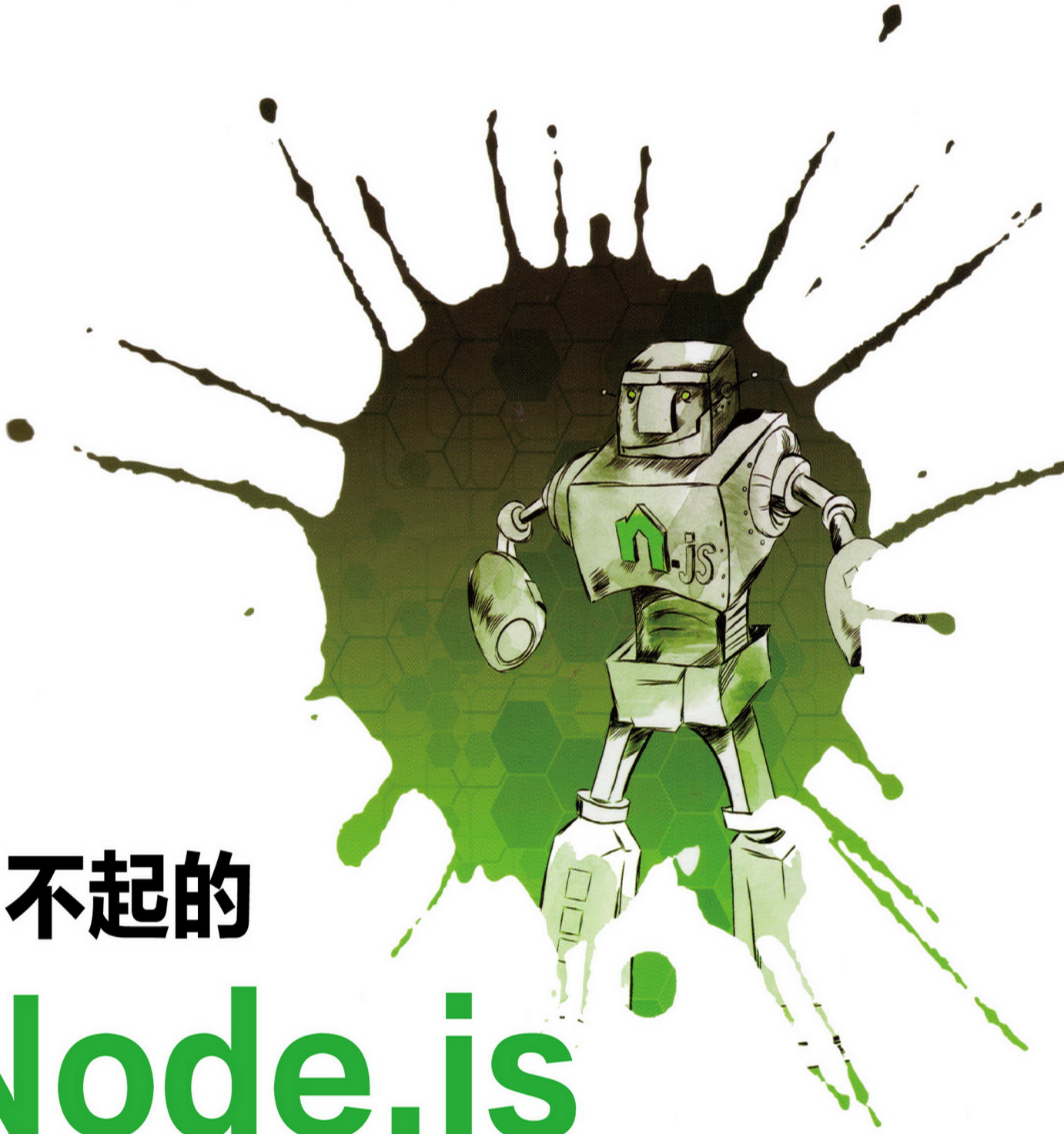
Broadview
www.broadview.com.cn

SMASHING MAGAZINE

SMASHING Node.js: JavaScript Everywhere



了不起的 Node.js

将JavaScript进行到底

Guillermo Rauch 著　Goddy Zhao 译

WILEY

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

内 容 简 介

本书是一本经典的Learning by Doing的书籍。它由Node社区著名的 Socket.IO作者——Guillermo Rauch，通过大量的实践案例撰写，并由 Node社区非常活跃的开发者——Goddy Zhao翻译而成。

本书内容主要由对五大部分的介绍组成：Node核心设计理念、Node核心模块API、Web开发、数据库以及测试。从前到后、由表及里地对使用 Node进行Web开发的每一个环节都进行了深入的讲解，并且最大的特点就是通过大量的实际案例、代码展示来剖析技术点，讲解最佳实践。

版权贸易合同登记号　图字：

图书在版编目（CIP）数据

策划编辑：张春雨
责任编辑：贾　莉
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787×980　1/16　　印张：00　　字数：000千字
印　　次：2013年12月第1次印刷
定　　价：79.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

译者序

从2009年Ryan Dahl着手开发Node.js开始，到现在Node已经快4岁了。尽管它至今还未发布1.0版本，甚至连alpha都还没有，但是Node这几年的发展大家都是有目共睹的。越来越多的开发者和公司开始尝试使用 Node.js：微软在其Azure云上支持了部署Node.js应用，它同时还是Node Windows版本的主要贡献者；雅虎主站大量使用了Node；LinkedIn也使用Node为其移动应用提供服务器端服务。除此之外，Node官方Wiki页面（https://github.com/joyent/node/wiki/Projects,-Applications,-and-Companies-Using-Node）列出了更多在使用Node的公司和项目。可以说，Node以其异步IO、服务器端JavaScript的特点为Web开发掀开了新的篇章。

