块的文件夹。也就是说，当前文件夹的改动会在运行的时候体现出来，所以也是开发npm模块时候的利器。

假设我们在 `package.json` 文件中将 `name` 命名为 `validator-test`，那么就已经可以通过如下代码使用了新鲜出炉的模块了：

```
1. var validator = require('validator-test');
2.
3. validator.isEmail('foo@bar.net'); // true
```

我们可以不断的在这个对象中添加函数，当然也可以根据不同的类别分成不同的文件模块，通过 `require` 的方式构成我们的第一个完整的模块。

编写测试用例

- 编写测试用例

编写测试用例

Node.js中已经有很多优秀的测试框架，例如：[mocha](#)，[jasmine](#)等。

这里，我们选择了mocha作为测试框架。

想要了解mocha的读者，可以访问其主页以获取更多信息：[mochajs.org](#)

- 安装mocha

```
1. npm install -g mocha
```

mocha集成了很多的特性，用户可以根据项目的特点选择合适的特性进行测试用例的编写。而在此，我们可以选择“assertion”（断言）来对“node-validator”进行测试。

- 引入 `assert`

```
1. var assert = require('assert');
```

“assert”中包含了很多Node.js中有关断言的模块，例如[shoud.js](#)，[expect](#)等。这些模块多数都是行为驱动开发（BDD）的实践。

- assert示例

```
1. describe('Array', function() {
2.     describe('#indexOf()', function() {
3.         it('should return -1 when the value is not present', function() {
4.             [1,2,3].indexOf(5).should.equal(-1);
5.             [1,2,3].indexOf(0).should.equal(-1);
6.         });
7.     });
8. });
```

这看起来就像是我们用英语描述了一件事情。没错，我们要做的就是描述“node-validator”中的函数运行正确是什么样的，运行错误是什么样的。

照着上面的例子，可以写出测试用例代码原型：

```
1. var assert = require('assert');
```



```
2. var validator = require('validator-test');
3.
4. describe('Validator', function () {
5.     describe('#isEmail', function () {
6.         it('should return true when the string is an email address', function () {
7.             if (validator.isEmail('foo@bar.net') !== true) {
8.                 throw new Error('Validator not right');
9.             }
10.        });
11.    });
12. });
```

然后在终端中输入mocha，会自动运行 `test` 目录下的 `test.js` 文件：

```
1. mocha
```

得到以下结果：

```
1. Validator
2. #isEmail
3. ✓ should return true when the string is an email address
4.
5.
6. 1 passing (6ms)
```

