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

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Node.js是一个由JavaScript书写而成的强大的Web开发框架，它让开发强壮的、伸缩性良好的服务器端Web应用变得更加简单、容易。本书向你展示了什么是Node以及如何让你在项目中使用它。本书包含大量实际应用中的示例程序，证明了为什么Node.js会快速成为Web开发首选工具，通过本书，你能够快速熟悉和掌握达到如下目标所需的Node知识和技能：

- 了解Node基于事件轮询的架构、无阻塞I/O以及事件驱动的编程方式
- 精通Node.js的API
- 轻松实现开发实时应用相关的技术，如Socket.IO和HTML5 WebSocket
- 编写能够支持跨多台服务器的高并发应用
- 通过Node来支持多种数据库以及数据存储工具
- 编写在单台服务器情况下能够处理万级并发量的程序
- 能够在一个包含更多Node知识和注解示例（含源代码）的网站上和其他开发者进行实时的沟通交流

本书包含大量全彩插图和实用的源代码，绝对是一本革命性Web开发工具——Node的实用指南。

Guillermo Rauch（旧金山，加利福尼亚州）是一家位于旧金山，为当地教育提供相关服务的创业公司LearnBoost的CTO和联合创始人。Rauch还是几个知名Node.js项目的发明者，曾在JSConf和一些Node.js workshop做过演讲。

WILEY

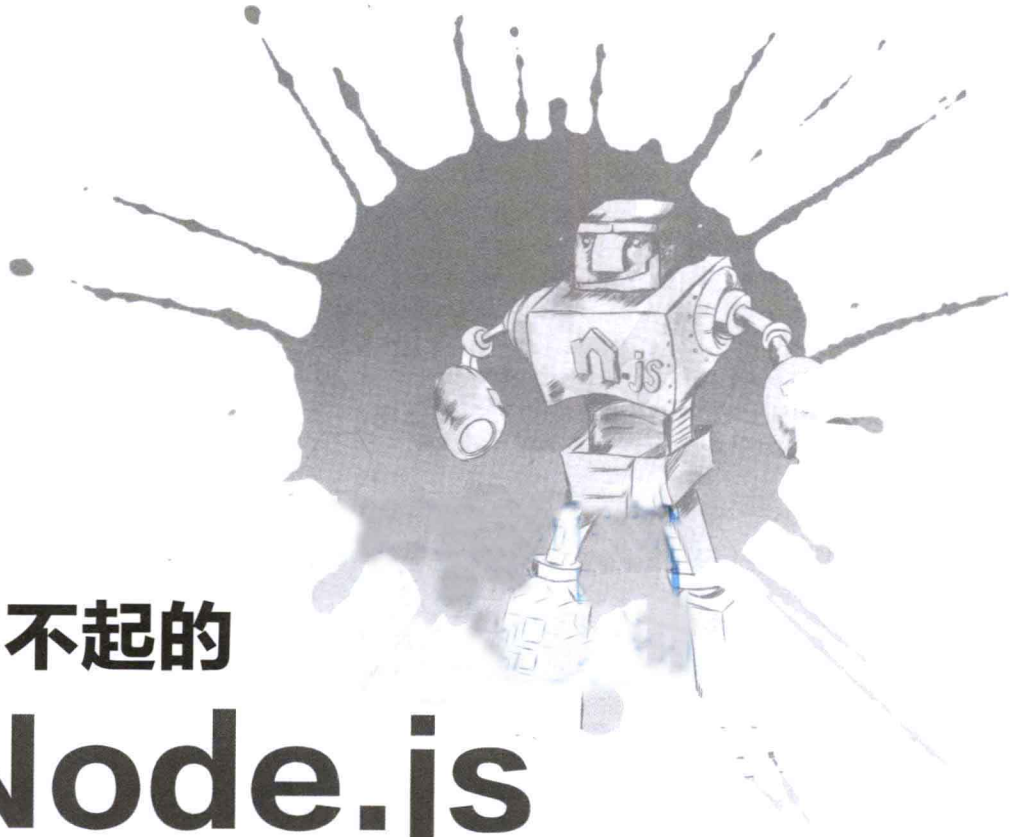
策划编辑：张春雨
责任编辑：贾　莉
封面设计：侯士卿

上架建议：Web开发/JavaScript

ISBN 978-7-121-21769-2

定价：79.00元

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。

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

PART II Node重要的API

PART Ⅳ 数据库

PART V 测试

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。

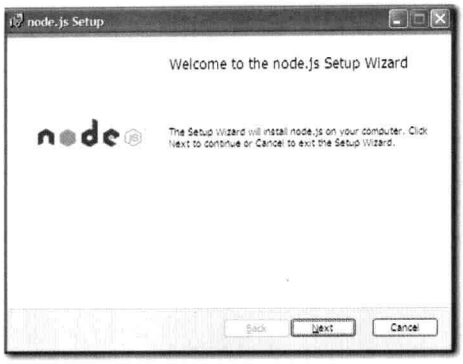


图1-1：Node.js安装指引

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。

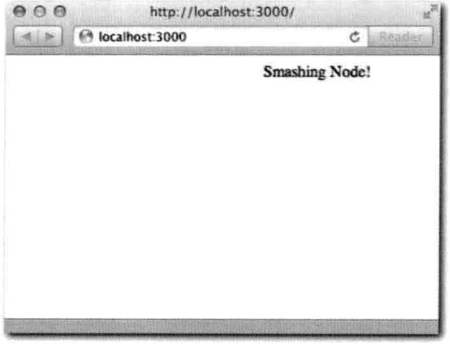


图1-3：使用Node托管一个简单的HTML文件

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。

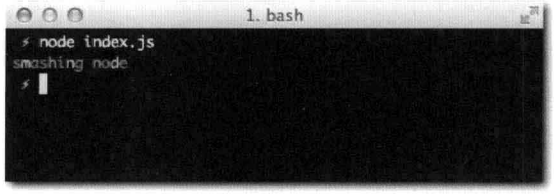


图1-4：模块安装成功验证结果

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。

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。

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;
}
```

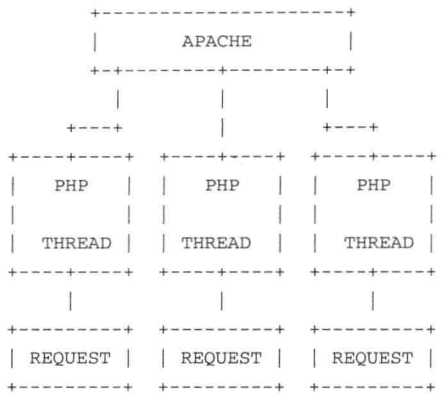
注意，在上述两例serveBooks函数中，都将books数组重置了。

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：

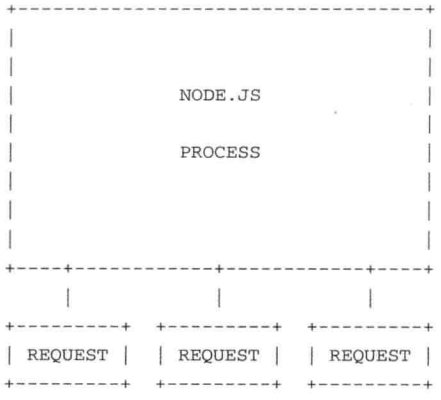
29

- 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）。



能力越强，责任也就越大。



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
31 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。

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。

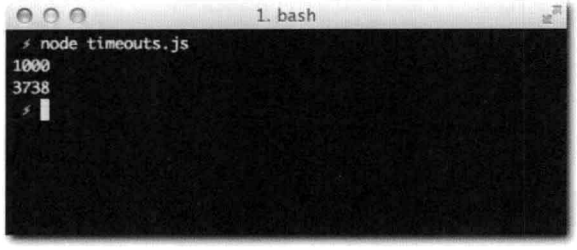


图3-1：程序显示了每个setTimeout执行的时间间隔，其结果和代码中设定的值并不相同

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

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。

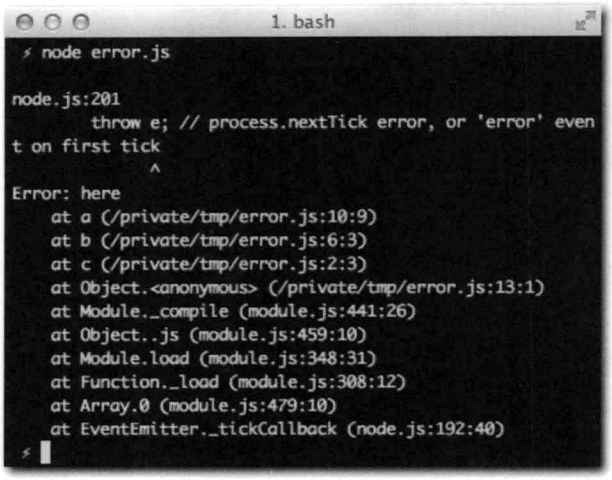


图3-3：针对上述代码，V8显示的堆栈追踪信息

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。

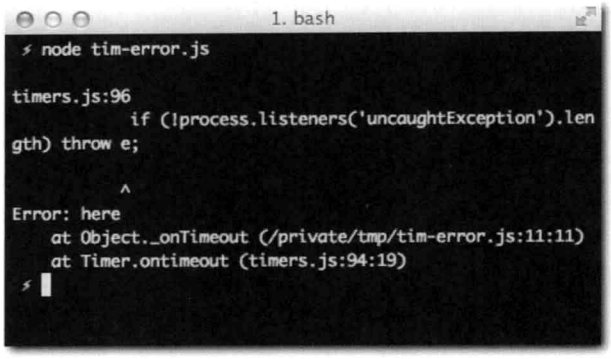


图3-4：堆栈信息显示的是从事件轮询开始的

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 42
一系列函数：

```
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

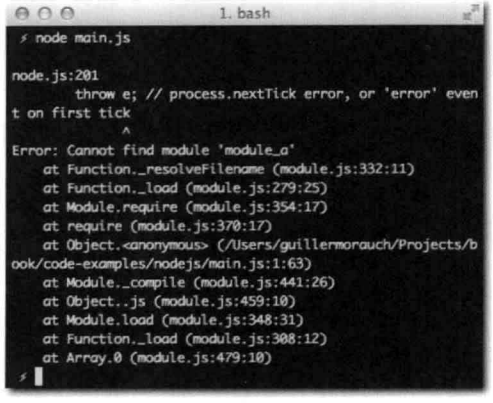


图4-1：加载module_a模块时，出错告知未能找到

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）。

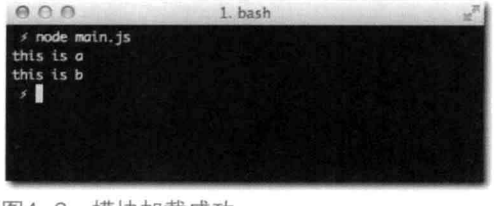


图4-2：模块加载成功

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());
```

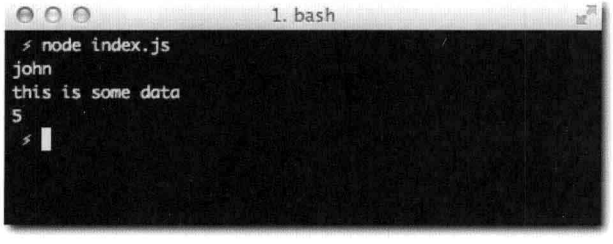


图4-3：查看module_a暴露出来的API

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();
```

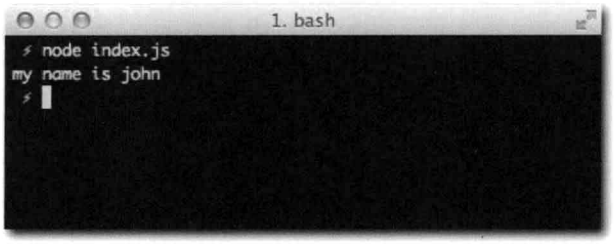


图4-4：一个具备JavaScript OOP风格的Node.js模块的例子

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;
```

47

```
  });
  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
```

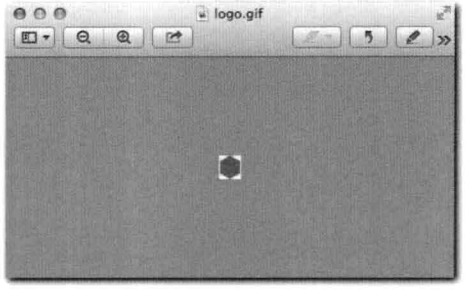


图4-5：上述脚本中，从用base64表示的buffer中创建的GIF图片显示了Node.js的logo

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 II

Node重要的API

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。

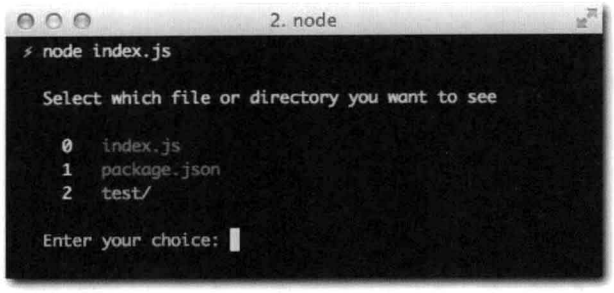


图5-1：程序启动后显示的目录列表

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）。

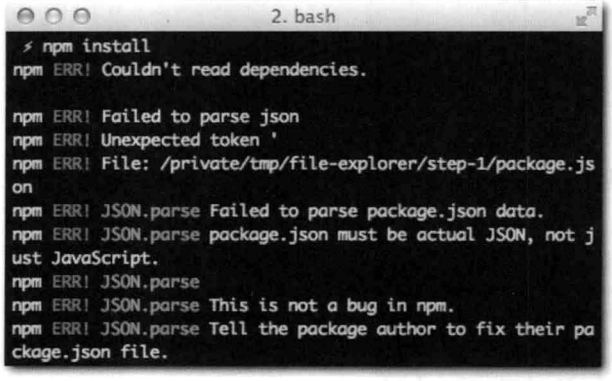


图5-2：package.json文件中有JSON错误的情况下运行npm install命令

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）。

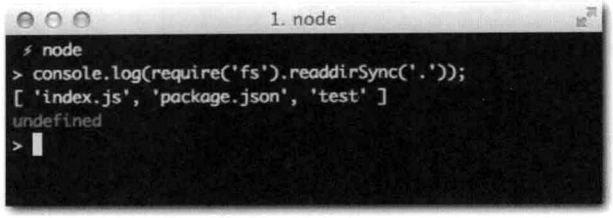


图5-3：查看readdirSync的返回值

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。

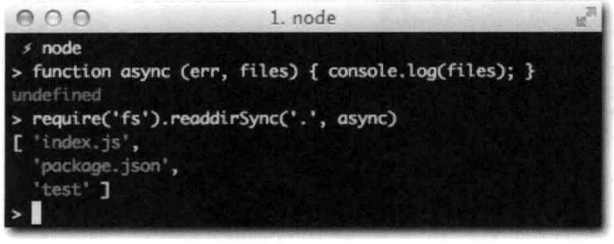


图5-4：异步版本的readdir

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。

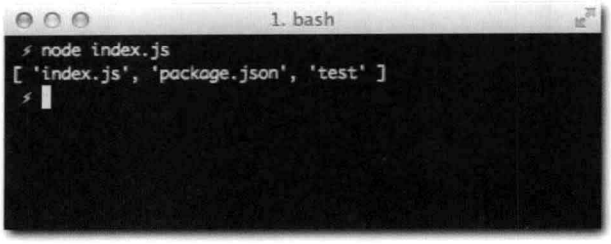


图5-5：运行自己的包含在index.js文件中的Node程序

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。

56

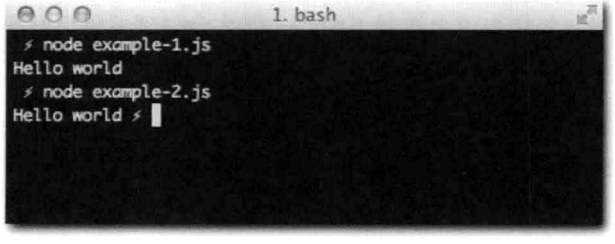


图5-6：第一个Hello World的程序加了一个换行符，第二个则没加

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。

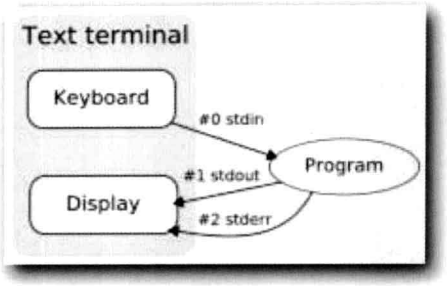


图5-7：传统文本终端下的stdio、stdout以及stderr对象

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');
```

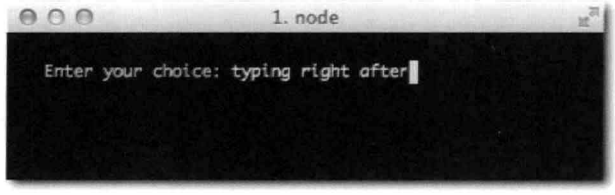


图5-8：程序提示用户向stdin进行输入

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）。

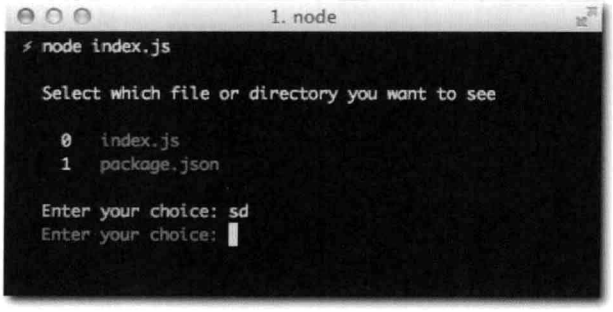


图5-9：用户选择错误的例子

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')
```

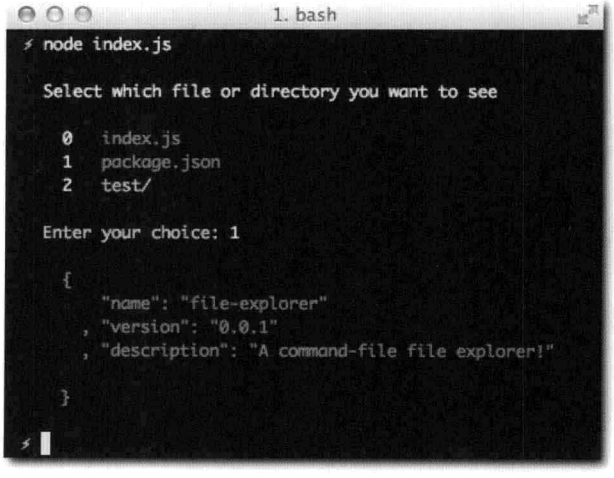


图5-10：读取简单文件的例子

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）。

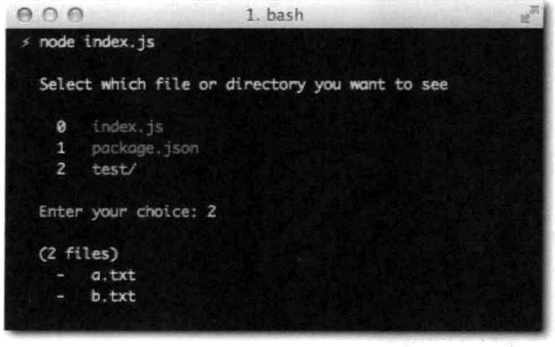


图5-11：读取test/文件夹的例子

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));
```

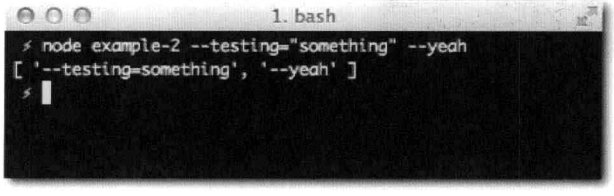


图5-13：去除argv前两个元素，显示真正元素的例子

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。

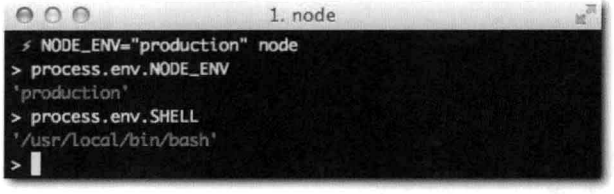


图5-14：NODE_ENV环境变量

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！

CHAPTER

6 TCP

传输控制协议（TCP）是一个面向连接的协议，它保证了两台计算机之间数据传输的可靠性和顺序。

换句话说，TCP是一种传输层协议，它可以让你将数据从一台计算机完整有序地传输到另一台计算机。

正是由于这些特点，很多我们现在使用的如HTTP这样的协议都是基于TCP协议的。当传输一个页面的HTML文件时，肯定是希望它传输到目的地时能够与传输前一致，要是出什么问题，就应该抛出错误。哪怕有一个字符（字节）传输错位了，浏览器都有可能无法渲染这个页面。

Node.js这个框架的出发点就是为了网络应用开发所设计的。如今，网络应用都是用TCP/IP协议进行通信的。所以，了解TCP/IP协议是如何工作的，以及Node.js是如何通过简单的API对其进行封装的，都是非常有帮助的。

首先要介绍的是该协议的特点。举例来说，使用TCP在两台电脑之间进行数据传输是如何做到的。当传输两条消息时，传输到目的地时还能保持发送前的顺序吗？理解协议本身对于理解使用该协议的软件也是很重要的。比如，大部分时候，连接如MySQL等的数据库以及与数

据库进行通信使用的都是TCP套接字。

Node HTTP服务器是构建于Node TCP服务器之上的。从编程角度来说，也就是Node中的`http.Server`继承自`net.Server`（`net`是TCP模块）。

除了Web浏览器和服务器（HTTP）之外，很多我们日常使用的如邮件客户端（SMTP/IMAP/POP）、聊天程序（IRC/XMPP）以及远程 shell（SSH）等都基于TCP协议。

尽可能多地了解TCP以及如何使用相关的Node.js API对书写和理解网络程序会大有裨益。

70 TCP有哪些特性

若只是使用TCP的话，那无须理解它内部的工作原理，以及其实现机制。

不过，理解这些东西对分析更高层的协议和服务器，如Web服务器、数据库等的问题有很大的帮助。

TCP的首要特性就是它是面向连接的。

面向连接的通信和保证顺序的传递

说到TCP，可以将客户端和服务器端的通信看作是一个连接或者数据流。这对开发面向服务的应用和流应用是很好的抽象，因为TCP协议做基于的IP协议是面向无连接的。

IP是基于数据报的传输。这些数据报是独立进行传输的，送达的顺序也是无序的。

那么TCP又是如何保证这些独立的数据报送达的时候是有序的呢？

使用IP协议意味着数据包送达时是无序的，这些数据包不属于任何的数据流或者连接，那么当使用TCP/IP和服务器建立连接后，是怎样做到让数据包送达时是有序的呢？

要回答上述问题其实就等于在解释为什么会有TCP。当在TCP连接内进行数据传递时，发送的IP数据报包含了标识该连接以及数据流顺序的信息。

假设一条消息分割为四个部分。当服务器从连接A收到第一部分和第四部分后，它就知道还要等待其他数据报中的第二部分和第三部分。

要是用Node来写一个TCP服务器，就完全没必要去考虑这些复杂的内部实现了。只要考虑连接以及往套接字中写数据即可。接收方会按序接收到传输的信息，要是发生网络错误，连接会失效或者终止。

面向字节

TCP对字符以及字符编码是完全无知的。正如第4章介绍的，不同的编码会导致传输的字节数不同。

所以，TCP允许数据以ASCII字符（每个字符一个字节）或者 Unicode（即每个字符四个字节）进行传输。

正是因为对消息格式没有严格的约束，使得TCP有很好的灵活性。 71

可靠性

由于TCP是基于底层不可靠的服务，因此，它必须要基于确认和超时实现一系列的机制来达到可靠性的要求。

当数据发送出去后，发送方就会等待一个确认消息（表示数据包已经收到的简短的确认消息）。如果过了指定的窗口时间，还未收到确认消息，发送方就会对数据进行重发。

这种机制有效地解决了如网络错误或者网络阻塞这样的不可预测的情况。

流控制

要是两台互相通信的计算机中，有一台速度远快于另一台的话，会怎么样呢?

TCP会通过一种叫流控制的方式来确保两点之间传输数据的平衡。

拥堵控制

TCP有一种内置的机制能够控制数据包的延迟率及丢包率不会太高，以此来确保服务的质量（QoS）。

举例来说，和流控制机制能够避免发送方压垮接收方一样，TCP会通过控制数据包的传输速率来避免拥堵的情况。

好了，介绍完TCP的基本工作原理，到实践的时候了。为了测试 TCP服务器，我们可以使用Telnet工具。

Telnet

Telnet是一个早期的网络协议，旨在提供双向的虚拟终端。在SSH出现前，它作为一种控制远程计算机的方式被广泛使用，如远程服务器管理。它是TCP协议上层的协议（不要惊讶）。

尽管从21世纪初就基本不用Telnet了，但如今绝大部分主流的操作系统都内置了telnet客户端（见图6-1）：

```
$ telnet
```

绝大部分Telnet使用的是23号端口。要是通过该端口连接到服务器（telnet host.com 23或者就简单地写成telnet host.com），就说明在通过TCP使用Telnet协议。

72

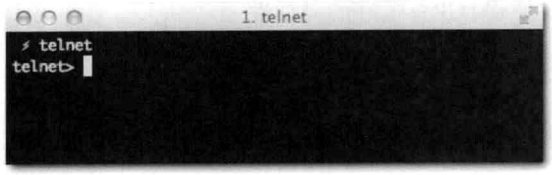


图6-1：运行telnet工具

不过，在本例中，telnet客户端还有更加有意思的功能。发送数据时，客户端要是发现服务器使用的并不是Telnet，这时，它不会关闭连接或者显示错误信息，相反，它会自动降级到低层的纯TCP模式。

所以，要是telnet到Web服务器会怎么样呢？要一探究竟，我们来看下面这个例子。

首先，我们用Node.js来写一个简单的hello world Web服务器，并监听3000端口：

```
# web-server.js
require('http').createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end('<h1>Hello world</h1>');
}).listen(3000);
```

接下来通过node server.js运行上述代码。要确保运行正常，可以打开一个典型的HTTP客户端——浏览器来查看，如图6-2所示。

图6-2：浏览器通过本地3000端口建立了一个TCP连接，然后开始使用HTTP协议进行通信

现在来实现客户端部分。使用telnet来建立一个连接（见图6-3）：

```
$ telnet localhost 3000
```

73

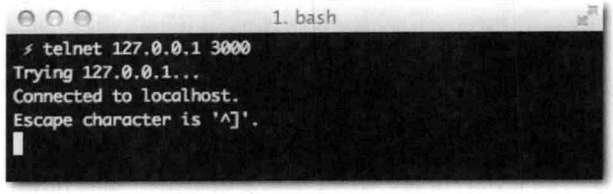


图6-3：telnet允许在终端手动建立一个TCP连接

尽管根据图6-3中所示的结果，看上去已经工作正常了，但是，服务器端的“Hello World”消息并未到客户端这里。原因在于，要往TCP连接中写数据，必须首先创建一个HTTP请求，这就是套接字（socket）。在终端输入GET / HTTP/1.1然后按两下回车键。

如图6-4所示，这个时候，服务器端的响应就出现了！

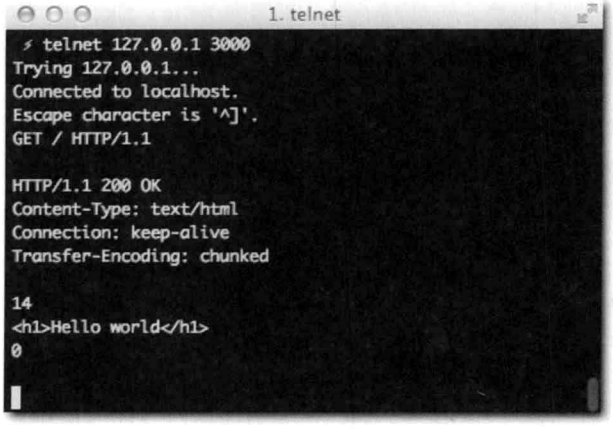


图6-4：Mac上IRC客户端（Textual.app）实践。Textual.app实现了基于TCP套接字的IRC协议

我们来总结一下：

- 成功建立了一个TCP连接。
- 创建了一个HTTP请求。
- 接收到了一个HTTP响应。
- 测试了一些TCP的特性。到达的数据和在Node.js中写的一样：先写了`Content-Type`响应头，然后是响应体，最后所有的信息都按序到达。

74 基于TCP的聊天程序

正如此前介绍的，TCP的主要目的就是为两台计算机通过提供可靠的网络进行通信。

本章选择一个聊天应用作为TCP的“Hello World”程序，因为，它是展示TCP最简单的方式之一。

下面，我们来创建一个基本的TCP服务器，任何人都可以连接到该服务器，无须实现任何协议或者指令：

- 成功连接到服务器后，服务器会显示欢迎信息，并要求输入用户名。同时还会告诉你当前还有多少其他客户端也连接到了该服务器。
- 输入用户名，按下回车键后，就认为成功连接上了。
- 连接后，就可以通过输入信息再按下回车键，来向其他客户端进行消息的收发。

为什么要按下回车键呢？事实上，Telnet中输入的任何信息都会立刻发送到服务器。按下回车键是为了输入\n字符。在Node服务器端，通过\n来判断消息是否已完全到达。所以，这其实是作为一个分隔符在使用。

换句话说，这里按下回车键和输入字符a没有什么区别。

创建模块

按照惯例，我们先来创建一个项目目录和package.json文件：

```
# package.json
{
    "name": "tcp-chat"
  , "description": "Our first TCP server"
  , "version": "0.0.1"
}
```

接着运行npm install测试一下。结果会输出一个空行，因为项目没有任何依赖。

理解NET.SERVER API

接下来，创建一个包含如下代码的index.js文件：

```
/**
 * 模块依赖
 */

var net = require('net')
/**
 * 创建服务器
 */
```

75

```
var server = net.createServer(function (conn) {
  // handle connection
  console.log('\033[90m   new connection!\033[39m');
});

/**
 * 监听
 */

server.listen(3000, function () {
  console.log('\033[96m   server listening on *:3000\033[39m');
});
```

注意，上述代码中为createServer指定了一个回调函数。该函数在每次有新的连接建立时都会被执行。

为了验证，我们来运行上述代码，启动一个TCP服务器。当listen执行时，它会将服务器绑定到3000端口，并最终在终端打印出一段消息。

```
$ node index.js
```

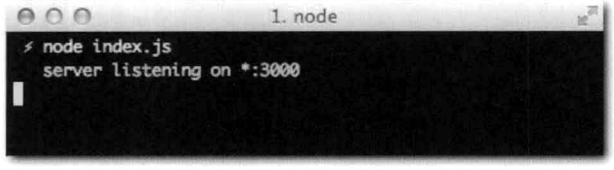


图6-5：服务器会绑定到3000端口，然后显示一段成功启动的信息

现在，我们尝试用telnet来进行连接：

```
$ telnet 127.0.0.1 3000
```

在图6-6中，能看到该命令，而且，“new connection！”消息也会显示出来。

如你所见，这个例子和此前HTTP的hello world程序类似。现在再去理解“HTTP是建立在TCP协议之上的”就不难了吧。不过，本例中，我们会创建自己的协议。

createServer回调函数会接收一个对象，该对象是Node中一个很常见的实例：流（Stream）。本例中，它传递的是net.Stream，该对象通常是既可读又可写的。

最后，还有一个重要的方法就是listen，它可以将服务器绑定要某个端口上。由于该方法也是异步的，所以也接收一个回调函数。

76

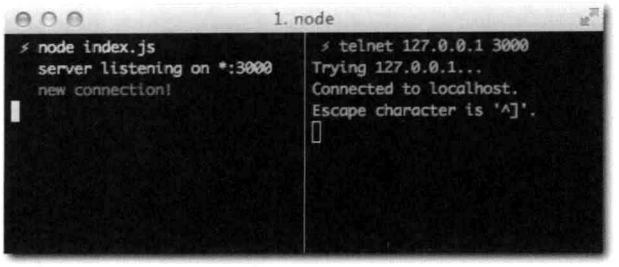


图6-6：左侧显示的是服务器进程的状态。右侧则是客户端，连接到服务器端后，服务器端就输出了“新连接（new connection）！”

接收连接

正如此前项目描述中的，一旦连接建立，就会向客户端回写欢迎语和当前连接数。

我们先在回调函数外部添加一个计数器：

```
/**
 * 追踪连接数
 */

var count = 0;
```

接着，我们需要修改回调函数内容，把计数器递增和打印出欢迎语的逻辑添加上去：

```
var server = net.createServer(function (conn) {
  conn.write(
      '\n > welcome to \033[92mnode-chat\033[39m!'
    + '\n > ' + count + ' other people are connected at this time.'
    + '\n > please write your name and press enter: '
  );
  count++;
});
```

如上述代码所示，我们仍旧使用shell转义码来控制输出文本的颜色。

接着重启服务器进行测试：

```
$ node index
```

再次通过`telnet`去连接（见图6-7）：

```
$ telnet 127.0.0.1 3000
```

77

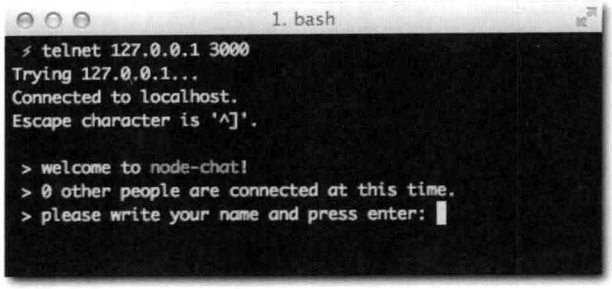


图6-7：现在客户端连接成功后能够接收到更多信息了

如图6-8所示，当第二个客户端连接进去后，计数器就增加了一个！

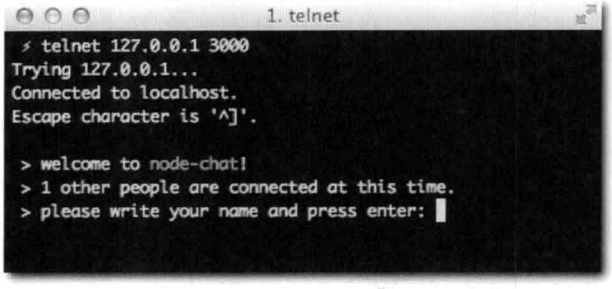


图6-8：连接计数器正确反映了当前连接数

当客户端请求关闭连接时，计数器变量就要进行递减操作：

```
conn.on('close', function () {
  count--;
});
```