然而，尽管Node这几年发展神速，但是相关的书籍、资料却很少，尤其在国内就更是寥寥无几。这就让我萌发了为国内Node爱好者翻译一本 Node书籍的想法，于是我就想到了*SMASHING Node.js: JavaScript Everywhere*。之所以选择这本书，是因为：首先，这本书的作者Guillermo Rauch在Node社区非常有名，作为Socket.IO的作者、Express的开发者之一，他为社区贡献了很多质量很高的Node模块；其次，我此前碰巧有幸受原书出版社WILEY之邀，担任了原书的技术审校，所以我对原书内容非常熟悉；最后，也是主要的原因，是因为这本书有大量的实践案例，我个人始终认为学习技术的最佳方式就是实践，而且本书中的案例在阐述技术点的同时，还非常具有实践价值。在上述三点原因的驱使下，最终让我决定向电子工业出版社引荐此书，最后也很高兴出版社能够认同这本书并决定引进此书。

本书根据Web开发的流程，从Node核心概念——事件轮询、V8中的 JavaScript的介绍，Node核心库——TCP、HTTP的讲解，到应用层开发——Connect、Express、Socket.IO的实践，再到数据库——MongoDB、Redis、MySQL的剖析，最后到测试——Mocha、BDD的阐述，每个环节都一一做了深入的讲解。另外，本书始终贯穿了 Learning by Doing的理念，每一章都有大量的实践案例、代码展示，以编写实际代码的方式让读者掌握技术、同时教会读者如何将其运用到实际项目中。总的来说，本书确实是一本学习Node的好书。

最后，在本书翻译过程中要特别感谢来自淘宝网工程师——易敛（花名）以及聚美优品工程师——邵信衡给予的帮助。另外，还要感谢本书的编辑张春雨和贾莉的辛苦工作，以及我太太的大力支持。

希望本书能够为广大Node开发者带来帮助，谢谢！

Goddy Zhao（赵静），SuccessFactors（SAP子公司）软件工程师。毕业于复旦大学，先后在IBM、淘宝工作过，专注于企业级富客户端Web应用的开发，擅长前后端相结合的技术解决方案。曾与人合译过多本前端图书，并曾在沪JS及D2前端技术论坛担任过主持人和演讲嘉宾。

前言

绝大部分Web应用都包含客户端和服务器端两部分。服务器端的实现往往比较复杂、麻烦。创建一个简单的服务器都要求对多线程、伸缩性以及服务器部署有专业的技术知识。除此之外，由于客户端软件是用HTML和JavaScript来实现的，而服务器端核心代码通常都是用静态编程语言实现的，所以，开发Web应用经常会有错乱的感觉。由于这种前后端开发语言的差异，不得不让开发者使用多种编程语言，同时还要对特定的程序逻辑事先做好设计选型。

几年前，要用JavaScript来实现服务端软件几乎是想都不敢想的一件事情。糟糕的性能、不成熟的内存管理以及缺乏操作系统层面的集成，不解决这些问题，JavaScript很难成为一门服务器端的语言。作为 Google Chrome浏览器的一部分，新的V8引擎能够解决前两个问题。V8是一个开源的项目，通过简单的API就可以将其集成进去。

Ryan Dahl洞察到了这样一个机会，可以通过将V8内嵌到操作系统的集成层，来让JavaScript享受到底层操作系统的异步接口，从而实现将其带到服务器端的目的。这就是Node.js的设计思路。这么做的好处是显而易见的。程序员们可以在客户端和服务器端使用同样的编程语言了。 JavaScript动态语言的特性使得开发和试验服务器端代码变得很自由，使得程序员们摆脱了传统那种又慢又重的编程模式。

Node.js迅速蹿红，衍生了一个强大的开源社区、支持企业，甚至还拥有属于自己的技术大会。我把这种成功归结于它的简洁，高效，同时提高了编程生产力。我很高兴V8成为其一小部分。

本书将带着读者学习如何基于Node.js为Web应用构建服务器端部分，同时还会带着大家学习如何组织服务器端异步代码以及如何与数据库进行交互。

好好享受这本书带来的乐趣吧！

Lars Bak,Virtual Machinist

介绍

2009年年末，Ryan Dahl在柏林的一个JavaScript大会上宣布了一项名为Node.js（http://nodejs.org/）的新技术。有意思的是，出乎所有参会者的意料，这项技术居然不是运行在浏览器端的，要知道浏览器端对于JavaScript来说绝对是拥有霸主地位的，这是毋庸置疑的。

这项技术是关于在服务器端运行JavaScript的。当时，这简单的一句描述，瞬间让听众眼前一亮，同时也宣告了这项新技术的发布大获成功。

如果成真的话，以后开发Web应用就只需要一种语言了。

毫无疑问，这是当时所有人的第一想法。毕竟，要开发一个现代富客户端Web应用，必须要对JavaScript非常熟悉和了解，然而，对于服务器端的技术来说，就有很多不同的选择，而且都需要专业的要求。拿Facebook来说，他们最近透露其总代码库中JS的代码量是服务器端语言PHP的四倍。