所以接下来要做的事情，就是为每一个“node-validator”中的函数编写测试用例，以期将所有情况都覆盖到。

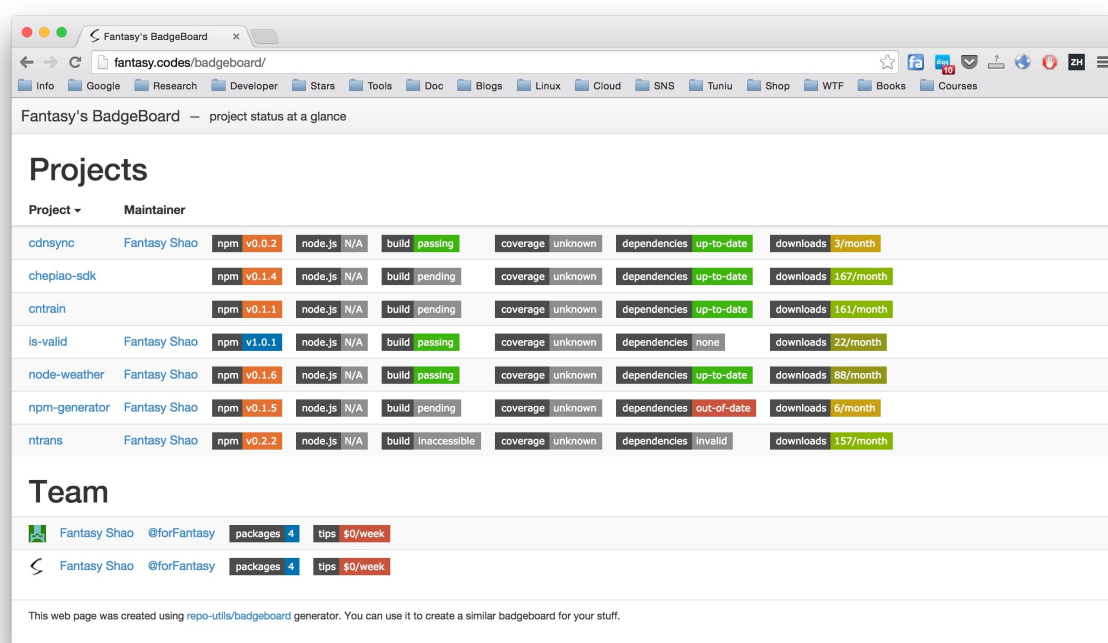
管理npm模块

- 管理npm模块
 - 跟踪Bug

管理npm模块

所有发布在npm上的模块，都可以在自己的npm主页上看到。

但是如果想要更为直观的管理，可以通过一个名为**badgeboard**的工具来浏览，以下是我的npm badgeboard：



对于npm的模块，已经有很多成熟的服务，方便开发者了解自己的模块状态。

正如上图中所示，常用的服务有：

- 持续集成：[travis](#)
- 测试覆盖：[coveralls](#)
- 查看模块依赖：[david-dm](#)

这些工具都可以非常好的帮助你管理自己的npm模块、了解模块的状态。

例如，使用“david-dm”可以了解模块的依赖是否为最新。若依赖有所落后，则最好及时跟进最新版。

跟踪Bug

即使拥有travis和coveralls这样的工具，还是难以避免测试用例没有覆盖等没有预料到的情况。

而对于npm而言，本身只是一个模块的平台，并不具备BUG跟踪、反馈的功能。所以还是需要配合Github的issues来收集用户反馈，及时修复Bug。

开发命令行工具

- [开发命令行工具](#)
 - [命令行参数](#)
 - [node-translator](#)

开发命令行工具

在第二章中，我们将会学习如何使用 Node.js 开发命令行工具。

在**nix* 系统中，命令行工具是平日里开发、工作、日常生活的必备品，而使用 Node.js 可以快速的发展一个自己所需要与喜好的命令行工具。

一个 Node.js 的命令行工具其实都是通过 `node` 的可执行文件来运行的，然后通过npm工具写入 `/usr/local/bin` 这样的可执行文件目录以达到可以通过命令行运行的目的，这与其他语言编写的命令行工具都是一样的。

以下是一个简单的 Node.js 的脚本：

```
1. #!/usr/bin/env node
2.
3. console.log('Hello CLI');
```

命令行参数

在命令行之中，我们往往需要加入很多的参数。[TJ](#)开发了一个非常实用的模块 — [Commander.js](#)，专门用于获取命令行参数，从而方便程序的编写。

例如，我们可以这样使用commander.js：

```
1. #!/usr/bin/env node
2.
3. var program = require('commander');
4.
5. program.version('0.0.1').parse(process.argv);
```