当底层套接字关闭时，Node.js会触发close事件。Node.js中有两个和连接终止相关的事件：end和close。前者是当客户端显示关闭TCP连接时触发。比如，当你关闭telnet时，它会发送一个名为“FIN”的包给服务器，意味着要结束连接。

当连接发生错误时（触发error事件），end事件不会触发，因为服务器端并未收到“FIN”包信息。不过这两种情况下，close事件都会被触发，所以，上述例子中使用close事件会比较好。

在Mac上，可以通过按下Alt + [来结束一个telnet连接，Windows上可以用Ctrl +]。

data事件

我们已经能在客户端打印出一些信息了，接着我们来看看如何处理客户端发送的数据。

首先要处理的数据是用户输入的昵称（nickname），所以，我们从监听data事件开始。与

78

其他Node中的API一样，net.Stream同时也是一个EventEmitter。

为了进行测试，我们在服务器端的控制台输出客户端发来的数据：

```
var server = net.createServer(function (conn) {
  conn.write(
      '\n > welcome to \033[92mnode-chat\033[39m!'
    + '\n > ' + count + ' other people are connected at this time.'
    + '\n > please write your name and press enter: '
  );
  count++;

  conn.on('data', function (data) {
    console.log(data);
  });
  conn.on('close', function () {
    count--;
  });
});
```

然后，我们启动该服务器，并使用客户端进行连接。如图6-9所示，在客户端输入一些数据。看左侧部分，当输入数据时，服务器端就直接通过console.log将其打印出来了。

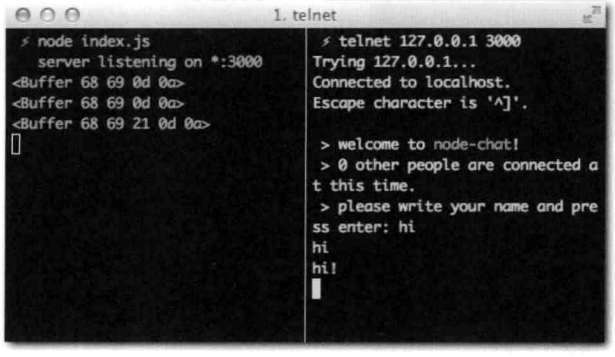


图6-9：图中左侧部分显示了右侧发送过来数据相对应的Buffer对象

如图6-9中所示，我们接收到的数据是一个Buffer。还记得此前介绍的，TCP是面向字节的协议吗？这里可以看得出来Node遵循了TCP的这一习惯！

这个时候，有多种选择可以获取字符串形式的数据。可以通过调用Buffer对象上的.toString('utf8')来获取utf8编码的字符串。

不过，由于我们这里无须获取utf8之外其他编码格式的数据，我们可以通过net.Stream#setEncoding方法来设置编码（如图6-10所示）：

```
# index.js
. . .
conn.setEncoding('utf8');
```

79

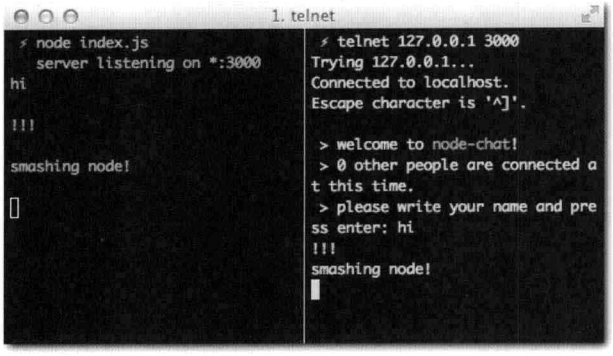


图6-10：聊天信息现在以utf8编码的字符串形式输出在左侧

好了，至此，我们已经可以让客户端和服务器进行交互了，接下来我们要让更多的客户端加入聊天。

状态以及记录连接情况

此前定义的计数器通常称为状态。因为，在本例中，两个不同连接的用户需要修改同一个状态变量，这在Node中称为共享状态的并发。

为了能够向其他连接进来的客户端发送和广播消息，我们需要对该状态进行扩展，来追踪到底谁连接进来了。

当客户端输入了昵称后，就认为该客户端已经连接成功，并可以接收消息了。

首先，我们要记录设置了昵称的用户。为此，我们需要引入一个新的状态变量，`users`：

```
var count = 0
  , users = {}
```

然后，在每个连接中，再引入一个`nickname`变量：

```
conn.setEncoding('utf8');

// 代表当前连接的昵称
var nickname;

conn.on('data', function (data) {
```

接收到数据时，确保将`\r\n`（相当于按下回车键）清除：

80

```
// 删除回车符
data = data.replace('\r\n', '');
```

对于尚未注册的用户，需要进行校验。如果昵称可用，则通知其他客户端当前用户已经连接进来了（见图6-11）：

```
// 接收到的第一份数据应当是用户输入的昵称
if (!nickname) {
  if (users[data]) {
    conn.write('\033[93m> nickname already in use. try again:\033[39m ');
    return;
  } else {
    nickname = data;
    users[nickname] = conn;

    for (var i in users) {
      users[i].write('\033[90m > ' + nickname + ' joined the room\033[39m\n');
    }
  }
}
```

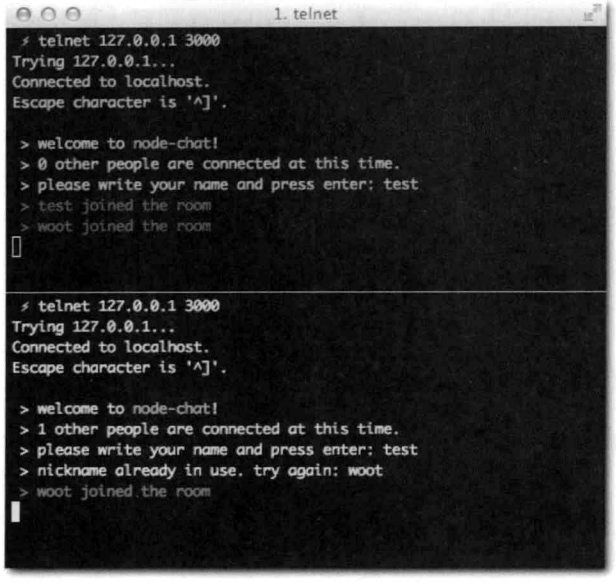


图6-11：消息已经广播给所有加入聊天服务器的客户端了

如果是已经验证通过的用户，那么接下来收到的数据就认为是聊天消息，需要显示给所有其他客户端：

81

```
else {
  // 否则，视为聊天消息
  for (var i in users) {
    if (i != nickname) {
      users[i].write('\033[96m > ' + nickname + ':\033[39m ' + data + '\n');
    }
```

```
  }
}
```

使用i != nickname来确保消息只发送给除了自己以外的其他客户端。

图6-12展示了两台客户端连接进来之后，互相聊天的场景。

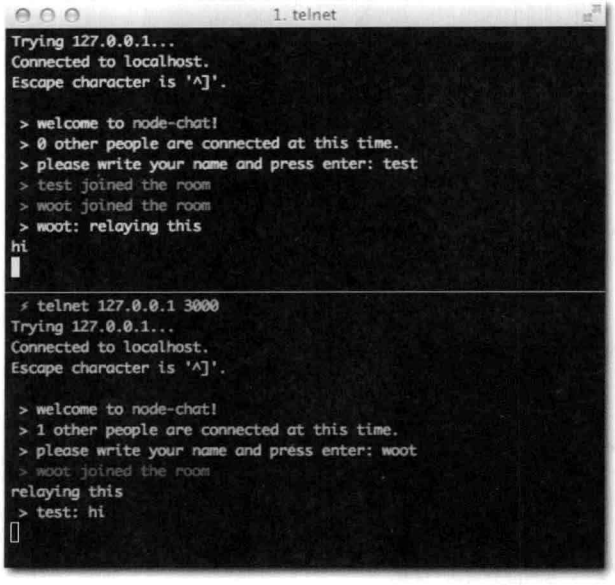


图6-12：展示了两台客户端连进来之后，互相聊天的场景

成功交换聊天消息后，我们来圆满完成这个程序。

圆满完成此程序

当有人断开连接时，我们需要清除users数组中对应的元素：

```
conn.on('close', function () {
  count--;
  delete users[nickname];
});
```

用户断开时通知其他用户也是个不错的想法。由于我们时不时就需要给所有的用户广播消息，所以，把这部分逻辑抽象出来也不错：

```
// . . .
function broadcast (msg, exceptMyself) {
  for (var i in users) {
    if (!exceptMyself || i != nickname) {
      users[i].write(msg);
    }
  }
```

82

```
}

conn.on('data', function (data) {
  // . . .
```

这样一来，就可以重用上面的函数进行消息广播了，简洁易懂，一目了然：

```
broadcast('\033[90m > ' + nickname + ' joined the room\033[39m\n');
// . . .
broadcast('\033[96m > ' + nickname + ':\033[39m ' + data + '\n', true);
```

我们把它加到close处理器中（见图6-13）：

```
conn.on('close', function () {
  // . . .
  broadcast('\033[90m > ' + nickname + ' left the room\033[39m\n');
});
```

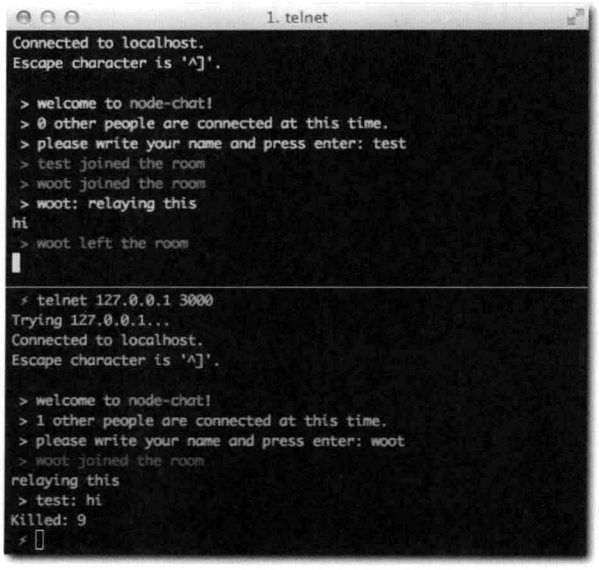


图6-13：在关掉第一个客户端时，会强制关闭该连接，接着其他客户端就会收到当前用户离开的消息

完成！

83 在成功实现了一个TCP服务器后，我们需要进一步学习在Node.js中如何实现一个TCP客户端。

客户端API和其他像Twitter这样用来查询Web服务的HTTP客户端类似，因此，完全明白这些非常重要。

一个IRC客户端程序

IRC是因特网中继聊天（Internet Relay Chat）的缩写，它也是一项常用的基于TCP的协议。它通常被使用在如图6-14所示的客户端程序中，作为连接到IRC服务器的客户端。

图6-14：正在运行的Mac上一款IRC客户端程序（Linkinus）。Linkinus实现了基于TCP套接字的IRC协议

此前我们成功地构建了一个TCP服务器，这次我们来写一个TCP客户端。

构建一个实现IRC协议的客户端意味着，需要实现通过一组命令来实现与IRC服务器进行“通信”，进行数据的交换。

比如，要设置昵称，客户端可以发送如下指令：

```
NICK mynick
```

IRC是一项非常直观、简单的协议。通过一些简单的命令就可以和现有的应用以及服务器（如图6-14中展示的）进行通信。

下面我们就来学习如何用Node.js书写一个非常基本的客户端，实现加入一个聊天室以及回复消息功能。

创建模块

和往常一样，我们先来创建一个项目目录，并在该目录下创建一个package.json文件：

```
{
    "name": "irc-client"
  , "description": "Our first TCP client"
  , "version": "0.0.1"
}
```

84

接着运行npm install。因为项目没有任何依赖的模块，所以控制台应当会输出一个空行。

理解NET#STREAM API

和createServer一样，net API提供了另外一个名为connect的方法，如下所示：

```
net.connect(port[[, host], callback]])
```

如果提供了回调函数，就等于是监听了该方法返回的对象上的connect事件。

```
var client = net.connect(3000, 'localhost');
client.on('connect', function () {});
```

上述代码和下面的代码是一样的：

```
net.connect(300, 'localhost', function () {});
```

另外，和此前使用的API类似，我们还可以监听data和close事件。

实现部分IRC协议

我们先来初始化IRC客户端。然后尝试去登录irc.freenode.net中的#node.js频道：

```
var client = net.connect(6667, 'irc.freenode.net')
```

设置编码为utf-8：

```
client.setEncoding('utf-8')
```

连接上服务器后，发送自己的昵称。除此之外，还要写上服务器要求的USER命令。采用类似如下所示的方式来发送数据：

```
NICK mynick
USER mynick 0 * :realname
JOIN #node.js
```

85

具体代码如下所示：

```
client.on('connect, function () {
  client.write('NICK mynick\r\n');
  client.write('USER mynick 0 * :realname\r\n');
  client.write('JOIN #node.js\r\n')
});
```

这里要注意，需要在每条命令后加上\r\n分割符。这和此前在Telnet下按下回车键是一样的。\r\n也是HTTP协议用来区分头信息的分隔符。

测试实际的IRC服务器

打开一个IRC客户端（如Windows上的mIRC、Linux上的xChat或者Mac上的Colloquy/Linkinus），并将其指向：

```
irc.freenode.net
#node.js
```

启动后，观察mynick是否连接上了：

```
![](http://f.cl.ly/items/1b3g3i1w120Z2U082I3G/Image%202011.11.07%202:31:35%20AM.png)
```

小结

本章介绍了一个简单的网络客户端的实现，成功使其与一台并非自己实现的TCP服务器进行通信。

作为习题，你可以尝试着去完成如下部分：监听data数据并尝试解析接收到的数据，将其他用户发送到#node.js频道的消息打印出来。你还可以进一步尝试着结合现有的代码去实现一个IRC机器人，来自动对命令做出响应。比如，当某人说日期（data）时（可以在data事件中检测到），你就可以输出new Date()的结果。

紧接着在下一章，你会学到HTTP这种Node.js因此闻名的Web协议。现在你对TCP中数据传递、数据块的构建已经很清楚了，那么接下来学习TCP上层的HTTP API，一定会让你对Node.js中的核心功能有更加深入的了解。

CHAPTER

7 HTTP

超文本传输协议，又称为HTTP，是一种Web协议，它为Web注入了很多强大的功能，正如在第6章中介绍的，它是属于TCP上层的协议。

87

本章会介绍如何使用Node.js服务器端和客户端的API。尽管两者都很容易上手，但是在构建真正的Web网站时，它们还是存在着一些不足的。正因如此，下面的章节还会给大家介绍如何实现HTTP服务器上层的抽象，完成可复用的模块。

注意，由于我们要编写的示例代码一部分属于服务器端，所以每次对文件进行修改，都需要重启Node进程来使其生效。在本章的最后，会给大家介绍如何使用工具来简化这种频繁重启的操作。

下面，就让我们从分析HTTP协议开始吧。

HTTP结构

88

HTTP协议构建在请求和响应的概念上，对应在Node.js中就是由`http.ServerRequest`和`http.ServerResponse`这两个构造器构造出来的对象。

当用户浏览一个网站时，用户代理（浏览器）会创建一个请求，该请求通过TCP发送给

Web服务器，随后服务器会给出响应。

那么，请求和响应是什么样的呢？我们先用Node创建一个Hello World的HTTP服务器，并监听http://localhost:3000：

```
require('http').createServer(function (req, res) {
  res.writeHead(200);
  res.end('Hello World');

}).listen(3000);
```

接着，建立一个telnet连接，并发送请求：

```
GET / HTTP/1.1
```

输入GET / HTTP/1.1后，按下两次回车键。

如图7-1所示，响应会立刻出现！

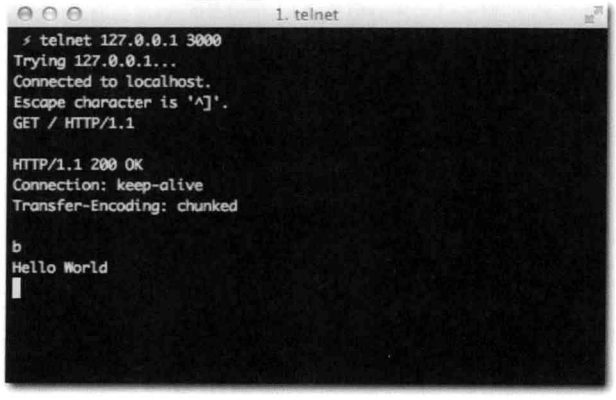


图7-1：HTTP服务器发送的响应

响应的内容如下所示：

```
HTTP/1.1 200 OK
Connection: keep-alive
Transfer-Encoding: chunked
```

89

```
b
Hello World
0
```

该响应中第一部分是头信息，下面会对头信息做相关介绍。

头信息

HTTP协议和IRC一样流行，其目的是进行文档交换。它在请求和响应消息前使用头信息

（header）来描述不同的消息内容。

举个例子，Web页面会分发许多不同类型的内容：文本（text）、HTML、XML、JSON、PNG以及JPEG图片，等等。

发送内容的类型（type）就是在著名的Content-Type头信息中标注的。

来看一个实践中的例子。还是回到hello world，不过这次我们在里面加点HTML：

```
require('http').createServer(function (req, res) {
  res.writeHead(200);
  res.end('Hello <b>World</b>');

}).listen(3000);
```

注意，World这个单词放在粗体的标签内。我们再通过简单的TCP客户端来看看效果（见图7-2）。

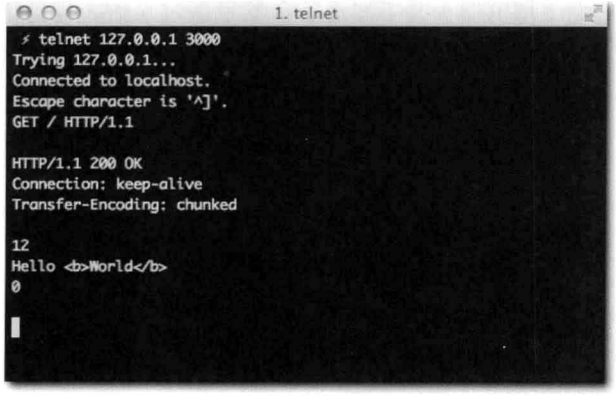


图7-2：Hello World的响应

响应结果和预期的一样：

```
GET / HTTP/1.1

HTTP/1.1 200 OK
Connection: keep-alive
Transfer-Encoding: chunked

12
Hello <b>World</b>
0
```

90

不过，现在我们再在浏览器中试试，看看结果会如何（见图7-3）。

图7-3：浏览器将响应结果作为纯文本来处理了

可是并没有看到富文本，怎么回事？

之所以会这样，是因为HTTP客户端（浏览器）并不知道服务器发送过来的内容是什么类型，我们没有把这部分信息告诉浏览器。于是，浏览器就认为它看到的内容是text/plain类型，也就是普通文本类型，就不会将它作为HTML来渲染了。

如果我们把代码稍做修改，加入正确的头信息，就能解决这个问题了（见图7-4）：

```
require('http').createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end('Hello <b>World</b>');

}).listen(3000);
```

91

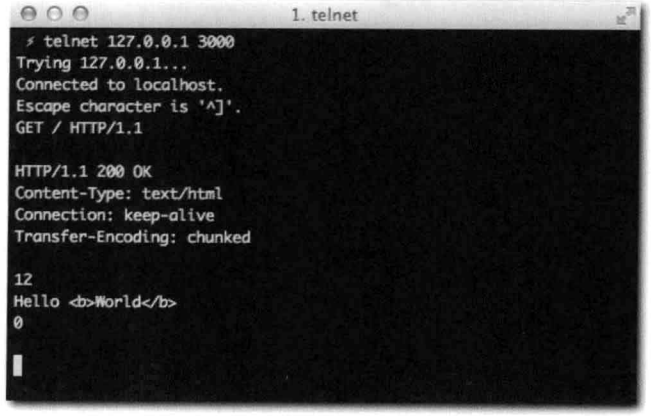


图7-4：带有头信息的响应消息

响应消息如下所示：

```
HTTP/1.1 200 OK

Content-Type: text/html
Connection: keep-alive
Transfer-Encoding: chunked

12
Hello <b>World</b>
0
```

注意，这里头信息包含在响应消息中。浏览器会对其进行解析（见图7-5），这就能够正确地渲染出HTML了。

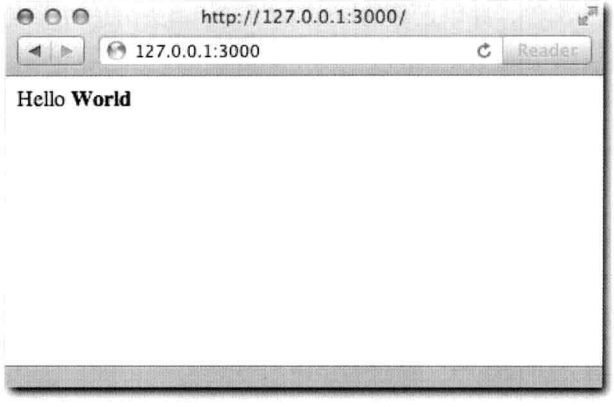


图7-5：现在浏览器将World这个单词展示成粗体了

注意，尽管我们只用writeHead API指定了一个头信息，但是，Node还是把另外两个头信息——Transfer-Encoding和Connection加进去了。 92

Transfer-Encoding头信息的默认值是chunked，主要的原因是Node天生的异步机制，这样响应就可以逐步产生。

来看下面这个例子：

```
require('http').createServer(function (req, res) {
  res.writeHead(200);
  res.write('Hello');

  setTimeout(function () {

    res.end('World');

  }, 500);

}).listen(3000);
```

注意，在调用end前，我们可以多次调用write方法来发送数据。为了尽可能快地响应客户端，在首次调用write时，Node就能把所有的响应头信息以及第一块数据（Hello）发送出去。

随后，在执行setTimeout回调函数时，又写入了另外一块数据。由于这次是使用end而不是write方法，因此，Node会结束响应，并不再允许往这次响应中发送数据了。

发送数据块的方式在涉及文件系统的情况下会非常高效。Web服务器对硬盘上的文件进行托管服务是很常见的。因为Node允许以数据块的形式往响应中写数据，同时它又允许以数据块的形式读取文件，所以我们就可以使用ReadStream文件系统API来实现。

下面这个例子用于读取image.png文件，并以正确的Content-Type头信息做出响应：

```
require('http').createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'image/png');
  var stream = require('fs').createReadStream('image.png');
  stream.on('data', function (data) {
    res.write(data);
  });
  stream.on('end', function () {
    res.end();
  });
}).listen(3000);
```

93 以一系列数据块的形式来将图片写入到响应中，有如下好处：

- 高效的内存分配。要是对每个请求在写入前都完全把图片信息读取完（通过fs.readFile），在处理大量请求时会消耗大量内存。
- 数据一旦就绪就可以立刻写入了。

另外，这里要注意的是，实际上我们做的就是把一个流（Stream）（文件系统）接（piping）到了另一个流上（一个http.ServerResponse对象）。正如此前介绍的，流是Node.js中非常重要的一种抽象。流的对接是很常见的行为，为此，Node.js提供了一个方法让上述例子代码变得非常简洁：

```
require('http').createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'image/png');
  require('fs').createReadStream('image.png').pipe(res);
}).listen(3000);
```

好了，现在你应当明白为什么Node默认要用chunked传输编码了，下面我们来看看连接（connection）。

连接

要是比对一下TCP服务器和HTTP服务器的实现，你可能会注意到它们很相似：都调用了createServer方法，并且当客户端连入时都会执行一个回调函数。

不过，它们有个本质的区别，即回调函数中对象的类型。在net服务器中，是个连接（connection）对象，而在HTTP服务器中，则是请求和响应对象。

之所以会这样，原因有两个。其一，HTTP服务器是更高层的API，提供了控制和HTTP协议相关的一些功能。

比如，当Web浏览器请求服务器时，我们来看下请求对象（例子中的req参数）中的headers属性（见图7-6）。作为实验代码，我们就用console.log将req.headers属性值打印出来：

```
require('http').createServer(function (req, res) {
  console.log(req.headers);
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end('Hello <b>World</b>');

}).listen(3000);
```

94

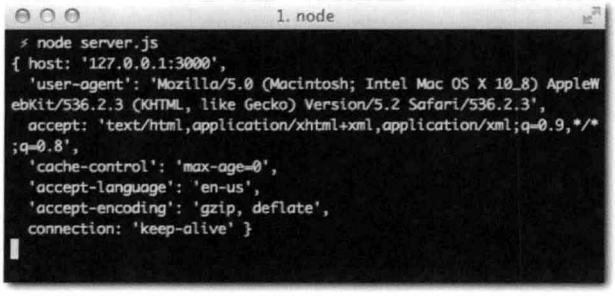


图7-6：利用console.log打印出的ServerRequest headers属性值

注意了，这里Node在内部做了很多的事情。它拿到浏览器发送的数据后，对其进行分析（解析），然后构造了一个JavaScript对象方便我们在脚本中使用。它甚至还将所有的头信息都变成了小写，这样我们就无须去记忆到底是Content-type、Content-Type还是Content-TYPE了。

其二，同时也是一个更为重要的原因是，浏览器在访问站点时不会就只用一个连接。很多主流的浏览器为了更快地加载网站内容，能向同一个主机打开八个不同的连接，并发送请求。

尽管我们可以通过req.connection获取TCP连接对象，Node为了不让我们担心到底这是请求还是连接，为我们提供了请求和响应的抽象。因此，即使你能通过req.connection

属性获得TCP连接对象，但大多情况下你还是在与请求和响应的抽象打交道。

默认情况下，Node会告诉浏览器始终保持连接，通过它发送更多的请求。这是通过此前我们看到的Connection头信息中的keep-alive值来通知浏览器的。为了提高性能（因为浏览器不想浪费时间去重新建立和关闭TCP连接），这样做通常都是对的，不过，我们也可以调用writeHead方法，传递一个不同的值，如Close，来将其重写掉。

下一个项目，我们会使用Node HTTP API来完成一个切实的任务：处理用户提交的表单。

一个简单的Web服务器

通过本项目，你能够学到如何使用此前列出的一些关键概念，如Content-Type头信息。

你还将学到Web浏览器是如何在表单提交时传递编码过的数据的，以及如何将它们解析为JavaScript中的数据结构。

95

创建模块

按照惯例，我们从创建一个项目目录开始，并在该目录下创建一个package.json文件：

```
{
    "name": "http-form"
  , "description": "An HTTP server that processes forms"
  , "version": "0.0.1"
}
```

接着运行npm install。因为项目没有任何依赖的模块，所以终端应该只会打印出一个空行。

输出表单

此前Hello World例子中，我们输出了一些HTML内容。本例中，我们要输出一个表单。在server.js文件中写入如下内容：

```
require('http').createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end([
      '<form method="POST" action="/url">'
    , '<h1>My form</h1>'
    ,   '<fieldset>'
    ,   '<label>Personal information</label>'
```

```
    ,   '<p>What is your name?</p>'
    ,   '<input type="text" name="name">'
    ,   '<p><button>Submit</button></p>'
    , '</form>'
  ].join('')); }).listen(3000);
```

注意，为了让HTML结构更加清楚，我把响应文本内容写在一个数组中，再用数组的join方法将其转为字符串。其他部分和Hello World例子一样。

还要注意的是，<form>标签中有/url以及POST方法。另外，供用户输入的输入框还有个叫name的名字。

下面，我们来运行服务器：

```
$ node server.js
```

然后，通过浏览器进行访问，如图7-7所示，就能看到表单了。 96

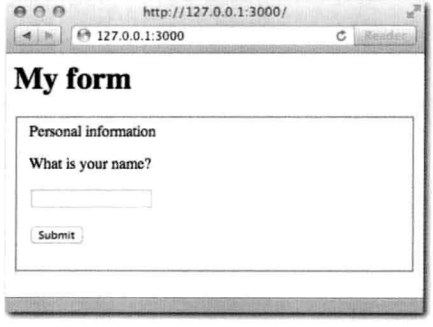


图7-7：渲染出的表单页面

你可以按下回车键试试。浏览器会发出一个新的请求（包含了相应数据），不过，因为现在所有的代码只用于输出表单的HTML，所以结果与图7-7看到的应该是一样的（见图7-8）。输入名字后，然后单击“Submit”（提交）按钮。

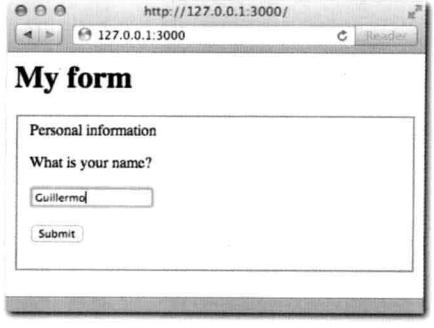


图7-8：表单提交的例子

提交后的结果是，URL变了，不过响应结果还是一样的，如图7-9所示。

97

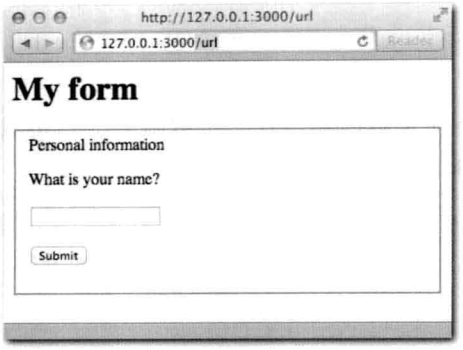


图7-9：尽管表单提交了，但Node还是以同样的方式对请求进行处理，这就是结果还是一样的原因

为了让Node能够对表单提交的请求做出正确的处理，我们需要学习关于检测请求方法和URL的相关内容。

method和URL

显然，在用户按下回车键（提交表单）后，我们得向用户展示些不同的东西，这个时候就需要处理表单。

在代码的最后，需要对请求对象上的`url`属性进行检测。server.js的代码如下所示：

```
require('http').createServer(function (req, res) {
  if ('/' == req.url) {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end([
        '<form method="POST" action="/url">'
      ,   '<h1>My form</h1>'
      ,   '<fieldset>'
      ,   '<label>Personal information</label>'
      ,   '<p>What is your name?</p>'
      ,   '<input type="text" name="name">'
      ,   '<p><button>Submit</button></p>'
      , '</form>'
    ].join(''));
  } else if ('/url' == req.url) {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end('You sent a  <em>' + req.method + '</em> request');
  }
}).listen(3000);
```

98

如果访问/ URL，就会看到如图7-10所示的页面，没有任何变化。

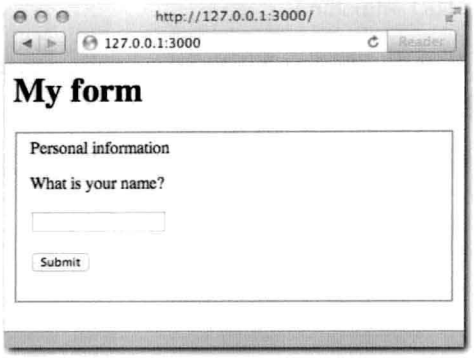


图7-10：当访问URL时，请求处理器仍旧显示同样的HTML内容

要是输入/url，就会看到如图7-11所示的页面。因为该URL匹配到了req.url在else if的这种情况，所以获得了这样的响应结果。

图7-11：当访问/url时，随着req.url值的变化导致了不同的响应结果

但是，当在表单中输入名字并提交后，就会看到如图7-12所示的页面。这是因为浏览器会通过<form>标签中action属性指定的HTTP方法将表单数据发送过去。于是，在本例中req.method值就是POST了，接着就会响应出如图7-12所示的内容了。

99

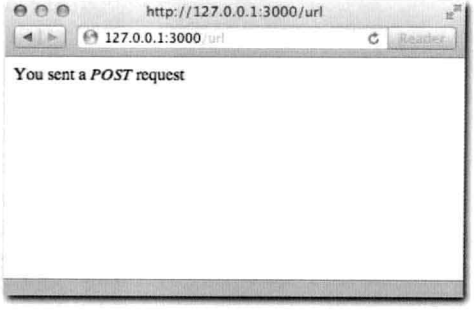


图7-12：本例中，req.method值为POST

如你所见，我们接触了请求对象的两个变量：URL和method。

Node.js会将主机名后所有的内容都放在url属性中。假设访问http://myhost.com/url?this+is+a+long+url，那么url的值就会是/url?this+is+a+long+url。

Web协议HTTP/1.1（你也许还记得第6章中介绍的telnet的例子），为请求定义了以下不同的方法：

- GET（默认）
- POST
- PUT
- DELETE
- PATCH（最新的）

定义这些不同方法的用意在于可以让HTTP客户端选择合适的方法，通过发送请求数据，修改服务器上的资源，该资源通过URL来指定。

数据

当发送HTML时，需要随着响应体定义Content-Type头信息。

100

和响应消息一样，请求消息也可以包含Content-Type头信息。为了更有效地处理表单，这两部分信息都是不可或缺的。就像在没有显式告知浏览器的情况下，浏览器不知道Hello World到底是HTML还是纯文本一样，我们怎样知道用户发送的用户名是JSON格式、XML格式，还是纯文本格式呢？server.js代码如下所示：

```
require('http').createServer(function (req, res) {
  if ('/' == req.url) {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end([
        '<form method="POST" action="/url">'
      ,   '<h1>My form</h1>'
      ,   '<fieldset>'
      ,   '<label>Personal information</label>'
      ,   '<p>What is your name?</p>'
      ,   '<input type="text" name="name">'
      ,   '<p><button>Submit</button></p>'
      , '</form>'
    ].join(''));
  } else if ('/url' == req.url && 'POST' == req.method) {
    var body = '';
    req.on('data', function (chunk) {
      body += chunk;
```

```
    });
    req.on('end', function () {
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end('<p>Content-Type: ' + req.headers['content-type'] + '</p>'
        + '<p>Data:</p><pre>' + body + '</pre>');
    });
  }
}).listen(3000);
```

上述代码有什么变化呢？我们监听了data和end事件。创建了一个body字符串用来接收数据块，仅当end事件触发时，我们就知道数据接收完全了。

之所以可以这样逐块接收数据，是因为Node.js允许在数据到达服务器时就可以对其进行处理。因为数据是以不同TCP包到达服务器的，这和现实情况也完全匹配，我们先获取一部分数据，然后在某个时刻再获取其余的数据。