不过Ryan感兴趣的是为大家展示一个简洁又强大的示例程序。他展示了一个Node.js中的“hello world”程序——创建一个Web服务器。

```
var http = require('http');
var server = http.createServer(function (req, res) {
  res.writeHead(200);
  res.end('Hello world');
});
server.listen(80);
```

这样一个Web服务器并非只是个“玩具”，相反，它是一个高性能的Web服务器，甚至，在某些场景下，比现有如Apache和Nginx这样的Web服务器性能还要好。Node.js被称为是一个将设计网络应用导向正确道路的特殊工具。

Node.js快速高效的优点得益于一种叫做事件轮询（event loop）的技术，以及其构建于V8之上，V8是Google为Chrome Web浏览器设计的 JavaScript解释器和虚拟机，它运行JavaScript非常快。

Node.js改变了Web开发模式。你无须再将书写部署到独立安装的Web服务器中去运行，如

传统的LAMP模式，它通常包含了PHP环境和 Apache服务器。

正如本书正文中将要介绍的，获得Web服务器完全的控制权催生了另外一类基于Node.js开发的应用：实时Web应用。在一个服务器端和众多客户端进行快速的数据传输，在Node开发中变得越来越常见。这意味着你既可以创建更高效的程序，又能成为社区的一部分，推进理想的Web开发模式。

有了Node，你就有了主动权。同时，本书也会详细介绍这种能力背后所带来的新的挑战和责任。

目标

首先最重要的是，本书是一本关于JavaScript的书。你必须具备一定的JavaScript知识，同时，一开始我也花了一章来介绍JavaScript的相关概念，根据我的经验来看，这是非常有帮助的。

正如你将会学到的，Node.js努力为浏览器端开发者提供一个舒服的开发环境。有些常用的表达式，如`setTimeout`和`console.log`，并非语言标准的一部分，而是浏览器添加的，它们在Node.js中也能使用。

在你理清思绪，准备就绪后，就可以开始Node之旅。作为其核心的一部分，Node自带了很多有用的模块，以及一个名为NPM的简单包管理器。本书从教你如何仅使用Node核心模块构建应用开始，随后教你使用一些最有用的社区开发者基于Node开发的模块来开发应用，这些模块都可以通过NPM安装获得。

在介绍如何用专门设计的模块解决特定问题前，我通常会先介绍如何在不使用模块的情况下解决此问题。理解一个工具最好的方式就是首先搞明白为什么会有这个工具。因此，在学习某个Web框架前，你会先学习为什么用它要比使用Node.js原生的HTTP模块要好。在学习如何使用如Socket.IO的跨浏览器的实时框架构建应用前，你会先学习 HTML5 WebSocket的缺陷。

本书包含大量示例。这些示例，会教你如何一步一步构建小应用或者测试不同的API。本书所有的示例代码都可以通过`node`命令运行，以下是两种不同的使用方式：

- 通过node REPL（Read-Eval-Print Loop）。和Firebug或者Web调试器中的JavaScript控制台类似，node REPL允许你从操作系统的命令行工具输入JavaScript代码，按下回车键，就能执行。
- 通过`node`命令运行node文件。这种方式要求你使用已有的文本编辑器。我个人推荐vim（http://vim.org）编辑器，不过，任何文本编辑器都是可以的。

绝大多数例子，会一步步教你书写示例代码，并且，首次书写会讲解其代码含义。我还会带领你经历不同的考验以及代码重构。当到了重要的里程碑时，我通常会展示一个截图，截图

内容取决于开发的应用，可能是终端的截图，也可能是浏览器端窗口的截图。

有的时候，讲解这些示例的时候，不管考虑得多周全，问题可能还是无法避免。所以，我给你提供了一个资源列表来帮助你解决问题。

资源

要是在阅读本书中，遇到问题，可以通过如下途径获得帮助。

要获得关于Node.js的问题的帮助，可以通过如下途径：

- Node.js邮件列表（http://groups.google.com/group/nodejs）。
- `irc.freenode.net`服务器，`#nodejs`频道。

要获得如`socket.io`或者`express`等的特定项目的帮助，可以通过官方支持频道；如果没有，可以通过像Stack Overflow（http:// stackoverflow.com/questions/tagged/node.js）这样的论坛，都会很有帮助。

绝大多数的Node.js模块都托管在GitHub上。如果你发现了bug，就可以通过GitHub报给他们，并贡献相应的测试用例。

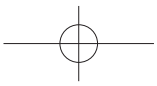
尽力弄清楚你的问题到底属于Node.js还是JavaScript。这对确保你寻求的Node.js帮助确实是与Node相关的问题很有帮助。

如果就本书中的某个问题想要讨论，可以直接通过`rauchg@gmail.com`联系我。

PART

1

从安装与概念开始


CHAPTER

1 安装

安装Node.js比较容易。其设计理念之一就是只维护少量的依赖，这使得编译、安装Node.js变得非常简单。 7

本章介绍如何在Windows、OS X、以及Linux系统下安装Node.js。在Linux系统下，要以编译源代码的方式进行安装[1]，得先确保正确安装了其依赖的软件包。

注意：在本书中，若看到代码片段前有$符号，就表示需要将其代码输入到操作系统的shell中。 8

在Windows下安装

Windows用户要安装Node.js，只需前往其官网http://nodejs.org下载MSI安装包即可。每个Node.js的发行版都有对应的MSI安装包供用户下载和安装。

