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

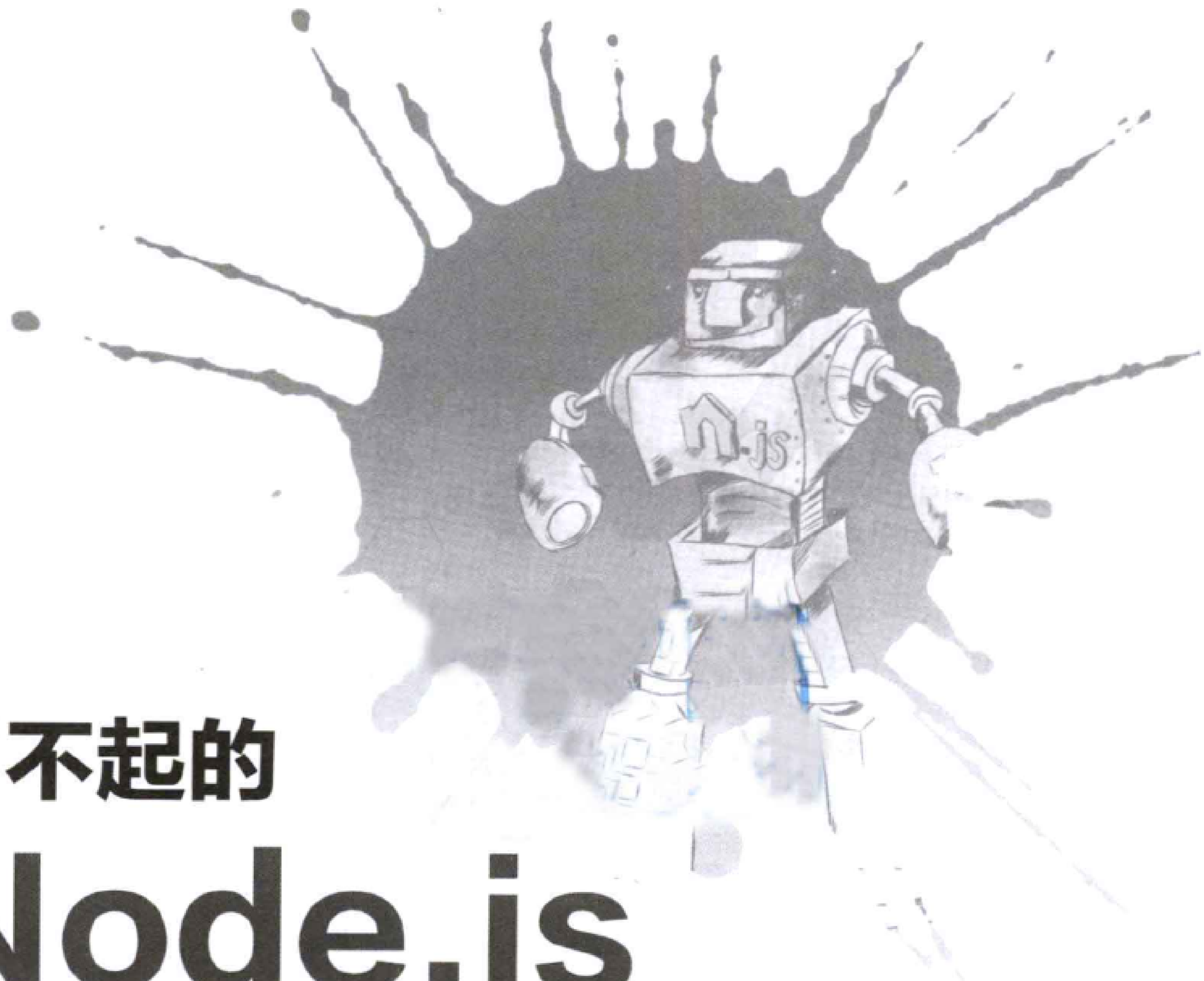
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

提供各种IT类书籍pdf下载，如有需要，请QQ:2404062482

注：链接至淘宝，不喜者勿入！整理那么多资料也不容易，请多多见谅！非诚勿扰！

更多此类书籍

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
北京•BEIJING

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。

SMASHING Node.js : JavaScript Everywhere, 978-1-119-96259-5, Guillermo Rauch.

©2012 Guillermo Rauch

All Rights Reserved. Authorised translation from the English language edition published by John Wiley and Sons Ltd. Responsibility for the accuracy of the translation rests solely with PHEI and is not the responsibility of John Wiley and Sons Ltd. No part of this book may be reproduced in any form without the written permission of the original copyright holder, John Wiley and Sons Ltd. Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

Trademarks: Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley and Sons, Inc. and/or its affiliates in the United States and/or other countries, and may not be used without written permission.

A catalogue record for this book is available from the British Library.

本书简体中文字版专有翻译出版权由John Wiley & Sons, Ltd.授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。本书封底贴有John Wiley & Sons, Inc. 防伪标签，无标签者不得销售。

版权贸易合同登记号　图字：01-2013-8005

图书在版编目（CIP）数据

了不起的Node.js：将JavaScript进行到底 / (美)劳奇 (Rauch,G.) 著；赵静译. —北京：电子工业出版社，2014.1
书名原文：SMASHING Node.js:JavaScript Everywhere
ISBN 978-7-121-21769-2

Ⅰ. ①了… Ⅱ. ①劳… ②赵… Ⅲ. ①JAVA语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字（2013）第257626号

策划编辑：张春雨
责任编辑：贾　莉
印　　刷：中国电影出版社印刷厂
装　　订：中国电影出版社印刷厂
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：19.5　字数：436千字
印　　次：2014年1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。

目录

PART Ⅰ 从安装与概念开始

CHAPTER 1 安装 3

在Windows下安装 3

在OS X下安装 4

在Linux下安装 5

编译 5

确保安装成功 5

Node REPL 5

执行文件 6

NPM 6

安装模块 7

自定义模块 8

安装二进制工具包 9

浏览NPM仓库 9

小结 10

CHAPTER 2 JavaScript概览 11

介绍 11

JavaScript基础 12

类型 12

类型的困惑 12

函数 13

THIS、FUNCTION#CALL以及FUNCTION#APPLY 14

函数的参数数量 14
闭包 14
类 15
继承 16
TRY {} CATCH {} 17
v8中的JavaScript 17
OBJECT#KEYS 18
ARRAY#ISARRAY 18
数组方法 18
字符串方法 19
JSON 19
FUNCTION#BIND 19
FUNCTION#NAME 19
PROTO（继承） 20
存取器 20
小结 21

CHAPTER 3 阻塞与非阻塞IO 23
能力越强，责任就越大 23
阻塞 25
单线程的世界 27
错误处理 29
堆栈追踪 30
小结 32

CHAPTER 4 Node中的JavaScript 33
global对象 33
实用的全局对象 34
模块系统 34
绝对和相对模块 35
暴露API 37
事件 38
buffer 40
小结 41

PART Ⅱ Node重要的API

CHAPTER 5 命令行工具（CLI）以及FS API：首个Node应用 45
需求 45
编写首个Node程序 46
创建模块 46
同步还是异步 47
理解什么是流（stream） 49
输入和输出 50
重构 53
用fs进行文件操作 55
对CLI一探究竟 56
argv 57
工作目录 57
环境变量 58
退出 58
信号 58
ANSI转义码 59
对fs一探究竟 59
Stream 59
监视 60
小结 61

CHAPTER 6 TCP 63
TCP有哪些特性 64
面向连接的通信和保证顺序的传递 64
面向字节 65
可靠性 65
流控制 65
拥堵控制 65
Telnet 65
基于TCP的聊天程序 68
创建模块 68
理解NET.SERVER API 68

接收连接 70
data事件 71
状态以及记录连接情况 73
圆满完成此程序 75
一个IRC客户端程序 77
创建模块 77
理解NET#STREAM API 78
实现部分IRC协议 78
测试实际的IRC服务器 78
小结 79

CHAPTER 7 HTTP 81

HTTP结构 81
头信息 82
连接 87
一个简单的Web服务器 88
创建模块 88
输出表单 88
method和URL 90
数据 92
整合 94
让程序更健壮 95
一个Twitter Web客户端 96
创建模块 96
发送一个简单的HTTP请求 97
发送数据 98
获取推文 99
superagent来拯救 102
使用up重启HTTP服务器 103
小结 104

PART Ⅲ Web开发

CHAPTER 8 Connect 107

使用HTTP构建一个简单的网站 108

通过Connect实现一个简单的网站 111
中间件 112
书写可重用的中间件 114
static中间件 119
query中间件 120
logger中间件 120
body parser中间件 122
cookie 125
会话（session） 126
Redis session 131
methodOverride中间件 132
basicAuth中间件 132
小结 134

CHAPTER 9 Express 135

一个小型Express应用 135
创建模块 136
HTML 136
SETUP 137
定义路由 137
查询 140
运行 141
设置 142
模板引擎 143
错误处理 144
快捷方法 144
路由 146
中间件 148
代码组织策略 149
小结 151

CHAPTER 10 WebSocket 153

Ajax 153
HTML5 WebSocket 156

一个ECHO示例 157
初始化项目 157
建立服务器 158
建立客户端 159
运行示例程序 160
鼠标光标 161
初始化示例程序 161
建立服务器 161
建立客户端 164
运行示例程序 166
面临一个挑战 166
关闭并不意味着断开连接 166
JSON 167
重连 167
广播 167
WebSocket属于HTML5：早期浏览器不支持 167
解决方案 167
小结 167

CHAPTER 11 Socket.IO 169
传输 170
断开 VS 关闭 170
事件 170
命名空间 171
聊天程序 172
初始化程序 172
构建服务器 172
构建客户端 173
事件和广播 175
消息接收确认 179
一个轮流做DJ的应用 180
扩展聊天应用 181
集成Grooveshark API 182
播放歌曲 185

小结 190

PART Ⅳ 数据库

CHAPTER 12 MongoDB 193
安装 195
使用MongoDB：一个用户认证的例子 195
构建应用程序 195
创建Express App 196
连接MongoDB 200
创建文档 201
查找文档 203
身份验证中间件 204
校验 205
原子性 206
安全模式 206
Mongoose介绍 206
定义模型 207
定义嵌套的键 208
定义嵌套文档 209
构建索引 209
中间件 209
探测模型状态 210
查询 210
扩展查询 210
排序 211
选择 211
限制 211
跳过 211
自动产生键 211
转换 212
一个使用Mongoose的例子 212
构建应用 212
重构 213

建立模型.....213
小结.....215

CHAPTER 13 MySQL.....217
node-mysql.....217
初始化项目.....217
Express应用.....218
连接MySQL.....219
初始化脚本.....220
创建数据.....224
获取数据.....228
sequelize.....229
初始化sequelize.....230
初始化Express应用.....230
连接sequelize.....233
定义模型和同步.....234
创建数据.....236
获取数据.....238
删除数据.....239
完整地完成应用.....240
小结.....241

CHAPTER 14 Redis.....243
安装Redis.....244
Redis查询语言.....245
数据类型.....245
字符串.....246
哈希.....246
列表.....248
数据集.....249
有序数据集.....249
Redis和Node.....249
使用node-redis实现一个社交图谱.....250
小结.....259

PART V 测试

CHAPTER 15 代码共享 263
什么样的代码可以共享 263
书写兼容的JavaScript代码 264
导出模块 264
模拟实现ECMA API 265
模拟实现Node API 267
模拟实现浏览器端API 267
跨浏览器的继承实现 268
集成到一起：browserbuild 268
基础案例 269
小结 271

CHAPTER 16 测试 273
简单测试 273
测试目标 274
测试策略 274
测试程序 275
expect.js 276
API一览 276
Mocha 278
测试异步代码 279
BDD风格 281
TDD风格 281
export风格 282
在浏览器端使用Mocha 282
小结 284