再次提交表单，响应结果如图7-13所示。

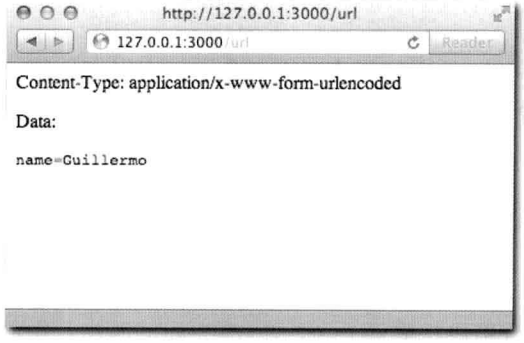


图7-13：本例中，我们将Content-Type内容以及请求的数据内容输出在页面上

举个例子，当使用Google进行搜索时，URL通常如图7-14所示。

图7-14：在搜索时，URL中高亮的部分为q=<search term>

注意，搜索部分的URL和表单内容一样都是经过编码的。这也解释了为什么Content-Type为urlencoded。

这部分URL片段又被称为查询字符串。

Node.js提供了一个称为querystring的模块，可以方便地对这类字符串进行解析，这样，我们就可以像处理头信息一样对其进行处理。我们来创建一个名为qs-example.js的文件，添加如下内容并运行（见图7-15）。

```
console.log(require('querystring').parse('name=Guillermo')); console.
 log(require('querystring').parse('q=guillermo+rauch'));
```

102

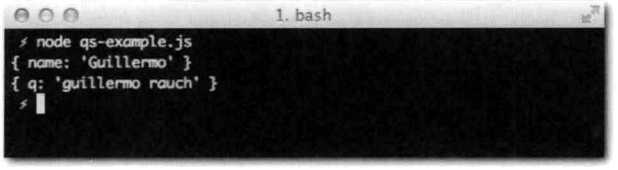


图7-15：调用parse函数后的结果

如图7-15所示，querystring模块将一个字符串解析成一个对象。这个解析处理方式和Node解析headers消息的方式类似，Node将HTTP请求数据中的headers信息从字符串解析成一个方便处理的headers对象。

接下来我们要使用querystring模块来获取提交表单的字段。

整合

现在我们要解析发送进来的数据并展示给用户。将server.js改为如下形式。注意了，这里我们在end事件中，使用querystring parse方法对请求内容进行解析，然后从解析生成的对象中获取name的值，并将其展示给用户。注意，这里的name是指<input>标签中的name值。我们来看一下截止到目前server.js的文件内容：

```
var qs = require('querystring');
require('http').createServer(function (req, res) {
   if ('/' == req.url) {
     res.writeHead(200, { 'Content-Type': 'text/html' });
     res.end([
         '<form method="POST" action="/url">'
       ,   '<h1>My form</h1>'
       ,   '<fieldset>'
       ,   '<label>Personal information</label>'
       ,   '<p>What is your name?</p>'
       ,   '<input type="text" name="name">'
       ,   '<p><button>Submit</button></p>'
```

```
      , '</form>'
    ].join(''));
  } else if ('/url' == req.url && 'POST' == req.method) {
    var body = '';
    req.on('data', function (chunk) {
      body += chunk;
    });
    req.on('end', function () {
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end('<p>Your name is <b>' + qs.parse(body).name + '</b></p>');
    });
  }
}).listen(3000);
```

接着打开浏览器访问，看到了吧（见图7-16）！ 103

图7-16：name字段值

让程序更健壮

这里还有一个问题：要是URL没有匹配到任何判断条件，怎么办？

要是我们访问/test URL，你会发现服务器端一直都没有响应，浏览器一直都处在挂起的状态。

要解决这个问题，我们可以在服务器不知道如何处理该请求时，发送404 (Not Found)状态码给客户端。注意在如下server.js代码中，我们添加了else的逻辑，并调用了writeHead写入404状态码。

```
var qs = require('querystring');
require('http').createServer(function (req, res) {
  if ('/' == req.url) {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end([
        '<form method="POST" action="/url">'
```

```
        ,   '<h1>My form</h1>'
        ,   '<fieldset>'
        ,   '<label>Personal information</label>'
        ,   '<p>What is your name?</p>'
        ,   '<input type="text" name="name">'
        ,   '<p><button>Submit</button></p>'
        , '</form>'
      ].join(''));
    } else if ('/url' == req.url && 'POST' == req.method) {
      var body = '';
      req.on('data', function (chunk) {
        body += chunk;
      });
      req.on('end', function () {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.end('<p>Your name is <b>' + qs.parse(body).name + '</b></p>');
      });
    } else {
      res.writeHead(404);
      res.end('Not Found');
    }
}).listen(3000);
```

104

好了，现在我们已经完成了第一个HTTP Web服务器了！尽管代码不是那么的整洁，不过不用担心，接下来的章节中会介绍如何更好地书写更复杂的HTTP服务器。

我们接下来介绍和服务器端API相对的HTTP客户端API。

一个Twitter Web客户端

学习如何使用Node.js向其他Web服务器发送请求是十分重要的。

HTTP已经演变成并非仅用于交换最终渲染、展示给用户的标记文本（如HTML），而且它还是服务器在不同网络环境传递数据的一种方式。同时，JSON因其语法衍生自JavaScrip线性对象，也快速成为了HTTP默认的标准数据格式，这也是Node.js在服务器端的优势之一。

在本例中，会介绍如何查询Twitter API，获取JSON数据，并解码为一种数据结构，以方便对其进行迭代后生成人类可读的形式。

创建模块

按照惯例，我们先从创建一个项目目录，并在该目录下创建一个`package.json`文件开始：

```
{
    "name": "tweet-client"
  , "description": "An HTTP tweets client"
```

```
    , "version": "0.0.1"
}
```

发送一个简单的HTTP请求

无独有偶，和我们创建的TCP客户端类似，我们通过http模块中的request静态方法创建一个Client对象。

为了让你更加熟悉，我们先回到此前典型的HTTP服务器：

```
require('http').createServer(function (req, res) {
  res.writeHead(200);
  res.end('Hello World');
}).listen(3000);
```

105

然后，写一个客户端来获取该响应，并将其在控制台以彩色的形式打印出来：

```
require('http').request({
    host: '127.0.0.1'
  , port: 3000
  , url: '/'
  , method: 'GET'
}, function (res) {
   var body = '';
   res.setEncoding('utf8');
   res.on('data', function (chunk) {
     body += chunk;
   });
   res.on('end', function () {
     console.log('\n  We got: \033[96m' + body + '\033[39m\n');
   });
 }).end();
```

上述代码中，首先调用了request方法。此方法用于初始化一个新的http.Client Request对象。

注意，我们收集信息块的方式和此前在服务器端收集客户端消息块的方式一样。连接的远程服务器会返回不同的数据块，我们需要将它们全部收集才能得到完整的响应。当然，也有可能所有的数据在一个 data事件中都到达了，不过我们无从得知。

所以，在本例中我们要监听end事件，然后将body数据打印到控制台。

除此之外，我们还通过响应对象上的setEncoding将编码设置为utf8，因为所有要输出到控制台的都是文本。你还可以试试用客户端下载一个PNG图片，这个时候用utf8来输出就不合适了。

下面，我们先运行服务器，然后再运行客户端（见图7-17）：

```
$ node client
```

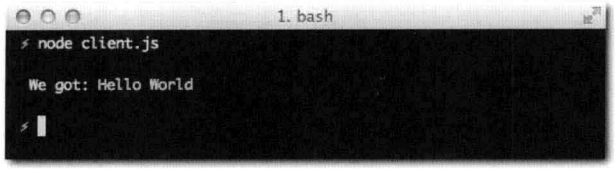


图7-17：从Hello World服务器返回的消息在客户端成功获取后显示了出来

106 下面，我们介绍如何通过请求发送数据。

发送数据

注意，在之前的例子中，调用完request之后，还需要调用end。

原因是在创建完一个请求之后，在发送给服务器前还可以和request对象进行交互。

比如，下面给你展示的就是这样一个发送数据给服务器的例子。

还记得之前我们在浏览器中创建的表单吗？这次仍旧创建该表单，不过不同的是，这次创建表单给客户端、使用Node，并且对于<form>，利用第5章学习的知识——stdin来处理。

服务器端处理表单：

```
var qs = require('querystring');
require('http').createServer(function (req, res) {
  var body = '';
  req.on('data', function (chunk) {
    body += chunk;
  });
  req.on('end', function () {
    res.writeHead(200);
    res.end('Done');
    console.log('\n  got name \033[90m' + qs.parse(body).name + '\033[39m\n');
 });
}).listen(3000);
```

客户端也要做对应的处理。通过使用querystring模块的stringify方法，可以将一个对象转化为url编码过的数据：

```
var http = require('http')
  , qs = require('querystring')

function send (theName) {
  http.request({
    host: '127.0.0.1'    , port: 3000
   , url: '/'
   , method: 'POST'
  }, function (res) {
```

```
        res.setEncoding('utf8');
        res.on('end', function () {
          console.log('\n  \033[90m  request complete!\033[39m');
process.stdout.write('\n  your name: ');
        });
  }).end(qs.stringify({ name: theName }));
}

process.stdout.write('\n  your name: ');
process.stdin.resume(); process.stdin.setEncoding('utf-8');
process.stdin.on('data', function (name) {
  send(name.replace('\n', ''));

});
```

107

注意在上述代码中，数据是通过end方法发送的，和我们在服务器端创建响应消息方式一样。

上述代码中，无须担心从服务器端获取数据块的问题。你知道，当end事件触发时，就可以将完整的请求数据打印出来，然后接着要求用户再次输入数据即可。

图7-18一步步展示了具体操作。在左侧，服务器端会显示由右侧提交的通过stdin输入的名字信息。

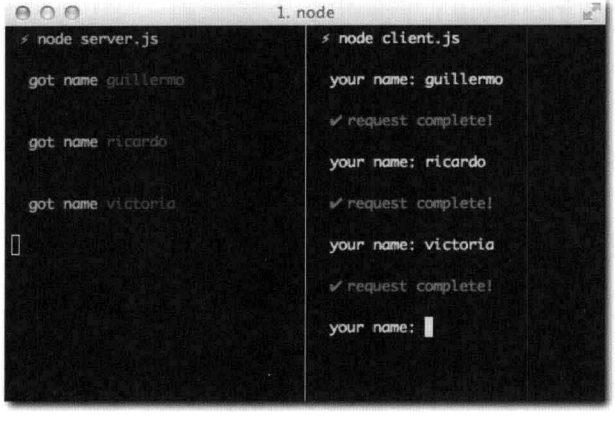


图7-18：当右侧提示输入名字时，我输入之后按下回车键，随后该名字就会在左侧服务器端显示出来

至此，你已经学会了如何通过请求发送数据了，已经了解了几乎全部请求API的用法。下面，让我们继续去完成目标！

获取推文

下面来点实际有用的东西！我们来创建一个tweets命令，该命令接受一个搜索参数，然后将最近相关话题的推文展示出来。

要是你看过Twitter开放的公共搜索API文档，你会发现其URL是这样的：`http://search.twitter.com/search.json?q=blue+angels`。

搜索结果数据如下所示（注意，这里我把后面的省略了）：

108

```
{
   "completed_in":0.031,
   "max_id":122078461840982016,
   "max_id_str":"122078461840982016",
   "next_page":"?page=2&max_id=122078461840982016&q=blue%20angels&rpp=5",
   "page":1,
   "query":"blue+angels",
   "refresh_url":"?since_id=122078461840982016&q=blue%20angels",
   "results":[     {
// …
```

同样的：搜索关键字也是url编码过的（q=blue+angels），并且结果是JSON数据格式。推文数据在响应结果对象中的`results`数组中。

由于我们要支持命令接收参数，就像第5章介绍的那样，我们需要访问`argv`。通过使用`querystring`模块，可以生成搜索URL，随后发送请求获取响应数据。这里访问资源请求的`method`就很明显是`GET`了，端口则是80，不过这些都是默认的，所以可以省略（为了让这首个HTTP客户端例子更清楚，我还是加上了`GET`选项）。

```
var qs = require('querystring')
  , http = require('http')

var search = process.argv.slice(2).join(' ').trim()

if (!search.length) {
  return console.log('\n  Usage: node tweets <search term>\n');
}
console.log('\n  searching for: \033[96m' + search + '\033[39m\n')
http.request({
    host: 'search.twitter.com'
  , path: '/search.json?' + qs.stringify({ q: search })
}, function (res) {
  var body = '';
  res.setEncoding('utf8');
  res.on('data', function (chunk) {
    body += chunk;
  });
  res.on('end', function () {
    var obj = JSON.parse(body);
    obj.results.forEach(function (tweet) {
      console.log('  \033[90m' + tweet.text + '\033[39m');
      console.log('  \033[94m' + tweet.from_user + '\033[39m');
      console.log('--');
```

```
    });
  });
}).end()
```

运行上述代码，会校验process.argv数组，看是否提供了搜索关键字（见图7-19），要是没有提供则会显示帮助信息。

109

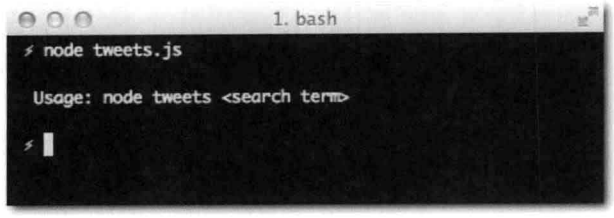


图7-19：不带关键字运行命令

当提供了搜索关键字时，就会执行搜索操作，如图7-20所示。Twitter会响应对应的JSON数据，在end事件处理器中会对该数据进行迭代，并将结果显示给用户。

图7-20：本例中，我搜索 Justin Bieber，相关有趣的推文就显示出来了

至此，我们一直在频繁使用http.request，创建了可以说是最常见的GET请求。Web服务通常也会提供相比POST、PUT而言更多的GET服务。随着请求发送数据（一个请求体）也是相对不怎么常见的。

Node.js也为发送最常见的请求提供了便利，它提供了request.get方法。调用Twitter API（使用http.request）的代码可以重写为如下形式：

```
http.get({
    host: 'search.twitter.com'
  , path: '/search.json?' + qs.stringify({ q: search })
}, function (res) {
  var body = '';
  res.setEncoding('utf8');
  res.on('data', function (chunk) {
    body += chunk;
  });
  res.on('end', function () {
    var obj = JSON.parse(body);
    obj.results.forEach(function (tweet) {
      console.log('  \033[90m' + tweet.text + '\033[39m');
      console.log('  \033[94m' + tweet.from_user + '\033[39m');
      console.log('--');
   });
  });
})
```

110

唯一本质的不同就是这种方式就无须调用end方法了，并且从语义上更显然能够看出是要获取数据。因为API要接收一个method参数，其默认值是GET，所以这种方式更简单有效。

尽管做了上述改进，但我们还是有很多冗余代码。接下来会介绍一款工具superagent，它是基于HTTP客户端API的更高层封装，可以让上述这些处理变得更加容易。

superagent来拯救

HTTP客户端往往都会有一些共性：获取所有响应数据，根据响应消息的Content-Type值进行数据解析，处理消息数据。

当向服务器发送数据时，情况也是类似的。我们会创建一个POST请求，然后将要发送的数据对象编码为JSON格式。

有一个叫做superagent的模块通过扩展response对象，为其添加一些有用的扩展来解决上述问题，其中部分扩展功能下面会进行介绍。

本例中，我们使用superagent 0.3.0版本。创建一个新的目录，然后本地安装superagent：

```
$ npm install superagent@0.3.0
```

通过request获取数据，如果服务器响应了正确的Content-Type并且表明响应数据是JSON格式，那么superagent就会自动将其进行缓存并解析，然后将数据放在res.body中。我们来创建一个名为tweet.js的文件，并将下述内容添加到该文件中：

```
var request = require('superagent');
request.get('http://twitter.com/search.json')
```

```
  .send({ q: 'justin bieber' })
  .end(function (res) { console.log(res.body); });
```

运行上述文件，就能看到从响应的JSON数据解码成的对象形式的数据。通过res.text还是可以获取到原始的响应文本。

注意，对于查询字符串也无须进行手动编码，因为，superagent知道，当发送一个GET请求，并且要发送数据时，就需要将其编码为查询字符串来作为URL一部分。 111

要设置请求头信息，可以调用set方法。如下例子，我通过set方法设置了请求的Date头信息：

```
var request = require('superagent');
request.get('http://twitter.com/search.json')
  .send({ q: 'justin bieber' })
  .set('Date, new Date)
  .end(function (res) { console.log(res.body); });
```

send和set方法均可被调用多次，并且均为渐进式API：可以进行链式调用，并最后通过end方法来结束。

这类API不仅仅只用于GET请求。类似的，superagent还提供了put、post、head以及del方法。

下面的例子POST一个JSON编码的对象：

```
var request = require('superagent');
request.post('http://example.com/')
  .send({ json: 'encoded' })
  .end();
```

JSON是默认的编码格式。要更改，只需简单地调用set方法来更改请求的Content-Type值即可。

使用up重启HTTP服务器

至此，或许你已经意识到，每次修改服务器端的代码后，为了让通过浏览器访问能够看到修改后的效果，都要手动去重启服务器，是件很让人懊恼的事情。

最直接的解决这个问题的方法就是一旦发现代码有改动就去重启该进程。这在开发模式下的确很好，不过，一旦将Web服务器部署到生产环境之后，就得要确保正在执行的请求（换句话说，即当要重启进程时，还处在处理过程中的请求）不能被杀掉。

为此，我开发了一个名为up的工具，以一种安全可靠的方式来解决这个问题。开发模式下，通过NPM安装即可：

```
$ npm install -g up
```

112 接下来，需要确保代码结构必须要将Node HTTP服务器暴露出来，而不是调用listen来启动。这是因为up会调用listen方法，并且它需要访问服务器实例。举个例子，创建一个新目录，将下述内容添加到server.js文件中：

```
module.exports = require('http').createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end('Hello <b>World</b>');
});
```

cd到该目录，通过up命令，同时传递--watch和--port选项来运行server.js文件：

```
$ up -watch -port 80 server.js
```

--watch意味着up会通过Node API来监听该目录下所有文件的更改。先试着通过浏览器访问该服务器，然后编辑server.js文件，更改Hello <b>World<b>文本。一旦保存修改的文件后，刷新浏览器就能立刻看到修改后的结果了！

小结

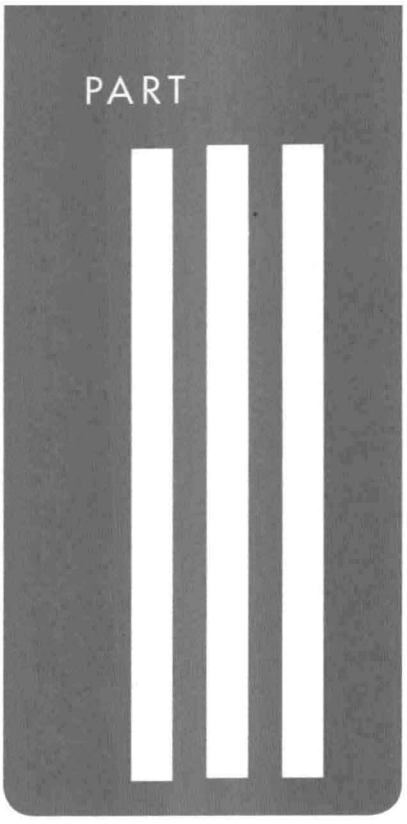
本章介绍了如何使用Node来书写HTTP服务器，本章从介绍基于TCP协议的HTTP协议基础开始。

然后，通过一个Hello World的例子展示了Node.js产生的默认响应结果。其中有默认的头信息，并介绍了为什么会有这些头信息。

之后介绍了HTTP请求中这些头信息的重要性，以及如何在服务器端响应中修改默认的头信息。还介绍了用于浏览器和服务器之间消息传递的编码，以及Node提供了哪些工具来对数据进行解析和处理。

成功完成了一个Web服务器之后，我们还介绍了Node客户端API，这对于在现代Web时代，与Web服务进行交互是非常有用的。再接着，我们在介绍了一些常规的用例后，成功地书写了一个查询Twitter API的客户端，不过在过程中也发现了很多代码都写重复了。为此，我们又介绍了一个核心Node.js之上的API来简化代码的书写。下一章要介绍的Connect模块也是类似的解决方案，它有很多值得学习的地方。

最后，介绍了up，一个命令行工具（同时也提供了JavaScript API），用它能够让开发过程变得更加简单，每次修改代码up就会自动重启服务器进程来让修改立即生效。要谨记：要使用up，必须确保模块要将通过调用createServer返回的http.Server实例暴露出来。



Web开发

CHAPTER

8 Connect

Node.js为常规的网络应用提供了基本的API。至此，你也已经了解了其为TCP服务器和基于此的HTTP服务器所提供的基本API了。 115

然而，实际情况下，绝大部分网络应用都需要完成一系列类似的操作，这些类似的操作你一定不想每次都重复地基于原始的API去实现。

Connect是一个基于HTTP服务器的工具集，它提供了一种新的组织代码的方式来与请求、响应对象进行交互，称为中间件（middleware）。

为了证明通过中间件进行代码复用的好处，假设我们有一个站点，其目录结构如下所示：

```
$ ls website/
index.html  images/
```

在images目录下，有四个图片文件：

```
$ ls website/images/
1.jpg  2.jpg  3.jpg  4.jpg
```

index.html简单地展示了这四张图片，并且可以通过http://localhost来访问（见图8-1）：

```
<h1>My website</h1>

<img src="/images/1.jpg">
<img src="/images/2.jpg">
<img src="/images/3.jpg">
<img src="/images/4.jpg">
```

116

图8-1：一个简单的静态网站，展示了Connect的能力

为了展示Connect为HTTP应用提供的便利，本章会介绍如何使用原生的`http` API书写一个简单的网站，之后再介绍如何使用`connect` API完成同样的事情。

使用HTTP构建一个简单的网站

按照惯例，我们先引入`http`模块用于创建服务器以及`fs`模块，用来读取文件：

```
/**
 * 模块依赖
 */

var http = require('http')
  , fs = require('fs')
```

接着，初始化服务器并处理请求—响应：

```
/**
 * 创建服务器
 */

var server = http.createServer(function (req, res) {
  // …
});
```

最后监听：

```
/**
 * 监听
 */

server.listen(3000);
```

117

回到createServer的回调函数。我们需要检查URL是否和服务器目录下的文件匹配，如果匹配，则读取该文件并展示出来。以后，可能会添加更多的图片，所以要确保足够灵活以支持这种情况。

首先要检查请求方法是GET并且URL以/images开始、.jpg结束。如果URL为'/'的话，则响应index.html（调用后面要实现的serve函数）。否则，发送404 Not Found（404未找到状态码），代码如下所示：

```
if ('GET' == req.method && '/images' == req.url.substr(0, 7)
  && '.jpg' == req.url.substr(-4)) {
  // …
} else if ('GET' == req.method && '/' == req.url) {
  serve(__dirname + '/index.html', 'text/html');
} else {
  res.writeHead(404);
  res.end('Not found');
}
```

接着使用fs.stat来检查文件是否存在。这里使用Node中的全局常量__dirname来获取当前服务器所在的路径。在首个if语句后，添加如下代码：

```
fs.stat(__dirname + req.url, function (err, stat) {

});
```

这里不使用同步版本的fs.stat（fs.statSync）。否则当处理磁盘文件时，会阻塞其他请求的处理，这是处理高并发的服务器的大忌。关于这些，我们早在第3章中就讨论过。

如果检查文件是否存在时发生错误，则终止进程并发送HTTP 404状态码告知无法找到请求的图片。对于stat成功但是路径所表示的并非是文件时，也要做此处理。如下所示的代码在fs.stat回调函数中。

```
if (err || !stat.isFile()) {
  res.writeHead(404);
  res.end('Not Found');
  return;
}
```

否则，就返回图片信息。下面这行代码紧随上述if之后：

118

```
serve(__dirname + req.url, 'application/jpg');
```

最后，我们完成serve方法，也许你已经猜到了，这个方法就是根据文件路径来获取文件内容，并添加必不可少的'Content-Type'头信息，如下所示，该信息告诉浏览器你发送的是什么类型的资源：

```
function serve (path, type) {
  res.writeHead(200, { 'Content-Type': type });
  fs.createReadStream(path).pipe(res);
}
```

还记得第6章中介绍的流（Stream）吗？HTTP响应对象是一个只写流。从文件创建出来的流是只读的。同时，可以将文件系统流接（pipe）到HTTP响应流中！上述简洁的代码其实就等同于如下这段代码：

```
fs.createReadStream(path)
  .on('data, function (data) {
    res.write(data);
  })
  .on('end', function () {
    res.end();
  })
```

这也是最有效的，推荐被用来实现静态文件托管功能的方法。

把所有代码整合起来，就变成了：

```
/**
 * 模块依赖
 */

var http = require('http')
  , fs = require('fs')

/**
 *创建服务器
 */

var server = http.createServer(function (req, res) {
  if ('GET' == req.method && '/images' == req.url.substr(0, 7)
    && '.jpg' == req.url.substr(-4)) {
    fs.stat(__dirname + req.url, function (err, stat) {
      if (err || !stat.isFile()) {
        res.writeHead(404);
        res.end('Not Found');
        return;
      }
      serve(__dirname + req.url, 'application/jpg');
    });
  } else if ('GET' == req.method && '/' == req.url) {
```

```
    serve(__dirname + '/index.html', 'text/html');
  } else {
    res.writeHead(404);
    res.end('Not found');
  }

  function serve (path, type) {
    res.writeHead(200, { 'Content-Type': type });
    fs.createReadStream(path).pipe(res);
  }
});

/**
 * 监听
 */

server.listen(3000);
```

完成！接下来就运行上述代码：

```
$ node server
```

然后通过浏览器访问http://127.0.0.1:3000，就能看到实现的网站了！

通过Connect实现一个简单的网站

接下来这个例子是要实现一个网站，该例子展示了创建网站时一些常见的任务：

- 托管静态文件。
- 处理错误以及损坏或者不存在的URL。
- 处理不同类型的请求。

基于http模块API之上的Connect，提供了一些工具方法能够让这些重复性的处理便于实现，以至于让开发者能够更加专注在应用本身。它很好地体现了DRY模式：不要重复自己（Don't Repeat Yourself）。

归功于Connect，本例实现起来会非常简单。首先在新目录下创建一 个package.json文件，并在其中声明对"connect"模块的依赖。

```
{
    "name": "my-website"
  , "version": "0.0.1"
  , "dependencies": {
        "connect": "1.8.7"
    }
}
```

接着，安装依赖：

```
$ npm install
```

然后，引入connect模块：

```
/**
 * 模块依赖
 */

var connect = require('connect')
```

通过Connect创建http.Server：

```
/**
 * 创建服务器
 */

var server = connect.createServer();
```

使用use()方法来添加static中间件。下一部分会介绍中间件的概念，并在下一章会对其做深入讲解。现在，最为重要的是要理解，中间件其实就是一个简单的JavaScript函数。本例中，我们这样配置static中间件——通过传递一些参数给connect.static方法，该方法本身会返回一个方法。

```
/**
 * 处理静态文件
 */

server.use(connect.static(__dirname + '/website'));
```

将index.html以及images目录放到/website下，确保没有将不想托管的文件放进去。

接着，调用listen()方法：

```
/**
 * 监听
 */

server.listen(3000);
```

完成！事实上，Connect还能处理404的情况，你可以通过访问/made-up-url来验证。

121 中间件

为了更好地理解中间件，我们回到node HTTP的例子。移除逻辑，我们来关注如下部分：

```
if ('GET' == req.method && '/images' == req.url.substr(0, 7)) {
  // serve image
} else if ('GET' == req.method && '/' == req.url) {
```

```
    // serve index
  } else {
    // display 404
  }
```

如上述代码所示，针对每个请求应用都只会做这三件事情中的一件。比如，若还要记录请求日志，就在顶部加上如下代码：

```
console.error(' %s %s ', req.method, req.url);
```

下面，我们来设计一个更大型的应用，它能够根据每个请求的不同情况处理以下这几种不同的任务：

- 记录请求处理时间。
- 托管静态文件。
- 处理授权。

当然，这些任务的处理代码都可以放在一个简单的事件处理器中（createServer的回调函数中），这将会是一个非常复杂的处理过程。

简单来说，中间件由函数组成，它除了处理req和res对象之外，还接收一个next函数来做流控制。

若用中间件模式来书写满足上述要求的应用，会是这样的：

```
server.use(function (req, res, next) {
  //记录日志
  console.error(' %s %s ', req.method, req.url);
  next();
});

server.use(function (req, res, next) {
  if ('GET' == req.method && '/images' == req.url.substr(0, 7)) {
    // 托管图片
  } else {
    // 交给其他的中间件去处理
    next();
  }
});

server.use(function (req, res, next) {
  if ('GET' == req.method && '/' == req.url) {
    // 响应index文件
  } else {
    // 交给其他中间件去处理
    next();
  }
```

122

```
});

server.use(function (req, res, next) {
  //最后一个中间件，如果到了这里，就意味着无能为力，只能返回404了
  res.writeHead(404);
  res.end('Not found');
});
```

使用中间件，不仅能够让代码有更强大的表达能力（将应用拆分为更小单元的能力），还能够实现很好的重用性。我们马上就会看到，Connect已经为处理常见的任务提供了对应的中间件。例如，要对请求进行日志记录，就可以简单地通过如下一行代码完成：

```
app.use(connect.logger('dev'))
```

帮你完成日志记录！

下一部分会介绍如何书写一个当请求处理时间过长而进行警告的中间件。

书写可重用的中间件

一个用于当请求时间过长而进行提醒的中间件在很多场景下都非常有用。比如，假设有个页面会向数据库发起一系列的请求。在测试过程中，所有响应都在100毫秒（ms）内完成，但是你要确保能够将响应时间大于100ms的请求记录下来。

为此，我们在一个名为`request-time.js`的独立模块中创建一个中间件。

这个模块暴露一个函数，此函数本身又返回一个函数。这对于可配置的中间件来说是很常见的写法。在前面的例子中，我们调用`connect.logger`时传递了一个参数，然后它自身会返回一个函数，最终用来处理请求。

目前，模块就接收一个超时时间阈值选项，该选项用来界定什么时候该将其记录下来。

123

```
/**
 * 请求时间中间件
 *
 * 选项:
 *    - 'time' ('Number'): 超时阈值（默认100）
 *
 * @param {Object} options
 * @api public
 */

module.exports = function (opts) {
  // …
};
```

首先，将默认阈值设为100：

```
var time = opts.time || 100;
```

随后，返回一个中间件函数：

```
return function (req, res, next) {
```

中间件本身创建一个计时器，并在指定时间内触发：

```
var timer = setTimeout(function () {
  console.log(
      '\033[90m%s %s\033[39m \033[91mis taking too long!\033[39m'
    , req.method
    , req.url
  );
}, time);
```

这里要确保当响应时间在100ms以内时要清除（停下来或者取消）计时器。另外一个在中间件中常用的模式叫做重写方法（也叫猴子补丁（monkey-patch）），能够在其他中间件调用它时，执行指定的行为。

在本例中，当响应结束时，我们需要清除计时器：

```
var end = res.end;
res.end = function (chunk, encoding) {
  res.end = end;
  res.end(chunk, encoding);
  clearTimeout(timer);
};
```

首先保持对原始函数的引用（`var end = res.end`）。然后，在重写的函数中，再恢复原始函数，并调用它，最后清除计时器。

最后，总是要让其他中间件能够处理请求，所以得调用`next`。否则，程序不会做任何事情！ 124

```
next();
```

完整版的中间件代码如下所示：

```
/**
 * 请求时间中间件
 *
 * 选项：
 *    - 'time' ('Number'): 超时阈值（默认100）
 *
 * @param {Object} options
 * @api public
 */
module.exports = function (opts) {
  var time = opts.time || 100;
```

```
  return function (req, res, next) {
    var timer = setTimeout(function () {
      console.log(
          '\033[90m%s %s\033[39m \033[91mis taking too long!\033[39m'
        , req.method
        , req.url
      );
    }, time);

    var end = res.end;
    res.end = function (chunk, encoding) {
      res.end = end;
      res.end(chunk, encoding);
      clearTimeout(timer);
    };
    next();
  };
};
```

为了测试上述例子，我们需要创建一个Connect应用，并创建两条路由：第一条很快得到响应，另外一条1秒后得到响应。

我们先来引入依赖的模块：

```
# sample.js
/**
 * 模块依赖
 */

var connect = require('connect')
  , time = require('./request-time')
```

125

接着，创建服务器：

```
/**
 * 创建服务器
 */

var server = connect.createServer();
```

记录请求情况：

```
/**
 * 记录请求情况
 */

server.use(connect.logger('dev'));
```

实现中间件：

```
/**
 * 实现时间中间件
 */

server.use(time({ time: 500 }));
```

实现快速响应：

```
/**
 * 快速响应
 */

server.use(function (req, res, next) {
  if ('/a' == req.url) {
    res.writeHead(200);
    res.end('Fast!');
  } else {
    next();
  }
});
```

实现模拟的慢速响应：

```
/**
 * 慢速响应
 */

server.use(function (req, res, next) {
  if ('/b' == req.url) {
    setTimeout(function () {
      res.writeHead(200);
      res.end('Slow!');
    }, 1000);
  } else {
    next();
  }
});
```

126

服务器监听端口：

```
/**
 * 监听
 */

server.listen(3000);
```

运行服务器：

```
$ node server
```

用浏览器访问http://localhost:3000/a（快速响应），如图8-2所示。

图8-2：一个简单的路由（/a）结果显示在浏览器中

要是看下控制台，就能看到logger中间件在工作了，如图8-3所示。

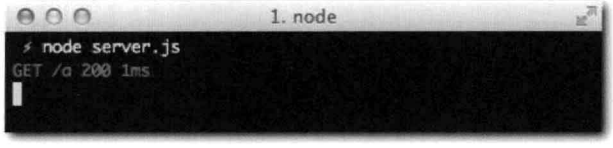


图8-3：在处理路由/a时，显示在控制台的日志

127

图8-4显示了慢速响应的结果（/b）。

图8-4：慢速响应/b结果显示在浏览器中

图8-5显示了对应的控制台信息。

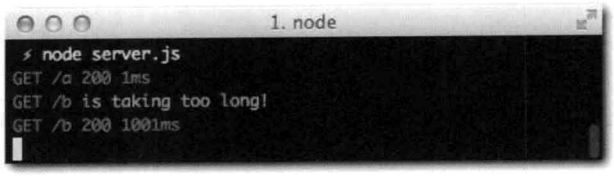


图8-5：处理/b后，控制台输出了警告日志

接下来，我们会介绍Connect内置的一些中间件，这些中间件在Web应用中都是非常常用的。

static中间件

static中间件可以算得上是使用Node开发Web应用时最常用的中间件之一了。

挂载

Connect允许中间件挂载到URL上。比如，static允许将任意一个URL匹配到文件系统中任意一个目录。

举例来说，假设要让/my-images URL和名为/images的目录对应起来，可以通过如下方式进行挂载：

```
server.use('/my-images', connect.static('/path/to/images'));
```

maxAge

128

static中间件接收一个名为maxAge的选项，这个选项代表一个资源在客户端缓存的时间。这非常实用，特别是对一些不经常改动的资源来说，浏览器就无须每次都去请求它了。