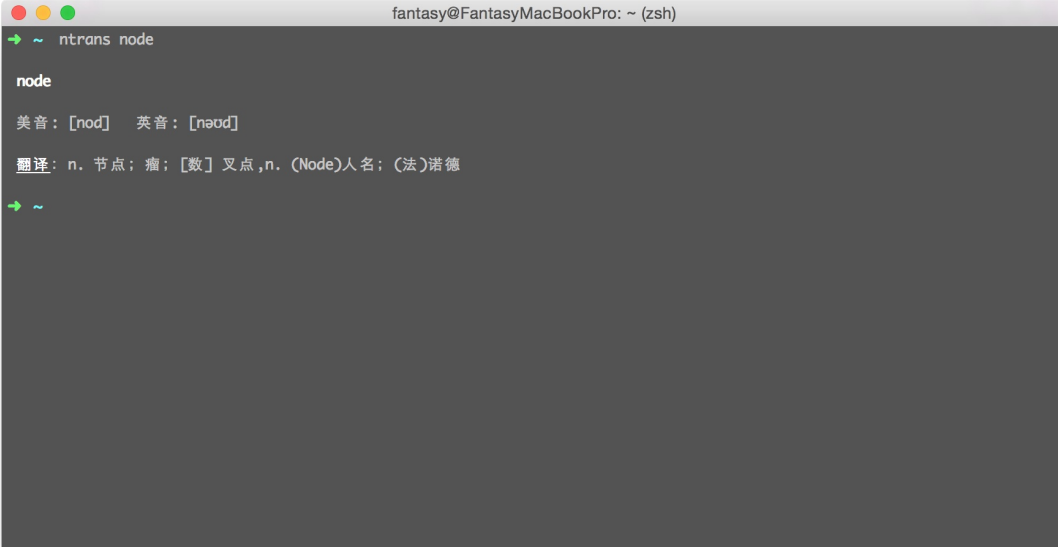
然后就可以这样运行可执行文件获取其版本：

```
1. node test --version
```

此工具适合那些需要传入很多参数并获取的程序。

node-translator

node-translator是一个在命令行中使用有道字典的API进行翻译的命令行工具，在本章中，我们就要完成这样一个非常实用的命令行工具：



```
fantasy@FantasyMacBookPro: ~ (zsh)
→ ~ ntrans node

node

美音：[nod] 英音：[nəʊd]

翻译：n. 节点；瘤；[数] 叉点，n. (Node)人名；(法)诺德

→ ~
```

本章中的章节都会基于这个工具的代码进行讲解。

调用接口

- [调用接口](#)
 - [使用request](#)
 - [有道词典](#)

调用接口

将要开发的这个命令行工具的核心功能就是通过请求有道词典的 API，然后将接口返回的结果通过一些处理之后输出到终端。

使用request

[request](#) 是 npm 中被使用最多的模块之一，每天有20~30万左右的下载量。

正如前端开发中 jQuery 的 Ajax 方法和 iOS 开发中的 AFNetworking 一样，在 Node.js 应用中被广泛当作请求接口之用。

request 的使用亦是非常简单易懂，以下是request的最简使用方式：

```
1. var request = require('request');
2.
3. request('http://www.google.com', function (error, response, body) {
4.   if (!error && response.statusCode == 200) {
5.     console.log(body); // 输出请求到的body
6.   }
7. });
```

有道词典

有道词典提供了开发者注册、申请调用API的[开放平台](#)，开发者在填写了必要的信息之后就可以获得不限时间的 API key。

基本的 API 请求形式也比较简单，可以在 URL 中添加参数后直接通过 GET 的方式请求之，例如有以下 URL：

```
1. http://fanyi.youdao.com/openapi.do?keyfrom=node-
   translator&key=2058911035&type=data&doctype=json&version=1.1&q=test
```

具体的参数就不一一分析了，这些都可以在开放平台的文档中找到。

接下来，可以使用 `request` 测试一下接口调用，我们将会通过接口获取单词“test”的中文释义：

```
1. var request = require('request');
2.
3. request('http://fanyi.youdao.com/openapi.do?keyfrom=node-
  translator&key=2058911035&type=data&doctype=json&version=1.1&q=test', function (error, response,
  body) {
4.   if (!error && response.statusCode == 200) {
5.     console.log(body);
6.   }
7. });
```

假定上述代码保存为 `request-test.js`，然后运行之：`node request-test.js`，便可得到以下结果：

```
1. {"translation":["测试"], "basic":{"us-phonetic":"test", "phonetic":"test", "uk-
  phonetic":"test", "explains":["n. 试验；检验", "vt. 试验；测试", "vi. 试验；测试", "n. (Test)人名；(英)特斯
  特"]}, "query":"test", "errorCode":0, "web":[{"value":["测试", "试验", "检验"], "key":"test"}, {"value":
  ["测试工程师", "测试员", "软件测试工程师"], "key":"Test engineer"}, {"value":["硬度试验", "硬度测试", "硬度实
  验"], "key":"hardness test"}]}
```