安装包文件名遵循`node-v?.?.?.msi`[2]的格式，运行安装包之后，简单地根据图1-1所示的安装指引进行安装即可。

1 译者注：在Linux下，官方还提供了二进制包进行安装。

2 译者注：截止到本书翻译期间，目前的格式为node-v?.?.?-bit.msi，这里的bit表示几位的操作系统，如32位就是x86、64位就是x64。

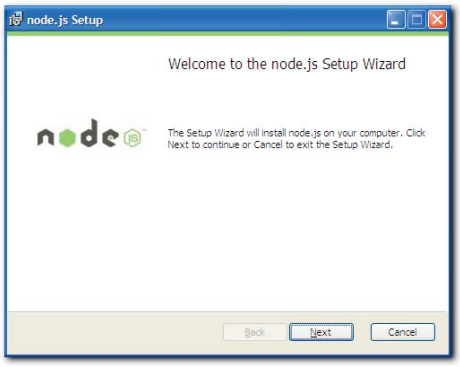


图1–1：Node.js安装指引

要验证是否安装成功，可以打开shell或者通过执行cmd.exe打开命令行工具并输入$ node -version。

如果安装成功的话，就会显示安装的Node.js的版本号。

在OS X下安装

在Mac下和在Windows下安装类似，可通过对应的安装包进行。从Node.js官网下载PKG文件，其文件名格式遵循node-v.?.?.?.pkg。若要通过手动编译来进行安装，请确保机器上已安装了XCode，然后根据Linux下的编译步骤进行编译安装。

运行下载好的安装包，并根据图1-2所示的安装步骤进行安装。

图1–2：Node.js安装包

要验证是否安装成功，打开shell或者运行Terminal.app打开终端工具（也可以在Spotlight中输入“Terminal”来搜索该软件），接着，输入$ node -version。

如果安装成功，就会显示安装的Node.js的版本号。

在Linux下安装

和直接用二进制包安装类似，编译安装Node.js也很简单。要在绝大多数*nix系的系统中编译Node.js，只需要确保系统中有C/C++编译器以及 OpenSSL库就可以了。

要是没有，安装起来也比较容易，大部分的Linux发行版都自带包管理器，通过它可以很方便地进行安装。 9

比方说，在Amazon Linux中，可以通过如下命令来安装依赖包：

```
> sudo yum install gcc gcc-c++ openssl-devel curl
```

在Ubuntu中，安装方式稍有不同，如下所示：

```
> sudo apt-get install g++ libssl-dev apache2-utils curl
```

编译

在操作系统终端下，运行如下命令：

注意：将下面例子中的?替换成最新的Node.js的版本号[3]。

```
$ curl -O http://nodejs.org/dist/node-v?.?.?.tar.gz
$ tar -xzvf node-v?.?.?.tar.gz
$ cd node-v?.?.?
$ ./configure
$ make
$ make test
$ make install
```

如果`make test`命令报错。我建议你停止安装，并将`./configure`、`make`以及`make test`命令产生的日志信息发送给 Node.js的邮件列表。

确保安装成功

打开终端或者类似XTerm这样的应用，并输入`$ node -version`。

如果安装成功的话，就会显示安装的Node.js的版本号。

Node REPL

要运行Node的REPL，在终端输入`node`即可。

可以试试运行一些JavaScript表达式。例如：

```
> Object.keys(global)
```

3 译者注：截止到本书翻译期间，Node.js发行版的下载目录已经更改为http://nodejs.org/dist/v?.?.?/node-v?.?.?.tar.gz。

注意：如果看到本书中的示例代码段前有>，就说明要在REPL中输入。

10 REPL是我最喜欢的工具之一，它能让我很方便地验证一些Node API和JavaScript API是否正确。若有时忘记了某个API的用法，就可以用REPL来验证下，非常有用，尤其是在开发大型模块的时候。我一般都新开一个单独的终端tab，快速在REPL中尝试一些JavaScript的原生用法，真的非常方便。

执行文件

和绝大多数脚本语言一样，Node.js可以通过node命令来执行Node脚本。

用你喜欢的编辑器，创建一个名为my-web-server.js的文件，输入如下内容：

```
var http = require('http');
var serv = http.createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end('<marquee>Smashing Node!</marquee>');
});
serv.listen(3000);
```

使用如下命令来执行此文件：

```
$ node my-web-server.js
```

接着，如图1-3所示，在浏览器中输入http://localhost:3000。

图1–3：使用Node托管一个简单的HTML文件

上述代码展示了如何使用Node书写一个完整的HTTP服务器，来托管一个简单的HTML文档。这是一个Node.js的经典例子，因为它证明了Node.js的强大，仅通过几行JavaScript代码就能创建出一个像Apache或者IIS的Web服务器。

NPM

Node包管理器（NPM）可以让你在项目中轻松地对模块进行管理，它会下载指定的包、

解决包的依赖、运行测试脚本以及安装命令行脚本。

尽管这些工作并非你项目的核心功能，但使用第三方发布的模块可以提高项目的开发效率。

NPM本身是用Node.js开发的，有二进制包的发布形式（Windows下有MSI安装器，Mac下有PKG文件）。若要从源码进行编译安装，可以使用如下命令[4]：