比如，一种Web应用常见的实践方式就是将所有的客户端JavaScript文件都合并到一个文件中，并在其文件名中加上修订号。这个时候，就可以设置maxAge选项，让其永远缓存起来。

```
server.use('/js', connect.static('/path/to/bundles', { maxAge: 10000000000000 }));
```

hidden

另一个static接收的参数是hidden。如果该选项为true，Connect就会托管那些文件名以点（.）开始的在UNIX文件系统中被认为是隐藏的文件：

```
server.use(connect.static('/path/to/resources', { hidden: true }));
```

query中间件

有时要发送给服务器的数据会以查询字符串形式，作为URL的一部分。

比如，`url /blog-posts?page=5`。当在浏览器中访问该URl时，Node会以字符串的形式将该URL存储到`req.url`变量中：

```
server.use(function (req) {
  // req.url == "/blog-posts?page=5"
});
```

而绝大多数情况下，真正想要获取的其实是查询字符串这部分的数据。

使用query中间件，就能够通过`req.query`对象自动获取这些数据：

```
server.use(connect.query);
server.use(function (req, res) {
  // req.query.page == "5"
});
```

同样的，解析查询字符串也是应用常见的需求，Connect也对其做了简化。就像刚刚提到的static中间件，你无须关心使用require去引入fs模块的问题一样，你也无须去担心使用querystring模块的问题。

query中间件在Express中默认就是启用的，Express是一个Web框架，我们会在下一章中做介绍。接下来要介绍另一个很实用的中间件——logger。

129 ### logger中间件

logger中间件是一个对Web应用非常有用的诊断工具。它将发送进来的请求信息和发送出去的响应信息打印在终端。

它提供了以下四种日志格式：

- default
- dev
- short
- tiny

比如，要使用dev日志格式，可以通过如下初始化logger中间件的方式：

```
server.use(connect.logger('dev'));
```

看下面这个例子，一个典型的使用了logger中间件的“Hello World”HTTP 服务器：

```
var connect = require('connect');
connect.createServer(
    connect.logger('dev')
  , function (req, res) {
      res.writeHead(200);
      res.end('Hello world');
    }
).listen(3000);
```

注意，在上述代码中，我传递了一系列中间件函数作为createServer的参数。这其实就等同于初始化Connect服务器时调用use两次。

当通过浏览器访问http://127.0.0.1:3000时，就能看到如下两行输出信息：

```
GET / 200 0ms
GET /favicon.ico 200 2ms
```

上述信息表示浏览器正在请求/favicon.ico和/，connect logger中间件将请求的方法、响应状态码以及处理时间输出。

dev是一种精准简短的日志格式，能够提供行为以及性能方面的信息，方便测试Web应用。

logger中间件还允许自定义日志输出格式。

比如，若只想记录请求方法和IP地址：

130

```
server.use(connect.logger(':method :remote-addr'));
```

另外，还能够通过动态的req和res来记录头信息。要记录响应的content-length和content-type信息，可以通过如下方式：

```
server.use(connect.logger('type is :res[content-type], length is '
  + ':res[content-length] and it took :response-time ms.'));
```

注意：记住，在Node中，请求/响应头都是小写的。

要是把上述代码应用到请求四个图片的例子，就能看到很有趣的输出结果，如图8-6所示。（确保在没有缓存的情况下。）

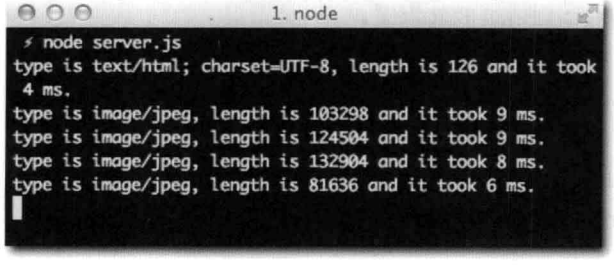


图8-6：自定义日志格式实践

下面是完整的可用token：

- :req[header]（如:req[Accept]）
- :res[header]（如:res[Content-Length]）
- :http-version
- :response-time
- :remote-addr
- :date
- :method
- :url
- :referrer
- :user-agent
- :status

131 logger还能够自定义token。比如，要给请求的Content-Type定义一个简写的:type token，就可以采用如下方式：

```
connect.logger.token('type', function (req, res) {
  return req.headers['content-type'];
});
```

接下来要介绍的是body parse中间件，它可以自动帮你完成另一件开发Web应用常见的任务。

body parser中间件

在一个使用http模块的例子中，我们用了qs模块来解析POST请求的消息体。

Connect也对这部分提供了帮助！它提供了bodyParser中间件：

```
server.use(connect.bodyParser());
```

然后，就能够在req.body中获取POST请求的数据了：

```
server.use(function (req, res) {
  // req.body.myinput
});
```

如果客户端在POST请求中使用了JSON格式，那么req.body也会对应地转换为JSON对象，因为bodyParser会检测Content-Type的值。

处理上传

bodyParser另一个功能就是使用formidable模块，它可以让你处理用户上传的文件。

本例中，我们可以使用createServer的快捷方式来创建一个服务器，并将所有要用的中间件都传递给它：

```
var server = connect(
    connect.bodyParser()
  , connect.static('static')
);
```

在static/文件夹中，我们创建一个包含用来处理文件上传表单的index.html文件：

```
<form action="/" method="POST" enctype="multipart/form-data">
  <input type="file" />
  <button>Send file!</button>
</form>
```

接着，我们可以添加一个简单的中间件来看一下req.body.file值是什么样子的：

132

```
function (req, res, next) {
  if ('POST' == req.method) {
    console.log(req.body.file);
  } else {
    next();
  }
}
```

到测试时间了！如图8-7所示，上传一个Hello.txt文件来测试一下。

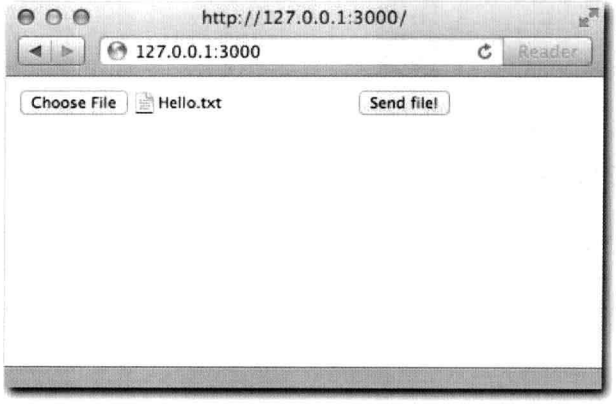


图8-7：从浏览器上传一个示例文本文件

接着，我们看看服务端的输出结果，如图8-8所示。

如图8-8中所示，我们获取到一个包含了许多有用的上传信息的对象。接着，我们把上传

的文件返回给用户：

```
if ('POST' == req.method && req.body.file) {
  fs.readFile(req.body.file.path, 'utf8', function (err, data) {
    if (err) {
      res.writeHead(500);
      res.end('Error!');
      return;
    }

    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end([
        '<h3>File: ' + req.body.file.name + '</h3>'
      , '<h4>Type: ' + req.body.file.type + '</h4>'
      , '<h4>Contents:</h4><pre>' + data + '</pre>'
    ].join(''));
```

133

```
  });
} else {
  next();
}
```

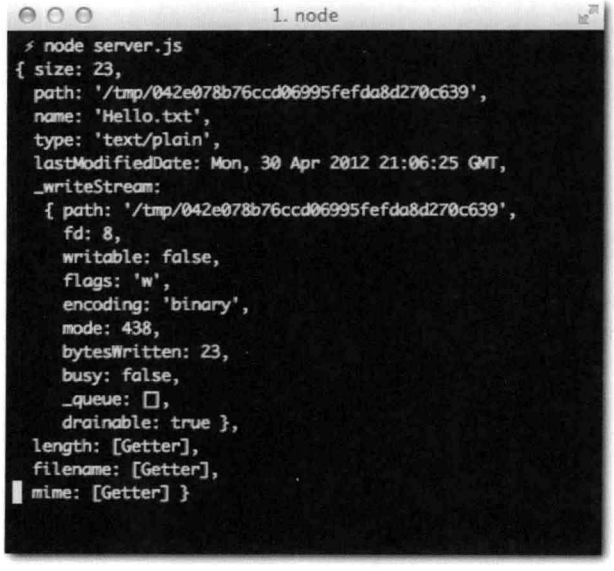


图8-8： 在req.body包含的上传文件对象，显示在控制台中

再次上传该文件来看看返回结果（见图8-9）。

图8-9：上传成功后，示例文件Hello.txt内容显示在浏览器中

多文件上传

134

要处理多文件上传也很简单，只要在input的name属性上加上[]即可：

```
<input type="file" name="files[]" />
<input type="file" name="files[]" />
```

req.body.files就包含了一个数组，数组中的元素就是此前单个Hello.txt示例的对象。

cookie

除了query之外，Connect还为读写cookie提供了便利。

当浏览器发送cookie数据时，会将其写到Cookie头信息中。其数据格式和URL中的查询字符串类似。我们来看下面这个包含了该头信息的请求：

```
GET /secret HTTP/1.1
Host: www.mywebsite.org
Cookie: secret1=value; secret2=value2
Accept: */*
```

要访问这些值（secret1和secret2），但又不想手动去解析，也不想使用正则表达式去抽取的话，就可以使用cookieParser中间件：

```
server.use(connect.cookieParser())
```

然后，就可以通过req.cookies对象来访问这些cookie数据了：

```
server.use(function (req, res, next) {
  // req.cookies.secret1 = "value"
  // req.cookies.secret2 = "value2"
})
```

会话（session）

在绝大多数Web应用中，多个请求间共享“用户会话”的概念是非常必要的。“登录”一个网站时，多多少少会使用某种形式的会话系统，它主要通过在浏览器中设置cookie来实现，该cookie信息会在随后所有的请求头信息中被带回到服务器。

Connect为此也提供了简单的实现方式。作为例子，我们创建了一个简单的登录系统。我们把用户凭证信息存放在一个名为users.json的文件中：

135

```
{
    "tobi": {
        "password": "ferret"
      , "name": "Tobi Holowaychuk"
    }
}
```

首先，通过require载入Connect和users文件：

```
/**
 * 模块依赖
 */

var connect = require('connect')
  , users = require('./users')
```

注意，这里直接require了JSON文件！当你只是要对外暴露数据时，就不需要加上module.exports，直接把数据文件以JSON形式暴露出来就好了。

接着，我们加上logger、bodyParser和session中间件。由于session中间件需要操作cookie，所以在这之前先要引入cookieParser中间件：

```
var server = connect(
    connect.logger('dev')
  , connect.bodyParser()
  , connect.cookieParser()
  , connect.session({ secret: 'my app secret' })
```

出于安全性的考虑，在初始化session中间件的时候需要提供secret选项。

接下来，我们首先检查用户是否已经登录。如果没有登录，则交给其他中间件处理：

```
, function (req, res, next) {
    if ('/' == req.url && req.session.logged_in) {
      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end(
          'Welcome back, <b>' + req.session.name + '</b>. '
        + '<a href="/logout">Logout</a>'
      );
    } else {
```

```
    next();
  }
}
```

紧接着的中间件，展示一个登录表单：

136

```
, function (req, res, next) {
  if ('/' == req.url && 'GET' == req.method) {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end([
        '<form action="/login" method="POST">'
      ,   '<fieldset>'
      ,     '<legend>Please log in</legend>'
      ,     '<p>User: <input type="text" name="user"></p>'
      ,     '<p>Password: <input type="password" name="password"></p>'
      ,     '<button>Submit</button>'
      ,   '</fieldset>'
      , '</form>'
    ].join(''));
  } else {
    next();
  }
}
```

再后面的中间件检查登录表单的信息是否与用户凭证匹配，匹配则认为登录成功：

```
, function (req, res, next) {
  if ('/login' == req.url && 'POST' == req.method) {
    res.writeHead(200);
    if (!users[req.body.user] || req.body.password != users[req.body.user].
password) {
      res.end('Bad username/password');
    } else {
      req.session.logged_in = true;
      req.session.name = users[req.body.user].name;
      res.end('Authenticated!');
    }
  } else {
    next();
  }
}
```

注意在上述代码中，修改了一个名为`req.session`的对象。该对象在响应发送出去时就会自动保存，无须我们手动处理。上述代码中将在`session`对象上存储`name`，以及将`logged_in`标记为`true`。

最后处理登出：

```
, function (req, res, next) {
  if ('/logout' == req.url) {
    req.session.logged_in = false;
```

```
      res.writeHead(200);
      res.end('Logged out!');
    } else {
      next();
    }
  }
```

137 完整的代码如下所示：

```
/**
 * Module dependencies
 */

var connect = require('connect')
  , users = require('./users')

/**
 * Create server
 */

var server = connect(
    connect.logger('dev')
  , connect.bodyParser()
  , connect.cookieParser()
  , connect.session({ secret: 'my app secret' })
  , function (req, res, next) {
      if ('/' == req.url && req.session.logged_in) {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.end(
            'Welcome back, <b>' + req.session.name + '</b>. '
          + '<a href="/logout">Logout</a>'
        );
      } else {
        next();
      }
    }
  , function (req, res, next) {
      if ('/' == req.url && 'GET' == req.method) {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.end([
            '<form action="/login" method="POST">'
          ,   '<fieldset>'
          ,     '<legend>Please log in</legend>'
          ,     '<p>User: <input type="text" name="user"></p>'
          ,     '<p>Password: <input type="password" name="password"></p>'
          ,     '<button>Submit</button>'
          ,   '</fieldset>'
          , '</form>'
        ].join(''));
      } else {
```

```
        next();
      }
    }
  , function (req, res, next) {
      if ('/login' == req.url && 'POST' == req.method) {
        res.writeHead(200);
        if (!users[req.body.user] || req.body.password != users[req.body.user].
password) {
          res.end('Bad username/password');
        } else {
          req.session.logged_in = true;
          req.session.name = users[req.body.user].name;
          res.end('Authenticated!');
        }
      } else {
        next();
      }
    }
  , function (req, res, next) {
      if ('/logout' == req.url) {
        req.session.logged_in = false;
        res.writeHead(200);
        res.end('Logged out!');
      } else {
        next();
      }
    }
);

/**
 * Listen.
 */

server.listen(3000);
```

138

下面来试试这个简单的登录系统。首先来测试一下基本的认证功能是否正常工作，如图8-10和图8-11所示。

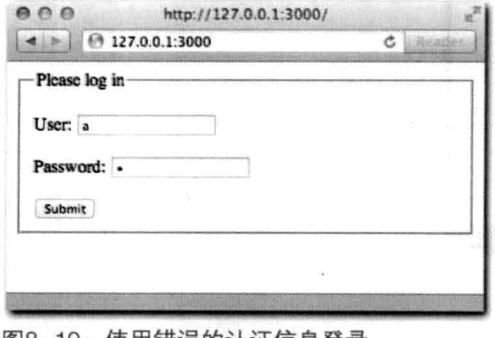


图8-10：使用错误的认证信息登录

139

图8-11：结果显示登录失败

下面，用正确的信息登录，如图8-12所示。

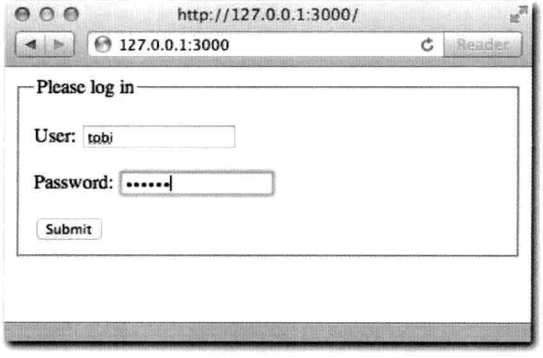


图8-12：有效用户登录

图8-13展示了成功登录的情况。

图8-13：登录成功

140

如图8-14所示，登录成功后，返回首页。

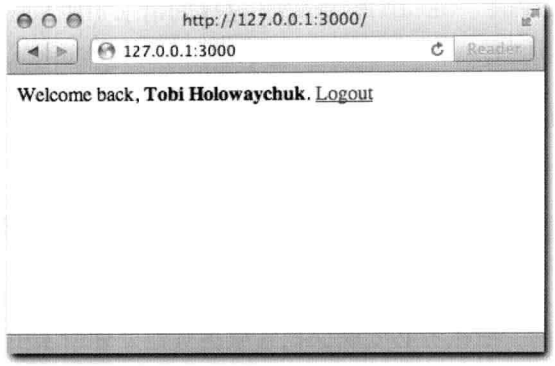


图8-14：登录成功后回到首页

为了让senssion能够在生产环境中也正常工作，我们需要学习如何通过Redis来实现一个持久化的session。

Redis session

尝试这样一件事情：登录后，重启node服务器，然后刷新浏览器。注意到没有，session丢了！

原因就在于session默认的存储方式是在内存。这就意味着session数据都是存储在内存中的，当进程退出后，session数据自然也就丢失了。

这对于开发过程并不算什么坏事，但是对于线上的应用来说，简直就是不能忍受的。生产环境中，需要使用一种当应用重启后，还能够将session信息持久化存储下来的机制，如Redis（第12章会详细介绍 Redis）。

Redis是一个既小又快的数据库，有一个`connect-redis`模块使用Redis来持久化session数据，这样就让session驻扎到了Node进程之外。

通过如下方式来使用该模块（必须要安装好Redis）：

```
var connect = require('connect')
  , RedisStore = require('connect-redis')(connect);
```

然后，将其作为`store`选项的值传递给`session`中间件：

```
server.use(connect.session({ store: new RedisStore, secret: 'my secret' }))
```

完成！现在session已经脱离Node进程了。

141 ## methodOverride中间件

一些比较早期的浏览器并不支持创建如PUT、DELETE、PATCH这样的请求（如Ajax）。常见的解决方案就是在GET或者POST请求中加上一个_method变量来模拟上述这些请求。

举例来说，要想在IE中发送一个PUT请求，可以采用如下方式：

```
POST /url?_method=PUT HTTP/1.1
```

为了让后台的处理程序能够觉得这是一个PUT请求，可以使用methodOverride中间件：

```
server.use(connect.methodOverride())
```

这里要记住的是，中间件是串行执行的，所以务必要确保把它放在其他处理请求中间件之前。

basicAuth中间件

有些项目需要对客户端进行基本的身份验证（见图8-15）。

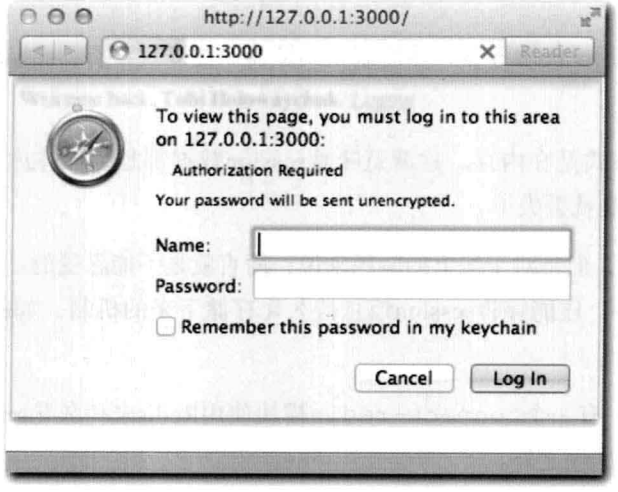


图8-15：浏览器端显示了一个登录框（基本身份验证）

142 为此，Connect也提供了一个名为basicAuth的中间件。

作为例子，我们来创建一个简单的身份验证系统，并且通过命令行对用户进行验证操作。

首先实现用户输入：

```
process.stdin.resume();
process.stdin.setEncoding('ascii');
```

接着，添加basicAuth中间件：

```
connect.basicAuth(function (user, pass, fn) {
  process.stdout.write('Allow user \033[96m' + user + '\033[39m '
    + 'with pass \033[90m' + pass + '\033[39m ? [y/n]: ');
  process.stdin.once('data', function (data) {
    if (data[0] == 'y') {
      fn(null, { username: user });
    }
    else fn(new Error('Unauthorized'));
  });
})
```

注意了，上述代码在stdin EventEmitter上使用了once方法，这是因为我们只需要对每个请求获取一次数据。

basicAuth使用起来非常简单。它提供了user和pass作为参数，同时还提供了一个回调函数来告诉你验证成功还是失败。

如果验证通过，就传递null作为第一个参数（反之，则传递一个Error对象），以及user对象用来生成req.remoteUser对象（供后续的中间件使用）。

然后，继续声明下一个中间件，它会在验证成功后执行。

```
, function (req, res) { res.writeHead(200); res.end('Welcome to the protected area,
  ' + req.remoteUser.username); }
```

好了，运行此例，浏览器会要求输入验证信息（见图8-16）。

143

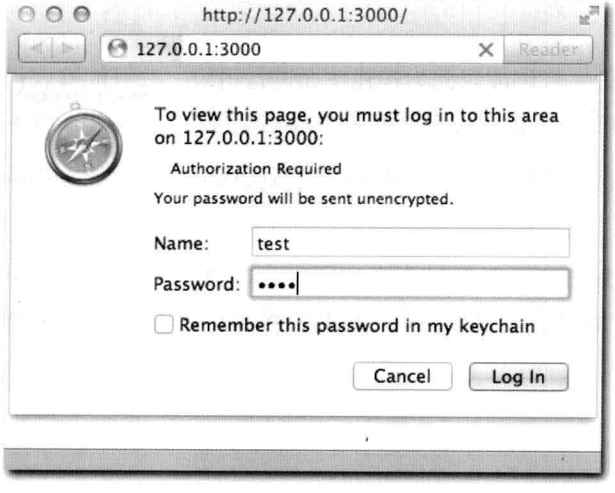


图8-16：用示例的验证信息填写登录对话框

最后，通过命令行来对验证进行处理（不安全），如图8-17所示。

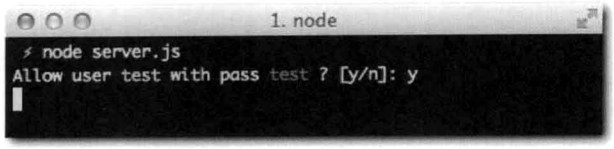


图8-17：服务器将用户输入的验证信息显示在命令行中供管理员处理

图8-18展示了验证通过后的Web站点。

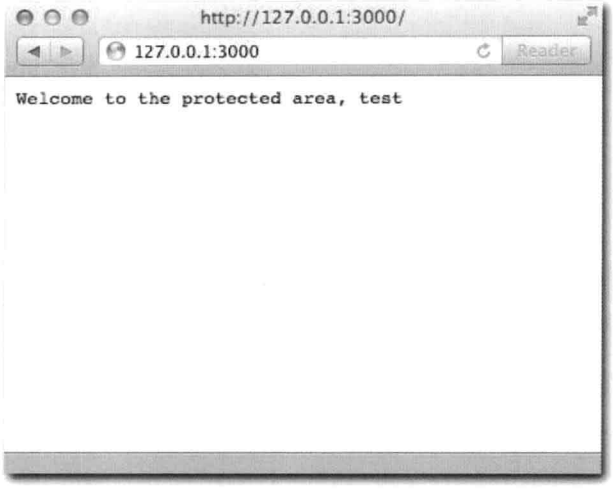


图8-18：在用户验证通过后，请求也验证通过了

144 ## 小结

本章介绍了使用中间件带来的好处：代码能以此为构建单元进行组织，并且能够获得高复用性。Connect是一个实现这种思路的模块，它为构建更具表达力的中间件提供了基础架构。

紧接着，本章又对同一个例子的两种实现——单个请求处理器实现以及拆分成更小构建单元的中间件实现，进行了对比。

最后，又对Connect自带的一些Web应用开发过程中实用的中间件一一进行了介绍。至此，你应当已经学会了如何自定义中间件，以及如何将中间件在Node.js模块系统中进行重用了。

CHAPTER

9 Express

鉴于Connect基于HTTP模块提供了开发Web应用常用的基础功能，Express基于Connect为构建整个网站以及Web应用提供了更为方便的 API。 145

纵观第8章中的例子，你可能已经发现了，绝大部分Web服务器和浏览器之间交互的任务都是通过URL和method来完成的。这两者的组合有时又称为路由，通过Express创建的应用中的一个基础概念。

Express基于Connect构建而成，因此，它也保持了重用中间件来完成基础任务的想法。这就意味着，通过Express的API方便地构建Web应用的同时，又不失构建于HTTP模块之上高可用中间件的生态系统。

接下来，我们通过Express构建一个查询Twitter API的小应用，来了解Express的用法和功能。

一个小型Express应用 146

应用虽小但五脏俱全。在用户通过查询关键词查询“推文”时，需要返回包含查询结果的HTML页面。另外，我们这次不在请求处理器中采用字符串拼接的方式来返回HTML页面内

容，而是使用一个简单的模板语言来实现，并将逻辑从视图层分离到控制层。

那么第一步就是要确保将所需的模块引入进来。

创建模块

按照惯例，先创建一个package.json文件，这次需要多添加两个额外的依赖：ejs。本例中要使用的模板引擎，以及superagent来简化向 Twitter发送HTTP查询请求的实现。

```
{
    "name": "express-tweet"
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.9"
      , "ejs": "0.4.2"
      , "superagent": "0.3.0"
    }
}
```

这里请注意，尽管本例中我们使用的是Express 2，但代码应该是能和 Express 3完全兼容的（截止本书撰写期间Express 3还处在开发阶段）。

定义好项目的元数据之后，接下来创建HTML模板。

HTML

和上一个应用不同的是，为了避免将HTML代码嵌入到应用逻辑（通常叫做处理器或者路由处理器）中，这次我们要使用一个简单的模板语言来处理。模板语言的名字是EJS（内嵌（embedded）的js），和在HTML中内嵌PHP类似。

从在views/文件夹中创建一个index.ejs文件开始。事实上，模板可以放在任何一个地方，不过考虑到本例的项目结构，我们就将其单独放到views目录下。

首个模板对应默认的路由路径（首页）。它提供一个入口让用户提交搜索推文的关键字：

```
<h1>Twitter app</h1>
<p>Please enter your search term:</p>
<form action="/search" method="GET">
  <input type="text" name="q">
  <button>Search</button>
</form>
```

147

另一个模板search.ejs用于展示查询结果。高亮查询关键词并将查询结果逐一显示出来（如果有的话），否则就显示一条消息：

```
<h1>Tweet results for <%= search %></h1>
<% if (results.length) { %>
  <ul>
```

```
    <% for (var i = 0; i < results.length; i++) { %>
    <li><%= results[i].text %> - <em><%= results[i].from_user %></li>
    <% } %>
  </ul>
<% } else { %>
  <p>No results</p>
<% } %>
```

如上面所示，我们将JavaScript代码嵌在<%和%> EJS标签中。另外，我们通过在<%之后加入“=”符号将变量值打印出来。

SETUP

我们现在server.js文件中引入依赖的模块：

```
var express = require('express')
```

引入Express之后，需要用它来初始化Web服务器。和Connect类似，Express提供了一个createServer快捷方法返回一个Express HTTP服务器。就像这样：

```
var app = express.createServer()
```

和其他流行的Web框架不同，Express并不要求任何必要的配置，也不对文件结构有特殊的要求。它足够灵活，允许你对每一个单独的功能点进行自定义。

本例中，我们需要指定使用的模板引擎（这样就不需要每次载入视图时都去引入）以及视图文件（模板）所在的目录。刚刚介绍的express.createServer方法返回的HTTP服务器自带配置系统。我们可以通过调用该对象上的set方法来修改默认的配置项。添加如下代码：

```
app.set('view engine', 'ejs');
app.set('views', __dirname + '/views');
app.set('view options', { layout: false });
```

第三个view options参数所定义的选项，在渲染视图时，会传递到每个模板中。这里layout的值设置为false，是为了匹配Express 3中的默认值。

要获取配置信息，可以调用app.set方法并传递对应要获取配置的标志。比如，要获取views的配置信息，可以通过如下方式： 148

```
console.log(app.set('views'));
```

接下来，我们会使用Express提供的方法来定义路由，关于路由的有关内容，我们其实在第7章和第8章就已经接触过了。

定义路由

通过使用Express来定义路由就无须手动地每次去检查method和url属性，只需调用

Express提供的对应HTTP method的方法，并将URL和对应的处理中间件传递进去就可以了。

Express支持的方法有get、put、post、del、patch以及head，分别对应HTTP的GET、PUT、POST、DELETE、PATCH以及HEAD。下面是定义路由的例子：

```
app.get('/', function (req, res, next) {});
app.put('/post/:name', function (req, res, next) {});
app.post('/signup', function (req, res, next) {});
app.del('/user/:id', function (req, res, next) {});
app.patch('/user/:id', function (req, res, next) {});
app.head('/user/:id', function (req, res, next) {});
```

第一个参数是路由地址，第二个参数是路由处理程序。路由处理程序就和中间件一样。

注意了，路由部分还可以通过特殊格式来定义变量。如上述例子中的/user/:id，哪怕id值不同，路由也能匹配到：如/user/2、/user/3，等等。下一章会对这部分内容做详细介绍。

下面我们来定义首页的路由：添加如下代码到server.js文件中：

```
app.get('/', function (req, res) {
  res.render('index');
});
```

到目前为止，完整代码如下所示：

```
/**
 * 模块依赖
 */
var express = require('express')
  , search = require('./search')

/**
 * 创建app
 */

var app = express.createServer();

/**
 * 配置
 */

app.set('view engine', 'ejs');
app.set('views', __dirname + '/views');
app.set('view options', { layout: false });

/**
 * 路由
 */

app.get('/', function (req, res) {
```

149

```
  res.render('index');
});

/**
 * 监听
 */
app.listen(3000);
```

Express为response对象提供了render方法。该方法完成如下三件事：

1. 初始化模板引擎。
2. 读取视图文件并将其传递给模板引擎。
3. 获取解析后的HTML页面并作为响应发送给客户端。

由于在前一步中将ejs指定为视图引擎，因此无须显式地指明index.ejs了。

运行之后结果如图9-1所示（别忘记调用listen方法）。

150

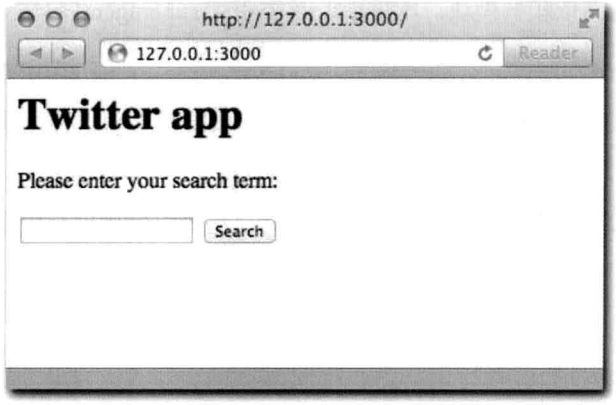


图9-1：路由处理器/渲染index视图

第二个路由中，我们调用一个名为search的方法（将在另外一个模块中定义）：

```
app.get('/search', function (req, res, next) {
  search(req.query.q, function (err, tweets) {
    if (err) return next(err);
    res.render('search', { results: tweets, search: req.query.q });
  });
});
```

接着，在express之后添加对search模块的依赖：

```
var express = require('express')
  , search = require('./search')
```

这里要注意，如果search方法回调函数中接收到错误对象，那么我们就直接把它传递给

next。后面章节会介绍错误处理的相关内容，到时你就能明白为什么要这么做了，目前，我们就假设Express会将错误信息通知给用户。

这个路由处理器中，我们也调用了render方法，不过不同的是，这次我们传递了一个对象作为第二个参数。该对象的内容会暴露给视图。注意这里我们是如何把tweets和search传递过去的，这两个变量可以在search.ejs视图中直接使用。这类对象称为本地变量，因为它的内容只对其传递的视图可见。

查询

查询模块暴露一个简单的方法提供对推文的查询，其内部调用了Twitter的查询API。本例中，我们将查询模块search.js文件和server.js文件放到同一目录中。

上述调用search方法的代码中，传递了一个查询关键字以及回调函数，回调函数接收两个参数，其一是错误对象（如果有的话），其二是一个包含了查询到的推文的数组。

151 要写这样一个模块，我们先来定义依赖的模块。本例中，只需用到 superagent：

```
var request = require('superagent')
```

由于向Twitter的Web服务发送HTTP请求是本例中重要的功能，我们需要确保查询模块正确地进行了错误处理。

比如，要是Twitter API服务宕机了，我们就发送一个错误对象，来给用户显示一个错误页面（如：显示HTTP错误状态码500）。

```
/**
 * Search function.
 *
 * @param {String} search query
 * @param {Function} callback
 * @api public
 */

module.exports = function search (query, fn) {
  request.get('http://search.twitter.com/search.json')
    .data({ q: query })
    .end(function (res) {
      if (res.body && Array.isArray(res.body.results)) {
        return fn(null, res.body.results);
      }
      fn(new Error('Bad twitter response');
    });
};
```

和其他superagnet例子类似，我们发送一个GET请求，将查询关键字作为数据字段q的值

以查询字符串的形式发送出去。以hello world为查询关键字的URL类似`http://search.twitter.com/search.json? q=hello+world`。

在响应处理程序中，我们确保请求发送成功并且完全符合我们的预期。相比查看HTTP响应码是否为200，我们采用更加聪明的方式：直接查看响应结果是否包含含有推文内容的数组。

第7章中我们就介绍过，如果superagent获取到一个JSON形式的响应消息，它会自动进行解码，并将解码后的内容放到res.body变量中。由于Twitter API会返回一个JSON对象，对象中的results字段内容就是包含了推文的数组，因此，下述代码段对于进行错误处理的判断来说足矣：

```
if (res.body && Array.isArray(res.body.results)) {
   return fn(null, res.body.results);
}
```

运行

152

运行上述服务器代码，并通过浏览器访问`http://localhost:3000`（见图9-2），试着输入推文关键字（见图9-3）。

图9-2：输入查询推文关键字并提交的例子

图9-3：查询结果

成功！向Twitter查询推文后，我们获得了一个包含推文的数组。最终它会显示在根据推文数组动态生成的search.ejs模板中。

HTML页面由Express的渲染引擎产生，/search路由成功地将完整的页面显示给用户，如图9-3所示。

在这个简单的示例后，接下来该是深入研究Express特性的时候了，我们来看一些新的功能。

153 设置

Express提供了最有意思的特性之一，同时也是对任意类型Web应用都不可或缺的，就是对环境设置的管理能力。

比如，在生产环境下，为了提高性能，我们可以让express将模板缓存起来，这样可以加快响应速度。然而，若在开发模式下开启这个功能，每次对模板稍做改动就需要重启Node服务程序才能生效。