索引 285

PART

I

从安装与概念开始

CHAPTER 1　安装

CHAPTER 2　JavaScript概览

CHAPTER 3　阻塞与非阻塞IO

CHAPTER 4　Node中的JavaScript

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 7
变得非常简单。

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。

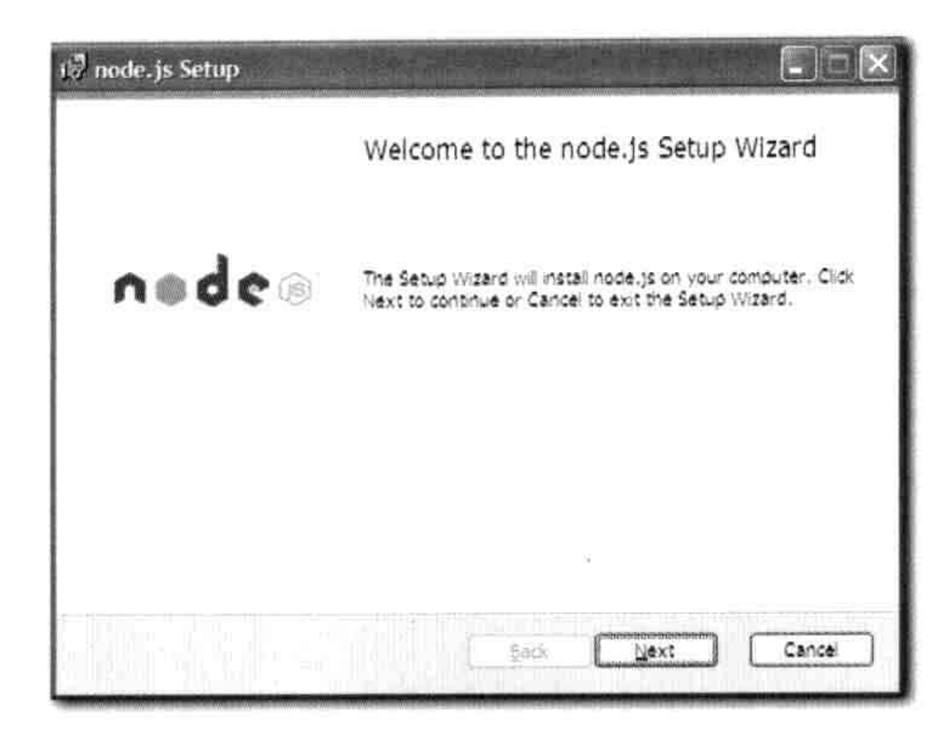


图1-1：Node.js安装指引

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。

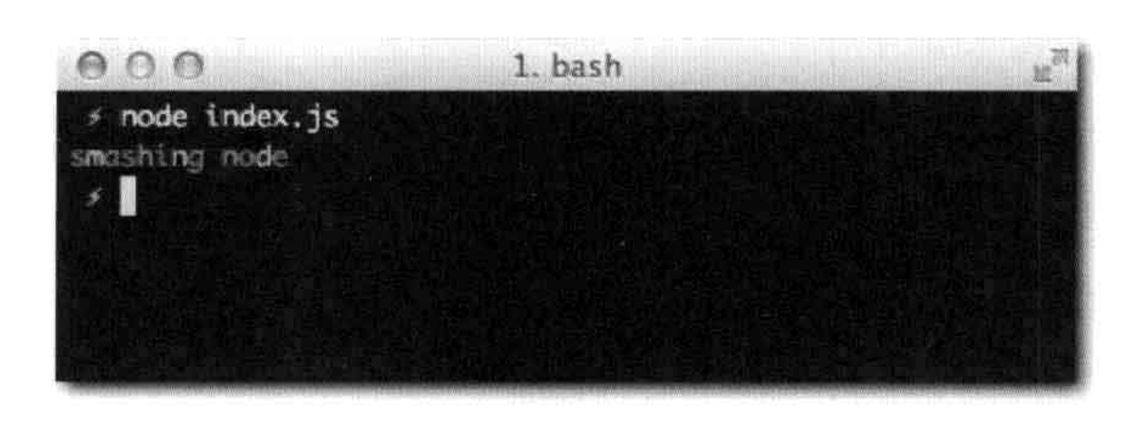


图1-4：模块安装成功验证结果

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。

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

26

```
        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。

CHAPTER

3 阻塞与非阻塞IO

绝大多数对Node.js的讨论都把关注点放在了其处理高并发的能力上。简单来说，相比其他同类解决方案，Node框架给开发者提供了构建高性能网络应用的强大能力，当然，开发者要明白Node内部所做出的权衡，以及用Node构建应用之所以性能好的原因。 27

能力越强，责任就越大 28

Node为JavaScript引入了一个复杂的概念，这在浏览器端从未有过：共享状态的并发。事实上，这种复杂度在像Apache与`mod_php`或者Nginx与FastCGI这样的Web应用开发模型下都从未有过。

通俗讲，Node中，你需要对回调函数如何修改当前内存中的变量（状态）特别小心。除此之外，你还要特别注意对错误的处理是否会潜在地修改这些状态，从而导致了整个进程不可用。

为了更好地掌握这个概念，我们来看如下的函数，该函数在每次请求`/books` URL的时候都会被执行。假设这里的“状态”就是存放图书的数组，该数组用来将图书列表以HTML的形式返回给客户端。

```
var books = [
    'Metamorphosis'
  , 'Crime and punishment'
];

function serveBooks () {
  // 给客户端返回HTML代码
  var html = '<b>' + books.join('</b><br><b>') + '</b>';

  // 恶魔出现了，把状态修改了！
  books = [];

  return html;
}
```

等价的PHP代码为：

```
$books = array(
    'Metamorphosis'
  , 'Crime and punishment'
);

function serveBooks () {
  $html = '<b>' . join($books, '</b><br><b>') . '</b>';
  $books = array();
  return $html;
}
```

注意，在上述两例serveBooks函数中，都将books数组重置了。

现在假设一个用户分别向Node服务器和PHP服务器各同时发起两次对/books的请求。试着预测下，两者结果会是如何：

29

- Node会将完整的图书列表返回给第一个请求，而第二个请求则返回一个空的图书列表。
- PHP都能将完整的图书列表返回给两个请求。

两者区别就在于基础架构上。Node采用一个长期运行的进程，相反，Apache会产出多个线程（每个请求一个线程），每次都会刷新状态。在PHP中，当解释器再次执行时，变量$books会被重新赋值，而 Node则不然，serveBooks函数会再次被调用，且作用域中的变量不受影响（此时$books数组仍为空）。

```
        +---------------------+
        |       APACHE        |
        +-+--------+--------+-+
          |        |        |
      +---+        |        +---+
+----+----+  +----+----+  +----+----+
|   PHP   |  |   PHP   |  |   PHP   |
|         |  |         |  |         |
| THREAD  |  | THREAD  |  | THREAD  |
+----+----+  +----+----+  +----+----+
     |            |            |
+---------+  +---------+  +---------+
| REQUEST |  | REQUEST |  | REQUEST |
+---------+  +---------+  +---------+
```

能力越强，责任也就越大。

```
+-----------------------------------+
|                                   |
|                                   |
|              NODE.JS              |
|                                   |
|              PROCESS              |
|                                   |
|                                   |
|                                   |
+----+-----------+-----------+----+
     |           |           |
+---------+  +---------+  +---------+
| REQUEST |  | REQUEST |  | REQUEST |
+---------+  +---------+  +---------+
```

始终牢记这点对书写出健壮的Node.js程序，避免运行时错误是非常重要的。

另外还有重要的一点是要弄清楚阻塞和非阻塞IO。

阻塞

尝试区分下面PHP代码和Node代码有什么不同：

```
// PHP
print('Hello');

sleep(5);

print('World');
```

30

Node代码示例：

```
// node
console.log('Hello');
```

```
setTimeout(function () {
  console.log('World');
}, 5000);
```

上述两段代码不仅仅是语义上的区别（Node.js使用了回调函数），两者区别集中体现在阻塞和非阻塞的区别上。在第一个例子中，PHP sleep()阻塞了线程的执行。当程序进入睡眠时，就什么事情都不做了。

而Node.js使用了事件轮询，因此这里setTimeout是非阻塞的。

换句话说，如果在setTimeout后再加入console.log语句的话，该语句会被立刻执行：

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

setTimeout(function () {
  console.log('World');
}, 5000);

console.log('Bye');

// 这段脚本会输出：
// Hello
// Bye
// World
```

采用了事件轮询意味着什么呢？从本质上来说，Node会先注册事件，随后不停地询问内核这些事件是否已经分发。当事件分发时，对应的回调函数就会被触发，然后继续执行下去。如果没有事件触发，则继续执行其他代码，直到有新事件时，再去执行对应的回调函数。

相反，在PHP中，sleep一旦执行，执行会被阻塞一段指定的时间，并且在阻塞时间未达到设定时间前，不会有任何操作，也就是说这是同步的。和阻塞相反，setTimeout仅仅只是注册了一个事件，而程序继续执行，所以，这是异步的。

Node并发实现也采用了事件轮询。与timeout所采用的技术一样，所有像http、net这样的原生模块中的IO部分也都采用了事件轮询技术。和timeout机制中Node内部会不停地等待，并当超时完成时，触发一个消息通知一样，Node使用事件轮询，触发一个和文件描述符相关的通知。

31

文件描述符是抽象的句柄，存有对打开的文件、socket、管道等的引用。本质上来说，当Node接收到从浏览器发来的HTTP请求时，底层的TCP连接会分配一个文件描述符。随后，如果客户端向服务器发送数据，Node就会收到该文件描述符上的通知，然后触发JavaScript的回调函数。

单线程的世界

有一点很重要，Node是单线程的。在没有第三方模块的帮助下是无法改变这一事实的。[1]

为了证明这一点，以及展示它和事件轮询之间的关系，我们来看如下例子：

```
var start = Date.now();

setTimeout(function () {
  console.log(Date.now() - start);

  for (var i = 0; i < 1000000000; i++){}
}, 1000);

setTimeout(function () {
  console.log(Date.now() - start);
}, 2000);
```

上述两段setTimeout代码，会打印出timeout设置与最终回调函数执行时，两者的时间差，以毫秒为单位。如图3-1所示是我电脑上打印出的结果。

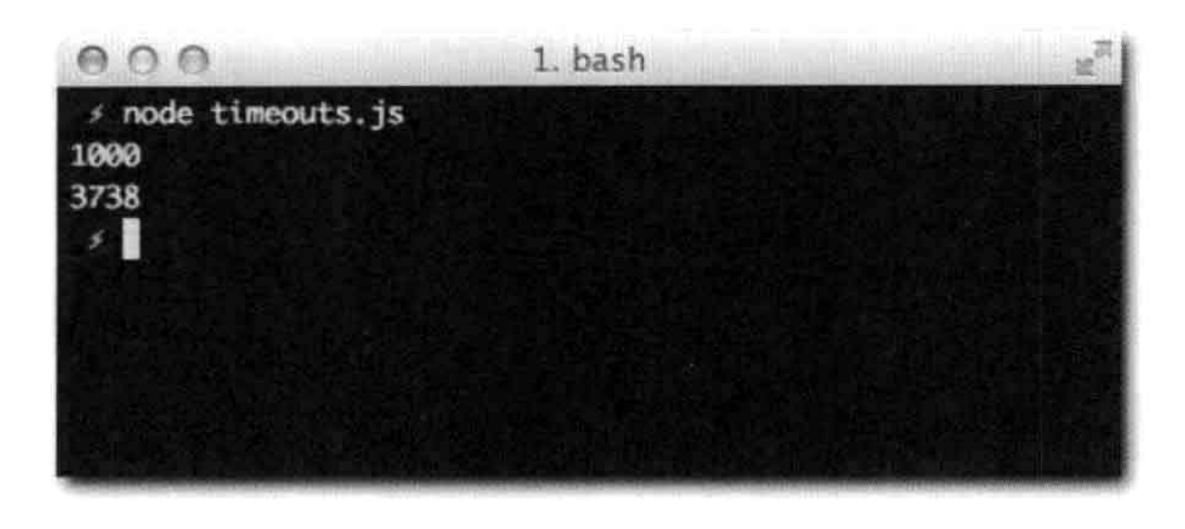


图3-1：程序显示了每个setTimeout执行的时间间隔，其结果和代码中设定的值并不相同

为什么会这样呢？究其原因，是事件轮询被JavaScript代码阻塞了。当第一个事件分发时，会执行JavaScript回调函数。由于回调函数需要执行很长的一段时间（循环次数很多），所以下一个事件轮询执行的时间就远远超过了2秒。因此，JavaScript的timeout并不能严格遵守时钟设置。

当然了，这样的行为方式并不理想。正如我此前介绍的，事件轮询是Node IO的基础核心。既然超时可以延迟，那HTTP请求以及其他形式的IO均可如此。也就意味着，HTTP服务器每秒处理的请求数量减少了，效率也就降低了。 32

正因如此，许多优秀的Node模块都是非阻塞的，执行任务也都采用了异步的方式。

既然执行时只有一个线程，也就是说，当一个函数执行时，同一时间不可能有第二个函数

1 译者注：Node早期版本的确不行，但是截止到本书翻译期间，Node 0.8.x和0.10.x都已经内置了child_process模块，允许创建子进程。

也在执行，那Node.js又是如何做到高并发的呢？比如，一台普通的笔记本电脑，用Node书写的简单的服务器就能够处理每秒上千个请求。

为了搞清楚这个问题，你首先要明白调用堆栈的概念。

当v8首次调用一个函数时，会创建一个众所周知的调用堆栈，或者称为执行堆栈。

如果该函数调用又去调用另外一个函数的话，v8就会把它添加到调用堆栈上。考虑如下例子：

```
function a () {
  b();
}
function b(){};
```

针对上述例子，调用堆栈是“a”后面跟着“b”。当“b”执行完，v8就不再执行任何代码了。

回到HTTP服务器的例子：

```
http.createServer(function () {
  a();
});
function a(){
  b();
};
function b(){};
```

在上述例子中，一旦HTTP请求到达服务器，Node就会分发一个通知。最终，回调函数会被执行，并且调用堆栈变为“a” > “b”。

由于Node是运行在单线程环境中，所以，当调用堆栈展开时，Node就无法处理其他的客户端或者HTTP请求了。

你也许在想，那照这样看来，Node的最大并发量不就是1了！是的。Node并不提供真正的并行操作，因为那样需要引入更多的并行执行线程。

33

关键在于，在调用堆栈执行非常快的情况下，同一时刻你无须处理多个请求。这也是为何说v8搭配非阻塞IO是最好的组合：v8执行JavaScript速度非常快，非阻塞IO确保了单线程执行时，不会因为有数据库访问或者硬盘访问等操作而导致被挂起。

一个真实世界的运用非阻塞IO的例子是云。在绝大多数如亚马逊云（“AWS”）这样的云部署系统中，操作系统都是虚拟出来的，硬件也是由租用者之间互相共享的（所以说你是在“租硬件”）。也就是说，假设硬盘正在为另外的租用者搜索文件，而你也要进行文件搜索，那么延迟就会变长。由于硬盘的IO效率是非常难预测的，所以，读文件时，如果把执行线程阻塞住，那么程序运行起来也会非常不稳定，而且很慢。

在我们的应用中，常见的IO例子就是从数据库中获取数据。假设我们需要为某个请求响应数据库获取的数据。

```
http.createServer(function (req, res) {
  database.getInformation(function (data) {
    res.writeHead(200);
    res.end(data);
  });
});
```

在上述例子中，当请求到达时，调用堆栈中只有数据库调用。由于调用是非阻塞的，当数据库IO完成时，就完全取决于事件轮询何时再初始化新的调用堆栈。不过，在告诉Node“当你获取数据库响应时记得通知我”之后，Node就可以继续处理其他事情了。也就是说，Node可以去处理更多的请求了！