11

```
$ curl http://npmjs.org/install.sh | sh
```

通过如下命令可以检查NPM是否安装成功：

```
$ npm --version
```

安装成功的话，会显示出所安装NPM的版本号。

安装模块

为了展示如何通过NPM来安装模块，我们创建一个`my-project`目录，安装`colors`模块，然后创建一个`index.js`文件：

```
$ mkdir my-project/
$ cd my-project/
$ npm install colors
```

要验证模块是否安装成功，可以在该目录下查看是否有`node_modules/colors`目录。

然后，用你最喜欢的编辑器编辑`index.js`文件：

```
$ vim index.js
```

在该文件中添加如下内容：

```
require('colors');
console.log('smashing node'.rainbow);
```

运行此文件的结果应该如图1-4所示。

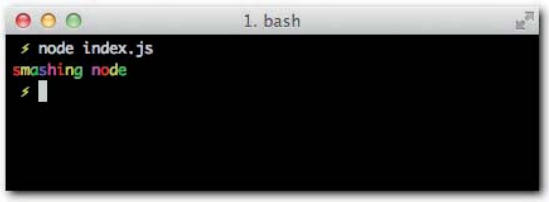


图1–4：模块安装成功验证结果

4 译者注：截止到本书翻译期间，NPM会随着Node.js的安装自动就安装好了，无须手动再去安装NPM，并且 http://nodejs.org/install.sh脚本已经被官方移除。

12 自定义模块

要自定义模块，你需要创建一个package.json文件。通过这种方式来定义模块有三种好处：

- 可以很方便地将项目中的模块分享给其他人，不需要将整个node_modules目录发给他们。因为有了package.json之后，其他人运行npm install就可以把依赖的模块都下载下来，直接将node_modules目录给别人根本就是个馊主意。特别是当用Git这样的SCM系统进行代码控制的时候。
- 可以很方便地记录所依赖模块的版本号。举个例子来说，当你的项目通过npm install colors安装的是0.5.0的colors。一年后，由于colors模块API的更改，可能导致与你的项目不兼容，如果你使用npm install并且不指定版本号来安装的话，你的项目就没法正常运行了。
- 让分享更简单。如果你的项目不错，你是否想将它分享给别人？这时，因为有package.json文件，通过npm publish就可以将其发布到NPM库中供所有人下载使用了。

在原先创建的目录（my-project）中，删除node_modules目录并创建一个package.json文件：

```
$ rm -r node_modules
$ vim package.json
```

然后，将如下内容添加到该文件中[5]：

```
{
    "name": "my-colors-project"
  , "version": "0.0.1"
  , "dependencies": {
      "colors": "0.5.0"
    }
}
```

注意：此文件内容必须遵循JSON格式。仅遵循JavaScript格式是不够的。举例来说，你必须要确保所有的字符串，包括属性名，都是使用双引号而不是单引号。

package.json文件是从Node.js和NPM两个层面来描述项目的。其中只有name和version是必要的字段。通常情况下，还会定义一些依赖的模块，通过使用一个对象，将依赖模块的模块名及版本号以对象的属性和值将其定义在package.json文件中。

保存上述文件，安装依赖的模块，然后再次运行index.js文件：

13

```
$ npm install
$ node index    # 注意了，这里文件名不需要加上“.js”后缀
```

5 译者注：不建议示例代码中逗号的书写风格，个人建议将逗号写在行末。

在本例中，自定义模块是内部使用的。不过，如果想发布出去，NPM提供了如下这种方式，可以很方便地发布模块：

```
$ npm publish
```

当别人使用require('my-colors-project')时，为了能够让Node知道该载入哪个文件，我们可以在package.json文件中使用main属性来指定：

```
{
    "name": "my-colors-project"
  , "version": "0.0.1"
  , "main": "./index"
  , "dependencies": {
      "colors": "0.5.0"
    }
}
```

当需要让模块暴露API的时候，main属性就会变得尤为重要，因为你需要为模块定义一个入口（有的时候，入口可能是多个文件）。

要查看package.json文件所有的属性文档，可以使用如下命令：

```
$ npm help json
```

小贴士：如果你不想发布你的模块，那么在package.json中加入"private": "true"。这样可以避免误发布。

14

安装二进制工具包

有的项目分发的是Node编写的命令行工具。这个时候，安装时要增加-g标志。

举例来说，本书中要介绍的Web框架express就包含一个用于创建项目的可执行工具。

```
$ npm install -g express
```

安装好后，新建一个目录，并在该目录下运行express命令：

```
$ mkdir my-site
$ cd mysite
$ express
```

小贴士：要想分发此类脚本，发布时，在package.json文件中添加"bin": "./path/to/script"项，并将其值指向可执行的脚本或者二进制文件。

浏览NPM仓库

等掌握第4章关于Node.js模块系统的内容后，你就能编写出可以使用Node生态系统中任意类型模块的程序了。

NPM有一个丰富的仓库，包含了上千个模块。NPM有两个命令可以用来在仓库中搜索和查看模块：search和view。

例如，要搜索和realtime相关的模块，就可以执行如下命令：

```
$ npm search realtime
```

该命令会在已发布模块的name、tags以及description字段中搜索此关键字，并返回匹配的模块。

找到了感兴趣的模块后，通过运行npm view命令，后面紧跟该模块名，就能看到package.json文件以及与NPM仓库相关的属性，举个例子：

```
$ npm view socket.io
```