通过如下方式就可以达到让Express对模板进行缓存的效果：

```
app.configure('production', function () {
  app.enable('view cache');
});
```

在上述代码中，app.enable就等同于此前例子中为了配置views config标志所用到的app.set。

```
app.set('view cache', true);
```

要查看配置标志是否启用，也可以调用app.enabled。对应的还有app.disable和app.disabled方法。

当环境变量NODE_ENV设置为production时，我们在app.configure定义的对应的回调函数就会被执行。

要测试上述代码，运行如下命令：

```
$ NODE_ENV=production node server
```

要是NODE_ENV没有设置，则默认会调用development的配置：

```
app.configure('development', function () {
  // gets called in the absence of NODE_ENV too!
});
```

下面还有一些内置的实用设置：

- 大小写敏感的路由：启用大小写敏感的路由。默认情况下，当定义如下路由：

```
app.get('/my/route', function (req, res) {}
```

Express能匹配到/my/route和/MY/ROUTE。开启之后，路由只会匹配定义的路由，上述例子中就只会匹配/my/route。

- 严格路由：启用后，最后的斜杠就不会自动忽略。什么意思呢？比如，此前例子中能匹配/my/route和/my/route/，而一旦开启严格路由模式，只会匹配到/my/route，因为路由就是这样在app.set中定义的。 154
- jsonp回调：启用res.send() / res.json()对jsonp的支持。JSONP是一项解决跨域JSON请求的技术，它将响应结果包裹在一个用户指定的回调函数中。
- 当有JSONP请求时，其URL类似这样：/my/route?callback=myCallback。Express会自动检测callback参数，并将相应结果包裹在myCallback文本中。要启用该功能，可以调用app.enable('jsonp callback')。这里要注意，这只作用于res.send和res.json，后面的章节会有相关介绍。

模板引擎

要像此前例子那样使用ejs，必须完成以下两个步骤：

1. 通过NPM安装ejs模块。
2. 声明view engine为ejs。

如下的其他模板引擎也可以在Express中使用：

- Haml
- Jade
- CoffeeKup
- jQuery Templates for node

Express会试着以模板文件扩展名或者以配置的view engine的值为名去调用require方法。

比如，可以调用：

```
res.render('index.html')
```

这时，Express会尝试着去require html引擎。找不到该引擎，就会报错。

也可以通过app.register API将扩展名匹配到指定的模板引擎。比如，通过如下方式

就可以将html扩展名匹配到jade模板引擎：[1]

```
app.register('.html', require('jade'));
```

Jade是最流行的模板语言之一，绝对值得一学。要想了解更多关于Jade的内容可以访问其官方网站http://jade-lang.org。

155 ## 错误处理

在Node中，将错误对象作为非阻塞I/O回调函数的第一个参数是很常见的。在本章的例子中，我们也有可能在调用Twitter API进行查询时接收到错误对象。

面对这种情况，在Express中常规的做法就是将该错误对象传递给next函数。默认情况下，Express会展示一个错误页面并发送500状态码。

绝大多数Web应用都会自定义错误页面，或者甚至自建一套后台的错误处理机制。

我们可以通过app.error方法定义一个特殊的错误处理器作为错误处理的中间件：

```
app.error(function (err, req, res, next) {
  if ('Bad twitter response' == err.message) {
    res.render('twitter-error');
  } else {
    next();
  }
});
```

注意在上述代码中，通过检测错误消息内容来判断是否要对错误进行处理，要是检测结果不匹配就直接调用next。

还可以设置多个.error处理器来执行不同的处理。比如，最后一个错误处理器就可以发送500 Internal Server Error并展示一个通用的错误页面。

```
app.error(function (err, req, res) {
  res.render('error', { status: 500 });
});
```

当调用next并且对应的处理器无法找到时，默认的Express错误处理器就会触发。

快捷方法

Express为Node中的Request和Response对象提供了一系列扩展来简化开发。

Request对象上的扩展如下。

1 译者注：Express 3.x中需要使用app.engine方法。

- header：此扩展能够让程序以函数的方式获取头信息，并且还是大小写不敏感的。

```
req.header('Host')
req.header('host')
```

- accepts：此扩展会分析请求中的Accept头信息，并根据提供的值返回true或者false。 156

```
req.accepts('html')
req.accepts('text/html')
```

- is：此扩展和accepts类似，但它检查Content-Type头信息。

```
req.is('json')
req.is('text/html')
```

Response对象上的扩展如下。

- header：此扩展接收一个参数来检查对应的头信息是否已经在 response上设置了。

```
res.header('content-type')
```

或设置header的两个参数：

```
res.header('content-type', 'application/json')
```

- render：对render相信你已经了解不少了。不过，在此前的例子中，你也许注意到了我们传递了status值。这是一个特殊的类型，设置了它就等于为response响应对象设置了状态码。
  - 除此之外，还可以提供第三个参数给render方法来获取 HTML内容而不是直接将其作为响应消息自动传递出去。

```
res.render('template', function (err, html) {
  // 处理收到的 html
});
```

- send：此扩展很奇特。它会根据提供参数的类型执行响应的行为。
  - Number：发送状态码。

```
res.send(500);
```

  - String：发送HTML内容。

```
res.send('<p>html</p>');
```

  - Object/Array：序列化成JSON对象，并设置相应的Content-Type头信息。

```
res.send({ hello: 'world' }); res.send([1,2,3]);
```

- json：此扩展在绝大多数场景下和send类似。只是它会显式地将内容作为JSON对象发送。

```
res.json(5);
```

在发送值类型未知的情况下可以使用此方法。res.send会判断发送值的类型，并且依据判断结果来选择是否调用JSON.stringify方法。如果是数字类型，那么会认为发送的是状态码，并直接结束响应。而res.json会把数字类型也进行JSON.stringify转换。

由于绝大部分情况下我们会对发送对象进行编码，所以，res.send还是很不错的选择。

157

- redirect：redirect等效于发送302（暂时移除）状态码以及 Location头信息。如下所示。

```
res.redirect('/some/other/url')
```

就等效于：

```
res.header('Location', '/some/other/url');
res.send(302);
```

上述代码在Node.js内部其实是这样处理的：

```
res.writeHead(302, { 'Location': '/some/other/url' });
```

- redirect还可以接收自定义的状态码作为第二个参数。假设你不想发送302而是发送表示永久性移除的301状态码，可以采取如下方式。

```
res.redirect('/some/other/url', 301)
```

- sendfile：此扩展和Connect中的static中间件类似，不同之处在于它用于单个文件。

```
res.sendfile('image.jpg')
```

路由曾在我们的示例应用中有所应用，事实上，路由还有很多内容有待我们了解，它们对大型Web应用都非常有帮助。

路由

在定义路由时，可以使用自定义参数：

```
app.get('/post/:name', function () {
  // . . .
})
```

上述代码中，name变量值会注入到req.params对象上。比如，当通过浏览器访问'/post/hello-world'时，req.params对象会变为如下形式：

```
app.get('/post/:name', function () {
  // req.params.name == "hello-world"
})
```

还可以通过在变量后添加问号（？）来表示该变量是可选的。在此前的路由示例中，要是通过浏览器访问/post，那么该路由是不会匹配到的。要匹配到可以采用如下方式：

```
app.get('/post/:name?', function (req, res, next) {
  // this will match for /post and /post/a-post-here
})
```

像这样定义了参数的路由，内部会当正则表达式处理。也就是说，定义路由时也可以直接使用RegExp对象。比如，只想匹配字母、数字以及中划线的话，可以这样： 158

```
app.get(/^\/post\/([a-z\d\-]*)/, function (req, res, next) {
  // req.params contains the matches set by the RegExp capture groups
})
```

和中间件一样，在路由处理程序中也可以使用next。即使当一个路由匹配并得到处理，还是可以强制Express去继续匹配其他路由的。

比如，让路由只接受以'h'开头的参数：

```
app.get('/post/:name', function (req, res, next) {
  if ('h' != req.params.name[0]) return next();
  // . . .
});
```

就是因为Express提供了灵活的路由，才能实现像上述这样的处理代码来解决多变的情况。

举例来说，许多Web应用允许如/home、/about这样的路由，但它们同时又希望能够让动态的内容也能有永久的URL。

在定义好所有的路由之后，可以再定义一个路由来获取用户名，并进行数据库调用。如果该用户未能找到，就调用next并发送404，否则就渲染该用户个人页面。

```
app.get('/home', function (req, res) {
  // . . .
});

app.get('/:username', function (req, res, next) {
  // if you got here, no prior application routes matched
  getUser(req.params.username, function (err, user) {
    if (err) return next(err);

    if (exists) {
      res.render('profile')
    } else {
```

```
        next();
      }
    });
});
```

如你所知，Express也沿袭了中间件的概念。下面我们就来深入了解一下。

159

中间件

由于Express是构建于Connect之上的，所以，当创建Express服务器时可以使用Connect兼容的中间件。比如，要托管images/目录下的图片，就可以像这样使用static中间件：

```
app.use(express.static(__dirname + '/images'));
```

或者，要想使用connect的session，也很容易，像这样就可以了：

```
app.use(express.cookieParser());
app.use(express.session());
```

注意了，在引入了Express之后就可以直接使用Connect的中间件了。不需要`require('connect')`或者把connect作为项目依赖添加到`package.json`文件中。

中间件是易理解的。

更有意思的是，和全局中间件（针对每个请求）不同，Express还允许只在特定匹配到的路由中才使用中间件。

想象一下这样一个场景：你需要检查用户是否已经登录，并且这部分检查只在特定受保护的路由中进行。这个时候，就可以定义一个secure中间件，判断若`req.session.logged_in`不为true时就发送`403 Not Authorized`状态码。

```
function secure (req, res, next) {
  if (!req.session.logged_in) {
    return res.send(403);
  }

  next();
}
```

然后，将它应用到对应的路由中：

```
app.get('/home', function () {
  // accessible to everyone
});

app.get('/financials', secure, function () {
  // secure!
});

app.get('/about', function () {
  // accessible to everyone
```

```
});

app.get('/roadmap', secure, function () {
  // secure!
});
```

还可以像这样为路由定义多个中间件：

160

```
''app.post('/route', a, b, c, function () { });
```

有的时候，在中间件中调用next就可以跳过该路由的其他中间件，这样Express就会紧接着在下一个路由中做相应处理。

比如，若相比发送403，你更希望Express去检查其他的路由，那么就可以采用如下方式来实现：

```
function secure (req, res, next) {
  if (!req.session.logged_in) {
    return next('route');
  }

  next();
}
```

通过调用next('route')，就能确保当前路由会被跳过。

随着项目规模的扩大，路由和中间件的数量也会越来越多，因此，如何更好地组织代码就变得非常有用。下面我们就来介绍这部分的内容。

代码组织策略

对于任意一个Node.js应用（包括Express Web应用）来说，第一条准则都是模块化。Node.js通过提供一个简单的require API来提供一个强大的代码组织策略。

比如，一个应用包含三块独立的内容：/blog、/pages以及/tags。每块都包含各自的路由。例如：/blog/search、/pages/list以及tagst/cloud。

好的代码组织方式应当是维护一个server.js文件，该文件中包含了路由表，同时将每一部分的路由处理器都通过模块化的方式来引入，如blog.js、pages.js以及tags.js。首先，定义依赖的模块并初始化 app、引入中间件等：

```
var express = require('express')
  , blog = require('./blog')
  , pages = require('./pages')
  , tags = require('./tags')

// initialize app
```

```
var app = express.createServer();

// here you would include middleware, settings, etc
```

161 接着定义之前提到的路由表，这里简单地将所有的路由信息都罗列出来放在一个地方：

```
// blog routes
app.get('/blog', blog.home);
app.get('/blog/search', blog.search);
app.post('/blog/create', blog.create);

// pages routes
app.get('/pages', pages.home);
app.get('/pages/list', pages.list);

// tags routes

app.get('/tags', tags.home);
app.get('/tags/search', tags.search);
```

然后，针对每个模块需要使用exports函数。以blog.js为例：

```
exports.home = function (req, res, next) {
  // home
};

exports.search = function (req, res, next) {
  // search functionality
};
```

模块化提供了很强大的扩展性。上述代码还可以根据http方法进行进一步扩展。例如：

```
exports.get = {};
exports.get.home = function (req, res, next) {})
exports.post = {};
exports.post.create = function (req, res, next) {})
```

另外一种对应用进行解耦的方式叫做app挂载。你可以将整个Express app作为一个模块（模块中依然可以使用NPM模块），并将其挂载到现有的应用中，这样就能够让路由无缝对接起来。

考虑这样一个博客的应用。你可以将一个博客相关的路由都通过/来定义，/categories以及/search，然后将其以blog.js文件暴露出来：

```
var app = module.exports = express.createServer();
app.get('/', function (req, res, next) {});
app.get('/categories', function (req, res, next) {});
app.get('/search', function (req, res, next) {});
```

注意，上述路由都是通过绝对路径来定义的，不包含任何前缀。接着，在主程序中，要做的就是将其引入并传递给app.use方法：

```
app.use(require('./blog'));
```

162

这样一来，所有博客相关的路由就都可以使用了。除此之外，你还可以为它们定义一个前缀：

```
app.use ('/blog', require('./blog'));
```

现在，/blog/、/blog/categories以及/blog/search以后都可以直接给其他express应用使用，并且它自身拥有完全独立的依赖、中间件、配置，等等。

小结

本章介绍了如何使用最流行的Node.js Web框架Express。

使用Express最大的益处就在于，它简洁、配置少但却又不失灵活，同时它和Connect一样构建于测试全面、抽象简洁的基础之上。

和其他Web框架或者类库不同，Express非常容易进行模块化来满足不同的需求、结构以及模式。本章介绍了如何使用Express以最少的代码来构建首个示例应用，就像Node.js Hello World程序那样。

事实上，你或许已经注意到了，与重新在Node.js core API基础上构建一套新的方式不同，Express尝试与core API靠拢，并扩展它。这就是路由处理器可以直接接收原生的Node request和response对象的原因，就像我们在首个HTTP服务器应用中那样。本章介绍了一些Express实用的扩展，比如，如何使用res.send来以JSON的形式进行响应等。

本章最后介绍了如何将不同功能的代码组织起来创建一个可维护的应用。介绍了最好的方式就是通过Node.js核心的API：通过使用require就是其中一个最强大的工具，来进行更上层的代码组织。

CHAPTER

10 WebSocket

目前，绝大部分网站和Web应用开发者都习惯了通过发送HTTP请求来和服务器进行通信，随后接收服务器的HTTP响应。 163

正如在上一章中看到的那样，通过指定URL、`Content-Type`以及设置其他属性来获取资源的模型是最常见的，也是万维网中最常见的方式。Web生来就是用以传递互相关联的文档的。URL之所以含有路径信息是因为文档通常在文件系统中是有层次结构的，并且，每一层的结构都包含了对超链接的索引。

如下述例子：

```
GET /animals/index.html
GET /animals/mammals/index.html
GET /animals/mammals/ferrets.html
```

然而，随着时间的推移，Web变得越来越注重用户体验。如今，特别是随着HTML5以及相关工具的诞生，传统的那种每次需要用户点击之后才能获取文档的方式正在逐步退出历史舞台。现在已经可以创建出如游戏、文本编辑器等这种非常酷的Web应用了，完全可以取代了传统的桌面应用。

Ajax

164

Web 2.0标志着Web应用的崛起。其中一个关键因素就是Ajax，其具体表现在于提高了用

户体验，这背后重要的原因就是：用户再也不用每次都通过交互操作才能从服务器端获取新的HTML文档。

比如，要想在社交应用中更新个人信息，发送一个异步的POST请求，随后会收到一个更新成功的返回消息。接着，使用一个简单易用的 JavaScript框架，更新视图以展现用户行为结果即可。

再比如，当用户点击一张表中的移除按钮时，就可以发送DELETE请求，并移除该行（<tr>）元素即可，无须刷新浏览器，也无须再去获取许多不必要的数据、图片、脚本以及样式文件以及重新渲染整个页面。

Ajax之所以重要，从本质上来说，它避免了许多原本需要在Web应用中处理的数据传递和渲染的开销。

然而，现在许多应用通过传统的HTTP请求+响应模型的方式来发送和接收数据依然会造成很大的开销。就拿本章中要构建的应用来说，我们要想实时地获取每位网站访问者当前鼠标的位置。那么每次当用户移动鼠标时，我们都要将坐标信息发送给服务器。

假设使用jQuery来发送Ajax请求。第一个想到的办法就是：每次mousemove事件触发时，就通过$.post向服务器发送一个POST请求，如下所示：

```
$(document).mousemove(function (ev) {
  $.post('/position', { x: ev.clientX, y: ev.clientY });
});
```

上述代码尽管看起来很直观，但是有个根本性的问题：无法控制服务器接收请求的先后顺序。

当执行代码发出请求时，浏览器会使用可用的socket来进行数据发送，为了提高性能，浏览器会和服务器之间建立多个socket通道。举例来说，在下载图片的时候，还是可以同时发送Ajax请求。要是浏览器只有一个socket通道，那么，网站渲染和使用都会非常慢。

若三个请求分别通过三个不同的socket通道发送，就无法保证服务器端接收的顺序了。所以，要解决这个问题，我们需要调整代码，在服务器接收到一个请求后再接着发送第二个请求，这样就能保证接收的顺序了：

165

```
var sending = false;

$(document).mousemove(function (ev) {
  if (sending) return;
  sending = true;
  $.post('/position', { x: ev.clientX, y: ev.clientY }, function () {
    sending = false;
  });
});
```

作为例子，下面显示了Firefox中TCP传输消息的内容：

Request

```
POST / HTTP/1.1
Host: localhost:3000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:8.0.1) Gecko/20100101
 Firefox/8.0.1
Accept: */*
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Referer: http://localhost:3000/
Content-Length: 7
Pragma: no-cache
Cache-Control: no-cache

x=6&y=7
```

Response

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 2
Connection: keep-alive

OK
```

如上述例子所示，除了一小段数据，还包含了很多的文本内容。对于上述例子而言，很多不需要的头信息依然被发送了出去，这些头信息的数据量远远超过了真正要发送数据本身的大小。

尽管可以移除其中一些头信息，但是对于上述例子来说，我们真的需要响应消息吗？在发送一些像鼠标位置这样无关紧要的信息时，其实根本不需要等到响应返回后再接着发送更多的消息。

对于这类Web应用来说，理想解决方案得从TCP而非HTTP入手（就像第6章中的聊天程序那样）。理想状态下，我们更希望直接将数据，另外附加最小的消息窗口（就是与真正发送的数据包裹在一起的数据）通过socket发送。 166

拿telnet举例的话，我们就希望浏览器可以发送如下面这样的数据：

```
x=6&y=7 \n
x=10&y=15 \n
. . .
```

归功于HTML5，现在我们有了这样的解决方案：WebSocket。WebSocket是Web下的TCP，

一个底层的双向socket，允许用户对消息传递进行控制。

HTML5 WebSocket

每次提到WebSocket的时候，其实是在讲两部分内容：一部分是浏览器实现的WebSocket API，另一部分是服务器端实现的WebSocket协议。这两部分是随着HTML5的推动一起被设计和开发的，但是两者都没有成为HTML5标准的一部分。前者被W3C标准化了，而后者被IETF标准化为RFC 6455。

浏览器端实现的API如下：

```
var ws = new WebSocket('ws://host/path');
ws.onopen = function () {
  ws.send('data');
}
ws.onclose = function () {}
ws.ondata = function (ev) {
  alert(ev.data);
}
```

上述简单的API不禁让我们想起第6章中写过的TCP客户端。和XMLHttpRequest（Ajax）不同，它并非面向请求和响应，而是可以直接通过send方法进行消息传递。通过data事件，发送和接收 UTF-8或者二进制编码的消息都非常简单，另外，通过open和close事件能够获知连接打开和关闭的状态。

首先，连接必须通过握手来建立。握手方面和普通的HTTP请求类似，但在服务器端响应后，客户端和服务器端收发数据时，数据本身之外的信息非常少：

167

Request

```
GET /ws HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Version: 6
Sec-WebSocket-Origin: http://pmx
Sec-WebSocket-Extensions: deflate-stream
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
```

Response

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
```

WebSocket还是建立在HTTP之上的，也就说，对于现有的服务器来说，实现WebSocket协议非常容易。它和HTTP之间主要的区别就是，握手完成后，客户端和服务器端就建立了类似TCP socket这样的通道。

为了更好地理解这部分内容，我们来写个示例应用。

一个ECHO示例

第一个示例包含一个服务器和一个客户端，主要完成互相之间消息的交换并输出接收的消息内容。当客户端发送消息时，记录相应的时间，用于计算服务器处理响应花费了多长时间。

初始化项目

本例中，我们将使用我在LearnBoost工作时开发的`websocket.io`模块。

这里有一点非常重要：`websocket.io`仅处理WebSocket相关的请求。其他请求仍由你的网站或者应用的常规Web服务器来处理，这就是我们在`package.json`文件中加入对`express`的依赖[1]的原因：

1 译者注：截止译者翻译期间，本章示例在最新稳定版node.js v0.10.x下运行会有错误抛出。具体原因是websocket.io中代码书写有问题，要解决需要将websocket.io/lib/hybi-16.js文件中如下左侧部分的代码中的`.on('xxx',fn)`，均改为`.onxxx = fn`的形式，如下右侧所示：

```
this.parser
.on('text', function (packet) {

 self.onMessage(packet);
})
.on('binary', function (packet) {

 self.onMessage(packet);
})
.on('ping', function () {

 // version 8 ping => pong
 try {

 self.socket.write('\u008a\u0000');
 }
 catch (e) {
 self.end();
 return;

 }
})
.on('close', function () {

 self.end();
})
.on('error', function (reason) {

 self.log.warn(self.name + 'parser error:'
+ reason);
 self.end();
})
```

```
this.parser.ontext = function (packet) {
 self.onMessage(packet);
};

this.parser.onbinary = function (packet)
{
 self.onMessage(packet);
};

this.parser.onping = function () {
 // version 8 ping => pong
 try {

 self.socket.write('\u008a\u0000');
 }
 catch (e) {

 self.end();
 return;
 }
};

this.parser.onclose = function () {
 self.end();
};

this.parser.onerror = function (reason){
self.log.warn(self.name + 'parser error:'
+ reason);
self.end();
};
```

```
"name": "ws-echo"
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.1"
      , "websocket.io": "0.1.6"
  }
}
```

168 服务器的处理就是简单地将浏览器端发送过来的消息再回传回去。浏览器端则计算服务器端处理响应的耗时。

建立服务器

首先要做的就是初始化express并将weksocket.io绑定到express上，这样它就能处理WebSocket的请求了：

```
var express = require('express')
  , wsio = require('websocket.io')

/**
 * Create express app.
 */

var app = express.createServer();

/**
 * Attach websocket server.
 */

var ws = wsio.attach(app);

/**
 * Serve your code
 */

app.use(express.static('public'));

/**
 * Listening on connections
 */

ws.on('connection', function (socket) {
  // . . .
});

/**
 * Listen
 */

app.listen(3000);
```

我们来重点看下connection处理程序。我特意将websocket.io设计为与实现一个net.Server类似。由于我们需要直接将接收到的消息返回给客户端，所以，我们需要做的就是监听message事件，并将其send回去。

```
ws.on('connection', function (socket) {
  socket.on('message', function (msg) {
    console.log(' \033[96mgot:\033[39m ' + msg);
    socket.send('pong');
  });
});
```

169

建立客户端

接着我们在public目录下添加index.html文件，输入如下内容：

index.html

```
<!doctype html>
<html>
  <head>
    <title>WebSocket echo test</title>
    <script>
      var lastMessage;

      window.onload = function () {
        // create socket
        var ws = new WebSocket('ws://localhost:3000');
        ws.onopen = function () {
          // send first ping
          ping();
        }
        ws.onmessage = function (ev) {
          console.log(' got: ' + ev.data);
          // you got echo back, measure latency
          document.getElementById('latency').innerHTML = new Date - lastMessage;
          // ping again
          ping();
        }
        function ping () {
          // record the timestamp
          lastMessage = +new Date;
          // send the message
          ws.send('ping');
              };
    </script>
  </head>
  <body>
    <h1>WebSocket Echo</h1>
    <h2>Latency: <span id="latency"></span>ms</h2>
```

```
</body>
</html>
```

上述HTML代码非常直观。它创建了一个占位符用于显示服务器端处理的延时（消息来回一次所需要的时间，以毫秒为单位）。

170 JavaScript代码相对来说也较直观。首先定义一个存储延时的变量：

```
var lastMessage
```

接着初始化WebSocket并和服务器端建立连接：

```
var ws = new WebSocket('ws://localhost:3000');
```

建立连接后，就向服务器端发送第一条消息：

```
ws.onopen = function () {
  ping();
}
```

收到服务器端响应后计算耗时，并再次发出一条消息：

```
ws.onmessage = function () {
  console.log(' got: ' + ev.data);
  // you got echo back, measure latency
  document.getElementById('latency').innerHTML = new Date - lastMessage;
  // ping again
  ping();
}
```

最后，就是定义ping函数，记录发送消息前的时间戳用于最终计算耗时（也就是此前说的延时），随后发送一条简单的消息给服务器：

```
function ping () {
  // record the timestamp
  lastMessage = +new Date;
  // send the message
  ws.send('ping');
};
```

运行示例程序

输入如下命令来运行服务器端代码：

```
$ node server.js
```

接着，打开浏览器访问http://localhost:3000（见图10-1）。请确保使用的浏览器支持WebSocket，像Chrome 15+或者IE 10+都支持。如果不确定的话，可以访问http://websocket.org，查看网站右上角的“Does your browser support WebSocket”显示框。

好了，至此我们就成功创建了一个单用户的实时应用程序。通过终端的输出和浏览器的控制台能够看到消息传输的日志。在绝大多数现代电脑中，消息传输大概耗时1~5毫秒。作为习题，大家可以尝试使用Ajax配合Express的路由功能书写一个同样功能的应用程序，然后对比一下完成同样一组消息的传递耗时多久。 171

图10-1：一组消息从客户端到服务器端，再返回到客户端所需的时间

接下来，我们要书写一个示例应用，该应用用于将多个用户连接显示到一个屏幕上。

鼠标光标

接下来我们要书写的程序是在一个屏幕上以鼠标光标样式的图片形式展示所有连接过来的用户光标的位置。

通过本例，你将学习到广播的概念，也就是将一个消息发给除了自己以外的所有人。

初始化示例程序

书写本示例程序所需要的模块和上例一样，我们在package.json定义这些依赖：

```
{

    "name": "ws-cursors"
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.1"
      , "websocket.io": "0.1.6"
    }

}
```

建立服务器 172

建立服务器的方式和上例类似，使用epxress托管静态HTML文件，同时将websocket.io

绑定到express服务器上：

```
var express = require('express')
  , wsio = require('websocket.io')

/**
 * Create express app.
 */

var app = express.createServer();

/**
 * Attach websocket server.
 */

var ws = wsio.attach(app);

/**
 * Serve your code
 */

app.use(express.static('public'))

/**
 * Listening on connections
 */

ws.on('connection', function (socket) {
  // . . .
});

/**
 * Listen
 */

app.listen(3000);
```

但是，本例中，在用户连接成功后的处理就不同了。我们通过使用一个简单的变量将所有连接过来的用户位置信息记录在内存中。除此之外，我们还记录下连接用户总数，来方便分配给每个用户一个唯一的ID号。这个ID号用来标识positions对象中用户光标位置信息。

```
var positions = {}
  , total = 0

ws.on('connection', function (socket) {
  // . . .
});
```

173 当用户连接过来后，我们首先将其他人的位置信息作为第一条消息内容发送给他。这样一

来，用户就能看到所有连接进来的用户。

最后发送前，我们将positions对象编码为JSON格式：

```
ws.on('connection', function (socket) {
  // you give the socket an id
  socket.id = ++total;

  // you send the positions of everyone else
  socket.send(JSON.stringify(positions));
});
```

当用户发送消息时，我们就假定发送的是JSON格式的位置信息（一个包含x、y坐标的对象），然后将其存储到positions对象中。

```
socket.on('message', function (msg) {
  try {
    var pos = JSON.parse(msg);
  } catch (e) {
    return;
  }

  positions[socket.id] = pos;
});
```

最后，当用户断开连接后，我们就把他的位置信息清除：

```
socket.on('close', function () {
  delete positions[socket.id];
});
```

是不是少了什么？没错！广播！当收到位置信息时，我们需要将其广播给所有其他的人。当socket关闭时，我们得要通知所有其他人并将该光标从屏幕上移除。

我们声明一个broadcast函数，遍历所有其他用户并逐个发送消息给他们。将该函数放在注册connection处理程序的后面：

```
function broadcast (msg) {
  for (var i = 0, l = ws.clients.length; i < l; i++) {
    // you avoid sending a message to the same socket that broadcasts
    if (ws.clients[i] && socket.id != ws.clients[i].id) {
      // you call 'send' on the other clients
      ws.clients[i].send(msg);
    }
  }
}
```

由于我们要发送两类数据，所以，我们在JSON数据包中包含type标示符。 174

发送位置信息时，数据结构如下所示：

```
{
    type: 'position'
  , pos: { x: <x>, y: <y> }
  , id: <socket id>
}
```

当用户断开连接时，发送：

```
{
    type: 'disconnect'
  , id: <socket id>
}
```

因此：

```
socket.on('message', function () {
  // . . .
  broadcast(JSON.stringify({ type: 'position', pos: pos, id: socket.id }));
});
```

并且在关闭时，发送：

```
socket.on('close', function () {
  // . . .
  broadcast(JSON.stringify({ type: 'disconnect', id: socket.id }));
});
```

至此我们完成了服务器部分代码，接着我们开始书写客户端部分。

建立客户端

客户端部分，我们还是从一个简单的HTML开始，监听onload事件：

```
<!doctype html>
<html>
  <head>
    <title>WebSocket cursors</title>
    <script>
      window.onload = function () {
        var ws = new WebSocket('ws://localhost');
        // . . .
      }
    </script>
  </head>
  <body>
    <h1>WebSocket cursors</h1>
  </body>
</html>
```

175

这次，我们主要集中在两个事件上：open和message。

连接建立后，开始监听mousemove事件，用于将用户鼠标的位置实时发送到服务器端，再由服务器发送到其他客户端。

```
ws.onopen = function () {
  document.onmousemove = function (ev) {
    ws.send(JSON.stringify({ x: ev.clientX, y: ev.clientY }));
  }
}
```

如在此前提到的，接收到的消息无外乎两种，要么是用户光标移动了，要么是用户断开连接了：

```
// we instantiate a variable to keep track of initialization for this client
var initialized;

ws.onmessage = function (ev) {
   var obj = JSON.parse(ev.data);

  // the first message is the position of all existing cursors
  if (!initialized) {
    initialized = true;
    for (var id in obj) {
      move(id, obj[id]);
    }
  } else {
    // other messages can either be a position change or
    // a disconnection
    if ('disconnect' == obj.type) {
      remove(obj.id);
    } else {
      move(obj.id, obj.pos);
    }
  }
}
```

接着声明move和remove函数。

对于move函数，我们首先要确保光标对应的元素是否存在。我们通过查找是否有ID为cursor-{id}的DOM元素来完成这个检查。如果元素不存在，那么就创建一个图片元素，设置图片URL以及浮动的样式。

接着就调整它在屏幕上的位置：

176

```
function move (id, pos) {
  var cursor = document.getElementById('cursor-' + id);

  if (!cursor) {
    cursor = document.createElement('img');
    cursor.id = 'cursor-' + id;
    cursor.src = '/cursor.png';
    cursor.style.position = 'absolute';
    document.body.appendChild(cursor);
```

```
  }

  cursor.style.left = pos.x + 'px';
  cursor.style.top = pos.y + 'px';
}
```

对于remove来说，只要简单地将DOM元素删除就好了：

```
function remove (id) {
  var cursor = document.getElementById('cursor-' + id);
  cursor.parentNode.removeChild(cursor);
}
```

运行示例程序

和此前一样，我们需要做的就是运行服务器程序，并通过浏览器来访问。确保打开了多个浏览器标签页（如图10-2所示），来体验实时交互。

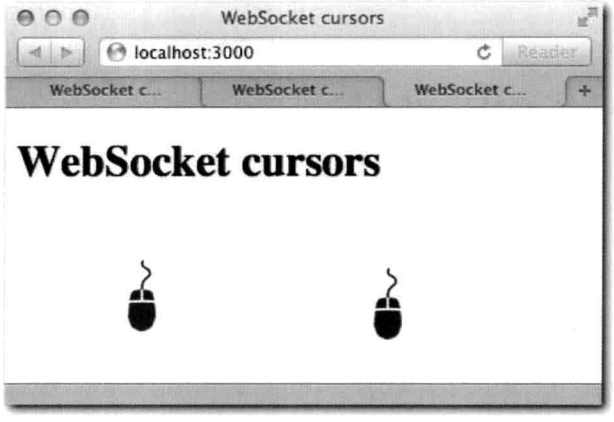


图10-2：移动的鼠标位置实时地反馈到了连入的所有客户端cursor.png图片来自http://thenounproject.com

177 面临一个挑战

尽管本例基本功能都已完备，但是要真的应用到实际产品中还有不少的工作要做。

关闭并不意味着断开连接

当服务器端或者客户端触发close事件时，意味着：TCP连接很可能已经关闭了。而事实上，并非总是如此。电脑有可能会意外关机，网络错误也有可能发生，甚至不小心把一杯水倒在了键盘上都有可能。在类似这样的情况下，close事件可能永远都不会触发！

解决这个问题的方法就是利用超时和心跳检查。要处理这样的情况，我们需要每隔几秒钟

向客户端发送一段消息来确保客户端还“活”着，要是发送失败则认为客户端已经强制断开连接了。

JSON

随着程序复杂度的提升，服务器端和客户端往返的数据量也会变大。

在第二个示例中，会严重依赖JSON进行手动编码和解码工作。因为这部分工作是一个应用中非常常见的任务，所以需要将其抽象出来。

重连

要是客户端临时断开怎么办？大部分应用程序会尝试让用户自动重连。而对于本章中的例子来说，一旦发生断开情况，就只能通过刷新浏览器来重新连接了。

广播

广播对于实时应用来说是非常常规的模式，用于和其他客户端进行交互。对于这部分功能，我们不需要手动去定义自己的广播机制。

WebSocket属于HTML5：早期浏览器不支持

WebSocket是一项新技术。大部分浏览器、代理、防火墙以及杀毒软件都还不完全支持这种新的协议。因此，我们需要一种支持早期浏览器的方案。

解决方案

178

幸运的是，所有这些问题都有解决方案。在下一章中，我们会介绍使用一个名为`socket.io`的模块，它的作用就是在保持简单、加速基于 WebSocket通信的前提下，解决所有上述这些问题。