接下来要介绍的，同时也是本书贯穿始终的一个话题，就是错误处理，这个话题和Node架构方式有着很大的关系。

错误处理

首先，很重要的一点，正如本章之前介绍的，Node应用依托在一个拥有大量共享状态的大进程中。

举例来说，在一个HTTP请求中，如果某个回调函数发生了错误，整个进程都会遭殃：

```
var http = require('http');

http.createServer(function () {
  throw new Error ( '错误不会被捕获' )
}).listen(3000)
```

因为错误未被捕获，若访问Web服务器，进程就会崩溃，如图3-2所示。 34

```
1. bash
⚡ node uncaught-http.js

/private/tmp/uncaught-http.js:4
  throw new Error('This will be uncaught')
        ^
Error: This will be uncaught
    at Server.<anonymous> (/private/tmp/uncaught-http.js:4:
9)
    at Server.emit (events.js:70:17)
    at HTTPParser.onIncoming (http.js:1514:12)
    at HTTPParser.onHeadersComplete (http.js:102:31)
    at Socket.ondata (http.js:1410:22)
    at TCP.onread (net.js:354:27)
⚡
```

图3-2：你能看到调用堆栈从事件轮询（IOWatcher）一路到回调函数

Node之所以这样处理是因为，在发生未被捕获的错误时，进程的状态就不确定了。之后就可能无法正常工作了，并且如果错误始终不处理的话，就会一直抛出意料之外的错误，这样很难调试。

如果添加了uncatchException处理器，就不一样了。这个时候，进程不会退出，并且之后的事情都在你的掌控中。

```
process.on('uncaughtException', function (err) {
  console.error(err);
  process.exit(1); // 手动退出
});
```

在上述例子中，行为方式和分发error事件的API行为方式一致。比如，考虑如下例子，创建一个TCP服务器，并用telnet工具发起连接：

```
var net = require('net');

net.createServer(function (connection) {
  connection.on('error', function (err) {
    // err是一个错误对象
  });
}).listen(400);
```

Node中，许多像http、net这样的原生模块都会分发error事件。如果该事件未处理，就会抛出未捕获的异常。

除了uncaughtException和error事件外，绝大部分Node异步API接收的回调函数，第一个参数都是错误对象或者是null：

35

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

fs.readFile('/etc/passwd', function (err, data) {
  if (err) return console.error(err);
  console.log(data);
});
```

错误处理中，每一步都很重要，因为它能让你书写更安全的程序，并且不丢失触发错误的上下文信息。

堆栈追踪

在JavaScript中，当错误发生时，在错误信息中可以看到一系列的函数调用，这称为堆栈追踪。看如下例子：

```
function c () {
  b();
};
```

```
function b () {
  a();
};

function a () {
  throw new Error('here');
};

c();
```

运行上述代码就能看到堆栈追踪信息，如图3-3所示。

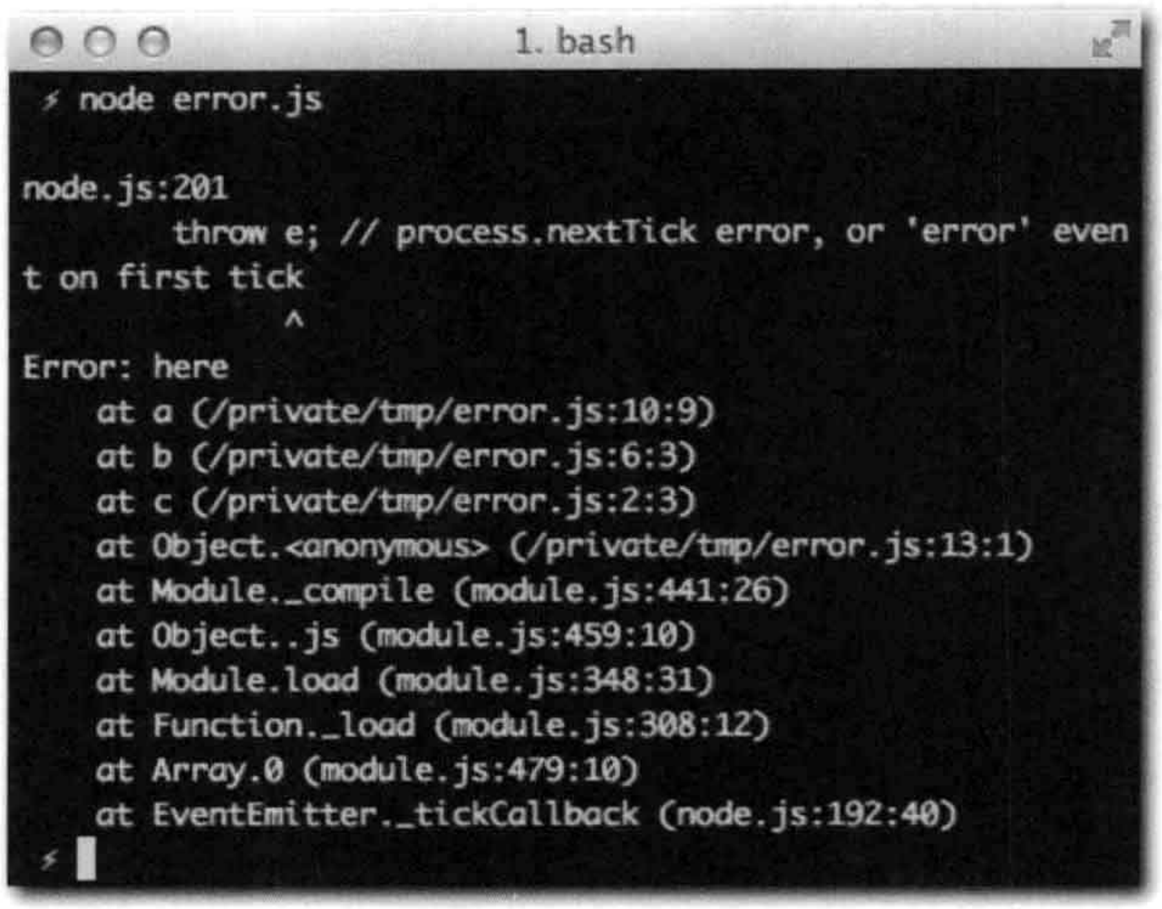


图3-3：针对上述代码，V8显示的堆栈追踪信息

在上图中，你能清晰地看到导致错误发生的函数调用路径。下面，我们来看一下引入事件轮询后会怎么样： 36

```
function c () {
  b();
};

function b () {
  a();
};

function a () {
  setTimeout(function () {
    throw new Error('here');
  }, 10);
};

c();
```

执行上述代码时（如图3-4所示），堆栈信息中有价值的信息就丢失了。

```
1. bash
⚡ node tim-error.js

timers.js:96
            if (!process.listeners('uncaughtException').len
gth) throw e;

           ^
Error: here
    at Object._onTimeout (/private/tmp/tim-error.js:11:11)
    at Timer.ontimeout (timers.js:94:19)
⚡
```

图3-4：堆栈信息显示的是从事件轮询开始的

同理，要捕获一个未来才会执行到的函数所抛出的错误是不可能的。这会直接抛出未捕获的异常，并且catch代码块永远都不会被执行：

```
try {
  setTimeout(function () {
    throw new Error('here');
  }, 10);
} catch (e) { }
```

这就是为什么在Node.js中，每步都要正确进行错误处理的原因了。一旦遗漏，你就会发现发生了错误后很难追踪，因为上下文信息都丢失了。

37 注意，有一点很重要，将来Node会让异步处理器抛出的异常更容易被追踪到。

小结

至此，你已经明白了事件轮询、非阻塞IO以及V8是如何互相配合为开发者提供书写高性能网络应用的能力。

相信你还学到了，Node通过单线程的执行环境，提供了极大的简便，不过，也正因如此，当你书写网络应用时，要尽可能地避免使用同步IO。除此之外，相信你也明白了，该线程中所有的状态都是维护在一个内存空间中的，换句话说，写程序的时候要格外小心。

相信你也清楚地看到了，非阻塞IO和回调引入了新的调试和错误处理的方式，这种方式与写阻塞式IO的程序时是截然不同的。

CHAPTER

4 Node中的JavaScript

在Node.js中写JavaScript和在浏览器中写JavaScript截然不同。Node.js除了提供和浏览器一样的基本语言之外，还在语言基础上提供了构建强大网络应用所需的API。 39

通过本章的介绍，你会看到一些API，这些API是Node和浏览器都有的，但却是语言本身之外、不在标准中定义的。而更重要的是，如本章标题——“Node中的JavaScript”所暗示的，本章会介绍Node.js的核心扩展相关内容。

首先我们来看看global对象的区别。

global对象

40

在浏览器中，全局对象指的就是`window`对象。在`window`对象上定义的任何内容都可以被全局访问到。比如，`setTimeout`其实就是`window.setTimeout`，`document`其实就是`window.document`。

Node中有两个类似但却各自代表着不同含义的对象，如下所示。

- `global`：和`window`一样，任何`global`对象上的属性都可以被全局访问到。

- process：所有全局执行上下文中的内容都在process对象中。在浏览器中，只有一个window对象，在Node中，也只有一个process对象。举例来说，在浏览器中窗口的名字是window.name，类似的，在Node中进程的名字是process.title。

下一章内容会深入介绍process对象，因为它提供了丰富有趣的功能，尤其是对于命令行程序来说。

实用的全局对象

浏览器中有些函数和工具虽然并非语言标准的一部分，但却非常实用，如今，它们已经被人们看作是JavaScript的一部分了。它们都是以全局的形式暴露出来的。

举例来说，setTimeout并非ECMAScript的一部分，但浏览器却仍将其视作重要的特性来实现。事实上，该函数是无法通过纯JavaScript重写的，不信的话你可以去试试。

另外有些API，人们还在讨论是否要加入到语言规范中（处在建议阶段），不过，Node.js为了让编写Node应用效率更高就把它们加进来了。setImmediate API就是一个例子，在Node中，它的作用和process.nextTick相当。

process.nextTick函数可以将一个函数的执行时间规划到下一个事件循环中：

```
console.log(1);
process.nextTick(function () {
  console.log(3);
});
console.log(2);
```

把它想象成是setTimeout(fn,1)或者“通过异步的方法在最近的将来调用该函数”，你就很容易能理解为什么上述例子的输出结果是1,2,3了。

41

还有一个类似的例子是console，console最早由Firefox中辅助开发的插件——Firebug实现。最后，Node也引入了一个全局console对象，该对象有一些如console.log和console.error这样的很有用的方法。

模块系统

JavaScript原生态是一个全局的世界。所有如setTimeout、document等这样在浏览器端使用的API，都是全局定义的。

当你引入第三方模块时，最好它们也暴露一个（或者多个）全局变量。比如，当你在HTML文档中引入<script src="http://code. jquery.com/jquery-1.6.0.js">后，你可以通过该模块上的jQuery对象来使用：

```
<script>
  jQuery(function () {
    alert('hello world!');
  });
</script>
```

之所以这样的根本原因是，JavaScript语言标准中并未为模块依赖以及模块独立定义专门的API。因此，就导致了通过这种方式引入的多个模块会出现对全局命名空间的污染及命名冲突的问题。

Node内置了很多实用的模块作为基础的工具集来帮助构建现代应用，包括http、net、fs，等等。此前在第1章“安装”中也介绍过，通过NPM你可以安装更多的模块。

Node摒弃了采用定义一堆全局变量（或者跑很多可能根本就不会用到的代码）的方式，转而引入了一个简单但却强大无比的模块系统，该模块系统有三个核心的全局对象：require、module和exports。

绝对和相对模块

这里，绝对模块是指Node通过在其内部node_modules查找到的模块，或者Node内置的如fs这样的模块。

正如在第1章中介绍的，当你安装好了colors模块，其路径就变成了./node_modules/colors。

这个时候，你就可以直接通过名字来require这个模块，无须添加路径名：

```
require('colors')
```

colors模块修改了String.prototype，因此，它无须暴露API。而fs模块，则暴露了一系列函数： 42

```
var fs = require('fs');
fs.readFile('/some/file', function (err, contents) {
  if (!err) console.log(contents);
});
```

模块还可以使用模块系统的功能来提供更加简洁独立的API以及抽象。然而，不一定非要将模块或者应用每一个部分都作为一个单独的模块和各自单独的package.json文件，你可以使用我所说的相对模块。

相对模块将require指向一个相对工作目录中的JavaScript文件。为了证明这一点，我们在同一目录中创建名为module_a.js、module_b.js以及main.js的三个文件。

module_a.js