小贴士：输入npm help可以查看某个NPM命令的帮助文档，如npm help publish就会教你如何发布模块。

小结

通过本章的学习，你应当已经搭建好了Node.js + NPM的环境。

除了能够运行node和npm命令外，你现在也应当学会了如何执行简单脚本以及如何声明模块依赖。

相信你还学会了Node.js中一个重要的关键词require，它用来载入模块和系统API，在快速介绍完语言基本知识后，第4章中会对这部分内容做着重介绍。

最后相信你了解了NPM仓库，它是Node.js模块生态系统的入口。Node.js是开源项目，所以大部分Node.js编写的程序也都是开源的，供其他人重用。

CHAPTER

2 JavaScript概览

介绍

JavaScript是基于原型、面向对象、弱类型的动态脚本语言。它从函数式语言中借鉴了一些强大的特性，如闭包和高阶函数，这也是 JavaScript语言本身有意思的地方。 15

从技术层面上来说，JavaScript是根据ECMAScript语言标准来实现的。这里有一点非常重要：由于Node使用了V8的原因，其实现很接近标准，另外，它还提供了一些标准之外实用的附加功能。换句话说，在Node中书写的JavaScript和浏览器上口碑不是很好的JavaScript有着重要的不同。

另外，Node中你书写的绝大多数JavaScript代码都符合Douglas Crockford在他那本著名的书《JavaScript语言精粹》中提到的 JavaScript语言的“精粹”。

本章分为以下两个部分：

- JavaScript基础。语言基础。适用于：Node、浏览器以及语言标准。
- V8中的JavaScript。V8提供的一些特性是浏览器不支持的，IE就更不用说了，因为这些特性是最近才纳入标准的。除此之外，V8还提供一些非标准的特性，它们能辅助解决一些常见的基本需求。

除此之外，下一章还会介绍Node中对语言的扩展和特性。

16

JavaScript基础

本章默认你对JavaScript及其语法有一定的了解。本章会介绍学习Node.js必须要掌握的JavaScript基础知识。

类型

JavaScript类型可以简单地分为两组：基本类型和复杂类型。访问基本类型，访问的是值，而访问复杂类型，访问的是对值的引用。

- 基本类型包括number、boolean、string、null以及undefined。
- 复杂类型包括array、function以及object。

如下述例子所示：

```
// 基本类型
var a = 5;
var b = a;
b = 6;
a; // 结果为5
b; // 结果为6

// 复杂类型
var a = ['hello', 'world'];
var b = a;
b[0] = 'bye';
a[0]; // 结果为“bye”
b[0]; // 结果为“bye”
```

上述例子中的第二部分，b和a包含了对值的相同引用。因此，当通过b修改数组的第一个元素时，a相应的值也更改了，也就说a[0] === b[0]。

类型的困惑

要在JavaScript中准确无误地判断变量值的类型并非易事。

因为对于绝大部分基本类型来说，JavaScript与其他面向对象语言一样有相应的构造器，比方说，你可以通过如下两种方式来创建一个字符串：

```
var a = 'woot';
var b = new String('woot');
a + b; // 'woot woot'
```

17

然而，要是对这两个变量使用typeof和instanceof操作符，事情就变得有意思了：

```
typeof a; // 'string'
typeof b; // 'object'
a instanceof String; // false
b instanceof String; // true
```

而事实上，这两个变量值绝对都是货真价实的字符串：

```
a.substr == b.substr; // true
```

并且使用==操作符判定时两者相等，而使用===操作符判定时并不相同：

```
a == b; // true
a === b; // false
```

考虑到有此差异，我建议你始终通过直观的方式进行定义，避免使用new。

有一点很重要，在条件表达式中，一些特定的值会被判定为false：null、undefined、' '，还有0：

```
var a = 0;
if (a) {
  // 这里始终不会被执行到
}
a == false; // true
a === false; // false
```

另外值得注意的是，typeof不会把null识别为类型为null：

```
typeof null == 'object'; // 很不幸，结果为true
```

数组也不例外，就算是通过[]这种方式定义数组也是如此：

```
typeof [] == 'object'; // true
```

这里要感谢V8给我们提供了判定是否为数组类型的方式，能够让我们免于使用hack的方式。

在浏览器环境中，我们通常要查看对象内部的[[Class]]值：Object.prototype.toString.call([]) == '[object Array]'。该值是不可变的，有利于我们在不同的上下文中（如浏览器窗口）对数组类型进行判定，而instanceof Array这种方式只适用于与数组初始化在相同上下文中才有效。

函数

18

在JavaScript中，函数最为重要。

它们都属于一等函数：可以作为引用存储在变量中，随后可以像其他对象一样，进行传递：

```
var a = function () {}
console.log(a); // 将函数作为参数传递
```

JavaScript中所有的函数都可以进行命名。有一点很重要，就是要能区分出函数名和变量名。

```
var a = function a () {
  'function' == typeof a; // true
};
```

THIS、FUNCTION#CALL以及FUNCTION#APPLY

下述代码中函数被调用时，this的值是全局对象。在浏览器中，就是window对象：

```
function a () {
  window == this; // true;
};

a();
```

调用以下函数时，使用.call和.apply方法可以改变this的值：

```
function a () {
  this.a == 'b'; // true
}

a.call({ a: 'b' });
```

call和apply的区别在于，call接受参数列表，而apply接受一个参数数组：

```
function a (b, c) {
  b == 'first'; // true
  c == 'second'; // true
}

a.call({ a: 'b' }, 'first', 'second')
a.apply({ a: 'b' }, ['first', 'second']);
```