可以发现调用接口后返回的body中的内容即为我们所需的JSON数据。

美化输出

- 美化输出
 - 增添色彩
 - 调整输出格式

美化输出

在获得了必要的数据之后，其实已经完成了整个命令行工具的核心功能了。

但是作为一个命令行的工具，还是比较好的输出形式以增强用户体验。

增添色彩

使用`colors`可以为命令行工具的输出增添色彩。

```
1. var colors = require('colors');
2.
3. console.log('Color'.green);
```

只需要简单的在字符串之后添加想要输出的颜色即可。具体支持的颜色可以移步其npm或者Github上的文档查阅。

调整输出格式

在上一节中，我们获取到的JSON返回值如下：

```
1. {"translation":["测试"],"basic":{"us-phonetic":"test","phonetic":"test","uk-phonetic":"test","explains":["n. 试验；检验","vt. 试验；测试","vi. 试验；测试","n. (Test)人名；(英)特斯特"],"query":"test","errorCode":0,"web":[{"value":["测试","试验","检验"],"key":"test"}, {"value":["测试工程师","测试员","软件测试工程师"],"key":"Test engineer"}, {"value":["硬度试验","硬度测试","硬度实验"],"key":"hardness test"}]}
```

格式化后可以得到：

```
1. {
2.   "translation":["测试"],
3.   "basic":{"
4.     "us-phonetic":"test",
5.     "phonetic":"test",
6.     "uk-phonetic":"test",
```



```
7.         "explains":[
8.             "n. 试验；检验",
9.             "vt. 试验；测试",
10.            "vi. 试验；测试",
11.            "n.(Test)人名；(英)特斯特"
12.        ]
13.    },
14.    "query":"test",
15.    "errorCode":0,
16.    "web":[{"
17.        "value":["测试","试验","检验"],
18.        "key":"test"
19.    },{
20.        "value":["测试工程师","测试员","软件测试工程师"],
21.        "key":"Test engineer"
22.    },{
23.        "value":["硬度试验","硬度测试","硬度实验"],
24.        "key":"hardness test"
25.    }]
26. }
```

比较好的输出格式应该包含合适的空行、空格与缩进，可以参考我在node-translator中写的 [output.js](#)。

完善工具

- 完善工具
 - 获取参数

完善工具

在前面的章节中，已经介绍了如何通过 `request` 请求有道词典的接口、使用 `colors` 给命令行工具添加颜色等。毕竟那些都是组成我们这个工具的一些部分内容，现在我们需要的是如何将之前学习到的内容组成一个完整的工具。

获取参数

在第一节中介绍了TJ的 `commander.js` 工具，不过由于node-translator并不需要获取很多参数，所以完全可以直接获取命令行中的参数。

查阅Node.js的文档后，可以了解到通过Node.js内置的 `process.argv` 获取Node.js的命令行参数的数组，例如：

```
1. node test.js a b
```

`process.argv` 的返回结果为 `['node', 'test.js', 'a', 'b']`。

在「调用接口」一节中，我们已经可以通过GET的方式获取到所需要的数据，但是在实际的使用中，所需要查阅的单词一定是通过命令行参数传递的方式获取而非直接写入到程序中的。

所以我们可以通过上述提到的方式获取参数：

```
1. var param = process.argv[2];
2. var word = param ? param : '';
```

然后，将获取到的参数通过拼接字符串的方式添加到URL中即可请求到需要的返回值：

```
1. var request = require('request');
2.
3. request('http://fanyi.youdao.com/openapi.do?keyfrom=node-translator&key=2058911035&type=data&doctype=json&version=1.1&q=' + word, function (error, response, body) {
4.     if (!error && response.statusCode == 200) {
5.         console.log(body);
6.     }
7. }
```

```
7. });
```

Express 站点开发

- [Express 站点开发](#)

Express 站点开发

在第三章中，我们将会学习如何使用最为流行的 Node.js Web 开发框架 — Express 开发一个小型的站点，并包含了如何部署到服务器的章节。

这部分的内容曾发布在我的个人博客上：[Express 开发 Web 应用](#)，[Express 开发 Web 应用 2](#)。本章中试图以更为清晰的语言讲解整个开发、部署的过程。