```
console.log('this is a');
```

module_b.js

```
console.log('this is b');
```

main.js

```
require('module_a');
require('module_b');
```

然后，运行main文件（见图4-1）：

```
$ node main
```

如图4-1所示，Node未能找到module_a和module_b。原因就在于它们并没有通过NPM来安装，也不在node_modules目录中，而且Node自带模块中没有以此为名的模块。

43

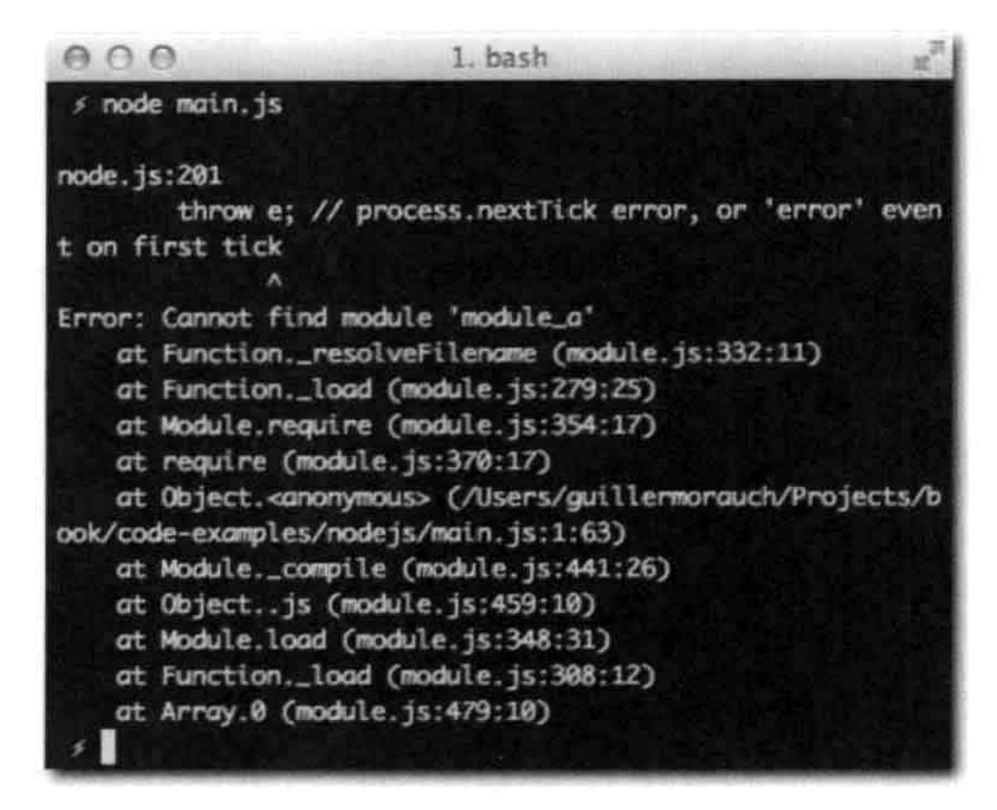


图4-1：加载module_a模块时，出错告知未能找到

要修复上述例子中这个问题，需要在require参数前加上./：

main.js

```
require('./module_a')
require('./module_b')
```

现在再次运行main文件（见图4-2）。

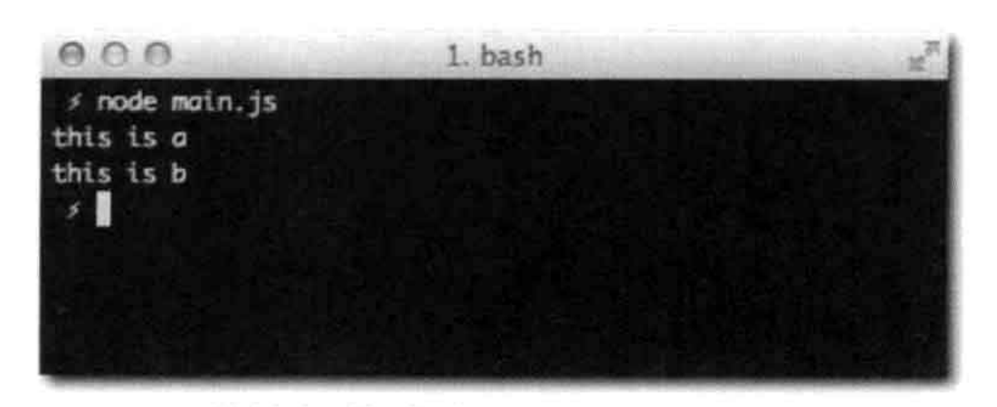


图4-2：模块加载成功

成功！这两个模块都执行了。接下来，我要介绍如何让这些模块暴露API，从而当调用require时，可以将其赋值给一个变量。

暴露API

44

要让模块暴露一个API成为require调用的返回值，就要依靠module和exports这两个全局变量。

在默认情况下，每个模块都会暴露出一个空对象。如果你想要在该对象上添加属性，那么简单地使用exports即可：

module_a.js

```
exports.name = 'john';
exports.data = 'this is some data';

var privateVariable = 5;

exports.getPrivate = function () {
  return privateVariable;
};
```

我们来测试下（见图4-3）：

index.js

```
var a = require('./module_a');
console.log(a.name);
console.log(a.data);
console.log(a.getPrivate());
```

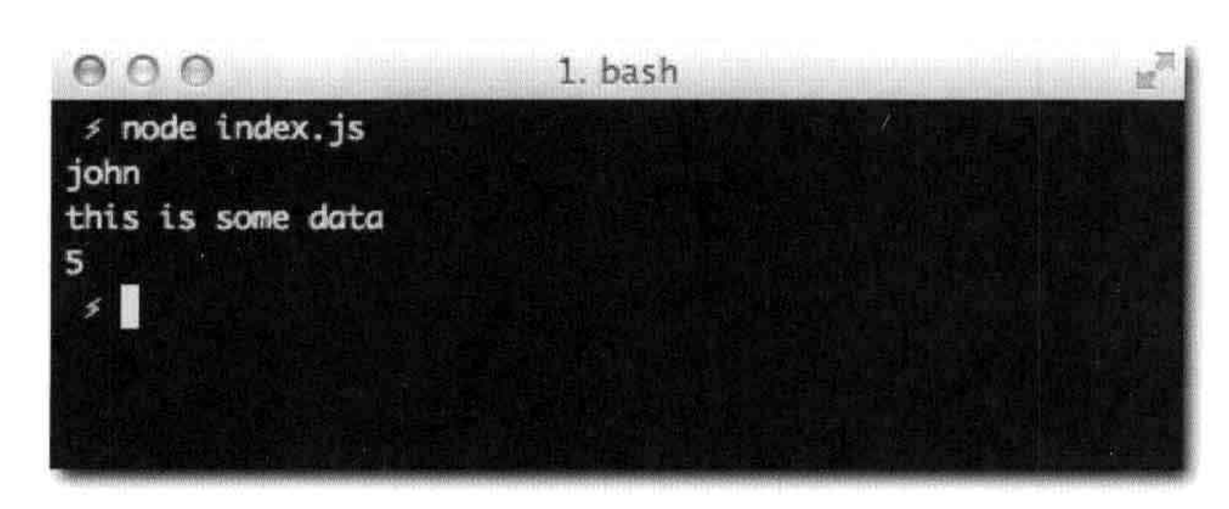


图4-3：查看module_a暴露出来的API

在上述例子中，exports其实就是对module.exports的引用，其在默认情况下是一个对象。要是在该对象上逐个添加属性无法满足你的需求，你还可以彻底重写module.exports。下面就是一个常见的将模块中构造器暴露出来的例子（见图4-4）：

45

person.js

```
module.exports = Person;

function Person (name) {
  this.name = name;
};

Person.prototype.talk = function () {
  console.log ( '我的名字是' , this.name ) ;
};
```

index.js

```
var Person = require('./person');
var john = new Person('john');
john.talk();
```

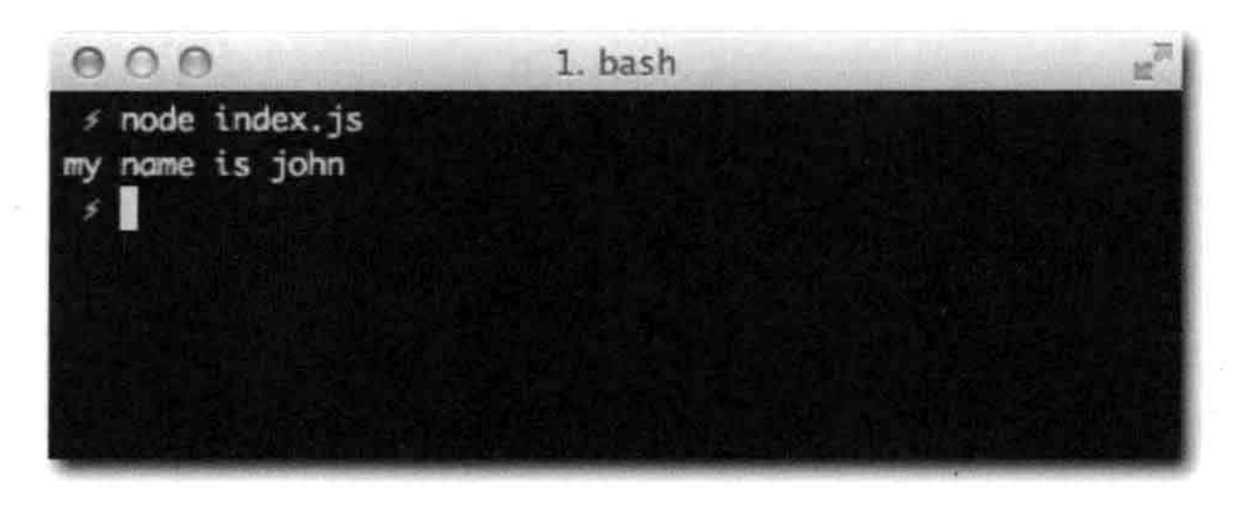


图4-4：一个具备JavaScript OOP风格的Node.js模块的例子

如上述例子所述，在index文件中，你不再是接收一个对象作为返回值，而是函数，这得归功于对module.exports的重写。

事件

Node.js中的基础API之一就是EventEmitter。无论是在Node中还是在浏览器中，大量代码都依赖于所监听或者分发的事件：

```
window.addEventListener('load', function () {
  alert ('窗口已加载！')  ;
});
```

浏览器中负责处理事件相关的DOM API主要包括addEventListener、removeEventListener以及dispatchEvent。它们还用在一系列从window到XMLHTTPRequest等的其他对象上。

46

下述例子发起一个Ajax请求（现代浏览器中），并通过监听stateChange事件来获知数据何时到达：

```
var ajax = new XMLHTTPRequest
ajax.addEventListener('stateChange', function () {
  if (ajax.readyState == 4 && ajax.responseText) {
    alert('we got some data: ' + ajax.responseText);
  }
});
ajax.open('GET', '/my-page');
ajax.send(null);
```

在Node中，你也希望可以随处进行事件的监听和分发。为此，Node暴露了Event EmitterAPI，该API上定义了on、emit以及removeListener方法。它以process.EventEmitter形式暴露出来：

eventemitter/index.js

```
var EventEmitter = require('events').EventEmitter
  , a = new EventEmitter;
a.on('event', function () {
  console.log('event called');
});
a.emit('event');
```

这个API相比DOM中的更简洁，Node内部在使用，你也可以很容易地将其添加到自己的类中：

```
var EventEmitter = process.EventEmitter
  , MyClass = function (){};

MyClass.prototype.__proto__ = EventEmitter.prototype;
```

这样，所有MyClass的实例都具备了事件功能：

```
var a = new MyClass;
a.on('某一事件', function () {
  // 做些什么
});
```

事件是Node非阻塞设计的重要体现。Node通常不会直接返回数据（因为这样可能会在等待某个资源的时候发生线程阻塞），而是采用分发事件来传递数据的方式。

我们再以HTTP服务器为例。当请求到达时，Node会调用一个回调函数，这个时候数据可能不会一下子都到达。POST请求（用户提交一个表单）就是这样的例子。

当用户提交表单时，你通常会监听请求的data和end事件：

```
http.Server(function (req, res) {
  var buf = '';
  req.on('data', function (data) {
    buf += data;
```

```
  });
  req.on('end', function () {
    console.log('数据接收完毕！');
  });
});
```

这是Node.js中很常见的例子：将请求数据内容进行缓冲（data事件），等到所有数据都接收完毕后（end事件）再对数据进行处理。

不管是否“所有的数据”都已到达，Node为了让你能够尽快知道请求已经到达服务器，都需要分发事件出来。在Node中，事件机制就是一个很好的机制，能够通知你尚未发生但即将要发生的事情。

事件是否会触发取决于实现它的API。比如，你知道了ServerRequest继承自EventEmitter，现在你也知道了它会分发data和end事件。

有些API会分发error事件，该事件也许根本不会发生。有些事件只会触发一次（如end事件），而有些则会触发多次（如data事件）。有些API只会在特定情况下触发某种事件。又比如，在特定的事件发生后，某些事件就不再触发。在上述HTTP的例子中，你肯定不希望在end事件触发后还触发data事件，否则，你的应用就会发生故障了。

同样的，有的时候，会有这样的需求：不管某个事件在将来会被触发多少次，我都希望只调用一次回调函数。Node为这类需求提供了一个名字简洁的方法：

```
a.once('某个事件' function () {
  //尽管事件会触发多次，但此方法只会执行一次
});
```

通常，要弄明白哪些事件是可用的，以及它们的“联系方式”（即触发它们的条件），需要查看模块的API文档。本书中会介绍核心Node模块的API以及一些重要的事件，不过，带上API手册会是个不错的习惯，帮助也会很大。

buffer

除了模块之外，Node还弥补了语言另外一个不足之处，那就是对二进制数据的处理。

48 buffer是一个表示固定内存分配的全局对象（也就是说，要放到缓冲区中的字节数需要提前定下），它就好比是一个由八位字节元素组成的数组，可以有效地在JavaScript中表示二进制数据。

该功能一部分作用就是可以对数据进行编码转换。比如，你可以创建一幅用base64表示的图片，然后将其作为二进制PNG图片的形式写入到文件中：

buffers/index.js

```
var mybuffer = new Buffer('==ii1j2i3h1i23h', 'base64'):
console.log(mybuffer);
require('fs').writeFile('logo.png', mybufffer);
```

base64主要是一种仅用ASCII字符书写二进制数据的方式。换句话说，它可以让你用简单的英文字符来表示像图片这样的复杂事物（所以会占用更多的硬盘空间）。

在Node.js中，绝大部分进行数据IO操作的API都用buffer来接收和返回数据。在上述例子中，filesystem模块中的`writeFile` API就接收buffer作为参数，并将其写到`logo.gif`文件中。

运行该代码，并打开gif图片（见图4-5）：