小结

本章介绍了WebSocket API和WebSocket协议的基础知识，以及如何在Node.js使用WebSocket进行消息的快速传递。同时，通过第一个示例程序，介绍了WebSocket的基本用法。

随后，通过一个多人示例应用展现了WebSocket的威力：它在消息传递时只附带极少的附加数据，使得它能够以尽可能快的速度完成消息传送。

最后，本章提出了WebSocket API并非所有浏览器都支持的弱点，并引出了下一章要介绍的能够解决这一问题的socket.io。

CHAPTER

11 Socket.IO

正如此前提过的，要将WebSocket用到应用中，并非简单地把 WebSocket实现了就可以的。 179

Socket.IO是我开发的一个项目，用于解决此前提过的使用WebSocket过程中一系列常见问题。它在保留了简洁API的前提下，提供了非常好的灵活性。

Server API

```
io.listen(app);
io.sockets.on('connection', function
(socket) {
  socket.emit('my event', { my: 'object' });
});
```

Browser/Client API

```
var socket = io.connect();
socket.on('my event', function (obj) {
  console.log(obj.my);
});
```

180

传输

Socket.IO最诱人的特性之一就是消息的传递是基于传输的，而非全部依靠WebSocket，也就是说，Socket.IO可以在绝大部分的浏览器和设备上运行，从IE6到iOS都支持。

例如，在使用一项称为long polling技术的时候，就可以通过Ajax来实现实时消息传输。简单来说，这项技术是通过持续发送一系列的Ajax请求来实现的，但是，当服务器端没有数据返回到客户端时，连接还会持续打开20～50秒，以确保不再有额外的数据通过HTTP请求/响应头传递过来。

Socket.IO会自动使用像long polling这样复杂的技术，但其API保持了与WebSocket一样的简洁。

另外，即使浏览器端支持的WebSocket被代理或者防火墙禁止了，Socket.IO依然能够通过采用别的技术来处理这类问题。

断开 VS 关闭

Socket.IO带来的另一个基础功能就是对超时的支持。正如我们在第6章和第10章中讨论的，在实际情况下，应用不能依赖TCP连接一定能够正常关闭。

本章中，我们使用Socket.IO，监听的是`connect`事件而不是`open`事件，以及`disconnect`事件而不是`close`事件。原因是Socket.IO提供了可靠的事件机制。若客户端停止传输数据，但在一定的时间内又没有正常地关闭连接，Socket.IO就认为它是断开连接了。

这样一来，就能够让你专注在应用逻辑本身，而无须去过多担心网络的各种不确定情况。

当连接丢失时，Socket.IO默认会自动重连。

事件

至此，你看到了，典型的Web通信方式是通过HTTP来收取（发送）文档（资源）的。但是，在实时Web世界中，都是基于事件传输的。

Socket.IO仍然允许你像WebSocket那样传输简单文本信息，除此之外，它还支持通过分发（emit）和监听（listen）事件来进行JSON数据的收发。下面这个例子展示了Socket.IO像WebSocket那样进行消息的收发：

```
io.sockets.on('connection', function (socket) {
  socket.send('a');
  socket.on('message', function (msg) {
181     console.log(msg);
  });
});
```

回想一下第10章光标的例子，要是用Socket.IO来实现的话，代码可以变得非常简单：

Client code

```
var socket = io.connect();

socket.on('position', move);

socket.on('remove', remove);
```

注意了，使用Socket.IO可以在应用中根据数据的含义进行频道分类，不再需要对单一事件（消息）中收到的事件进行解析。事件可以接收任意数量的JSON编码的参数：Number、Array、String、Object，等等。

命名空间

Socket.IO还提供了另一个强大的特性，它允许在单个连接中利用命名空间来将消息彼此区分开来。

有的时候，应用程序需要进行逻辑拆分，但考虑到性能、速度之类的原因，使用同一个连接还是可以接受的。考虑到我们无法事先获悉客户端的速度的快慢、浏览器的好坏，不依赖同时打开过多的连接通常是个不错的主意。

因此，Socket.IO允许监听多个命名空间中的connection事件。

```
io.sockets.on('connection');
io.of('/some/namespace').on('connection')
io.of('/some/other/namespace').on('connection')
```

尽管当通过如下方式从浏览器中获取连接时，可以获取到不同的连接对象，但是，通常只会使用一个传输通道（像WebSocket连接一样）：

```
var socket = io.connect();
var socket2 = io.connect('/some/namespace');
var socket3 = io.connect('/some/other/namespace');
```

某些场景下，为了更好地抽象，应用程序的部分代码或模块书写的时候完全是互相独立的。部分客户端JavaScript代码可能完全不知道另外一部分并行执行的代码。

比如，构建一个社交网络，在农场游戏旁边展示一个实时聊天程序。尽管，它们可以共享一些如授权用户的信息这样的通用的数据，但书写代码时让它们都能够完全控制一个socket依然是个很好的主意。 182

归功于命名空间（也可以称为多路传输），那样的socket不必非得是自己分配的真正的TCP socket。Socket.IO对同样的资源（为用户选择的传输通道）进行频道切分，并将数据传输给对应的回调函数。

至此，你已经了解了Socket.IO和WebSocket之间主要的不同点，接下来，是时候开始构建第一个示例程序——聊天程序了。

聊天程序

初始化程序

和websocket.io一样，将socket.io绑定到常规的http.Server就可以处理socket.io的请求和响应了：

package.json

```
{
    "name": "chat.io"
  , "version": "0.0.1"
  , "dependencies": {

        "express": "2.5.1"
      , "socket.io": "0.9.2"
    }
}
```

按照惯例，创建完package.json文件后，确保要运行npm install来安装所有的依赖。

构建服务器

和websocket.io一样，现在构建一个带static中间件的普通Express应用[1]：

server.js

```
/**
 * 模块依赖
 */

var express = require('express')
  , sio = require('socket.io')

/**
 * 创建 app
.*/

app = express.createServer(
    express.bodyParser()
, express.static('public')
);
```

183

1 译者注：Express3中需要首先用http.createServer来将app绑到server上，需要添加：var server = require("http").createServer(app); sio.listen(server);。同时，Express3中不能将中间件放在express(…)构造器参数中，需要使用app.use(express.bodyParser());。

```
/**
 * 监听
 */

app.listen(3000);
```

接下来，是时候将socket.io绑定到APP上了。和websocket.io一样，调用sio.listen即可：

```
var io = sio.listen(app);
```

接着，设置连接监听器：

```
io.sockets.on('connection', function (socket) {
  console.log('Someone connected');
});
```

好了，现在一旦有连接进来，就会在控制台输出简单的信息了。由于 Socket.IO是自定义的API，所以必须要在浏览器中载入Socket.IO客户端。

构建客户端

因为使用了static中间件，并将public目录设置为了要托管的目录，接下来我们就在该目录下创建一个index.html文件。

这次，为了方便，我们将聊天程序的逻辑部分从HTML代码中分离出来，单独放到chat.js文件中。

Socket.IO中方便的一点在于，当它绑定到http.Server后，所有以/socket.io开始的URL都会被其拦截。

Socket.IO还自带了其浏览器端运用的代码。因此，我们无须担心如何获取和托管客户端代码。

注意，下面这段代码，我们创建了一个<script>标签，并添加对/socket.io/socket.io.js的引用：

184

index.html

```
<!doctype html>
<html>
  <head>
    <title>Socket.IO chat</title>
    <script src="/socket.io/socket.io.js"></script>
    <script src="/chat.js"></script>
    <link href="/chat.css" rel="stylesheet" />
  </head>
  <body>
```

```
    <div id="chat">
      <ul id="messages"></ul>
      <form id="form">
        <input type="text" id="input" />
        <button>Send</button>
      </form>
    </div>
  </body>
</html>
```

现在，chat.js确保客户端载入完成后就开始连接。若一切都正常，就应该能在控制台看到Someone connected这样的输出了。

chat.js

```
window.onload = function () {
  var socket = io.connect();
}
```

所有Socket.IO客户端代码暴露出来的方法和类都在io命名空间中。

io.connect和new WebSocket类似，不过更智能。本例中，因为没有传递参数给它，所以，它会尝试向页面所在的主机发起连接，这也符合本例的要求。

使用如下命令来运行上述应用：

```
$ node server
```

接着，通过浏览器访问http://localhost:3000。应当就能看到 Socket.IO的日志器输出的关于Socket.IO内部发生的情况；比如，从输出结果中能够看到客户端使用的传输方式（见图11-1）。

185

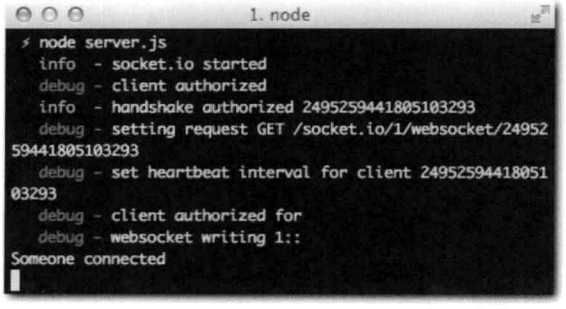


图11-1：socket.io自身的调试信息以及应用通过console.log输出的信息

像本例中那样，如果通过主流浏览器来连接Socket.IO服务器，那么 Socket.IO就会使用WebSocket来通信。

Socket.IO总会尝试选择对用户来说速度最快、对服务器性能来说最好的方法来建立连接，要是条件达不到，那么首先会保证连接正常。

事件和广播

现在已经成功连接上了，那么接下来就应该开始编写Socket.IO服务器端基础代码了。

广播用户加入信息

当有用户连接到服务器时，我们要通知其他人有人连接进来了。由于这属于非用户发送的特殊消息，所以我们称之为通告，并用相应的样式标识出来。

客户端首先要做的就是询问用户的名字。

因为要在用户成功连接后才能聊天，所以我们需要先将聊天窗口隐藏：

chat.css

```
/* … */
#chat { display: none }
```

接着，在用户连接成功后就将聊天窗口显示出来。这里我们需要监听已创建socket上的connect事件（此前定义在window.onload中的函数）：

186

chat.js

```
socket.on('connect', function () {
  // 通过join事件发送昵称
  socket.emit('join', prompt('What is your nickname?'));

  // 显示聊天窗口
  document.getElementById('chat').style.display = 'block';
});
```

服务器端则需要监听join事件，并将收到的消息通知给其他人，告诉他们有新用户连接进来了。我们将此前的io.sockets的connection处理器替换为如下形式即可：

server.js

```
// …
io.sockets.on('connection', function (socket) {
  socket.on('join', function (name) {
    socket.nickname = name;
    socket.broadcast.emit('announcement', name + ' joined the chat.');
  });
});
```

注意，socket.broadcast.emit中的broadcast是一种标志信息，它改变了emit函数的行为。

要是在上述例子中，我们直接调用socket.emit，那么仅仅是将消息返回给客户端。而我们真正需要的是将消息广播给所有其他的用户，所以这里需要broadcast标志。

再次回到客户端，我们需要监听announcement事件，并在DOM的消息列表中创建一个元素。将下述代码添加到connect处理器的最后：

chat.js

```
socket.on('announcement', function (msg) {
  var li = document.createElement('li');
  li.className = 'announcement';
  li.innerHTML = msg;
  document.getElementById('messages').appendChild(li);
});
```

广播聊天消息

接下来，我们要实现让用户发消息给其他人。

187 当用户在表单中输入消息并提交时，我们需要分发一个text事件，并将输入的消息发送出去：

chat.js

```
var input = document.getElementById('input');
document.getElementById('form').onsubmit = function () {
  socket.emit('text', input.value);

  // 重置输入框
  input.value = '';
  input.focus();

  return false;
}
```

很显然，当用户书写完消息并提交后，肯定不希望服务器再将该消息发回来。所以，在消息发送后，我们就立刻调用addMessage将消息显示出来：

chat.js

```
function addMessage (from, text) {
  var li = document.createElement('li');
  li.className = 'message';
  li.innerHTML = '<b>' + from + '</b>: ' + text;
  document.getElementById('messages').appendChild(li);
```

```
}
document.getElementById('form').onsubmit = function () {
  addMessage('me', input.value);
  // . . .
}
```

在从其他用户处收到消息后，我们也需要做同样的处理。这里，我们可以简单地传递一个对addMessage函数的引用，同时在服务器端，要确保广播消息时参数都正确。

chat.js

```
// …
socket.on('text', addMessage);
```

server.js

```
socket.on('text', function (msg) {
  socket.broadcast.emit('text', socket.nickname, msg);
});
```

至此，客户端和服务器端的代码大致如下所示：

188

chat.js

```
window.onload = function () {
  var socket = io.connect();
  socket.on('connect', function () {
    // 通过join事件发送昵称
    socket.emit('join', prompt('What is your nickname?'));

    // 显示聊天窗口
    document.getElementById('chat').style.display = 'block';

    socket.on('announcement', function (msg) {
      var li = document.createElement('li');
      li.className = 'announcement';
      li.innerHTML = msg;
      document.getElementById('messages').appendChild(li);
    });
  });

  function addMessage (from, text) {
    var li = document.createElement('li');
    li.className = 'message';
    li.innerHTML = '<b>' + from + '</b>: ' + text;
    document.getElementById('messages').appendChild(li);
  }
```

```
  var input = document.getElementById('input');
  document.getElementById('form').onsubmit = function () {
    addMessage('me', input.value);
    socket.emit('text', input.value);

    // 重置输入框
    input.value = '';
    input.focus();

    return false;
  }

  socket.on('text', addMessage);
}
```

server.js

```
/**
 * 模块依赖
 */

var express = require('express')
  , sio = require('socket.io')

/**
 * 创建app
 */

app = express.createServer(
    express.bodyParser()
  , express.static('public')
);

/**
 * 监听
 */

app.listen(3000);

var io = sio.listen(app);

io.sockets.on('connection', function (socket) {
  socket.on('join', function (name) {
    socket.nickname = name;
    socket.broadcast.emit('announcement', name + ' joined the chat.');
  });

  socket.on('text', function (msg) {
    socket.broadcast.emit('text', socket.nickname, msg);
```

189

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

运行server.js应当就能看到一个如图11-2所示的功能完整的实时聊天应用。

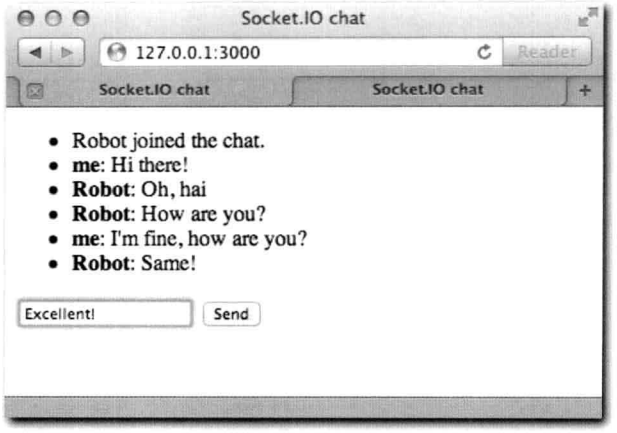


图11-2：聊天程序实战。这里通过多个浏览器标签页来聊天

接下来，我们要了解更多关于事件回调函数的内容，以及如何使用它们实现新特性。 190

消息接收确认

在聊天应用中，我们在用户按下回车键后立刻调用addMessage，这会让人产生一种错觉，感觉消息瞬间就发送成功了。

和WebSocket一样，Socket.IO并不强制对每条发送的消息做回应。不过，有的时候，我们需要确认消息是否达到。Socket.IO把这类确认消息叫做确认（acknowledgment）。

要实现这样的通知响应，我们要做的就是在分发事件的时候提供一个回调函数。

首先，我们要获得通过addMessage函数创建的消息元素，以便于在收到确认响应后，对该消息添加一个CSS类。接着，就可以在该消息后显示一个漂亮的小图标。

/chat.js

```
function addMessage (from, text) {
  // …
  return li;
}
```

接下来，我们添加一个回调函数。Socket.IO也允许在接收确认响应的回调函数中接收数据。本例中，可以在接收到消息后发送一个时间戳：

/chat.js

```
document.getElementById('form').onsubmit = function () {
  var li = addMessage('me', input.value);
  socket.emit('text', input.value, function (date) {
    li.className = 'confirmed';
    li.title = date;
  });
```

在服务器端，Socket.IO会添加一个回调函数作为事件的最后一个参数：

/server.js/

```
// …
socket.on('text', function (msg, fn) {
  // …
  // 确认消息已接收
  fn(Date.now());
});
```

191 现在，当服务器端接收到客户端发送的消息，发送确认响应后，一个CSS类，以及`title`属性就会添加到消息列表的最后一个元素中。这样做会带来两个好处：用户按下回车键后立刻就看到输入的消息，这使得应用获得最好的响应；另外还能通过CSS给用户反馈（比如：在消息后面显示一个如图11-3所示的对钩图标）。

图11-3：本例中，在确认响应收到后，利用CSS设置一个背景图

一个轮流做DJ的应用

要是把我们的聊天应用扩展为一个DJ的应用，那得有多酷啊？

- 服务器初始选择一名DJ。

- DJ有权利请求查询API，获取查询结果，选择一首歌。然后将这首歌广播给所有其他听众。
- 当DJ离开时，系统就会开放DJ人选给下一个用户。

扩展聊天应用

聊天应用的基础足够强健，可以添加DJ特性。

首先要做的是，初始化的时候选择一名DJ。因为我们还要记录当前播放的歌曲，所以需要申明两个变量：currentSong和dj。

由于DJ是可以更换的，所以，我们定义一个elect函数来执行选择DJ和发布公告的任务。当join事件分发时，DJ就会被选出来，或者（已经有DJ的情况下）将当前播放的歌曲（currentSong）发送给刚加入的用户。后面，实现了搜索之后，currentSong会被包含在一个对象中发送出去。

192

server.js

```
var io = sio.listen(app)
, currentSong
  , dj

function elect (socket) {
  dj = socket;
  io.sockets.emit('announcement', socket.nickname + ' is the new dj');
  socket.emit('elected');
  socket.dj = true;
  socket.on('disconnect', function () {
    dj = null;
    io.sockets.emit('announcement', 'the dj left - next one to join becomes dj');
  });
}

io.sockets.on('connection', function (socket) {
  socket.on('join', function (name) {
    socket.nickname = name;
    socket.broadcast.emit('announcement', name + ' joined the chat.');
    if (!dj) {
      elect(socket);
    } else {
      socket.emit('song', currentSong);
    }
  });
  // …
});
```

elect函数完成如下几件事情：

1. 将当前用户选为DJ。
2. 分发公告给所有人DJ已经选取完毕。
3. 通过分发elected事件，让DJ知道自己被选中了。
4. 当DJ断开连接时，将DJ的名额留给下一个进来的人。

客户端，将歌曲选择的界面代码添加到聊天表单下面即可：

index.html

```
<div id="playing"></div>
<form id="dj">
  <h3>Search songs</h3>
  <input type="text" id="s" />
  <ul id="results"></ul>
  <button type=submit>Search</button>
</form>
```

193 集成Grooveshark API

Grooveshark（http://grooveshark.com）提供了一个简单易用的API ——TinySong，能满足我们的需求。

TinySong允许如下的查询方式：

```
GET http://tinysong.com/s/Beethoven?key={apiKey}&format=json
```

返回结果如下：

```
[
  {
    "Url": "http:\/\/tinysong.com\/7Wm7",
    "SongID": 8815585,
    "SongName": "Moonlight Sonata",
    "ArtistID": 1833,
    "ArtistName": "Beethoven",
    "AlbumID": 258724,
    "AlbumName": "Beethoven"
  },
  {
    "Url": "http:\/\/tinysong.com\/6Jk3",
    "SongID": 564004,
    "SongName": "Fur Elise",
    "ArtistID": 1833,
    "ArtistName": "Beethoven",
```

```
    "AlbumID": 268605,
    "AlbumName": "Beethoven"
  },
  {
    "Url": "http:\/\/tinysong.com\/8We2",
    "SongID": 269743,
    "SongName": "The Legend Of Lil' Beethoven",
    "ArtistID": 7620,
    "ArtistName": "Sparks",
    "AlbumID": 204019,
    "AlbumName": "Sparks"
  }
]
```

因此，我需要暴露一个叫search的Socket.IO事件，内部使用superagent模块来进行对查询API的调用并返回其结果。

将superagent模块添加到package.json中，并添加模块依赖：

server.js

194

```
var express = require('express')
  , sio = require('socket.io')
  , request = require('superagent')
```

package.json

```
  , "dependencies": {
        "express": "2.5.1"
      , "socket.io": "0.9.2"
      , "superagent": "0.4.0"
    }
```

注意了，在查询URL中必须要包含API key，API key可以去 http://tinysong.com网站申请。

定义apiKey的方式如下：

server.js

```
var io = sio.listen(app)
  , apiKey = '{ your API key }'
  , currentSong
  , dj
```

接着，定义search事件：

```
socket.on('search', function (q, fn) {
    request('http://tinysong.com/s/' + encodeURIComponent(q)
      + '?key=' + apiKey + '&format=json', function (res) {
```

```
        if (200 == res.status) fn(JSON.parse(res.text));
      });});
```

注意，上述代码中需要手动解析JSON返回结果。这是因为TinySong目前没有发送正确的Content-Type响应头信息，导致 superagent无法自动启用JSON解析功能。

接着，我们要将查询功能添加到应用中，但是只能让DJ能够选择歌曲。

在chat.css文件中，添加如下两行代码：

```
#results a { display: none; }
form.isDJ #results a { display: inline; }
```

195 接着我们要添加查询逻辑，将从Socket.IO回调函数中获取到的查询结果展示出来供其选择。

在chat.js文件中，添加如下内容：

```
// search form
var form = document.getElementById('dj');
var results = document.getElementById('results');
form.style.display = 'block';
form.onsubmit = function () {
  results.innerHTML = '';
  socket.emit('search', document.getElementById('s').value, function (songs) {
    for (var i = 0, l = songs.length; i < l; i++) {
      (function (song) {
        var result = document.createElement('li');
        result.innerHTML = song.ArtistName + ' - <b>' + song.SongName + '</b> ';
        var a = document.createElement('a');
        a.href = '#';
        a.innerHTML = 'Select';
        a.onclick = function () {
          socket.emit('song', song);
          return false;
        }
        result.appendChild(a);
        results.appendChild(result);
      })(songs[i]);
    }
  });
  return false;
};

socket.on('elected', function () {
  form.className = 'isDJ';
});
```

上述代码中大部分都在处理DOM。因为服务器端从TinyURL的API中把所有的歌曲都发送到了客户端，因此可任由客户端对其进行渲染。在本例中，我们首先显示歌手名，紧跟着歌曲

名（对应的是ArtistName和SongName）。

在收到elected事件后，通过改变表单的className来显示每首歌曲的选择链接。

点击Select链接后，客户端发送song事件到服务器端，服务器端要做的就是记录当前歌曲，并广播给所有人。在server.js文件中，添加如下代码：

196

server.js

```
socket.on('song', function (song) {
  if (socket.dj) {
    currentSong = song;
    socket.broadcast.emit('song', song);
  }
});
```

至此，用户已经可以搜索和接收歌曲了，最后就剩下播放歌曲的功能了。 元素就是为此而预留的。

播放歌曲

和addMessage函数一样，我们也需要定义一个函数来标记当前正在播放的歌曲。

将下述代码添加到chat.js文件中。play函数就是简单地将当前播放的歌曲按照歌手、歌名的方式展现出来，与此同时，它还注入了一个 iframe，指向TinySong的Url字段，用来播放歌曲。

```
var playing = document.getElementById('playing');
function play (song) {
  if (!song) return;
  playing.innerHTML = '<hr><b>Now Playing: </b> '
    + song.ArtistName + ' ' + song.SongName + '<br>';

  var iframe = document.createElement('iframe');
  iframe.frameborder = 0;
  iframe.src = song.Url;
  playing.appendChild(iframe);
};
```

与此前一样，该函数用于两个场景：DJ（为自己）选择了一首歌后，以及DJ向其他普通用户发送song事件的时候。

对于第二个场景，我们只需要在chat.js中，将play函数作为回调函数传递给song事件。

```
socket.on('song', play);
```

对于DJ要立刻听到所选歌曲，我们就需要在他选择之后调用play函数。回到onclick处理器，在那里需要分发song事件给服务器并调用play函数，因此，该处理器代码就会是如下

所示的样子：

```
a.onclick = function () {
  socket.emit('song', song);
197 >  play(song);
  return false;
}
```

完成！回忆一下一开始服务器端join事件处理器部分代码，要是有currentSong就会分发song事件。也就是说，不仅在DJ选歌曲前加入的用户能播放歌曲，在这之后填写完昵称加入的用户也能播放歌曲（见图11-4）。

聊天+DJ应用完整代码大致如下所示：

server.js

```
var express = require('express')
  , sio = require('socket.io')
  , request = require('superagent')

app = express.createServer(
    express.bodyParser()
  , express.static('public')
);

app.listen(3000);

var io = sio.listen(app)
  , apiKey = '{ your API key }'
  , currentSong
  , dj

function elect (socket) {
  dj = socket;
  io.sockets.emit('announcement', socket.nickname + ' is the new dj');
  socket.emit('elected');
  socket.dj = true;
  socket.on('disconnect', function () {
    dj = null;
    io.sockets.emit('announcement', 'the dj left - next one to join becomes dj');
  });
}

io.sockets.on('connection', function (socket) {
  socket.on('join', function (name) {
    socket.nickname = name;
    socket.broadcast.emit('announcement', name + ' joined the chat.');
    if (!dj) {
      elect(socket);
```

```
    } else {
      socket.emit('song', currentSong);
    }
  });
  socket.on('song', function (song) {
    if (socket.dj) {
      currentSong = song;
      socket.broadcast.emit('song', song);
    }
  });

  socket.on('search', function (q, fn) {
    request('http://tinysong.com/s/' + encodeURIComponent(q)
      + '?key=' + apiKey + '&format=json', function (res) {
      if (200 == res.status) fn(JSON.parse(res.text));
    });
  });

  socket.on('text', function (msg) {
    socket.broadcast.emit('text', socket.nickname, msg);
  });
});
```

198

chat.js

```
window.onload = function () {
  var socket = io.connect();
  socket.on('connect', function () {
    // 通过join事件发送昵称
    socket.emit('join', prompt('What is your nickname?'));

    // 显示聊天窗口
    document.getElementById('chat').style.display = 'block';

    socket.on('announcement', function (msg) {
      var li = document.createElement('li');
      li.className = 'announcement';
      li.innerHTML = msg;
      document.getElementById('messages').appendChild(li);
    });
  });

  function addMessage (from, text) {
    var li = document.createElement('li');
    li.className = 'message';
    li.innerHTML = '<b>' + from + '</b>: ' + text;
    document.getElementById('messages').appendChild(li);
  }
```

```
  var input = document.getElementById('input');
  document.getElementById('form').onsubmit = function () {
    addMessage('me', input.value);
    socket.emit('text', input.value);
199 // 重置输入框
  input.value = '';
  input.focus();

  return false;
}

socket.on('text', addMessage);

// 播放歌曲
var playing = document.getElementById('playing');
function play (song) {
  if (!song) return;
  playing.innerHTML = '<hr><b>Now Playing: </b> '
    + song.ArtistName + ' ' + song.SongName + '<br>';

  var iframe = document.createElement('iframe');
  iframe.frameborder = 0;
  iframe.src = song.Url;
  playing.appendChild(iframe);
};
socket.on('song', play);

// 查询表单
var form = document.getElementById('dj');
var results = document.getElementById('results');
form.style.display = 'block';
form.onsubmit = function () {
  results.innerHTML = '';
  socket.emit('search', document.getElementById('s').value, function (songs) {
    for (var i = 0, l = songs.length; i < l; i++) {
      (function (song) {
        var result = document.createElement('li');
        result.innerHTML = song.ArtistName + ' - <b>' + song.SongName + '</b> ';
        var a = document.createElement('a');
        a.href = '#';
        a.innerHTML = 'Select';
        a.onclick = function () {
          socket.emit('song', song);
          play(song);
          return false;
        }
        result.appendChild(a);
        results.appendChild(result);
      })(songs[i]);
```

```
      }
    });
    return false;
  };
  socket.on('elected', function () {
    form.className = 'isDJ';
  });
}
```

200

index.html

```
<!doctype html>
<html>
  <head>
    <title>Socket.IO chat</title>
    <script src="/socket.io/socket.io.js"></script>
    <script src="/chat.js"></script>
    <link href="/chat.css" rel="stylesheet" />
  </head>
  <body>
    <div id="chat">
      <ul id="messages"></ul>
      <form id="form">
        <input type="text" id="input" />
        <button>Send</button>
      </form>
      <div id="playing"></div>

      <form id="dj">
        <h3>Search songs</h3>
        <input type="text" id="s" />
        <ul id="results"></ul>
        <button>Search</button>
      </form>
    </div>
  </body>
</html>
```

201

图11-4：DJ+聊天应用实战

小结

Socket.IO提供了足够简单但却十分强大的API，用于构建实时消息快速通信的应用。Socket.IO不仅保证了消息会尽可能快地进行传输，而且还能在所有的浏览器以及绝大部分移动设备上运行。

本章介绍了如何构建一个简单的应用以及如何使用Socket.IO提供的语法糖。还介绍了通过使用事件的方式来组织用户和服务器端传输的不同类型的数据。

书写实时应用最基础的部分就是广播。本章介绍了如何在服务器端分发事件给所有人，以及如何将一个用户的消息传递给其他人。作为例子，我们使用该技术实现了把DJ挑选的歌曲播放给其他所有的用户。

还有一件值得一提的事情是，绝大部分的功能都在客户端实现：编写代码来根据不同的消息类型进行相应的界面展现。由于本章只关注Socket.IO，所以没有介绍模板引擎以及其他更高层的框架，使用这些可以避免和DOM API直接打交道，不过，在实际情况下，随着应用程序复杂度的提高，这些也会变得更加复杂。

PART

IV 数据库

CHAPTER

12 MongoDB

MongoDB是一个面向文档，schema无关（schema-less）的数据库，它非常适合于Node.js应用以及云端部署。 205

与MySQL及PostgreSQL是根据固定的结构设计（schema）将数据存储在表中不同，MongoDB可以将任意类型的文档数据存储到集合中（schema无关），这也是MongoDB最有意思的特性之一。

例如，创建下面这张为Web应用保存用户信息的表： 206

First	Last	Email	Twitter
Guillermo	Rauch	rauchg@gmail.com	rauchg

在构建应用时，决定将用户信息按照上面这样的结构设计进行存储。需要如下这些信息：first name、last name、email以及Twitter ID。

随着应用的发展、需求发生了改变，或者随着时间的推移，又有了新的需求，可能需要增加或者删除表中的某些列。

然而，这样一个基础性问题，若要通过传统的（SQL）数据库来实现，从操作上和性能上

来讲都需要耗费非常高的成本来修改表设计。

比如，在MySQL中，每一次修改表的设计结构，都需要运行如下这个命令才能实现添加一个新的列：

```
$ mysql
  > ALTER TABLE profiles ADD COLUMN . . .
```

对于删除一列或多列的情况也是如此。

在MongoDB中，则可以将数据都看作文档，其设计非常灵活。当有数据存储后，这些文档就会以一种非常接近（或者说在绝大多数情况下就是JSON格式）JSON格式的形式存储：

```
{
    "name": "Guillermo"
  , "last": "Rauch"
  , "email": "rauchg@gmail.com"
  , "age": 21
  , "twitter": "rauchg"
}
```

MongoDB还有一个非常重要的特性，能够将其与其他键—值形式的 NoSQL数据库区别开来，就是文档可以是任意深度的。

例如，可以将社交信息以如下结构进行存储，而不是全都将它们直接作为文档的键来存储：

```
{
    "name": "Guillermo"
  , "last": "Rauch"
  , "email": "rauchg@gmail.com"
  , "age": 21
  , "social_networks": {
        "twitter": "rauchg"
      , "facebook": "rauchg@gmail.com"
      , "linkedin": 27760647
    }
}
```

207

如上述代码所示，数据类型可以混用。这里，twitter和facebook信息都是字符串类型的，而linkedin是数字类型。当通过Node.js获取到存储的文档数据后，拿到的数据类型也是和存储时一模一样的。

本章将会介绍MongoDB最常用的功能，以及如何获得最灵活、最高效（通过索引）存储方案的最佳实践。本章还会介绍多种查询文档的方法，以及如何使用Mongoose来简化使用方式，Mongoose是我和 Nathan White一同书写的Node.js模块，它为MongoDB和JavaScript提供了传统关系型数据库ORM（Object-Relational Mapper）的部分功能。对MongoDB来说，这类项目

更贴切的叫法应该是ODM： Object Document Mapper。

安装

记住，本章使用的是MongoDB分支上最新的2.x版本。[1]

通过官网：www.mongodb.org/的下载页面就能获取到MongoDB。与此同时，你或许也会有兴趣看下各平台下MongoDB的快速指南： www.mongodb.org/display/DOCS/Quickstart。

通过执行mongo客户端，看到如图12-1所示的界面就表示安装成功了。

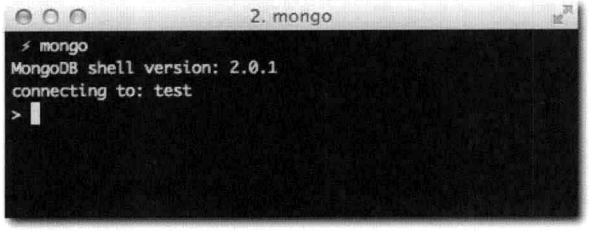


图12-1：MongoDB shell

要是无法连接到MongoDB，再检查下安装过程是否正确，同时通过进程管理器查看确保MongoDB服务器（mongod）正常运行。

使用MongoDB：一个用户认证的例子

208

通过Node.js操作MongoDB文档数据最主要的方式就是通过驱动器（driver）。通常在Node.js中驱动器指的就是一些基本的API，它懂得数据库网络层协议和其通信，并知道如何编码和解码存储的数据。

本例中选择的是由Christian Amor Kvalheim开发的node-mongodb-native。我们可以在Node包管理器（NPM）通过mongodb名字查找到这个驱动器。

第一个例子，我们创建一个简单的Express应用，实现将用户信息存储到MongoDB中，并实现用户注册、登录的功能。

构建应用程序

首先为项目创建package.json文件，声明项目所依赖的包。本例中，我们只需要express和mongodb。除此之外，我们还需要使用jade模板引擎：

```
{
    "name": "user-auth-example"
```

1 译者注：截止到本文翻译期间，MongoDB最新版本已经是2.4.6版本了。

```
  , "version": "0.0.1"
  , "dependencies": {
      "express": "2.5.8"
    , "mongodb": "0.9.9"
    , "jade": "0.20.3"
    }
}
```