整个网站的代码库开源于：github.com/SFantasy/Riki。

使用 Express

- [使用 Express](#)
 - [Express的"Hello World"](#)
 - [Express应用生成器](#)

使用 Express

[Express](#) 堪称是 Node.js 领域最为流行的Web开发框架，由著名的开发者 [TJ](#) 开发，现在已经衍生到4.x版本。

Express的"Hello World"

以下是一个简单的使用 Express 作为服务器的代码，通过字符串作为 Response 返回：

`res.send()`

方法将

`Hello World`

```
1. var express = require('express');
2. var app = express();
3.
4. app.get('/', function (req, res) {
5.     res.send('Hello World');
6. });
7.
8. app.listen(3000, function () {
9.     console.log('Express server started.');
10. });
```

运行 `app.js`：

```
1. node app.js
```

Express应用生成器

Express亦提供了诸如其他著名Web框架一样的快速生成项目的工具：[Express Generator](#)

- 安装：

```
1. npm install -g express-generator
```

- 快速生成Express项目：

```
1. express test
```

随后进入到该目录，并安装所依赖的模块：

```
1. cd test  
2. npm install
```

此时，已经可以通过 `npm start` 启动这个自动生成的项目了。

开发 Express 应用

- [开发Express应用](#)
 - [定义路由](#)
 - [MongoDB](#)
 - [参考](#)

开发Express应用

MVC 是很多应用开发时都会采用的一种「架构模式」，会把一个应用分成 Model-View-Controller，每一部分各自负责：

- Model - 应用的功能实现、数据库相关操作等
- Controller - 负责转发请求，对请求进行处理等
- View - 应用的界面部分，与用户的交互等

同样的，我们在开发Express应用的时候也可以采取这样的清晰明了的开发模式，所以我们可以先构建好应用的文件目录，大致如下：

```
1. - models/  
2. - controllers/  
3. - views/  
4. - app.js  
5. - package.json
```

这次我也是用的 MVC 的结构，同时稍微扩充了一个 Service 层把数据定义和操作分割：Model 只定义数据，Service 定义操作数据的方法。

定义路由

在项目中，我们可以在 Controller 中定义路由的规则，但是我觉得将项目中路由的部分从中拆出来单独的作为一个文件，即将所有的路由都定义在一起会比较清晰。

一般而言，项目的路由的代码相对于其他的代码实际上并不多，一般一个页面的操作的路由代码仅仅只有1~3行，对于小型的项目而言，一个路由文件就已足矣。

而且如果将路由放在不同的目录中会需要同时在这些文件中 `require('express')`，增加了不必要的重复代码。

```
1. app.get('/', site.index);  
2. app.post('/register', user.register);
```

其中 `site` 与 `user` 分别为两个 Controller 的名称，分别定义不同的 `GET` 或者 `POST` 路由规则。

MongoDB

MongoDB 对于 Node 而言确实很方便，由于 MongoDB 使用的是 JSON 风格的文档存储结构，所以特别是处理数据的时候就像是在处理 JavaScript 的对象一样。

有不知道 MongoDB 为何物的同学可以先点击本节末尾的链接了解一番。

在 Express 中使用 Mongoose 来操作 MongoDB 也特别的方便，例如以下是使用 Mongoose 连接数据库的一个操作：

`Mongoose` 是一个用于操作 MongoDB 的 ORM 模块。

```
1. var mongoose = require('mongoose');
2.
3. mongoose.connect('mongodb://127.0.0.1/test', function (err) {
4.   if (err) {
5.     console.log('connect to %s error: ', err.message);
6.     process.exit(1);
7.   }
8. });
```

再比如定义 Schema，以简单的 User 为例：

```
1. var mongoose = require('mongoose');
2. var Schema = mongoose.Schema;
3.
4. var UserSchema = new Schema({
5.   name: { type: String },
6.   password: { type: String }
7. });
8.
9. mongoose.model('User', UserSchema);
```