```
$ node index
$ open logo.png
```

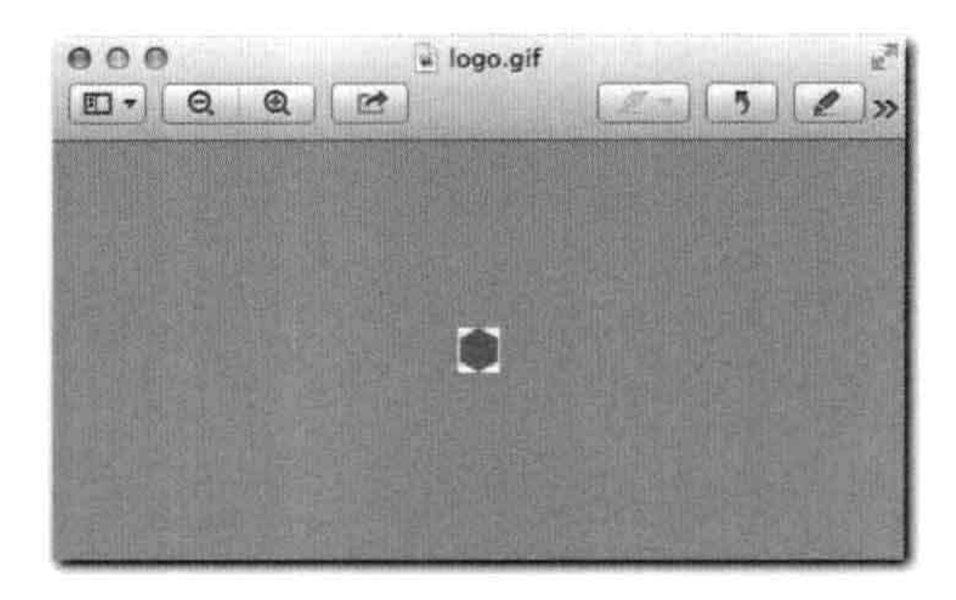


图4-5：上述脚本中，从用base64表示的buffer中创建的GIF图片显示了Node.js的logo

如上图所示，对于要调用`console.log`方法输出buffer对象而言，`writeFile`的确是个简单的接口，让原生字节数据生成图片。

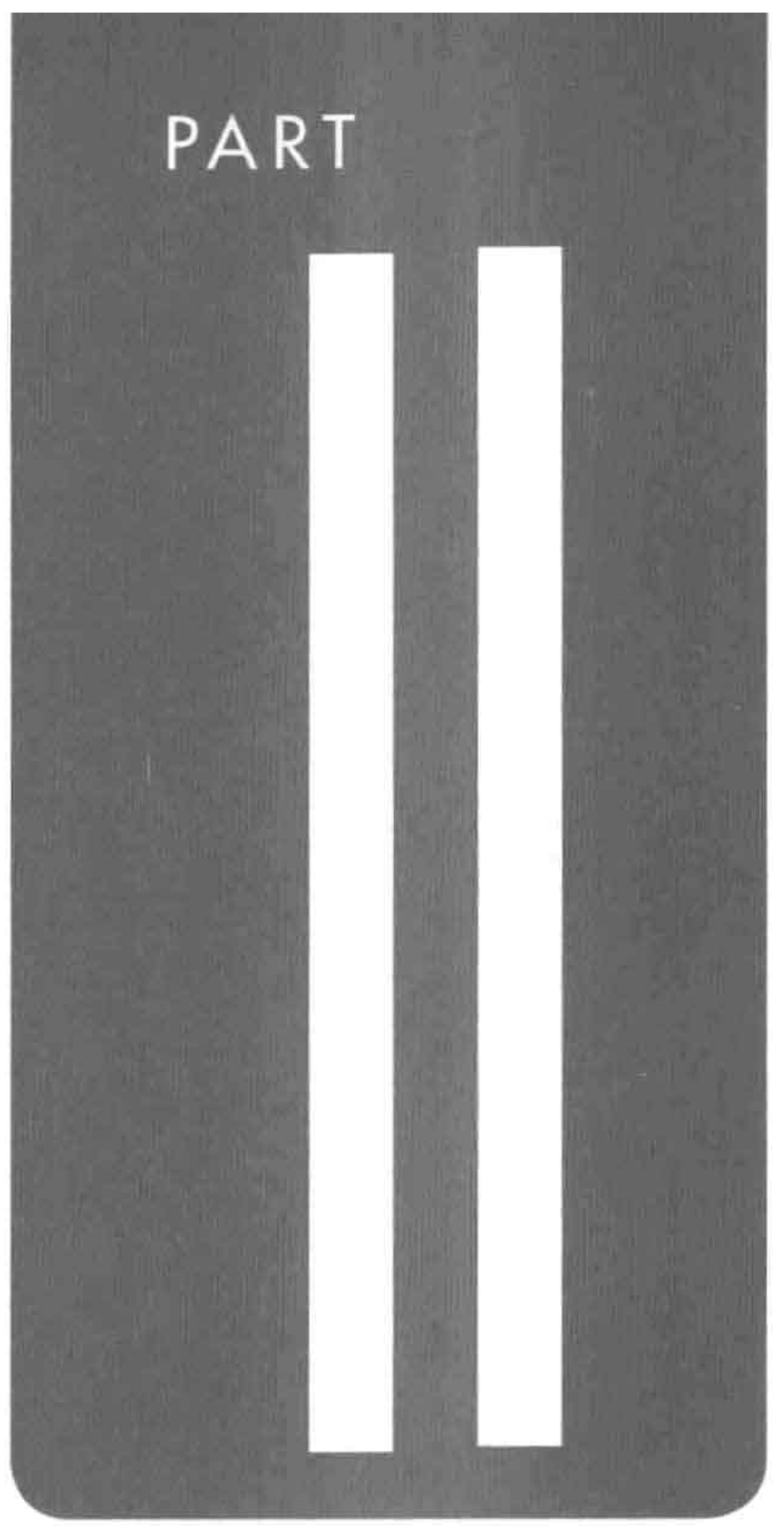
小结

通过本章，你应该了解到了在浏览器端和Node.js中书写JavaScript的主要区别。

你还应该了解了Node添加的但在语言标准中没有的JavaScript中的常用API，如定时器、事件、二进制数据以及模块。

你也应该知道了Node中也有类似window的对象，也可以使用如console这样的开发者工具。

PART



Node重要的API

CHAPTER 5 命令行工具（CLI）以及FS API：首个Node应用

CHAPTER 6 TCP

CHAPTER 7 HTTP

CHAPTER

5 命令行工具（CLI）以及FS API：首个Node应用

本章将介绍使用Node.js中一些重量级API：处理进程（stdio）的stdin以及stdout相关的API，还有那些与文件系统（fs）相关的 API。 51

第4章中我们介绍过，Node通过使用回调和事件机制来实现并发。这些API会让你首次接触到基于非阻塞事件的I/O编程中的流控制。

除了介绍如何使用这些API之外，你还可以通过本章来构建首个应用：一个简单的命令行文件浏览器，其功能是允许用户读取和创建文件。

需求

52

我们从定义需求开始：

- 程序需要在命令行运行。这就意味着程序要么通过node命令来执行，要么直接执行，然后通过终端提供交互给用户进行输入、输出。
- 程序启动后，需要显示当前目录下列表（见图5-1）。
- 选择某个文件时，程序需要显示该文件内容。

- 选择一个目录时，程序需要显示该目录下的信息。
- 运行结束后程序退出。

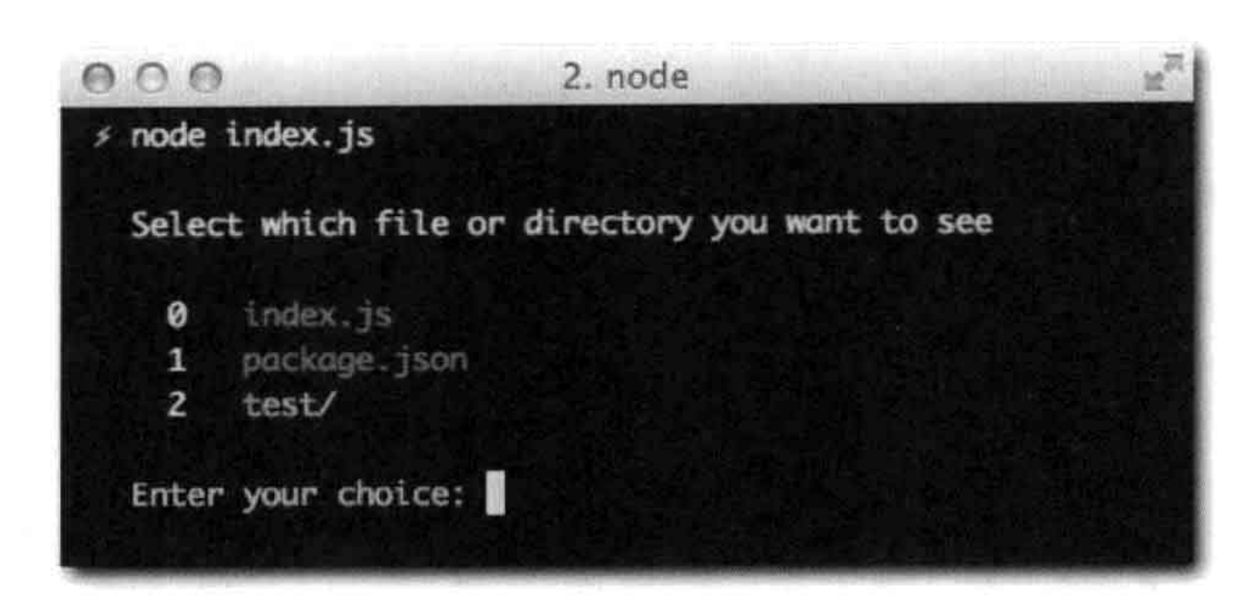


图5-1：程序启动后显示的目录列表

根据上述需求，你可以将此项目细分到如下几个步骤：

1. 创建模块。
2. 决定采用同步的fs还是异步的fs。
3. 理解什么是流（Stream）。
4. 实现输入输出。
5. 重构。
6. 使用fs进行文件交互。
7. 完成。

编写首个Node程序

现在我们开始基于上述步骤来编写一个模块。模块由几个文件组成，编写时可以使用任意文本编辑器。

通过本章，我们会完成一个具备完整功能的纯Node.js应用。

53 创建模块

和本书其他例子一样，我们从创建一个项目目录开始。按照此项目的需求，我们将该目录命名为file-explorer。

正如其他章节所述，在项目中定义package.json文件始终是最佳实践。这样，既可以方便地对NPM中注册的模块依赖进行管理，将来也能对模块进行发布。

尽管此项目仅仅用到Node.js的核心模块API（因此，不会从NPM仓库中获取模块），但

是，我们还是需要创建一个简单的package.json文件：

```
# package.json
{
    "name": "file-explorer"
  , "version": "0.0.1"
  , "description": "A command-file file explorer!"
}
```

注意：NPM遵循一个名为semver[1]的版本控制标准。这就是为何不使用“0.1”或者“1”作为版本号，而是用“0.0.1”的原因。

要验证你的package.json文件是否有效，可以运行$ npm install。

要是没有问题，就不会有任何输出内容，否则会抛出JSON异常的错误（见图5-2）。

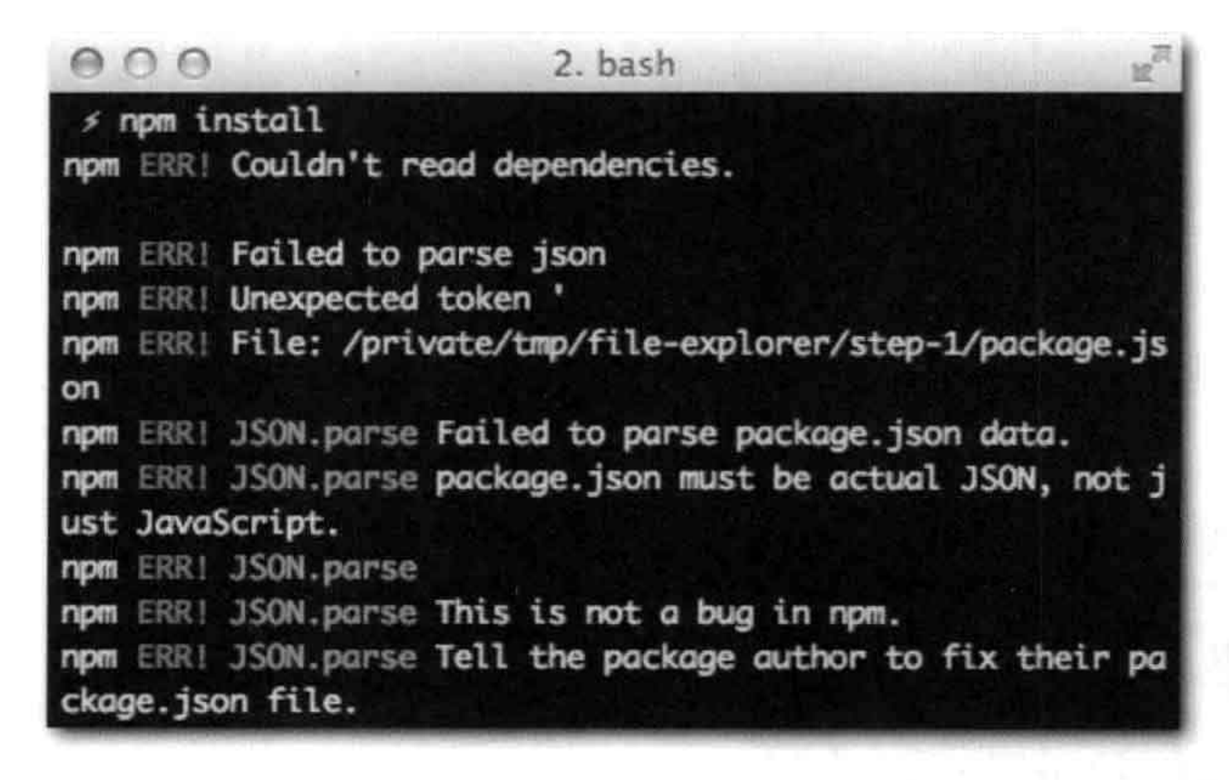


图5-2：package.json文件中有JSON错误的情况下运行npm install命令

接着，我们要创建一个简单的包含程序完整功能的JavaScript文件：index.js。

同步还是异步

54

我们从声明依赖关系开始。由于stdio API是全局process对象的一部分，所以，我们的程序唯一的依赖就是fs模块：