创建Express App

首先，我们通过使用require引入所需的依赖包：

```
/**
 * 模块依赖
 */

var express = require('express')
  , mongodb = require('mongodb')
```

由于本应用需要进行表单处理，所以，要用到bodyParser中间件。

因为我们要对用户进行认证，并将该信息保留下来，所以还需要用到 session中间件（它依赖Connect中的cookieParser中间件，在第8章中做过介绍）。

```
/**
 * 构建应用程序
 */
```

209

```
app = express.createServer()

/**
 * 中间件
 */

app.use(express.bodyParser());
app.use(express.cookieParser());
app.use(express.session({ secret: 'my secret' }));
```

因为本例中选用了jade模板引擎，所以还需要设置Express的view engine配置：

```
/**
 * 指定视图选项
 */

app.set('view engine', 'jade');

// 若使用了Express 3，则不需要下面这行代码


app.set('view options', { layout: false });
```

默认情况下，视图查找路径是views/。我们需要创建该目录，并在该目录下创建一个layout.jade文件来嵌入所有其他的视图：

```
doctype 5
html
  head
    title MongoDB example
  body
    h1 My first MongoDB app
    hr
    block body
```

尽管这并不在本章范畴，但是，作为Node.js应用中最流行的模板引擎之一，学习一些关于jade的内容还是很重要的。

- jade使用的是缩进（默认两个空格，应当避免使用tab），而不是复杂的嵌套XML、HTML标签。代码如下所示：

  ```
  p
     span Hello world
  ```

  等效于 Hello world 。

- 使用jade只需输入标签名，后面紧跟内容h1 My First MongoDB app即可，不需要这样完整地书写<h1>My first MongoDB app</h1>。
  这里使用doctype 5自动插入了HTML5的doctype。 210

- 代码中还使用了特殊的关键字block，这样其他视图文件就能嵌入到这个位置。这就是为什么要称该文件为layout的原因。其他还包括if和else这样特殊的关键字。

- 属性的写法看起来就像是HTML和JavaScript代码的混合体，并且非常容易嵌入变量（或者locals，express将从controller中暴露给视图层的变量称为locals）：

  ```
  a(href=#, another=attribute, dynamic=someVariable) My link
  ```

- 可以通过#{}这样的写法来嵌入变量：

  ```
  p Welcome back, #{user.name}
  ```

接着定义路由。我们需要一个主页的路由（/）、注册页面的路由（/signup）以及登录页面的路由（/login）：

```
/**
 * 默认路由
 */

app.get('/', function (req, res) {
  res.render('index', { authenticated: false });
});
```

```
/**
 * 登录路由
 */

app.get('/login', function (req, res) {
  res.render('login');
});

/**
 * 注册路由
 */

app.get('/signup', function (req, res) {
  res.render('signup');
});
```

在主页的路由中（/），我们传递了一个值为false的本地变量authenticated。等实现了登录功能后，我们会动态输出该变量。

我们在index模板中需要用到authenticated变量：

index.jade

```
extends layout
block body
if (authenticated)
  p Welcome back, #{me.first}
  a(href="/logout") Logout
else
  p Welcome new visitor!
  ul
    li: a(href="/login") Login
    li: a(href="/signup") Signup
```

211

如图12-2所示，注册和登录视图就是简单的表单视图：

signup.jade

```
extends layout
block body
form(action="/signup", method="POST")
  fieldset
    legend Sign up
    p
      label First
      input(name="user[first]", type="text")
    p
      label Last
```

```
      input(name="user[last]", type="text")
    p
      label Email
      input(name="user[email]", type="text")
    p
      label Password
      input(name="user[password]", type="password")
    p
      button Submit
    p
      a(href="/") Go back
```

login.jade

```
extends layout
block body
form(action="/login", method="POST")
  fieldset
    legend Log in
    p
      label Email
      input(name="user[email]", type="text")
    p
      label Password
      input(name="user[password]", type="password")
    p
      button Submit
    p
      a(href="/") Go back
```

图12-2：/signup路由

212 最后，我们需要让应用程序监听3000端口：

```
/**
 * 监听
 */

app.listen(3000);
```

要是通过浏览器访问，就能够很容易地访问到所有定义好的路由。

连接MongoDB

在查找文档（登录功能所需）和插入文档（注册功能所需）之前，先要连接MongoDB服务器并选择正确的数据库。

我们需要在服务器监听前就连接数据库。因为应用逻辑完全依赖数据库的操作，在还未能查询数据前就接收请求显然是不行的。

由于我们直接使用了MongoDB的驱动器，所以API有些长。不过，我们的目的是要暴露像`app.users`这样MongoDB集合的API，方便在路由定义中轻松对数据库进行操作。

首先通过创建一个`mongodb.Server`并提供IP和端口来初始化服务器：

213

```
/**
 * 连接数据库
 */

var server = new mongodb.Server('127.0.0.1', 27017)
```

接着告诉驱动器去连接数据库。比如，取一个叫`my-website`的数据库名字。在MongoDB中，要是指定的名字不存在，就会创建一个数据库。

```
new mongodb.Db('my-website', server).open(function (err, client) {
```

要是连接数据库失败，我们就得终止进程：

```
// 在有错误的情况下不允许应用程序启动
if (err) throw err;
```

要是连接成功则打印成功的消息：

```
console.log('\033[96m  + \033[39m connected to mongodb');
```

接着，建立集合：

```
// 建立集合快捷方式
app.users = new mongodb.Collection(client, 'users');
```

最后，让Express服务器监听端口准备操作集合：

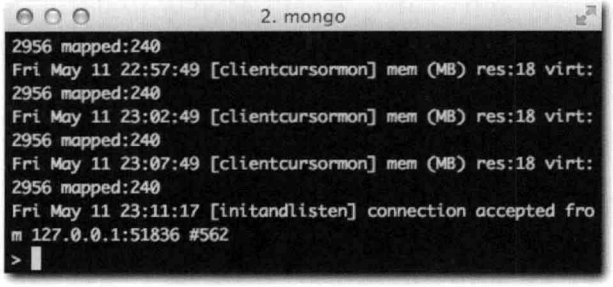
```
  // 监听
  app.listen(3000, function () {
    console.log('\033[96m  + \033[39m app listening on *:3000');
  });
});
```

如果运行应用程序（确保数据库运行正常），大致会输出如下形式的消息：

```
$ node server.js
  +  connected to mongodb
  +  app listening on *:3000
```

要是这个时候通过mongo客户端，运行show log global命令，应当会看到如图12-3所示的连接成功的消息！

```
$ mongo
> show log global;
[. . .]
{date} [initandlisten] connection accepted from 127.0.0.1:53649 #16
```



214

图12-3：如最后一行日志所示，Mongo从你的本地Web服务器获取了连接请求

创建文档

插入文档的API很简单。简单地调用Collection#insert方法，并提供要插入的文档以及一个回调函数就可以了。和绝大多数Node中的回调函数一样，第一个参数是一个错误对象，本例中，第二个参数是一个插入的文档数组：

```
collection.insert({ my: 'document' }, function (err, docs) {
  // . . .
});
```

另外还有一个options对象，是可选的第二个参数，后面会做相应介绍。

要是回头再来看注册表单，就会发现输入框的名字遵循这样的格式：user[field]。例如：

```
input(name="user[name]", type="text")
```

下面会看到，当bodyParser遇到这样的格式，会产生req.body.user.name这样的字段。

这个功能可以很方便地让我们直接将文档插入到MongoDB中。对于本例，我们忽略数据校验（非常重要）。

这样一来，处理注册的路由就变得非常简单：

```
/**
 * 处理注册的路由
 */

app.post('/signup', function (req, res, next) {
  app.users.insert(req.body.user, function (err, doc) {
    if (err) return next(err);
    res.redirect('/login/' + doc[0].email);
  });
});
```

215 若遇到错误，需要调用next，这样就可以显示一个“错误500”页面。尽管错误不会频繁发生，但重要的是必须要对其进行处理。

在处理了错误之后，最常犯的错误就是忘记return。这种错误会在应用程序中产生无法预知的行为。比如，这个时候有错误发生了，doc变量就会是undefined，这样一来代码就会抛出无法捕获的异常。

插入文档成功后，将应用程序重定向到登录路由，并提供email字段。

在登录路由中，我们获取email参数，并将其暴露给视图：

```
/**
 * 登录路由
 */

app.get('/login/:signupEmail', function (req, res) {
  res.render('login', { signupEmail: req.params.signupEmail });
});
```

在视图中，我们显示一个消息：

```
if (signupEmail)
  p Congratulations on signing up! Please login below.
```

然后，将email变量输出到email输入框中：

```
input(name="user[email]", type="text", value=signupEmail)
```

现在启动应用程序验证一下注册功能！这时，若启动mongo客户端，在新创建的集合上运行find命令，应该就能看到刚刚创建的文档内容了：

```
 Brian: in the mongo client things appear like this$ mongo my-website
> db.users.find()
{ "first" : "A", "last" : "B", "email" : "a@b.com", "password" : "d", "_id" : Object
 Id("4ef2cbd77bb50163a7000001") }
```

注意，上述文档看起来和你插入的完全一样，相当直观。除此之外， Mongo自动添加了_id字段，用来唯一确定文档内容。非常方便！

查找文档

至此我们已经创建好了文档，接下来就可以在/login路由中对文档进行查询了。我们要获取的是匹配对应email和password的文档。

在MongoDB中，没有固定的schema能确定一个集合，因此，每次对集合进行特定方式查询时，确保正确地对其进行了索引会是个不错的主意。特别是当某个键是在嵌套结构中时，要是没有对其进行索引，会导致一次扫描整表的查询操作，会导致应用程序性能的下降。 216

MongoDB有一个ensureIndex命令，顾名思义，不管索引是否存在，都可以调用这个命令来确保在查询前建立了索引。我们可以在应用初始化的时候调用它。

在设置了app.users后，我们应当调用两次ensureIndex：

```
client.ensureIndex('users', 'email', function (err) {
  if (err) throw err;
  client.ensureIndex('users', 'password', function (err) {
    if (err) throw err;

    console.log('\033[96m  + \033[39m ensured indexes');

    // 监听
    app.listen(3000, function () {
      console.log('\033[96m  + \033[39m app listening on *:3000');
    });
  });
});
```

重启应用后，会发现有额外日志：

```
$ node server.js
+  connected to mongodb
+  ensured indexes
+  app listening on :3000
```

现在就可以查询了！

```
/**
 * 登录处理路由
 */

app.post('/login', function (req, res) {
  app.users.findOne({ email: req.body.user.email, password: req.body.user.password
 }, function (err, doc) {
    if (err) return next(err);
    if (!doc) return res.send('<p>User not found. Go back and try again</p>');
```

```
      req.session.loggedIn = doc._id.toString();
      res.redirect('/');
    });
});
```

和insert命令一样，findOne命令可以对MongoDB进行查询文档的操作。

217 我们将_id存储为session的一部分，这样可以在用户访问其他路由时，能够获取当前登录用户的信息。注意这里，我们显式地将MongoDB ObjectId存储为字符串类型，其表现形式是十六进制的。

最后，还要实现/logout路由，处理非常简单，清除session就好了。记住，req.session对象可以随意修改，在做出响应后（比如本例中的重定向），Express会自动将其保存下来。

```
/**
 * 登出路由
 */

app.get('/logout', function (req, res) {
  req.session.loggedIn = null;
  res.redirect('/');
});
```

在这个例子中，我们保留了session，并将其ID设置成了null。或者，若想完全清楚session，可以直接调用req.session.regenerate()。

身份验证中间件

我们开发的绝大多数应用貌似在不止一处都需要访问验证后登录用户的信息。

若回过头来再看一下index.jade，我们需要访问me对象来获取匹配登录用户的文档信息，与此同时，还要通过authenticated变量来判断用户是否已经通过了身份验证。

```
if (authenticated)
  p Welcome back, #{me.name}
  a(href="/logout") Logout
```

我们可以定义一个中间件，将这两个变量（authenticated、me）暴露给视图使用。这里需要用到Express的res.local API：

```
/**
 * 身份验证中间件
 */

app.use(function (req, res, next) {
  if (req.session.loggedIn) {
    res.local('authenticated', true);
    app.users.findOne({ _id: { $oid: req.session.loggedIn } }, function (err, doc) {
      if (err) return next(err);
      res.local('me', doc);
```

```
        next();
      });
    } else {
      res.local('authenticated', false);
      next();
    }
});
```

218

注意了，在调用findOne时，我们传递了$oid修饰符。它允许直接传递一个字符串而不用传递一个ObjectId对象。回忆一下，之前我们调用了toString来确保存储的loggedIn是字符串。

记得移除此前在index路由中用来测试的{authenticated:false}，现在这部分代码应该是这样的（见图12-4）：

```
app.get('/', function (req, res) {
  res.render('index');
});
```

图12-4：用户成功登录后的界面

此前的应用没有考虑实际场景下必要的一些基础特性。接下来的三部分内容会对其进行介绍。

校验

若用户提交的表单太大，该怎么办呢？按照此前例子的处理方式，就会直接向数据库中插入这么大的文档数据了。

除此之外，我们也许还应该在存储数据前确保email字段确实是email字段，密码应该是不少于6个字符的字符串，而不是Date或者是Number。

我们也不想在每次对数据库进行插入、更新、查询操作的时候都要重复上述校验规则。

Mongoose通过允许在应用层定义schema来解决这个问题，它在保持文档灵活性和易改动的前提下，引入了特定的属性对其做一定的约束，称为模型。

219

原子性

假设我们基于Express和MongoDB书写一个博客引擎。可想而知，其中一部分功能会是允许用户修改博文的标题和内容，可能还有一部分功能是允许编辑和删除标签。

面向文档设计的MongoDB非常适合这样的场景。在posts集合中，文档可能会是这样的：

```
{
    "title": "I just bought Smashing Node.JS"
  , "author": "John Ward"
  , "content": "I went to the bookstore and picked up. . ."
  , "tags": ["node.js", "learning", "book"]
}
```

假设，这个时候，有两个人，用户A想要编辑文档的标题，与此同时，用户B想要添加一个标签。

如果两个用户都传递了一份完整的文档拷贝来进行更新操作，那么只有一个会胜出。另外一个将无法成功完成对文档的修改。

要确保某个操作的原子性，MongoDB提供了$set和$push这样不同的操作符：

```
db.blogposts.update({ _id: <id> }, { $set: { title: 'My title' } })
db.blogposts.update({ _id: <id> }, { tags: { $push: "new tag" })
```

Mongoose则是通过检查要对文档做的修改，并只修改受影响的字段来解决这个问题。就算操作的是数组（包括文档数组），原子性依然能够得到保证。

安全模式

此前介绍过，在使用驱动器时，我们可以在操作文档时提供一个可选的options参数：

```
app.users.insert({ }, { <options> })
```

其中一个选项叫safe，它会在对数据库进行修改时启动安全模式。

220 默认情况下，在操作完成后，如果有错误发生，MongoDB不会及时通知你。驱动器需要在操作完成后进行一个特殊的函数调用db.getLastError，来验证数据修改是否成功。

这背后的原因在于对于许多应用来说，相比于要知道某个操作是否失败而言，速度更为重要。比如，丢了某些日志并不是什么世界末日，但是导致性能低下就无法接受了。

Mongoose默认会对所有操作启用安全模式，当然了，你可以关闭这个选项。

Mongoose介绍

按照惯例，在开始使用Mongoose前，先要在package.json文件中定义对Mongoose的依赖，

然后通过require将其引入：

```
var mongoose = require('mongoose')
```

相比原生的驱动器，Mongoose做的第一个简化的事情就是它假定绝大部分的应用程序都是用一个数据库，这大大简化了使用方式。要连接数据库，只需要调用mongoose.connect并提供mongodb://URI即可：

```
mongoose.connect('mongodb://localhost/my_database');
```

另外，使用Mongoose，就无须关心连接是否真的已经建立了，因为，它会先把数据库操作指令缓存起来，在连接上数据库后就会把这些操作发送给MongoDB。这就意味着，我们无须监听connection的回调函数。连接后就可以直接像下面要介绍的这样开始查询数据了。

定义模型

模型是Schema类的简单实例。在指定字段时，简单地使用对应类型的JavaScript原生的构造器即可：

```
var Schema = mongoose.Schema
  , ObjectId = Schema.ObjectId;

var PostSchema = new Schema({
    author    : ObjectId
  , title     : String
  , body      : String
  , date      : Date
});
```

这些类型是：

- Date
- String
- Number
- Array
- Object

221

除此之外，MongoDB还提供了一种ObjectId类型，这种类型可以通过Schema.ObjectId来获得。

在这个博客例子中，我们可以将创建博文的用户存储为ObjectId类型。

Mongoose还为给定的字段提供了一些不同的选项。在提供选项时，需要在一个对象中引用对应字段类型的构造器。比如，要给字段提供了一个默认值，可以按照如下方式提供

default和type选项：

```
var PostSchema = new Schema({
    author    : ObjectId
  , title     : { type: String, default: 'Untitled' }
  , body      : String
  , date      : Date
});
```

创建好Schema后，通过mongoose来注册一个模型：

```
var Post = mongoose.model('BlogPost', PostSchema);
```

对于本例，Mongoose将集合名字设置为blogposts，除非我们通过第三个参数来指定集合名。Mongoose默认会对模型名字使用小写复数形式。

随后要想获取模型，可以通过调用mongoose.model方法并提供模型名：

```
var Post = mongoose.model('BlogPost');
```

接着就可以操作模型了。要创建一个博客文章，只要使用new操作符就可以了：

```
new Post({ title: 'My title' }).save(function (err) {
  console.log('that was easy!');
});
```

有一点很重要：Schema只是一种简单的抽象，用以描述模型的样子以及它是如何工作的。数据的交互发生在模型上，而不是Schema上。

因此，若要查询，与使用new关键字来初始化博文相对，需要执行静态的Post.find方法（或者其他一些会在下面介绍的方法）。

222 定义嵌套的键

考虑到数据的组织，有的时候以子结构的形式来组织键也非常有帮助：

```
var BlogPost = new Schema({
    author    : ObjectId
  , title     : String
  , body      : String
  , meta      : {
      votes : Number
    , favs  : Number
  }
});
```

在MongoDB中，可以使用点来操作这些属性。比如，要查找拥有指定投票数的博文，可以通过如下方式来实现：

```
db.blogposts.find({ 'meta.votes': 5 })
```

定义嵌套文档

在MongoDB中，文档可以很大，也可以层次很深。也就是说，如果博文有留言的话，可以直接将留言定义在博文中，而不需要将其定义为单独的集合。

```
var Comments = new Schema({
    title     : String
  , body      : String
  , date      : Date
});

var BlogPost = new Schema({
    author    : ObjectId
  , title     : String
  , body      : String
  , buf       : Buffer
  , date      : Date
  , comments  : [Comments]
  , meta      : {
      votes : Number
    , favs  : Number
  }
});
```

Mongoose还允许为该字段定义想要的类型。

构建索引

正如此前提到过的，索引是在MongoDB数据库中确保快速查询的重要因素。

要对指定的键做索引，需要传递一个index选项，并将值设置为true。 223

比如，要对title键做索引，并将uid键设置为唯一，可以这样：

```
var BlogPost = new Schema({
    author    : ObjectId
  , title     : { type: String, index: true }
  , uid       : { type: Number, unique: true }
});
```

要设置更复杂的索引（如组合索引），可以使用静态的index方法：

```
BlogPost.index({ key: -1, otherKey: 1 });
```

中间件

在相当一部分应用中，有的时候会在不同的地方以不同的方式对同样的数据进行修改。

通过模型接口将这部分对数据库的交互集中处理是避免代码重复很有效的方式。

Mongoose通过引入中间件来实现。Mongoose中间件的工作方式和 Express中间件非常相似。你可以定义一些方法，在某些特定动作前执行：save和remove。

比如，要在博文删除时，发送电子邮件给作者，可以通过下面的方式来实现：

```
Blogpost.pre('remove', function (next) {
  emailAuthor(this.email, 'Blog post removed!');
  next();
});
```

还可以对单个动作定义多次中间件来执行各类操作，特别是异步操作。

探测模型状态

很多时候，我们需要根据要对当前模型做的不同更改进行不同的操作：

```
Blogpost.pre('save', function (next) {
  if (this.isNew) {
    // doSomething
  } else {
    // doSomethingElse
  }
});
```

224

还可以通过this.dirtyPaths来探测什么键被修改了。

查询

在Model实例上暴露的所有常见的操作有：

- find
- findOne
- remove
- update
- count

Mongoose还添加了findById，该方法接受一个ObjectId去匹配文档的_id属性。

扩展查询

如果对某个查询不提供回调函数，那么直到调用run它才会执行：

```
Post.find({ author: '4ef2cbffb1d9807fa7000001' })
  .where('title', 'My title')
  .sort('content', -1)
  .limit(5)
  .run(function(err, post) {
    // . . .
  })
```

排序

要进行排序，只需提供排序的键以及排序的顺序即可：

```
query.sort('key', 1)
query.sort('some.key', -1)
```

选择

若文档很大，而想要的只是部分指定的键，那么就可以调用Query#select方法。

比如，要显示带链接的博文列表，无须获取所有的字段（有些字段可能数据量很大）：

```
Post.find()
  .select('field', 'field2')
```

限制

225

要限制查询结果的数量，可以调用Query#limit方法：

```
query.limit(5)
```

跳过

要跳过指定数量的文档数据，可以通过如下方式：

```
query.skip(10);
```

这个功能结合Model#count对做分页非常有用：

```
Post.count(function (err, totalPosts) {
  var numPages = Math.ceil(totalPosts / 10);
});
```

自动产生键

在BlogPost模型例子中，我们将博文作者的ID存储为author属性。

很多时候，在查询一个博文时，我们还需要获取对应的作者。这个时候，就可以为ObjectId类型提供一个ref属性：

```
var BlogPost = new Schema({
    author    : { type: ObjectId, ref: 'Author' }
  , title     : String
  , body      : String
  , meta      : {
      votes : Number
    , favs  : Number
  }
});
```

之后，查询文档时就能自动产生作者数据！通过简单地对指定键调用populate方法即可：

```
BlogPost.find({ title: 'My title' })
  .populate('author')
  .run(function (err, doc) {
    console.log(doc.author.email);
  })
```

转换

因为Mongoose提前知道需要什么样的数据类型，所以它总是会尝试去做类型转换。

226 例如，有一个年龄字段，在Schema中描述的是Number类型。如果有人在网站中提交了一个普通表单，那么在没有JSON或者特定逻辑处理的情况下，我们得到的是字符串而不是数字。Mongoose利用了动态语言的特性，所以对它来说在存储前将'21'（字符串）转化为21（数字）就没有什么问题了。

同样的情况还会发生在ObjectId。在此前的例子中，我们不得不使用$oid修饰符来让使用字符串形式ObjectID的查询操作成功执行，然而，这样的方式过于冗长。我们可以直接传递"4ef2cbd77bb50163a7000001"给Mongoose，它会自动将其转化为ObjectId("4ef2cbd77bb50163a7000001")。

除此之外，当类型不匹配并且转换失败时，Mongoose会抛出一个校验错误，并且放弃对文档的存储。这种行为在一致性和文档的整洁性上确保了易用性。

一个使用Mongoose的例子

和此前我们做的一样：先使用node http模块，然后再使用Connect和 Express来对其改进。接下来，我们要用Mongoose对此前的例子进行重构，以此来展现Mongoose在表现形式上对数据表提供的好处。

我们从创建一个新的package.json并添加对mongoose的依赖开始。

构建应用

新的package.json文件如下所示：

```
{
    "name": "mongoose-example"
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.2"
      , "mongoose": "2.5.10"
    }
}
```

和往常一样，运行npm install来安装所需的依赖。接下来，我们重构一下服务器端主要的代码。

重构

我们从此前例子中将server.js文件和views目录复制过来。

首先要重构的就是将对mongodb的依赖替换为mongoose，因为mongodb并未出现在package.json文件中。事实上，mongoose内部就是使用了mongodb。

server.js顶部代码如下所示：

227

```
/**
 * 模块依赖
 */

var express = require('express')
  , mongoose = require('mongoose')
```

现在，我们要将注意力放在该文件底部代码上，也就是连接数据库的代码：

```
/**
 * 连接数据库
 */

var server = new mongodb.Server('127.0.0.1', 27017)
// . . .
```

正如此前提到的，mongoose大大简化了连接数据库、操作集合、建立索引等的操作。server.js文件最后一部分代码就可以简化为如下形式：

```
/**
 * 连接数据库
 */

mongoose.connect('mongodb://127.0.0.1/my-website');
app.listen(3000, function () {
  console.log('\033[96m  + \033[39m app listening on *:3000');
});
```

接下来，我们要定义模型来取代app.users这种引用方式，并建立此前建立的索引。

建立模型

模型可以在文件任意位置通过Mongoose定义。无所谓Mongoose是否已经与数据库连接。

在文件最后，我们加上对模型的定义：

```
/**
 * 定义模型
```

```
 */

var Schema = mongoose.Schema

var User = mongoose.model('User', new Schema({
    first: String
  , last: String
  , email: { type: String, unique: true }
  , password: { type: String, index: true }
}));
```

228

接下来我们再看用它来替换app.users的地方。第一处就是身份认证中间件。我们用Mongoose提供的便利的findById方法来替换使用$oid的地方：

```
app.use(function (req, res, next) {
  if (req.session.loggedIn) {
    res.local('authenticated', true);
    User.findById(req.session.loggedIn, function (err, doc) {
      if (err) return next(err);
      res.local('me', doc);
      next();
    });
  } else {
    res.local('authenticated', false);
    next();
  }
});
```

在登录的POST路由中，需要再次用到模型方法。这次是findOne：

```
app.post('/login', function (req, res) {
  User.findOne({ email: req.body.user.email, password: req.body.user.password },
function (err, doc) {
    if (err) return next(err);
    if (!doc) return res.send('<p>User not found. Go back and try again');
    req.session.loggedIn = doc._id.toString();
    res.redirect('/');
  });
});
```

正如此前提到过的，模型可以静态方式使用（正如那两个例子所示的），也可以当构造器使用。

注册的POST路由应该重构成如下形式：

```
app.post('/signup', function (req, res, next) {
  var user = new User(req.body.user).save(function (err) {
    if (err) return next(err);
    res.redirect('/login/' + user.email);
  });
});
```

注意，我们不再需要一个包含了文档信息的回调函数。我们只需在回调函数中使用创建的模型实例即可（使用user.email而不是doc[0].email）。 229

重构完成！再次运行server.js，一切都应该运行正常，不过，代码变得更为整洁，更容易理解了。

小结

本章介绍了Node.js世界中最流行的数据库之一：MongoDB。

本章介绍了关于文档数据库是如何工作的基本知识，还介绍了Node.js中 MongoDB的驱动器。

你会注意到，它处理数据的方式非常自然，另外，对数据在浏览器和Web服务器之间传递的映射也非常好。

通过重构第一个例子，现在，你应当能够体会到本章介绍的引入了模型概念以及非常方便的API框架——Mongoose的好处了。

CHAPTER

13 MySQL

尽管现如今NoSQL渐行其道，越来越流行，但是SQL数据库依旧是绝大多数应用的选择。 231

Node.js丰富的生态系统中，有许多模块都是为SQL数据库设计开发的，特别是本章要介绍的：MySQL。

与第12章中介绍MongoDB一样，本章会先介绍一个原生的MySQL驱动器（一个名为node-mysql的项目）。

通过node-mysql，我们可以书写自己的SQL查询语句来操作数据库。

除了驱动器之外，本章还会介绍如何使用MySQL的对象关系映射器（ORM）——node-sequelize。正如上一章介绍的，ORM提供了一个MySQL数据库中数据到JavaScript模型对象的映射，使得操作数据关系、数据处理等变得更加容易。

node-mysql

232

要学习如何使用node-mysql，我们从创建一个简单的购物车应用的模型开始。

初始化项目

按照惯例，我们从添加对`express`、`jade`还有`node-mysql`的依赖开始：

package.json

```
{
    "name": "shopping-cart-example"
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.2"
      , "jade": "0.19.0"
      , "mysql": "0.9.5"
    }
}
```

Express应用

接下来，我们创建一个简单的Express应用，并添加如下路由：

- /：展示所有的商品以及创建商品的表单。
- /item/<id>：展示指定的商品以及用户评价。
- /item//review (POST)：创建一个评价。
- /item/create (POST)：创建一个商品。

对于首页以及商品的路由，我们将渲染简单的模板。注意了，我们配置了Express的view options，将模板布局取消了，这符合Express 3的行为。模板布局将直接通过jade来实现。

server.js

```
/**
 * 模块依赖
 */

var express = require('express')

/**
 * 创建应用
 */

app = express.createServer();

/**
 * 配置应用
 */

app.set('view engine', 'jade');
app.set('views', __dirname + '/views');
app.set('view options', { layout: false });
```

233

```
/**
 * 首页路由
 */

app.get('/', function (req, res, next) {
  res.render('index');
});

/**
 * 创建商品路由
 */

app.post('/create', function (req, res, next) {
});

/**
 * 查看商品路由
 */

app.get('/item/:id', function (req, res, next) {
  res.render('item');
});

/**
 * 创建商品评价路由
 */

app.post('/item/:id/review', function (req, res, next) {
});

/**
 * 监听
 */

app.listen(3000, function () {
  console.log(' - listening on http://*:3000');
});
```

连接MySQL

234

下一步是添加对node-mysql的依赖：

server.js

```
var express = require('express')
  , mysql = require('mysql')
```

要初始化连接，和创建net客户端所使用的Node API方式一样，我们通过调用createClient

方法来实现。

和Mongoose对mongodb的处理方式一样，node-mysql在真正连接到MySQL前就可以接收指令，并将它们缓存起来（也就是将它们存储在内存中），当连接建立后，就一次性将它们全部发送给MySQL。

所以，我们无须监听connection事件或者提供回调函数，只需简单地初始化客户端，提供相应的设置即可。将下列代码添加到应用配置部分的下方：

server.js

```
/**
 * 连接MySQL
 */

var db = mysql.createClient({
    host: 'localhost'
  , database: 'cart-example'
});
```

若要设置数据库的用户名和密码，就在调用createClient方法时在参数中传递user和password选项。要了解更多关于node-mysql的用法，可以查看其官方文档http://github.com/felixge/node-mysql。

初始化脚本

在应用程序中使用SQL数据库前，我们总是先要创建必要的数据库和表。

为了让代码重用，我们创建一个名为setup.js的简单node脚本来运行必要的CREATE TABLE命令。

由于连接数据库所需参数和此前书写在应用中的是一样的，所以我们先将这些参数的配置抽象到一个config.json文件中：

235

config.json

```
{
    "host": "localhost"
  , "database": "cart-example"
}
```

注意了，有效的JavaScript代码未必一定是有效的JSON。本例中，我们将所有的键都用引号括起来，并且要确保所有的值都使用的是双引号而不是单引号。

从Node 0.6开始，就可以直接使用require来引入JSON文件，而无须再用JSON.parse和

fs#readFileSync了。接下来，编辑修改依赖的模块：

server.js

```
/**
 * 模块依赖
 */

var express = require('express')
  , mysql = require('mysql')
  , config = require('./config')

// . . .
```

在连接数据库的代码部分，使用config来替换原先的参数对象：

server.js

```
var db = mysql.createClient(config);
```

接下来准备创建启动脚本。该脚本只依赖mysql和config，因为它是从命令行直接运行的。

setup.js

```
/**
 * 模块依赖
 */

var mysql = require('mysql')
  , config = require('./config')
```

下面初始化客户端，由于数据库还未创建好，所以需要将config中的database字段删除：

236

setup.js

```
/**
 * 初始化客户端
 */
delete config.database;
var db = mysql.createClient(config);
```

node-mysql提供的执行查询语句的API非常简单：client.query(<sql>, <callback>)。关闭连接的API是client.end。

由于我们使用单个TCP连接，所以服务器接收到的指令顺序和我们书写的顺序是一致的。也就意味着，不需要为了确保执行顺序而嵌套回调函数：

```
// 没有必要这么做!
db.query('CREATE TABLE. . .', function (err) {
  db.query('CREATE TABLE. . .', function (err) {
    db.query('CREATE TABLE. . .', function (err) { });
  });
});
```

为了确保能够将错误报告给用户，还需要监听db error事件：

```
db.on('error', function () {
  // handle error
});
```

对于中断程序而言，当错误发生时，最好的处理方式是将错误和调用堆都展示给用户并终止程序。你或许还记得第4章中介绍的，当错误对象通过EventEmitter分发出来，但又没有对应的监听器时（也就是说事件未处理），Node会将错误抛出，确保开发者能够意识到错误的发生，而不是将错误“吞”掉。因此，事实上，对于本例，我们无须专门添加一个错误处理器，因为Node会将未处理的错误直接抛出。

首先，我们需要创建数据库并告诉MySQL要使用该数据库：

setup.js

```
/**
 * 创建数据库
 */

db.query('CREATE DATABASE IF NOT EXISTS `cart-example`');
 db.query('USE `cart-example`');
```

237

setup.js

```
/**
 * 创建表
 */

db.query('DROP TABLE IF EXISTS item');
 db.query('CREATE TABLE item (' +
  'id INT(11) AUTO_INCREMENT,' +
  'title VARCHAR(255),' +
  'description TEXT,' +
  'created DATETIME,' +
  'PRIMARY KEY (id))');
db.query('DROP TABLE IF EXISTS review');
db.query('CREATE TABLE review (' +
  'id INT(11) AUTO_INCREMENT,' +
  'item_id INT(11),' +
  'text TEXT,' +
```

```
  'stars INT(1),' +
  'created DATETIME,' +
  'PRIMARY KEY (id))');
```

setup.js

```
/**
 * 关闭客户端
 */

db.end();
```

正如我们在第3章中介绍的，当事件轮询中没有任务要处理时，Node就会退出该进程。在连接MySQL服务器的过程中，我们打开了一个文件描述符，于是Node事件轮询机制就会开始监听。当调用结束客户端时，文件描述符也会被关闭，因此，也要结束程序。

代码如下所示：

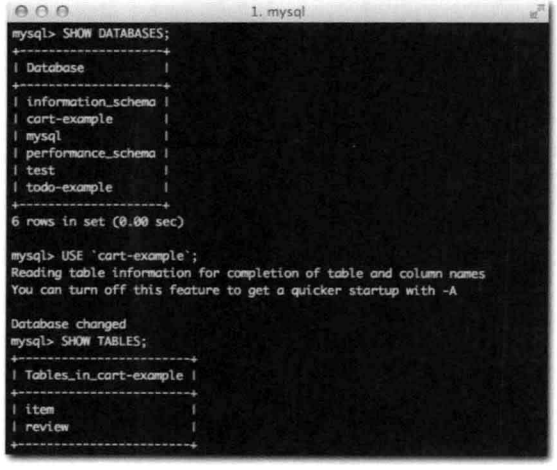
setup.js

```
db.end(function () {
  process.exit();
});
```