参考

- MongoDB: <https://www.mongodb.org/>
- Mongoose: <http://mongoosejs.com/>

服务器部署

- 服务器部署
 - 部署应用
 - Nginx

服务器部署

本章将介绍如何在 AWS(Amazon Web Service) 上部署上文中开发的 Node.js 的应用。

Node.js 在 Ubuntu 等 Linux 发行版上大概有这么两种安装方式：

1. （更改源后）使用发行版自带的包管理器，例如 Debian 系的 apt-get, CentOS 系的 yum 等
Clone GitHub 上 Node.js 的源码，编译安装
2. Node.js 官方给出了使用包管理方式安装 Node.js 的比较方便的方法：[Install instruction](#)，而我也正是使用这种方式安装的：

```
1. curl -sL https://deb.nodesource.com/setup | sudo bash -  
2. sudo apt-get install nodejs
```

在顺利安装完 Node.js 之后可以用 Vim 编辑一个简单的 JavaScript 文件测试一下：

```
1. console.log('Hello Node.js and AWS.');
```

部署应用

首先我需要checkout GitHub 上的 repo：

```
1. git clone https://github.com/SFantasy/Riki.git
```

安装依赖：`npm install`

由于原来 Riki 中是使用 supervisor 启动的，在EC2上还是选择使用PM2来运行。

安装PM2：`sudo npm install -g pm2`

修改原来项目种的package.json中的scripts：

```
1. pm2 start ./bin/www
```

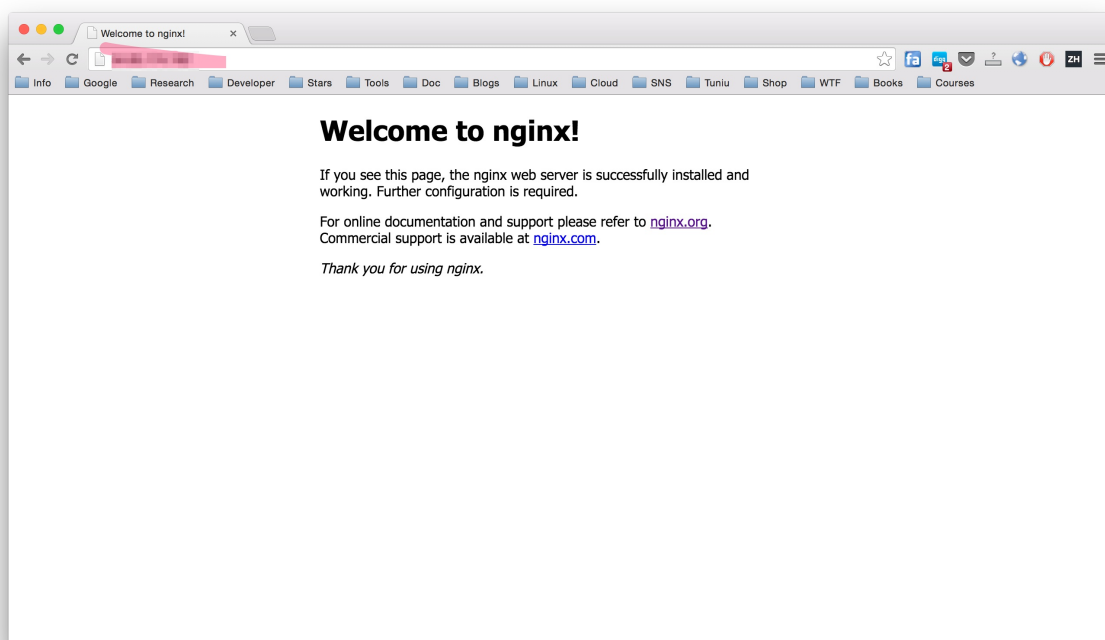
这样就可以通过 `npm start` 来运行我们的项目了，如果使用项目默认的3000端口的话就可以直接运行了。