```
# index.js
/**
 * Module dependencies.
 */

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

我们首先要做的就是获取当前目录的文件列表。

1 译者注：http://semver.org/。

有件重要的事情要记住：fs模块是唯一一个同时提供同步和异步API的模块。举个例子来说，要想获取当前目录的文件列表，可以使用如下方式：

```
> console.log(require('fs').readdirSync(__dirname));
```

它会立刻返回内容或者当有错误发生时抛出相应异常（见图5-3）。

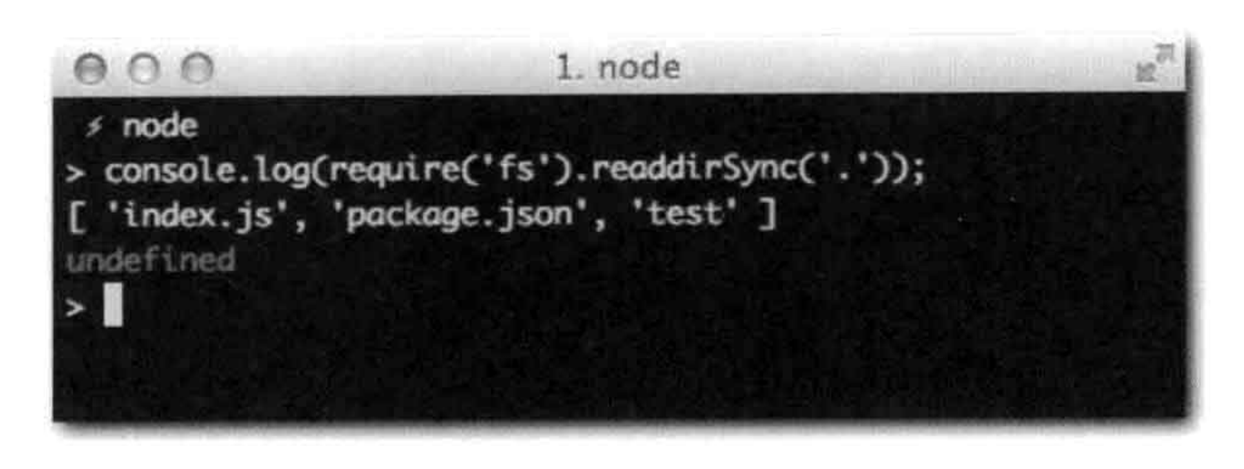


图5-3：查看readdirSync的返回值

下面是异步的版本：

```
> function async (err, files) { console.log(files); };
> require('fs').readdir('.', async);
```

如图5-4所示，上述两个例子结果是一样的。

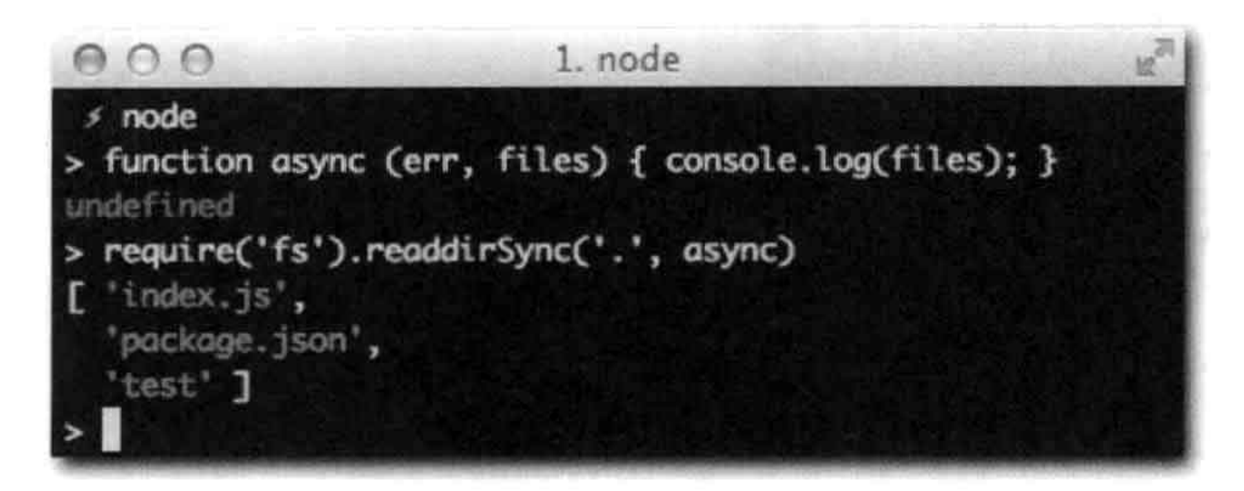


图5-4：异步版本的readdir

55 第3章中提过，要在单线程中创建能够处理高并发的高效程序，就得采用异步、事件驱动的程序。

尽管本章中这个命令行程序并非此类型程序（因为同一时间只会有一个人在读取文件），但是，为了学习Node.js中最重要也是最具挑战的部分，我们还是保持这种异步的代码风格。

为了获取文件列表，我们需要使用fs.readdir。我们提供的回调函数首个参数是一个错误对象（如果没有错误发生，该对象为null），另外一个参数是一个files数组：

```
# index.js
// . . .
fs.readdir(__dirname, function (err, files) {
  console.log(files);
});
```

运行上述代码，会得到如图5-5所示的结果。

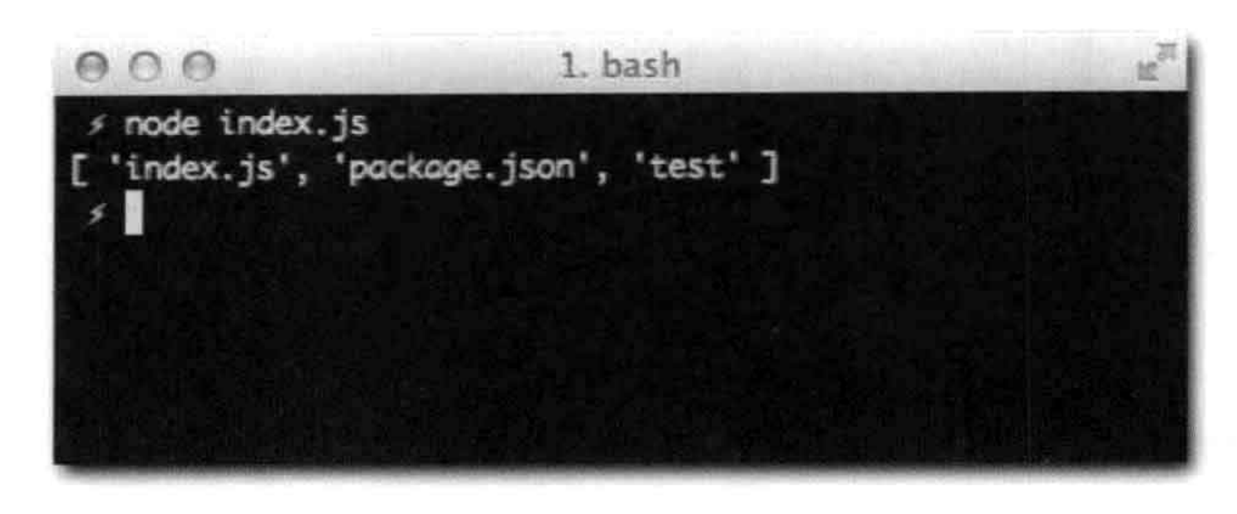


图5-5：运行自己的包含在index.js文件中的Node程序

好了，至此，你已经知道了fs模块同时提供了同步和异步的API来操作文件系统，接下来你需要学习Node.js中一个基础概念——流。

理解什么是流（stream）

我们已经知道，console.log会输出到控制台。事实上，console.log内部做了这样的事情：它在指定的字符串后加上\n（换行）字符，并将其写到stdout流中。

观察图5-6中显示的两个程序的不同点。

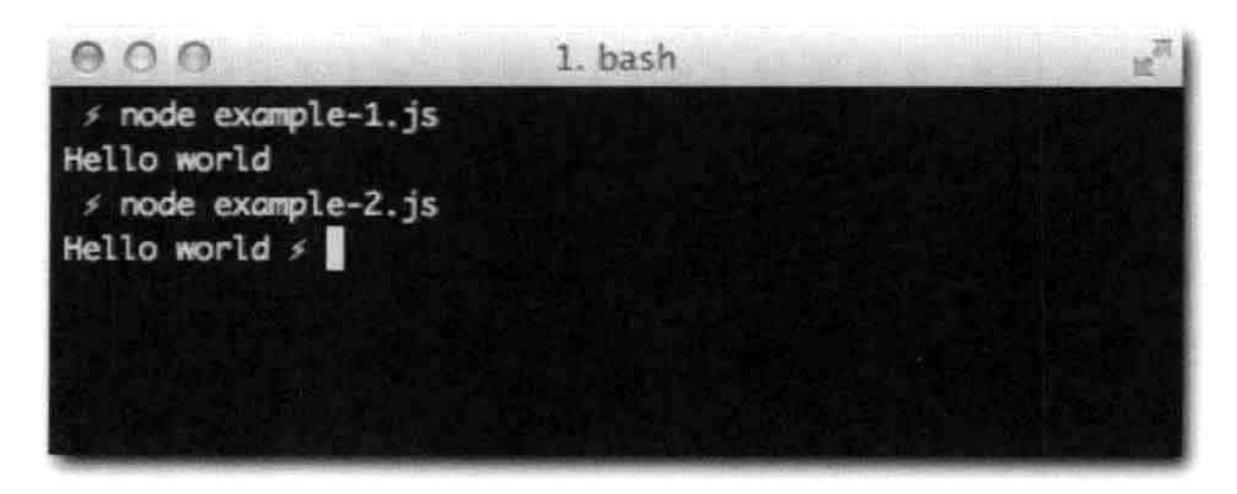


图5-6：第一个Hello World的程序加了一个换行符，第二个则没加

我们再来看下面的源代码：

```
# example-1.js
console.log('Hello world');
```

和：

```
# example-2.js
process.stdout.write('Hello world');
```

process全局对象中包含了三个流对象，分别对应三个UNIX标准流：

```
- **stdin**:  标准输入
- **stdout**: 标准输出
- **stderr**: 标准错误
```

图5-7描述了这三个流对象。

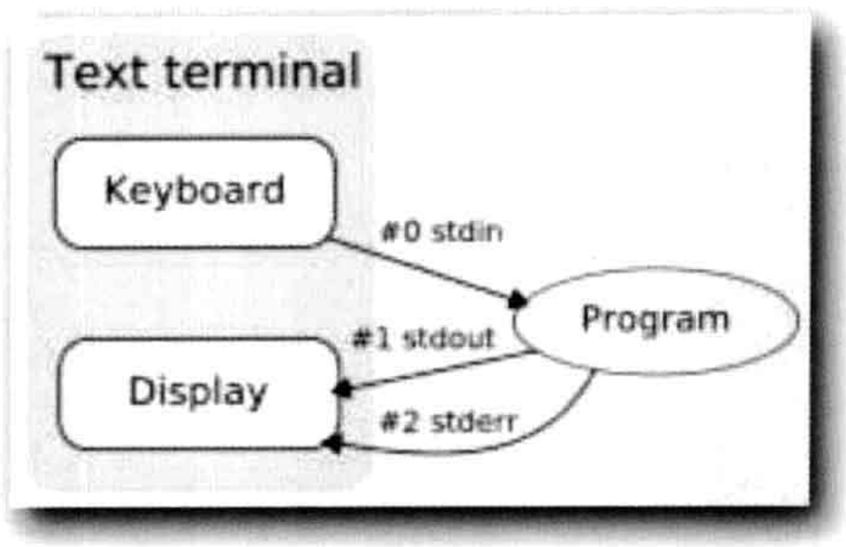


图5-7：传统文本终端下的stdio、stdout以及stderr对象

第一个stdin是一个可读流，而stdout和stderr都是可写流。

stdin流默认的状态是暂停的（paused）。通常，执行一个程序，程序会做一些处理，然后退出。不过，有些时候，就像本章中的这个应用一样，程序需要一直处在运行状态来接收用户输入的数据。

当恢复那个流时，Node会观察对应的文件描述符（在UNIX下为0），随后保持事件循环的运行，同时保持程序不退出，等待事件的触发。除非有IO等待，否则Node.js总是会自动退出。

57 流的另外一个属性是它默认的编码。如果在流上设置了编码，那么会得到编码后的字符串（utf-8、ascii等）而不是原始的Buffer作为事件参数。

Steam对象和EventEmitter很像（事实上，前者继承自后者）。在Node中，你会接触到各种类型流，如TCP套接字、HTTP请求等。简而言之，当涉及持续不断地对数据进行读写时，流就出现了。

输入和输出

既然已经知道运行程序后大概是怎样的一个情形了，我们来尝试写第一部分，列出当前目录下的文件，然后等待用户输入：

```
# index.js
// . . .
fs.readdir(process.cwd(), function (err, files) {
  console.log('');

  if (!files.length) {
    return console.log('    \033[31m No files to show!\033[39m\n');
  }

  console.log('   Select which file or directory you want to see\n');
```

```
function file(i) {
  var filename = files[i];

  fs.stat(__dirname + '/' + filename, function (err, stat) {
    if (stat.isDirectory()) {
      console.log('     '+i+'   \033[36m' + filename + '/\033[39m');
    } else {
      console.log('     '+i+'   \033[90m' + filename + '\033[39m');
    }

    i++;
    if (i == files.length) {
      console.log('');
      process.stdout.write('   \033[33mEnter your choice: \033[39m');
      process.stdin.resume();
    } else {
      file(i);
    }
  });
}

file(0);
});
```