19

函数的参数数量

函数有一个很有意思的属性——参数数量，该属性指明函数声明时可接收的参数数量。在JavaScript中，该属性名为length：

```
var a = function (a, b, c);
a.length == 3; // true
```

尽管这在浏览器端很少使用，但是，它对我们非常重要，因为一些流行的Node.js框架就是通过此属性来根据不同参数个数提供不同的功能的。

闭包

在JavaScript中，每次函数调用时，新的作用域就会产生。

在某个作用域中定义的变量只能在该作用域或其内部作用域（该作用域中定义的作用域）中才能访问到：

```
var a = 5;

function woot () {
  a == 5; // true

  var a = 6;

  function test () {
    a == 6; // true
  }

  test();
};

woot();
```

自执行函数是一种机制，通过这种机制声明和调用一个匿名函数，能够达到仅定义一个新作用域的作用。

```
var a = 3;

(function () {
  var a = 5;
})();

a == 3 // true;
```

自执行函数对声明私有变量是很有用的，这样可以让私有变量不被其他代码访问。

类

20

JavaScript中没有class关键词。类只能通过函数来定义：

```
function Animal () { }
```

要给所有Animal的实例定义函数，可以通过prototype属性来完成：

```
Animal.prototype.eat = function (food) {
  // eat method
}
```

这里值得一提的是，在prototype的函数内部，this并非像普通函数那样指向global对象，而是指向通过该类创建的实例对象：

```
function Animal (name) {
  this.name = name;
}

Animal.prototype.getName () {
```

```
  return this.name;
};

var animal = new Animal('tobi');
a.getName() == 'tobi'; // true
```

继承

JavaScript有基于原型的继承的特点。通常，你可以通过以下方式来模拟类继承。

定义一个要继承自Animal的构造器：

```
function Ferret () { };
```

要定义继承链，首先创建一个Animal对象，然后将其赋值给Ferret.prototype。

```
// 实现继承
Ferret.prototype = new Animal();
```

随后，可以为子类定义属性和方法：

```
// 为所有ferrets实例定义type属性
Ferret.prototype.type = 'domestic';
```

21

还可以通过prototype来重写和调用父类函数：

```
Ferret.prototype.eat = function (food) {
  Animal.prototype.eat.call(this, food);
  // ferret特有的逻辑写在这里
}
```

这项技术很赞。它是同类方案中最好的（相比其他函数式技巧），而且它不会破坏instanceof操作符的结果：

```
var animal = new Animal();
animal instanceof Animal // true
animal instanceof Ferret // false

var ferret = new Ferret();
ferret instanceof Animal // true
ferret instanceof Ferret // true
```

它最大的不足就是声明继承的时候创建的对象总要进行初始化（Ferret.prototype = new Animal），这种方式不好。一种解决该问题的方法就是在构造器中添加判断条件：

```
function Animal (a) {
  if (false !== a) return;
  // 初始化
}

Ferret.prototype = new Animal(false)
```

另外一个办法就是再定义一个新的空构造器，并重写它的原型：

```
function Animal () {
  // constructor stuff
}

function f () {};
f.prototype = Animal.prototype;
Ferret.prototype = new f;
```

幸运的是，V8提供了更简洁的解决方案，本章后续部分会做介绍。

TRY {} CATCH {}

try/catch允许进行异常捕获。下述代码会抛出异常：

```
> var a = 5;
> a()
TypeError: Property 'a' of object #<Object> is not a function
```

当函数抛出错误时，代码就停止执行了：

22

```
function () {
  throw new Error('hi');
  console.log('hi'); // 这里永远不会被执行到
}
```

若使用try/catch则可以进行错误处理，并让代码继续执行下去：

```
function () {
  var a = 5;
  try {
    a();
  } catch (e) {
    e instanceof Error; // true
  }

  console.log('you got here!');
}
```

v8中的JavaScript

至此，你已经了解了JavaScript在绝大多数环境下（包括早期浏览器中）的语言特性。

随着Chrome浏览器的发布，它带来了一个全新的JavaScript引擎——V8，它以极速的执行环境，加之时刻保持最新并支持最新ECMAScript特性的优势，快速地在浏览器市场中占据了重要的位置。

其中有些特性弥补了语言本身的不足。另外一部分特性的引入则要归功于像jQuery和PrototypeJS这样的前端类库，因为它们提供了非常实用的扩展和工具，如今，很难想象

JavaScript中要是没有了这些会是什么样子。

本章介绍V8中最有用的特性，并使用这些特性书写出更精准、更高效的代码，与此同时，代码的风格也将借鉴最流行的Node.js框架和库的代码风格。

OBJECT#KEYS

要想获取下述对象的键（a和c）：

```
var a = { a: 'b', c: 'd' };
```

通常会使用如下迭代的方式：

```
for (var i in a) { }
```

23 通过对键进行迭代，可以将它们收集到一个数组中。不过，如果采用如下方式对Object.prototype进行过扩展：

```
Object.prototype.c = 'd';
```

为了避免在迭代过程中把c也获取到，就需要使用hasOwnProperty来进行检查：

```
for (var i in a) {
  if (a.hasOwnProperty(i)) {}
}
```

在V8中，要获取对象上所有的自有键，还有更简单的方法：

```
var a = { a: 'b', c: 'd' };
Object.keys(a); // ['a', 'c']
```