Nginx

在Ubuntu的机器上安装Nginx非常简单：

```
1. sudo apt-get install nginx
```

安装完后 Nginx 就已经启动了，访问 EC2 的public IP可以看到如下界面：



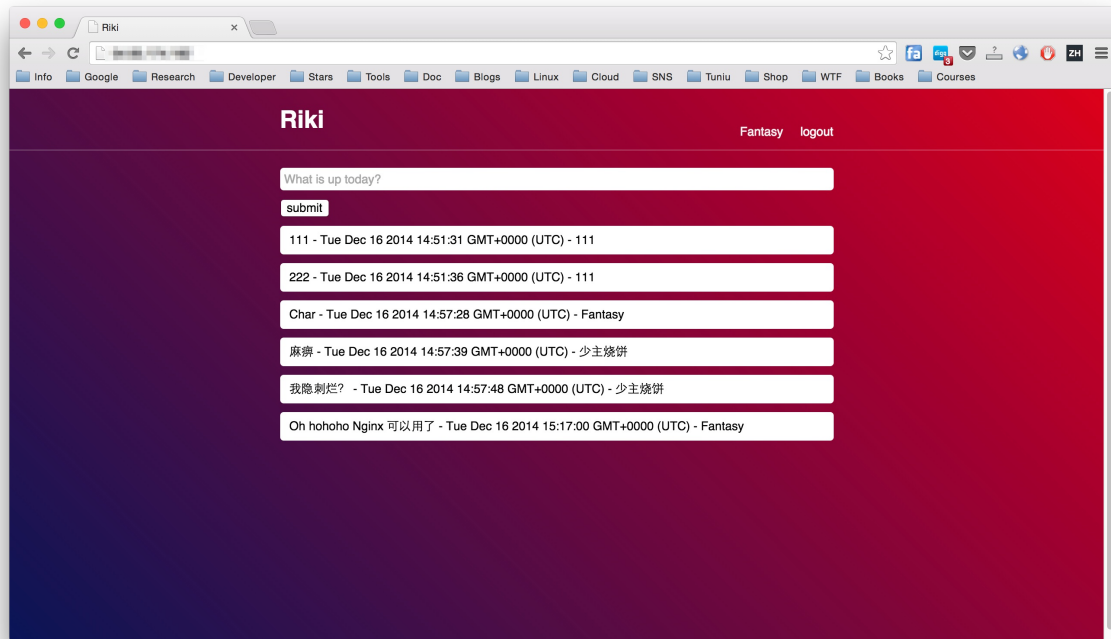
接下来需要修改Nginx的配置文件才能将3000端口转发到Nginx的80端口上，配置文件默认是在 `/etc/nginx/nginx.conf`：

```
1. upstream riki {  
2.     server 127.0.0.1:3000;  
3. }  
4.  
5. server {  
6.     listen 80;  
7.     server_name localhost;  
8.  
9.     location / {  
10.         proxy_pass http://riki;
```

```
11.     }  
12. }
```

这里我是简单粗暴的把原有配置文件中的include的内容注释掉了。

于是乎，我们就可以直接通过IP访问Riki了：



附录

- [附录](#)

附录

Node.js 实战之附录。

本章主要内容为介绍 Node.js 与 npm 的一些基础知识。

Node.js 基础

- [Node.js 基础](#)
 - [Node.js 版本的 Hello World](#)
 - [Node.js 的版本管理](#)
 - [模块系统](#)
 - [最简单的Node.js服务器](#)

Node.js 基础

如果你已经有 Node.js 的编程基础，那么可以跳过此章节，但也不妨将其作为一个参考。

Node.js 版本的 Hello World

```
1. console.log('Hello World');
```

- 运行Node.js

```
1. node test.js
```

Node.js 的版本管理

Node.js 的稳定版，亦即 LTS (Long Time Support) 版本是最为主流使用的版本，一般而言版本号号为偶数 — 4, 6 等，而奇数版本的则为两个稳定版之间的中间产物。因此建议在使用的时候安装 LTS 的版本。

不同版本的 Node.js 还是有不少差异的，而造成这些差异的原因，包含了诸如 V8 的版本，诸如对 ES6 等标准中特性的实现等，所以，切换 Node.js 的版本可能是经常会做的一件事情。