下面，我们来逐行分析上述代码。 58

为了输出更加友好，我们首先输出一个空行：

```
console.log('')
```

如果files数组为空，告知用户当前目录没有文件。文本周围的\033[31m和\033[39m是为了让文本呈现为红色。例子中最后一个字符又是换行符\n，也是为了输出更友好。

```
if (!files.length) {
  return console.log('    \033[31m No files to show!\033[39m\n');
}
```

下一行很简单，一眼就看明白了：

```
console.log('   Select which file or directory you want to see\n');
```

紧接着，定义了一个函数，数组中每个元素都会执行该函数。这里也出现了贯穿本书始终的第一种异步流控制模式：串行执行。本章最后会对此做详细介绍。

```
function file (i) {
  // . . .
}
```

然后，先获取文件名，再查看文件名对应路径的情况。fs.stat会给出文件或者目录的

元数据：

```
var filename = files[i];

fs.stat(__dirname + '/' + filename, function (err, stat) {
  // . . .
});
```

回调函数中，同时还给出了错误对象（如果有的话）和一个Stat对象。本例中所使用到的Stat对象上的方法是isDirectory：

```
if (stat.isDirectory()) {
  console.log('     '+i+'   \033[36m' + filename + '/\033[39m');
} else {
  console.log('     '+i+'   \033[90m' + filename + '\033[39m');
}
```

如果路径所代表的是目录，我们就用有别于文件的颜色标识出来。

接下来就到了流控制中的核心部分了。计数器不断递增，与此同时，检查是否还有未处理的文件：

59

```
i++;
if (i == files.length) {
  console.log('');
  process.stdout.write('   \033[33mEnter your choice: \033[39m');
  process.stdin.resume();
  process.stdin.setEncoding('utf8');
} else {
  file(i);
}
```

如果所有文件都处理完毕，此时提示用户进行选择。注意，这里使用的是process.stdout.write而不是console.log；这样就无须换行，让用户可以直接在提示语后进行输入（见图5-8）：

```
console.log('');
process.stdout.write('   \033[33mEnter your choice: \033[39m');
```

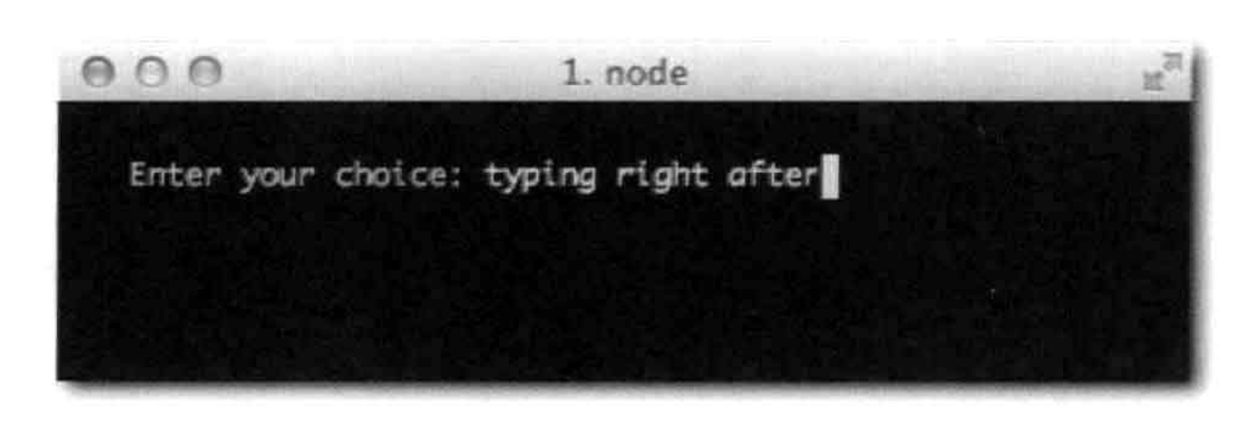


图5-8：程序提示用户向stdin进行输入

下面这行代码，此前介绍过，是用户等待用户输入：

```
process.stdin.resume();
```

紧跟着这行代码的是设置流编码为utf8，这样就能支持特殊字符了：

```
process.stdin.setEncoding('utf8');
```

要是还有未处理的文件，则递归调用该函数来进行处理：

```
file(i);
```

直到列出所有文件、用户输入完毕后，紧接着进行下一步串行处理。这是本章介绍的首个重要的模式。

重构

要做重构，我们从为几个常用的变量（如stdin和stdout）创建快捷变量开始：

```
# index.js
// . . .
var fs = require('fs')
  , stdin = process.stdin
  , stdout = process.stdout
```

60

由于我们书写的代码都是异步的，因此，会有这样的问题：随着函数量的增长（特别是流控制层的增加），过多的函数嵌套会让程序的可读性变差。

为了避免此类问题，我们可以为每一个异步操作预先定义一个函数。

首先，我们抽离出一个读取stdin函数：

```
# index.js
// called for each file walked in the directory
function file(i) {
  var filename = files[i];

  fs.stat(__dirname + '/' + filename, function (err, stat) {
    if (stat.isDirectory()) {
      console.log('     '+i+'   \033[36m' + filename + '/\033[39m');
    } else {
      console.log('     '+i+'   \033[90m' + filename + '\033[39m');
    }

    if (++i == files.length) {
      read();
    } else {
      file(i);
    }
  });
}

// read user input when files are shown
function read () {
  console.log('');
  stdout.write('   \033[33mEnter your choice: \033[39m');
```

```
  stdin.resume();
  stdin.setEncoding('utf8');
}
```

注意，上述代码所使用的是新的stdin和stdout引用。

读取用户输入后，接下来要做的就是根据用户输入做出相应处理。用户需要选择要读取的文件，所以，代码层面，我们设置了stdin的编码后，就开始监听其data事件：

61

```
function read () {
  // . . .
  stdin.on('data', option);
}

// called with the option supplied by the user
function option (data) {
  if (!files[Number(data)]) {
    stdout.write('   \033[31mEnter your choice: \033[39m');
  } else {
    stdin.pause();
  }
}
```

这里，我们检查用户的输入是否匹配files数组中的下标。还记得files数组是fs.readdir回调函数中的一部分吧。另外，要注意的是，上述代码中，我们将utf-8编码的字符串类型data转化为Number类型来方便做检查。

如果检查通过，我们要确保再次将流暂停（回到默认状态），以便于之后做完fs操作后，程序能够顺利退出（见图5-9）。

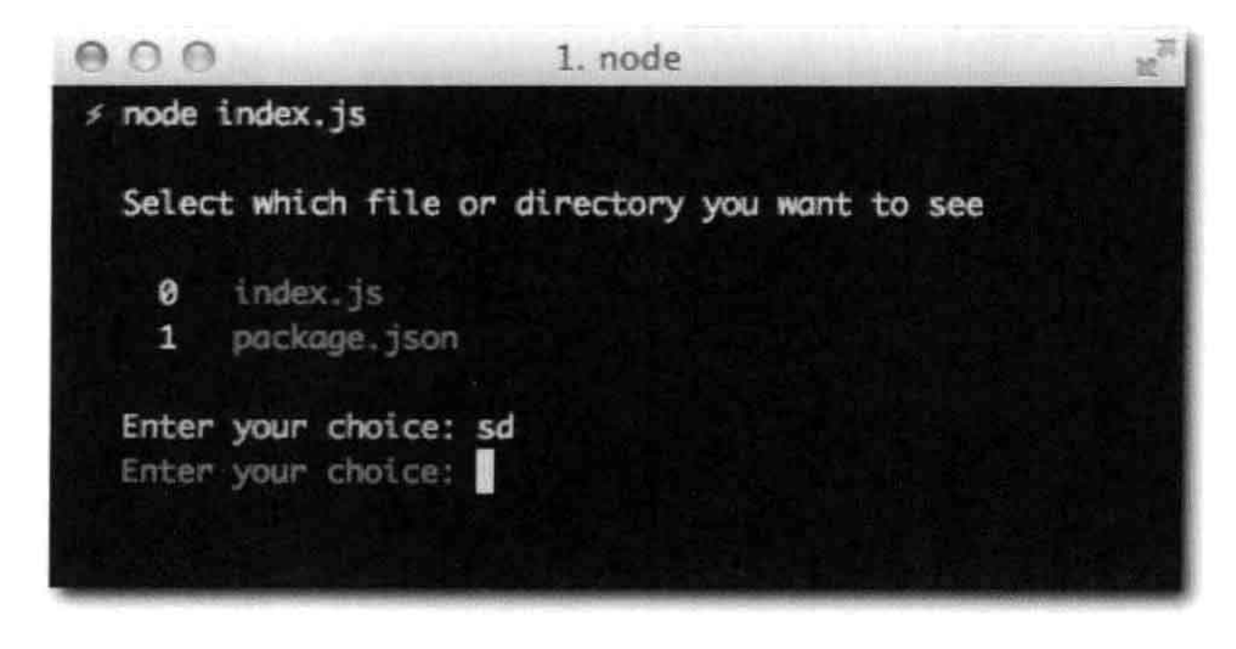


图5-9：用户选择错误的例子

至此，我们的程序已经能够与用户进行交互了，将当前目录中的文件列表展现给用户，下面我们来实现读取和显示文件内容。

用fs进行文件操作

既然都能定位到文件了，那是时候去读取它了！

```
function option (data) {
  var filename = files[Number(data)];
  if (!filename) {
    stdout.write('   \033[31mEnter your choice: \033[39m');
  } else {
    stdin.pause();
    fs.readFile(__dirname + '/' + filename, 'utf8', function (err, data) {

      console.log('');
      console.log('\033[90m' + data.replace(/(.*)/g, '     $1') + '\033[39m');
    });
  }
}
```

62

再次提醒，我们可以事先指定编码，这样我们得到的数据就是相应的字符串了：

```
fs.readFile(__dirname + '/' + filename, 'utf8', function (err, data) {
```

接着，可以使用正则表达式添加一些辅助缩进后将文件内容进行输出（见图5-10）：

```
data.replace(/(.*)/g, '     $1')
```

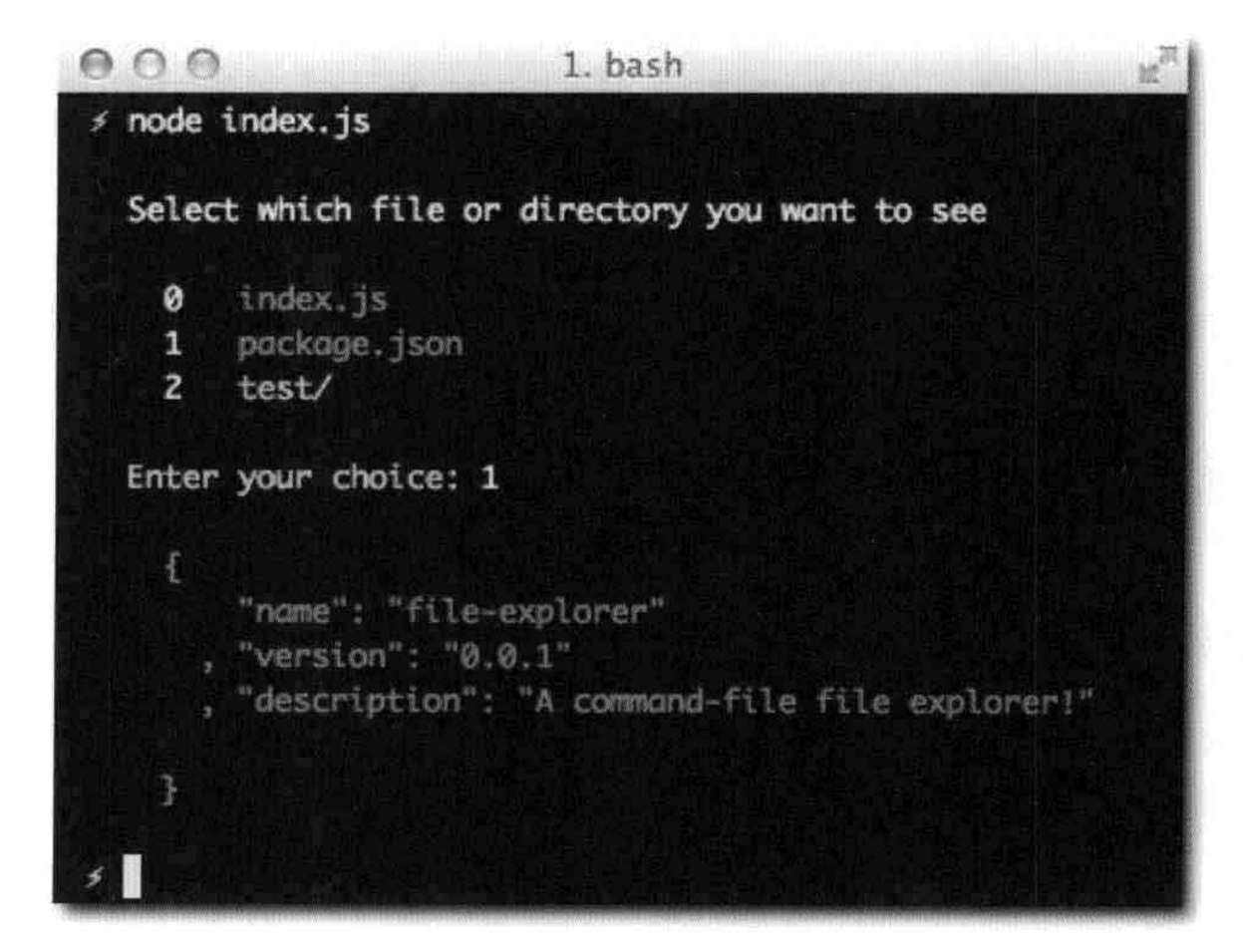


图5-10：读取简单文件的例子

不过，要是选择的是目录呢？这种情况下，我们就应当将其目录下的文件列表显示出来。为了避免再次执行fs.stat，我们在file函数中，将Stat对象保存下来：

```
// . . .
var stats = [];
```

```
function file(i) {
  var filename = files[i];

  fs.stat(__dirname + '/' + filename, function (err, stat) {
    stats[i] = stat;
    // . . .
```