ARRAY#ISARRAY

正如你此前看到的，对数组使用typeof操作符会返回object。然而大部分情况下，我们要检查数组是否真的是数组。

Array.isArray对数组返回true，对其他值则返回false：

```
Array.isArray(new Array) // true
Array.isArray([]) // true
Array.isArray(null) // false
Array.isArray(arguments) // false
```

数组方法

要遍历数组，可以使用forEach（类似jQuery的$.each）：

```
// 会打印出1, 2, 3
[1, 2, 3].forEach(function (v) {
  console.log(v);
});
```

要过滤数组元素，可以使用filter（类似jQuery的$.grep）：

```
[1, 2, 3].forEach(function (v) {
  return v < 3;
}); // 会返回[1,2]
```

要改变数组中每个元素的值，可以使用map（类似jQuery的$.map）：

```
[5, 10, 15].map(function (v) {
  return v * 2;
}); // 会返回 [10, 20, 30]
```

V8还提供了一些不太常用的方法，如reduce、reduceRight以及lastIndexOf。

24

字符串方法

要移除字符串首末的空格，可以使用：

```
'  hello  '.trim(); // 'hello'
```

JSON

V8提供了JSON.stringify和JSON.parse方法来对JSON数据进行解码和编码。

JSON是一种编码标准，和JavaScript对象字面量很相近，它用于大部分的Web服务和API服务：

```
var obj = JSON.parse('{"a":"b"}')
obj.a == 'b'; // true
```

FUNCTION#BIND

.bind（类似jQuery的$.proxy）允许改变对this的引用：

```
function a () {
  this.hello == 'world'; // true
};

var b = a.bind({ hello: 'world' });
b();
```

FUNCTION#NAME

V8还支持非标准的函数属性名：

```
var a = function woot () {};
a.name == 'woot'; // true
```

该属性用于V8内部的堆栈追踪。当有错误抛出时，V8会显示一个堆栈追踪的信息，会告诉你是哪个函数调用导致了错误的发生：

```
> var woot = function () { throw new Error(); };
```

```
> woot()
Error
    at [object Context]:1:32
```

25

在上述例子中，V8无法为函数引用指派名字。然而，如果对函数进行了命名，v8就能在显示堆栈追踪信息时将名字显示出来：

```
> var woot = function buggy () { throw new Error(); };
> woot()
Error
    at buggy ([object Context]:1:34)
```

为函数命名有助于调试，因此，推荐始终对函数进行命名。

PROTO（继承）

__proto__使得定义继承链变得更加容易：

```
function Animal () { }
function Ferret () { }
Ferret.prototype.__proto__ = Animal.prototype;
```

这是非常有用的特性，可以免去如下的工作：

- 像上一章节介绍的，借助中间构造器。
- 借助OOP的工具类库。无须再引入第三方模块来进行基于原型继承的声明。

存取器

你可以通过调用方法来定义属性，访问属性就使用__defineGetter__、设置属性就使用__defineSetter__。

比如，为Date对象定义一个称为ago的属性，返回以自然语言描述的日期间隔。

很多时候，特别是在软件中，想要用自然语言来描述日期距离某个特定时间点的时间间隔。比如，“某件事情发生在三秒钟前”这种表达，远要比“某件事情发生在×年×月×日”这种表达更容易理解。

下面的例子，为所有的Date实例都添加了ago获取器，它会返回以自然语言描述的日期距离现在的时间间隔。简单地访问该属性就会调用事先定义好的函数，无须显式调用。

```
// 基于John Resig的prettyDate（遵循MIT协议）
Date.prototype.__defineGetter__('ago', function () {
  var diff = (((new Date()).getTime() - this.getTime()) / 1000)
    , day_diff = Math.floor(diff / 86400);
```

26

```
  return day_diff == 0 && (
      diff < 60 && "just now" ||
      diff < 120 && "1 minute ago" ||
```

```
        diff < 3600 && Math.floor( diff / 60 ) + " minutes ago" ||
        diff < 7200 && "1 hour ago" ||
        diff < 86400 && Math.floor( diff / 3600 ) + " hours ago") ||
        day_diff == 1 && "Yesterday" ||
        day_diff < 7 && day_diff + " days ago" ||
        Math.ceil( day_diff / 7 ) + " weeks ago";
});
```

然后，简单地访问ago属性即可。注意，访问属性实际上还会调用定义的函数，只是这个过程透明了而已：

```
var a = new Date('12/12/1990'); // my birth date
a.ago // 1071 weeks ago
```

小结

理解掌握本章的内容对了解语言本身的不足以及大多数糟糕的JavaScript运行环境，如老版本的浏览器，至关重要。

由于JavaScript多年来自身发展缓慢且多少有种被人忽略的感觉，许多开发者投入了大量的时间来开发相应的技术以书写出更高效、可维护的JavaScript代码，同时也总结出了JavaScript一些诡异的工作方式。

V8做了一件很酷的事情，它始终坚定不移地实现最新版本的ECMA标准。Node.js的核心团队也是如此，只要你安装的是最新版本的Node，你总能使用到最新版本的V8。它开启了服务器端开发的新篇章，我们可以使用它提供的更易理解且执行效率更高的API。

希望通过本章的学习，你已掌握了Node开发者常用的一些特性，这些特性是诸多未来JavaScript拥有的特性中的一部分。