Node.js 的版本可以通过 [NVM](#) 以及 [N](#) 进行管理。

相比这两个工具而言，前者可能做的更为完善、易用。当然你也不妨将两者都尝试一下，选择更为适合自己的。

模块系统

Node.js 实现的是[CommonJS规范](#)：

- 使用 `global` 定义全局对象

```
1. global.test = true;
```

- 使用 `require` 引入模块

```
1. var fs = require('fs');
```

- 使用 `module.exports` 输出模块

```
1. module.exports = function () {  
2.     console.log('Hello Node.js');  
3. };
```

- 使用 `exports` 输出多个方法或者对象：

```
1. exports.foo = function () {};  
2. exports.bar = function () {};
```

最简单的Node.js服务器

使用Node.js的 `http` 模块可以创建一个最简易的HTTP服务器：

```
1. var http = require('http');  
2.  
3. http.createServer(function (req, res) {  
4.     res.send('Hello');  
5.     res.end();  
6. }).listen(3000);
```

随后访问浏览器：`localhost:3000` 便可看到“Hello”字样。

npm 基础

- npm 基础
 - 简介
 - 基本使用
 - 安装模块依赖
 - 管理模块
 - npm 镜像

npm 基础

简介

npm，一般认为是 Node Package Manager 的缩写，当然也有很多其他的别称，这一点可以在 npm 的主页的左上角作为一个彩蛋看到。

npm 类似 Java 中的 Maven，Python 中的 pip，Ruby 中的 Gem 等，可以方便的管理 Node.js 项目中的依赖，在项目中以 `package.json` 的形式展示。

基本使用

以下介绍 npm 的基本使用方式，简单来说包含了 Node.js 模块的依赖管理、模块的管理以及脚本的运行。

当然更好的方式是参阅npm的文档，或者通过 `npm -l` 与 `man npm` 本地查阅各种命令的用途。

安装模块依赖

- 安装项目所依赖的模块

```
1. npm install
```

- 安装指定模块

```
1. npm install express
```

- 安装指定模块并将其保存为项目依赖（会写入 `package.json` 文件的 `dependencies` 字段）


```
1. npm install express --save
```

- 安装指定模块并将其保存为项目的开发依赖（会写入 `package.json` 文件的 `devDependencies` 字段）

```
1. npm install express --save-dev
```

- 全局安装指定模块

```
1. npm install express -g
```

上述命令有些可以写成更为简略的形式。

例如 `npm install express --save` 可以缩写为 `npm i express -S`，而 `npm install express --save-dev` 可以缩写为 `npm i express -D`，是不是很方便呢。

管理模块

- 初始化模块

```
1. npm init
```

这个命令其实只是通过一种交互式的方式生成一个项目的 `package.json` 文件，而并不会创建一些常见的 npm 模块目录的文件夹。

- 发布模块

```
1. npm publish
```

无论是第一次发布模块，亦或是更新模块之后发布更新到npm上，都需要这个命令。不过要注意的是，在发布之前需要通过 `npm adduser` 添加注册过的npm账号。

- 取消发布模块

```
1. npm unpublish
```

npm 在 `left-pad` 事件后，规定在模块发布后的一定时间后就不能 `unpublish`。

npm 镜像

鉴于 npm在国内会受到 GFW 的影响，可以选择使用 [Taonpm](#) 作为 npm 的镜像，方便安装 npm 中的模块。

或者可以选择直接安装 cnpm 加速项目依赖的安装：

```
1. npm install -g cnpm
```

参考

- [参考](#)

参考

- [Node.js API](#)
- [npm Documentation](#)
- [Wikipedia MVC](#)

更新说明

- [更新说明](#)
 - [2016-10](#)
 - [2016-08](#)
 - [2016-07](#)
 - [2016-05](#)

更新说明

2016-10

- 将所有的 NPM 替换为 npm
- 更新了一点 npm 的基础知识

2016-08

- 更新附录中关于 npm 和 Node.js 基础的部分内容
- 移除封面图片

2016-07

- 修改一些排版问题

2016-05

- 修改一些措辞