63

好了，现在可以轻松地在option函数中进行检查操作了。下述代码对应原先执行fs.readFile的代码位置：

```
if (stats[Number(data)].isDirectory()) {
  fs.readdir(__dirname + '/' + filename, function (err, files) {
    console.log('');
    console.log('    (' + files.length + ' files)');
    files.forEach(function (file) {
      console.log('      -   ' + file);
    });
    console.log('');
  });
} else {
  fs.readFile(__dirname + '/' + filename, 'utf8', function (err, data) {
    console.log('');
    console.log('\033[90m' + data.replace(/(.*)/g, '      $1') + '\033[39m');
  });
}
```

现在再运行该程序，就能够进行目录选择，并能看到该目录下的文件列表信息了（见图5-11）。

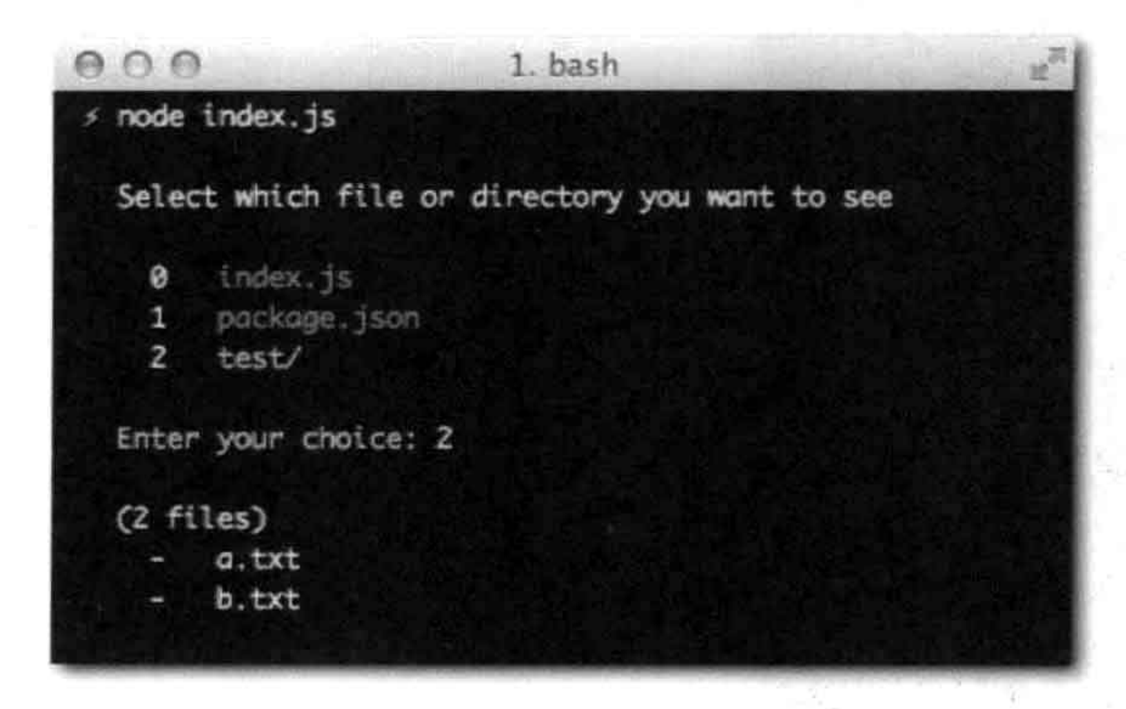


图5-11：读取test/文件夹的例子

完成！恭喜你完成了首个Node命令行（CLI）程序。

对CLI一探究竟

完成了首个命令行程序之后，有必要学习一些API，它们对于书写在终端运行的类似程序很有帮助。

argv

process.argv包含了所有Node程序运行时的参数值：

```
# example.js
console.log(process.argv);
```

64

如图5-12所示，第一个元素始终是node、第二个元素始终是执行的文件路径。紧接着是命令行后紧跟着的参数。

图5-12：展示process.argv内容的例子

要获取这些真正的元素，需要首先将数组的前两个元素去除掉（见图 5-13）：

```
# example-2.js
console.log(process.argv.slice(2));
```

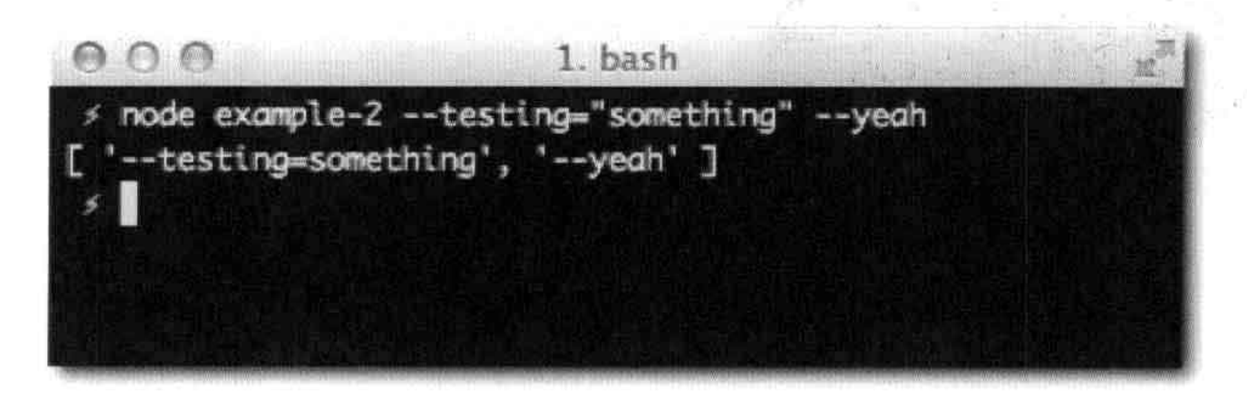


图5-13：去除argv前两个元素，显示真正元素的例子

接下来，你需要学会如何获取两个不同的目录：一个是程序本身所在的目录，另外一个是程序运行时的目录。

工作目录

在此前的例子中，我们使用__dirname来获取执行文件时该文件在文件系统中所在的目录。

不过，有的时候，更希望获得程序运行时的当前工作目录。以此前例子而言，如果在home目录下运行该程序，获得的当前工作目录和在其他目录下运行是一样的，因为index.js文件的路径始终没变，因此__dirname也不会变。

要获取当前工作目录，可以调用process.cwd方法：

```
> process.cwd()
/Users/guillermo
```

65 Node还提供了process.chdir方法，允许灵活地更改工作目录：

```
> process.cwd()
/Users/guillermo
> process.chdir('/')
> process.cwd()
/
```

还有另外一个和程序运行上下文有关的方面就是环境变量。请接着往下看。

环境变量

Node允许通过process.env变量来轻松访问shell环境下的变量。

举例来说，一个最常见的环境变量就是NODE_ENV（见图5-14），该变量用来控制程序是运行在开发模式下还是产品模式下。

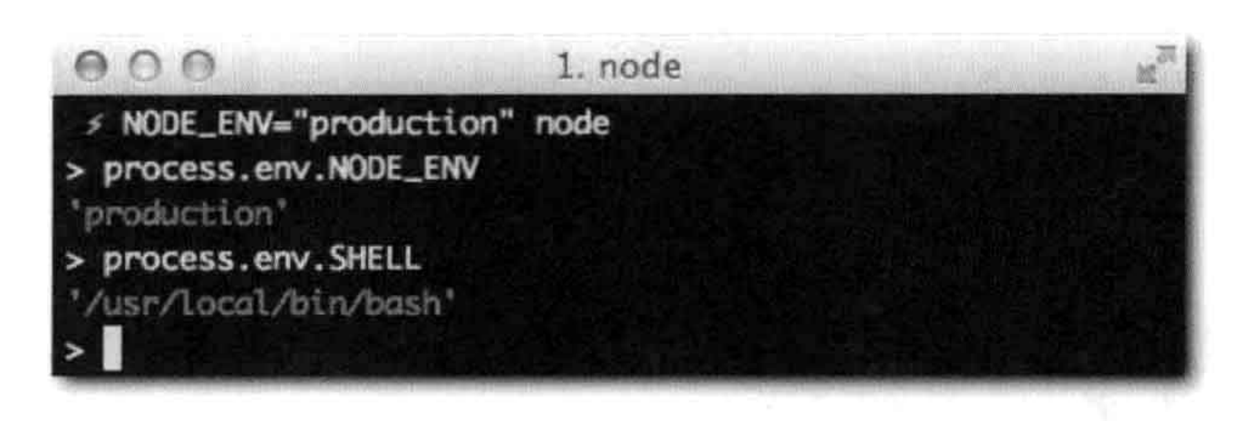


图5-14：NODE_ENV环境变量

除此之外，在程序中控制程序自身的退出也是很有必要的。

退出

要让一个应用退出，可以调用process.exit并提供一个退出代码。比如，当发生错误时，要退出程序，这个时候最好是使用退出代码1：

```
console.error('An error occurred');
process.exit(1);
```

这样可以让Node命令行程序和操作系统中其他工具进行更好的协同。

另外还有一点就是进程信号。

信号

进程和操作系统进行通信的其中一种方式就是通过信号。比如，要让进程终止，可以发送SIGKILL信号。

66 Node程序是通过在process对象上以事件分发的形式来发送信号的：

```
process.on('SIGKILL', function () {
  // 信号已收到
});
```

接下来我们看看此前的程序是如何使用转义码让终端文本呈现不同颜色的。

ANSI转义码

要在文本终端下控制格式、颜色以及其他输出选项，可以使用ANSI转义码。

在文本周围添加的明显不用于输出的字符，称为非打印字符。

比如，看下面这个例子：

```
console.log('\033[90m' + data.replace(/(.*)/g, '     $1') + '\033[39m');
```

- \033表示转义序列的开始。
- [表示开始颜色设置。
- 90表示前景色为亮灰色。
- m表示颜色设置结束。

或许你已经注意到了，最后用的是39，没错，这是用来将颜色再设置回去的，我们这里只想对部分文本着色。

http://en.wikipedia.org/wiki/ansi_escape_code列出了一张完整的ANSI转义码表。

对fs一探究竟

fs模块允许你通过Stream API来对数据进行读写操作。与readFile及writeFile方法不同，它对内存的分配不是一次完成的。

比如，考虑这样一个例子，有一个大文件，文件内容由上百万行逗号分割文本组成。要完整地读取该文件来进行解析，意味着要一次性分配很大的内存。更好的方式应当是一次只读取一块内容，以行尾结束符（"\n"）来切分，然后再逐块进行解析。

下面要介绍的Node Stream就是对上述解决方案完美的实现。

Stream

67

fs.createReadStream方法允许为一个文件创建一个可读的Stream对象。

为了更好地理解stream的威力，我们来看如下两个例子。

```
fs.readFile('my-file.txt', function (err, contents){
  // 对文件进行处理
});
```

上述例子中，回调函数必须要等到整个文件读取完毕、载入到RAM、可用的情况下才会触发。

而下面这个例子，每次会读取可变大小的内容块，并且每次读取后会触发回调函数：

```
var stream = fs.createReadStream('my-file.txt');
stream.on('data', function(chunk){
  // 处理文件部分内容
});
stream.on('end', function(chunk){
  // 文件读取完毕
});
```

为什么这种能力很重要呢？假设有个很大的视频文件需要上传到某个Web服务。这时，你无须在读取完整的视频内容后再开始上传，使用`Stream`就可以大大提速上传过程。

这对日志记录的例子也很有效，特别是使用可写stream。假设有个应用需要记录网站上的访问情况，这时，为了将记录写到文件中，让操作系统进行打开/关闭文件的操作可能就很低效（每次都得要在磁盘上进行查找文件操作）。

所以，这就是一个很好的使用`fs.WriteStream`的例子。打开文件操作只做一次，然后写入每个日志项时都调用`.write`方法。

另外一个很好的符合Node非阻塞设计的例子就是监视（Watch）。

监视

Node允许监视文件或目录是否发生变化。监视意味着当文件系统中的文件（或者目录）发生变化时，会分发一个事件，然后触发指定的回调函数。

该功能在Node生态系统中被广泛使用。举例来说，有人喜欢用一种可以编译为CSS的语言来书写CSS样式。这个时候，就可以使用监视功能，当源文件发生改变时，就将其编译为CSS文件。

68 我们来看下面这个例子。首先，查找工作目录下所有的CSS文件，然后监视其是否发生改变。一旦文件发生更改，就将该文件名输出到控制台：

```
var stream = fs.createReadStream('my-file.txt');
var fs = require('fs');
// 获取工作目录下所有的文件
var files = fs.readdirSync(process.cwd());
files.forEach(function (file) {
  // 监听“.css”后缀的文件
  if (/\.css/.test(file)) {
    fs.watchFile(process.cwd() + '/' + file, function () {
      console.log(' - ' + file + ' changed!');
```

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

除了fs.watchFile之外，还可以使用fs.watch来监视整个目录。

小结

本章介绍了书写Node.js程序的基础知识，特别是如何书写一个与文件系统进行交互的命令行程序。

尽管用同步的fs API来完成本章首个示例程序也没有什么不妥，但本章着重是要介绍如何使用异步API来帮助大家掌握书写包含多层回调的复杂代码的方法。不管怎么样，我们还是成功地完成了一个包含完整功能、代码整洁的应用。

本章还介绍了Node中最为重要的API之一——Stream，Stream会在本书中频繁出现。几乎所有涉及到I/O的地方都有它的身影，Stream真的非常棒。

除此之外，本章还教会了你使用工具来创建有用的命令行程序，与文件系统、其他程序进行交互，以及获取用户的输入。

作为Node.js开发者，不论是写Web应用还是写更加复杂的应用，这些 API会经常使用到（特别是process对象上的API）。好好地记住这些 API！