接下来测试刚刚书写的脚本：

```
$ node setup.js
```

然后，我们可以用mysql客户端确认下，数据库和表都已经创建好了（见图13-1）：



238

图13-1：通过MySQL命令行客户端来查看利用setup.js创建的数据库和表是否创建成功了

```
$ mysql
> show databases;
. . .
> use cart-examples;
. . .
> SHOW TABLES;
. . .
```

创建数据

接下来，我们在views目录下创建一个简单的布局。如下所示，该文件包含一个特殊的jade `block body`声明，用于将其他视图嵌入：

views/layout.jade

```
doctype 5
html
  head
    title My shopping cart
  body
    h1 My shopping cart
    #cart

block body
```

index文件展示了一个包含所有商品的列表，以及用于创建新商品的表单：

239

views/index.jade

```
extends layout
block body
h2 All items
if (items.length)
  ul
    each item in items
      li
        h3: a(href="/item/#{item.id}")= item.title
        = item.description
else
  p No items to show

h2 Create new item

form(action="/create", method="POST")
  p
    label Title
    input(type="text", name="title")
  p
    label Description
```

```
    textarea(name="description")
  p
    button Submit
```

因为在上述代码中，通过检查length属性来展示items数组项，所以，我们暂且先确保在/路由中传递一个空数组，如下所示。当然了，真正的数据稍后肯定是从数据库中获得的。

server.js

```
app.get('/', function (req, res, next) {
  res.render('index', { items: [] });
});. . .
```

对于商品查看页面，我们需要商品本身、关于它的评价以及创建新评价的表单。

views/item.jade

```
extends layout

block body
  a(href="/") Go back

  h2= item.title
  p= item.description
  h3 User reviews

  if (reviews.length)
    each review in reviews
      .review
        b #{review.stars} stars
        p= review.text
      hr
  else
    p No reviews to show. Write one!

  form(action="/item/#{item.id}/review", method="POST")
    fieldset
      legend Create review
      p
        label Stars
        select(name="stars")
          option 1
          option 2
          option 3
          option 4
          option 5
      p
        label Review
```

240

```
      textarea(name="text")
    p
      button(type="submit") Send
```

注意在上述代码中，表单的action属性部分使用了jade的插补特性。通过使用#{}以一种安全的方式（HTML的实体会被转义）来引入变量。如果想要引入不需要转义的字符串变量，可以使用!{}。

在开始获取数据前，我们需要先创建数据来做简单的测试。

在项目配置代码下方，添加bodyParser中间件来处理POST请求：

server.js

```
/**
 * 中间件
 */

app.use(express.bodyParser());
```

241 接着，完成/create路由：

server.js

```
/**
 * 创建商品路由
 */

app.post('/create', function (req, res, next) {
  db.query('INSERT INTO item SET title = ?, description = ?',
  [req.body.title, req.body.description], function (err, info) {
    if (err) return next(err);
    console.log(' - item created with id %s', info.insertId);
    res.redirect('/');
  });
});
```

上述代码有两部分非常意思。第一部分是db.query允许用后面提供的数据替换?记号。通过这种替换记号的方式，可以有效地避免字符串的拼接，从而避免SQL注入攻击。如果在查询语句中包含了?记号，那么需要提供一个包含要替换数据的数组作为第二个参数。

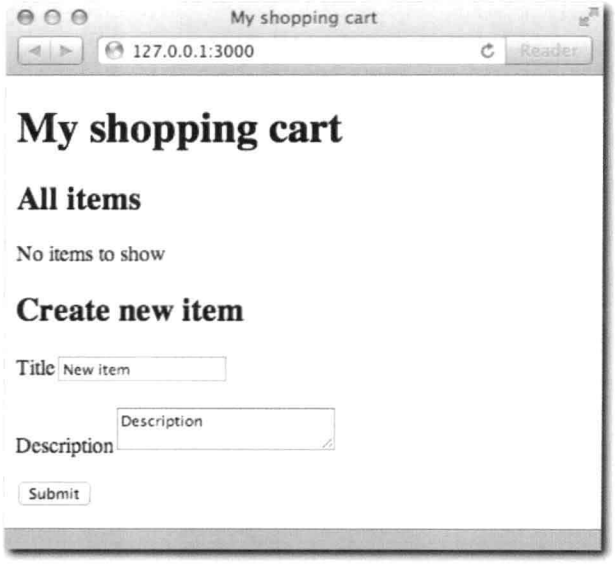
另外一部分有意思的是info对象。本例中，我们通过insertId来获取创建商品的ID。只要不发生错误，这个属性一直都在。如果有错误发生，我们就终止处理，并调用next方法。

创建评价的路由也类似：

```
/**
 * 创建商品评价路由
 */

app.post('/item/:id/review', function (req, res, next) {
  db.query('INSERT INTO review SET item_id = ?, stars = ?, text = ?',
  [req.params.id, req.body.stars, req.body.text], function (err, info) {
    if (err) return next(err);
    console.log(' - review created with id %s', info.insertId);
    res.redirect('/item/' + req.params.id);
  });
});
```

通过运行上述应用并创建一个商品来做测试（见图13-2）。



242

图13-2：在首页路由中，填写创建商品的表单

之后，就能在控制台看到如图13-3所示的内容了。

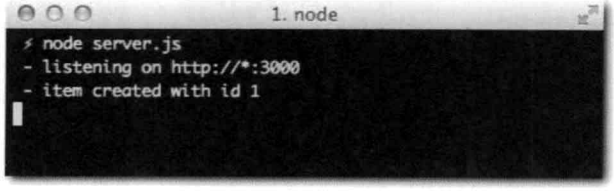


图13-3：商品创建成功后，在控制台显示了其id

获取数据

通过node-mysql从MySQL中获取数据是非常简单直观的。当执行的命令是SELECT时，回调函数中会接收一个包含查询结果对象的数组。数组中的对象包含了指定返回的字段。根据本章所属范畴，我们将只讨论SELECT指令。

回调函数中接收到的是数组，这其实和我们在index.jade模板中想要的数据结构是一致的，模板中，我们需要遍历数组，并显示其id、title以及description属性。

因此，我们要做的就是检查是否有错误发生，如果没有，就把查询结果传递给视图：

```
/**
 * 首页路由
 */
app.get('/', function (req, res, next) {
  db.query('SELECT id, title, description FROM item', function (err, results) {
res.render('index', { items: results });
  });
});
```

/路由完成后，就可以重启应用，查看所有商品的列表了。

对于列表中包含的查看商品路由，我们需要获取商品数据，确保它存在，并获取相关的评价。如果商品不存在，就返回404状态码。

为了让代码更可读，我们将逻辑按照执行顺序拆分到几个函数中：

server.js

```
/**
 * 查看商品路由
 */

app.get('/item/:id', function (req, res, next) {
  function getItem (fn) {
    db.query('SELECT id, title, description FROM item WHERE id = ? LIMIT 1',
    [req.params.id], function (err, results) {
      if (err) return next(err);
      if (!results[0]) return res.send(404);
      fn(results[0]);
    });
  }

  function getReviews (item_id, fn) {
    db.query('SELECT text, stars FROM review WHERE item_id = ?',
    [item_id], function (err, results) {
      if (err) return next(err);
      fn(results);
    });
```

```
  }

  getItem(function (item) {
    getReviews(item.id, function (reviews) {
      res.render('item', { item: item, reviews: reviews });
    });
  });
});
```

图13-4展示了完成后的应用。现在可以浏览商品、提交评价以及在界面上看到它们的信息了。

244

图13-4：完整应用展示

sequelize

在此前的例子中，直接操作SQL数据库的方式多少有些问题。

第一个问题就是建表的过程是手动的（耗时），而且表的定义并非项目本身的一部分。应用程序根本无法得知商品的title属性只能允许最多255个字符。如果能够知道，那么应用程序就可以对用户的输入做校验，并显示错误信息。

解决这个问题的方式就是使用sequelize：通过它可以定义schema和模型，同时还可以使用其同步的特性，根据那些定义来创建要使用的数据库表。这样一来，此前例子中的`setup.js`就不再需要了。

因为schema也是应用程序的一部分，我们可以使用它们来做类型转化。要存储包含指定数据的商品，我们可以直接传递一个JavaScript的Date对象，而无须手动构造MySQL需要的日期格式。

最后一点，同时也是很重要的一点就是联合。在此前的例子中，我们是手动去获取商品的评价信息的，而通过sequelize，可以自动获取这部分数据。

245 我们通过创建一个简单的任务列表应用，来运用sequelize为数据库表带来的不同的概念和特性。任务可以根据项目来分组。我们可以添加、创建和删除项目，同时也可以在指定项目中，添加、创建和删除任务。

初始化sequelize

由于sequelize内部使用了node-mysql驱动器，所以依赖列表就可以是如下形式：

package.json

```
{
    "name": "todo-list-example"
  , "version": "0.0.1"
  , "dependencies": {
        "express": "2.5.2"
      , "jade": "0.19.0"
      , "sequelize": "1.3.7"
    }
}
```

初始化Express应用

这部分应用将与以往传统的应用有所不同，这次在创建和删除商品时将采用Ajax的方式。我们可以通过使用DELETE方法让应用程序更加RESTful。若你对REST还不熟悉，那么简单来说它就是一系列准则，引入了一种HTTP协议更宽泛的使用方式，使得像HTTP的PATCH、DELETE以及平时不常使用的状态码为我们所用。

定义的路由如下所示：

- / (GET)：获取所有项目。
- /projects (POST)：创建项目。
- /project/:id (DELETE)：删除项目。
- /project/:id/tasks (GET)：获取任务。
- /project/:id/tasks (POST)：添加任务。
- /task/:id (DELETE)：删除任务。

```
/**
 * 模块依赖
 */

var express = require('express')

/**
 * 创建应用
 */

app = express.createServer();

/**
 * 配置应用
 */

app.set('view engine', 'jade');
app.set('views', __dirname + '/views');
app.set('view options', { layout: false });

/**
 * 首页路由
 */

app.get('/', function (req, res, next) {
  res.render('index');
});

/**
 * 删除项目路由
 */

app.del('/project/:id', function (req, res, next) {
});

/**
 * 创建项目路由
 */

app.post('/projects', function (req, res, next) {
});

/**
 * 展示指定项目中的任务
 */

app.get('/project/:id/tasks, function (req, res, next) {
});

/**
```

246

```
 * 为指定项目添加任务
 */

app.post('/project/:id/tasks, function (req, res, next) {
});

/**
 * 删除任务路由
 */

app.del('/task/:id', function (req, res, next) {
});
/**
 * 监听
 */

app.listen(3000, function () {
  console.log(' - listening on http://*:3000');
});
```

247

我们再定义一个简单的布局，这次使用jQuery来更容易地发送Ajax请求：

views/layout.jade

```
doctype 5
html
  head
    title TODO list app
    script(src="http://code.jquery.com/jquery-1.7.2.js")
    script(src="/js/main.js")
  body
    h1 TODO list app
    #todo
      block body
```

注意了，在上述代码中，载入了`main.js`文件，该文件将包含所有的客户端逻辑（如：发送Ajax请求来提交表单）。

项目和任务列表展示方式一样，因为它们都有添加和删除操作：

views/index.jade

```
extends layout

block body
  h2 Projects

  #list
    ul#projects-list
```

```
    each project in projects
      li
        a(href="/project/#{project.id}/items")= project.title
        a.delete(href="/project/#{project.id}") x

  form#add(action="/projects", method="POST")
    input(type="text", name="title")
```

248

button(type="submit") Addviews/tasks.jade

```
h2 Tasks for project #{project.title}

#list
  ul#tasks-list
    each task in tasks
      li
        span= task.title
        a.delete(href="/task/#{task.id} ") x

  form#add(action="/project/#{project.id}/tasks", method="POST")
    input(type="text", name="title")
    button Add
```

连接sequelize

现在，我们需要将sequelize添加到模块依赖中：

server.js

```
/**
 * 模块依赖
 */

var express = require('express')
  , Sequelize = require('sequelize')
```

接下来初始化主类。我们可以直接在模块依赖后做这个初始化工作，也可以出于对程序结构清晰的考虑，在应用设置后做。

server.js

```
/**
 * 初始化 sequelize
 */

var sequelize = new Sequelize('todo-example', 'root')
```

Sequelize构造器接收如下参数：

- database (String)
- username (String) - 必要
- password (String) - 可选
- other options (Object) - 可选
  - host (String)
  - port (Number)

249 可以使用如下命令行来创建数据库。记住将root替换为在Sequelize构造器中使用的MySQL用户名：

```
$ mysqldmin -u root -p create todo-exmaple
```

定义模型和同步

要定义模型，需要调用sequelize.define方法。我们可以在引入sequelize后直接调用该方法。该方法第一个参数是模型名，第二个参数是包含了属性的对象。

server.js

```
var Project = sequelize.define('Project', {
    title: Sequelize.STRING
  , description: Sequelize.TEXT
  , created: Sequelize.DATE
});
```

在上述代码中，属性的类型对应如下sequelize中的类型。接下来，每种类型都对应MySQL中的类型：

- Sequelize.STRING // VARCHAR(255)
- Sequelize.BOOLEAN // TINYINT(1)
- Sequelize.TEXT // TEXT
- Sequelize.DATE // DATETIME
- Sequelize.INTEGER // INT

除了传递类型之外，还可以传递一个包含选项的对象，比如，要设置默认值，就可以传递：

```
title: { type: Sequelize.STRING, defaultValue: 'No title' }
```

接下来定义任务模型：

server.js

```
/**
 * 定义任务模型
 */

var Task = sequelize.define('Task', {
    title: Sequelize.STRING
});
```

最后，设置一个hasMany的联合：

250

server.js

```
/**
 * 设置联合
 */

Task.belongsTo(Project);
 Project.hasMany(Task);
```

为了设置联合，Sequelize会处理构建响应的列、主键以及索引的任务。

belongsTo联合意味着每个Task都有一个指向它所属项目的字段。另外，每个任务模型都会有一个名为getProject的方法来获取其所属的项目。

对于hasMany而言，调用find查询到项目后，它们都会有一个名为getTasks的方法来获取项目中的任务。

除此之外，sequelize还支持另一种关系：hasOne。不过本例中用不到这种联合，它与belongsTo是相对的。

最后，我们要确保schema都同步到数据库中了，并且不需要手动运行CREATE TABLE命令：

```
/**
 * 同步
 */

sequelize.sync();
```

在开发阶段，我们会经常对数据库表做修改操作。因此，我们可以在调用sync方法时，传递一个{force: true}参数来让sequelize始终先删除已有的表，再重新创建，以确保数据变化总能同步到数据库中。

server.js

```
sequelize.sync();
```

创建数据

对于项目和任务列表，我们都需要在表单提交时，绑定一个jQuery的监听器。

当发送Ajax调用时，我们希望返回的是JSON格式的模型实例数据。

251 获取数据后，我们就将其添加到DOM中。

在这之前，我们需要添加一个static中间件来托管public/js目录（也需要创建好）。因为还需要使用jQuery来POST数据，所以还需要bodyParser中间件。

server.js

```
/**
 * 中间件
 */

app.use(express.static(__dirname + '/public'));
app.use(express.bodyParser());
```

public/js/main.js

```
$(function () {
  $('form').submit(function (ev) {
    ev.preventDefault();
    var form = $(this);
    $.ajax({
        url: form.attr('action')
      , type: 'POST'
      , data: form.serialize()
      , success: function (obj) {
          var el = $('<li>');
          if ($('#projects-list').length) {
            el
              .append($('<a>').attr('href', '/project/' + obj.id
+ '/tasks').text(obj.title + ' '))
              .append($('<a>').attr('href', '/project/' + obj.id)
.attr('class', 'delete').text('x'));
          } else {
            el
              .append($('<span>').text(obj.title + ' '))
              .append($('<a>').attr('href', '/task/' + obj.id)
.attr('class', 'delete').text('x'));
          }
          $('ul').append(el);
        }
    });
form.find('input').val(''); // clear the input
  });
});
```

代码很简单。捕获到网站中所有的表单提交，并利用Ajax的方式来提交：

1. 捕获表单提交。
2. 通过调用preventDefault方法来阻止默认行为。也就是说，阻止浏览器试图自动POST表单，因为我们想要用Ajax的方式来提交。 252
3. 调用jQuery的$.ajax方法来提交一个POST请求，同时将表单数据序列化成查询字符串发送过去（通过form.serialize序列化数据，并将其作为data属性值）。

JSON数据返回后，我们重新构造该数据项，并将其添加到项目列表或者任务列表中。如果该数据项是项目，那么我们将其链接添加到任务列表中，同时再添加一个删除链接。若是任务，则简单地添加一个span和删除链接。

现在我们来实现Express应用中的.post路由。这里我们使用模型上的.build方法：

server.js

```
/**
 * 创建项目路由
 */

app.post('/projects', function (req, res, next) {
  Project.build(req.body).save()
    .success(function (obj) {
      res.send(obj);
     })
    ..error(next)
});

/**
 * 为项目添加任务
 */

app.post('/project/:id/tasks', function (req, res, next) {
   res.body.ProjectId = req.params.id;
  Task.build(req.body).save()
    .success(function (obj) {
      res.send(obj);
     })
    .error(next)
});
```

需要记住很重要的一点：像本例这样，如果用户可以设置数据库中的字段，并且没有安全隐患的情况下，我们只需传递整个请求体即可（如：传递req.body）。哪怕我们在表单中只创建了一些输入项，但是不要忘记，用户是可以手动伪造任意请求类型的。

253 如第9章中介绍的，在Express中使用res.send方法可以很容易地发送JSON数据。

如下所示，当调用模型实例上的.save方法时，sequelize会分发一个success事件并传递构建好的对象，或者一个failure事件并传递一个错误对象。在sequelize中，如下代码是有效的：

```
Task.build(req.body).save()
    .on('success', function (obj) {
       res.send(obj);
     })
    .on('failure', next)
```

可以使用success和error方法来更容易地添加事件处理器：

```
Task.build(req.body).save()
  .success(function (obj) {
     res.send(obj);
   })
  .error(next)
```

注意了，为了确保任务和项目之前的关系能够保留，我们在使用Task.build创建任务时，在Task对象上添加了ProjectId字段。回过头来，当我们在模型上设置了belongsTo关系后，sequelize会自动在schema定义中添加ProjectId字段。

获取数据

每个sequelize模型都有简单的方法可以获取指定数据表中单个或多个实例。

调用Model#find方法时，可以直接提供一个主键，并监听success和failure事件：

```
/**
 * 首页路由
 */

app.get('/', function (req, res, next) {
  Project.findAll()
    .success(function (projects) {
      res.render('index', { projects: projects });
    })
    .error(next);
});
```

254 由于此前建立了项目——任务的联合，在/project/:id/items路由，我们可以使用getTasks方法将项目和任务都传递给视图层：

server.js

```
app.get('/project/:id/tasks', function (req, res, next) {
  Project.find(Number(req.params.id))
    .success(function (project) {
```

```
      project.getTasks().on('success', function (tasks) {
        res.render('tasks', { project: project, tasks: tasks });
      })
    })
    .error(next)
});
```

另外，还要注意的是，当使用模型实例的find方法时，需要将参数转化为Number类型。这很重要，因为这样sequelize才能知道这里是用主键去查询。

接下来，我们实现剩下的路由：删除项目和删除任务。

删除数据

接下来，我们利用事件委派来捕获所有delete类的链接，并发送DELETE请求。将如下代码添加到$(form).submit处理器中：

public/js/main.js

```
$('ul').delegate('a.delete', 'click', function (ev) {
  ev.preventDefault();
  var li = $(this).closest('li');
  $.ajax({
      url: $(this).attr('href')
    , type: 'DELETE'
    , success: function () {
        li.remove();
      }
  });
});
```

jQuery的delegate方法允许捕获任意包含了delete类的超链接，不管它是本来就在DOM中的还是后台动态加进去的。

注意，在点击了含有delete类的超链接后，我们查找它的父元素li，并在Ajax请求成功后，将其删除。

接着，定义删除路由：

255

```
/**
 * 删除项目路由
 */

app.del('/project/:id', function (req, res, next) {
  Project.find(Number(req.params.id)).success(function (proj) {
    proj.destroy()
      .success(function () {
```

```
        res.send(200);
      })
      .error(next);
  }).error(next);
});

/**
 * 删除任务路由
 */

app.del('/task/:id', function (req, res, next) {
  Task.find(Number(req.params.id)).success(function (task) {
    task.destroy()
      .success(function () {
        res.send(200);
      })
      .error(next)
  }).error(next);
});
```

如上述代码所示，首先获取任务或者项目的实例，然后，调用destroy将其删除。当destroy指令成功后，发送200状态码给浏览器。

同样的，要想修改获取到的数据项的属性，可以调用updateAttributes。下述代码修改指定任务实例的标题：

```
task.updateAttributes({
   title: 'a new title'
});
```

图13-5展示了完整的应用。可以浏览项目和任务，异步地对它们进行添加和删除。

256

图13-5：为某个项目创建一个新任务

完整地完成应用

sequelize还有许多功能。就像Mongoose操作MongoDB那样，Sequelize可以在MySQL和模

型数据之间添加一层验证层，除了定义类型之外，这个功能也非常有用。

在模型中定义字段时，可以通过传递validate选项来设置验证机制。类型则定义在type中。

比如，要想任务标题只允许大写字母，那么模型定义可以采用如下方式：

```
var Task = sequelize.define('Task', {
    title: { type: Sequelize.STRING, isUppercase: true }
});
```

要设置自定义验证，只需传递任意的函数名和验证函数。要想查看Sequelize自带的完整验证函数列表，可以前往其官方文档查看：http://sequelizejs.com/?active=validations#validations。

还可以通过自定义的类和实例方法来扩展模型：

```
var Task = sequelize.define('Task', {
    title: { type: Sequelize.STRING, isUppercase: true }
  , classMethods: {
      staticMethod: function(){}
    }
  , instanceMethods: {
      instanceMethod: function(){}
    }
});
```

上述例子中的staticMethod方法可以通过如下方式进行调用：

257

```
Task.staticMethod()
```

instanceMethod则可以在查询到的实例上使用：

```
Task.find(4).success(function (task) {
  task.instanceMethod();
});
```

小结

MySQL仍旧是最流行、最可靠的开源数据库之一。无论新的趋势如何，毋庸置疑，MySQL依然是构建各类应用不错的选择。

本章介绍了一个优秀的MySQL Node.js驱动器。不过，需要手动书写 SQL语句来创建数据库、表才能查询。

对于开发Web应用，ORM通常是一种非常有用的武器。在本章第二个例子中，得益于Sequelize，我们无须再手写查询语句，而是可以直接通过模型类和实例来操作数据了。

尽管总要在选择正确的工具这件事情上面多加谨慎，不过，现在你应该懂得了在Node.js中什么样的项目适合使用MySQL。

CHAPTER

14 Redis

既然你已经学会了如何通过Node.js来使用两大主流数据库——MongoDB和MySQL，那么接下来是时候介绍Redis了。 259

Redis是一种数据库，不过更准确地来说，它更像一台结构化的数据服务器，从定义上来说相比MySQL更接近MongoDB。

和操作表中的行或者集合中的文档不同，在Redis中是通过键来访问数据的。因此，可以将Redis想象成是通过如下所示的JavaScript对象的方式来存储数据的： 260

```
{
    'key': 'some value'
  , 'key.2': 'some other value'
}
```

不过，正因为它是一个结构化的数据服务器，能存储的值自然不仅仅是简单的字符串。如下都是Redis支持的数据类型：

- 字符串（string）
- 列表（list）
- 数据集（set）

- 哈希（hash）
- 有序数据集（sorted set）

然而，Redis和MongoDB最显著的不同点就是，Redis中的文档结构总是扁平的。举例来说，即使一个键包含类似哈希的JavaScript对象，却不能包含像MongoDB支持的那种嵌套的数据结构。

另一个不同点在于持久化数据方式的不同。Redis设计的初衷是内存存储，搭配可配置的磁盘持久化思路，所以速度很快。有一点很重要，需要记住：持久化到磁盘是很重要的，因为任何存储在内存中的东西都是不稳定的，而且会随着系统崩溃或者重启而受到影响。

这就意味着，对于敏感的数据系统而言（如处理金融交易），在开发之前，使用以及配置Redis都要非常仔细。尽管Redis的数据集（我们所操作的所有数据）存储在内存中，但是它有多重策略来将数据备份到磁盘中。问题就在于，其默认的配置和行为并不像MySQL那样非常适合数据敏感的系统。这就是Redis被冠以“仅仅算是一种数据库，但却不怎么耐用”的原因了。

如果你是第一次尝试部署Redis，我建议你先看看官网提供的几种不同持久化的选择：http://redis.io/topics/persistence。总的来说，你可以把Redis看作是简单、庞大、扁平（键—值）的JavaScript对象，其中值可以是特殊的数据类型（哈希、数据集、字符串等），它是为高速读写数据孕育而生的（所有数据都存储在内存中）。数据写入后的安全级别也是可配置的，但是，对某类系统而言，它并非是一个好的选择。

261 安装Redis

Redis官方以tar包的形式发布，包含所有源代码，支持Mac OS X和 Linux。如果你想自己编译，可以直接通过http://redis.io/download来下载最新的稳定版本。

而对于Mac来说，最简单的安装Redis的方式是通过`homebrew`：

```
$ brew install redis
```

或者通过`ports`：

```
$ sudo port install redis
```

与从源代码安装不同，这两个包管理器会在载入完毕后，自动安装。通过如下命令可以运行Redis服务器：

```
$ nohup redis-server &
```

对于Windows，也有非官方但维护得很好的版本。通过上面的 URL来查看最新文档了解对

于Windows的支持情况。

Redis查询语言

要开始学习Redis查询语言（换句话说，就好比是Redis中的SQL），首先确保服务器正常运行，随后执行如下命令：

```
$ redis-cli
```

和执行node一样，redis-cli其实就是和Redis服务器之间建立了telnet连接。换句话说，当建立到Redis的TCP连接时，接下来要执行的命令和Node客户端执行的几乎是一样的。

第一个要执行的命令是KEYS。在Redis中，命令都是大小写不敏感的，不过通常都约定了用大写方式。

和函数调用一样，命令可以接收任意数量的参数。如果执行KEYS时不添加任何参数，Redis就会抛出如下错误：

```
redis 127.0.0.1:6379> KEYS
(error) ERR wrong number of arguments for 'keys' command
```

KEYS接收一个匹配键的模式，并返回匹配到的键。*表示匹配所有键：

```
redis 127.0.0.1:6379> KEYS *
(empty list or set)
```

262

由于Redis刚装好，所以上述代码返回空。

现在我们来用SET命令将一个字符串赋值给某个键。Redis会返回OK：

```
redis 127.0.0.1:6379> SET my.key test
OK
```

运行GET my.key会返回刚刚保存的值：

```
redis 127.0.0.1:6379> GET my.key
"test"
```

再次执行KEYS *就会返回新的键：

```
redis 127.0.0.1:6379> KEYS *
1) "my.key"
```

绝大部分Redis指令都是依赖于数据类型的。使用GET和SET可以操作字符串，但是，使用GET却不能获取哈希类型的值。

数据类型

Redis简单的设计带来的最基本的好处之一就是开发者可以很容易地预测出性能。数据库

并不是一个黑盒，而是一个简单的进程，它将某些已知的数据结构存储到内容中，让其他程序通过简单的协议就能够获取到。

如果你在Redis官方文档手册中查看HEXISTS命令，你就会发现其中有一部分是关于时间复杂度的。

HEXISTS命令的时间复杂度是$O(1)$，也就是固定的时长。这就意味着，不管数据集有多大，执行HEXISTS命令总是需要这些时间。

如果查看SMEMBERS，就会发现它的时间复杂度是$O(n)$，也就是线性的时长，所需时间是随着数据集的大小而改变的。这就意味着Redis完成该指令所需的时间直接取决于数据有多少量。

由于Redis的对象模型大致就是一个大的扁平的JSON对象，所以，理解不同数据类型最简单的方式就是把它们想象成JavaScript中的数据类型。对于每一种数据类型，下面都会介绍与之类似的JS世界中的类型。

263 字符串

Redis中的字符串类似于JavaScript中的Number和String。

除了使用SET和GET外，还可以对数字进行递增和递减：

```
redis 127.0.0.1:6379> SET online.users 0
OK
redis 127.0.0.1:6379> INCR online.users
(integer) 1
redis 127.0.0.1:6379> INCR online.users
(integer) 2
```

哈希

在Redis中，哈希类似于子对象。不过和MongoDB不同的是，这些子对象只能局限于字符串形式的键和值。

要在Redis中存储用户信息，如下所示：

```
{
    "name": "Guillermo"
  , "last": "Rauch"
  , "age": "21"
}
```

由于键和值都是字符串（或者数字），所以，用哈希来描述这种数据结构最合适不过了。

正如此前所述，在Redis中，大对象中的所有数据都是通过唯一的键来获取的。要存储用户信息，我们需要将用户ID作为键的一部分来唯一确定存储的值。Redis数据库存储的数据如

下所示：

```
{
    "profile:1": { name: "Guillermo", "last": "Rauch", . . . }
  , "profile:2": { name: "Tobi", "last": "Rauch", . . . }
}
```

上述例子中是否使用冒号（:）取决于你自己。你也可以使用点、下画线或者干脆不用。只要保证在操作文档时每个键都能唯一，避免冲突，同时键名又包含了足够多的信息，能够从应用层面简单地获取到，就可以了。

操作哈希的基本命令是HSET：

```
redis 127.0.0.1:6379> HSET profile.1 name Guillermo
(integer) 1
```

这个命令就等于在JavaScript中设置一个键：

```
obj['profile.1'].name = 'Guillermo';
```

264

要获取一个指定哈希中所有的键和值，可以使用HGETALL，并提供一个键名：

```
redis 127.0.0.1:6379> HGETALL profile.1
1) "name"
2) "Guillermo"
```

Redis会返回一个包含了修改过的键和值的列表：

```
redis 127.0.0.1:6379> HSET profile.1 last Rauch
(integer) 1
redis 127.0.0.1:6379> HGETALL profile.1
1) "name"
2) "Guillermo"
3) "last"
4) "Rauch"
```

要在哈希中删除一个键，可以调用HDEL：

```
redis 127.0.0.1:6379> HSET profile.1 programmer 1
(integer) 1
redis 127.0.0.1:6379> HGETALL profile.1
1) "name"
2) "Guillermo"
3) "last"
4) "Rauch"
5) "programmer"
6) "1"
redis 127.0.0.1:6379> HDEL profile.1 programmer
(integer) 1
redis 127.0.0.1:6379> HGETALL profile.1
1) "name"
2) "Guillermo"
```

```
3) "last"
4) "Rauch"
```

在JavaScript中，上述指令就等同于在哈希中使用了delete操作符：

```
delete obj['profile.1'].programmer
```

还可以使用HEXISTS来检查指定的字段是否存在：

```
redis 127.0.0.1:6379> HEXISTS profile.1 programmer
(integer) 0
```

上述命令等同于检查某个值是否不等于undefined：

```
'undefined' != typeof obj['profile.1'].programmer
```

265 列表

Redis中的列表就等同于JavaScript中的字符串数组。

对于列表，在Redis中有两个基本的操作命令是RPUSH（push到右侧，也就是列表的尾端）和LPUSH（push到左侧，也就是列表的顶端）。

操作列表和操作哈希类似：

```
redis 127.0.0.1:6379> RPUSH profile.1.jobs "job 1"
(integer) 1
redis 127.0.0.1:6379> RPUSH profile.1.jobs "job 2"
(integer) 2
```

然后可以获取指定返回的数组：

```
redis 127.0.0.1:6379> LRANGE profile.1.jobs 0 -1
1) "job 1"
2) "job 2"
```

LPUSH也类似：

```
redis 127.0.0.1:6379> LPUSH profile.1.jobs "job 0"
(integer) 3
redis 127.0.0.1:6379> LRANGE profile.1.jobs 0 -1
1) "job 0"
2) "job 1"
3) "job 2"
```

RPUSH等同于在JavaScript中push一个元素到数组中：

```
obj['profile.1.jobs'].push('job 2');
```

LPUSH等同于unshift：

```
obj['profile.1.jobs'].unshift('job 2');
```

LRANGE命令返回一个在列表中指定范围的元素。它和JavaScript中数组的slice类似但不完全一样。特别是，当第二个参数是-1时，它会返回列表中所有的值。

数据集

数据集处于列表和哈希之间。它拥有哈希的属性，即数据集中的每一项（或者哈希中的键）都是唯一不重复的。既然是等同于哈希中的键，操作数据集中的元素都只需要固定时长（也就是说，无论数据集多大，删除、添加、查找数据集中的元素都只需同等的时间）。

不像哈希但类似列表的是，数据集保存的是单个值（字符串），没有键。不过，数据集还 266
有它专属的有意思的特性。Redis允许在数据集、联合（union）、获取到的随机元素等之间做交集操作。

要添加一个元素到数据集中，可以使用SADD：

```
redis 127.0.0.1:6379> SADD myset "a member"
(integer) 1
```

获取数据集的所有元素，可以使用SMEMBERS：

```
redis 127.0.0.1:6379> SMEMBERS myset
1) "a member"
```

以相同值再次调用SADD不会发生任何事情：

```
redis 127.0.0.1:6379> SADD myset "a member"
(integer) 0
redis 127.0.0.1:6379> SMEMBERS myset
1) "a member"
```

移除数据集中的某个元素，可以使用SREM：

```
redis 127.0.0.1:6379> SREM myset "a member"
(integer) 1
```

有序数据集

有序数据集拥有所有数据集的特性，不过，顾名思义，它是有序的。在Redis中，使用有序数据集的情况较少，属于高级用法。

Redis和Node

既然这些数据结构JavaScript都有（或者可以很容易地构建出来），并且操作它们也不需要什么协议或者服务器，那么Redis的作用究竟在哪里呢？

其中有一个原因就是当关闭Node进程后，所有在内存中的数据都会消失。

在第9章中，我们介绍过如何使用Redis来存储用户的session数据。要是把这些数据存储在

Node进程中，会有两大缺点，如下所示。

- 应用程序永远都无法享受到多线程带来的好处。随着应用程序负载不断增长，单进程无法承受所有的负载，我们需要将应用程序扩展到多进程或者多台计算机。
- 每次重启应用都会丢失session数据：比如，在部署新代码的时候总是需要重启的。

267

Redis还有许多其他非常重要的好处，比如：多种编程语言间的协同、持久性以及其他一些通过书写完整的数据存储方案才能获取的特性。

使用node-redis实现一个社交图谱

作为一个使用Redis和Node的示例程序，我们可以通过使用数据集和交集的强大功能，来构建一个关注（follows）和粉丝（followers）的社交图谱，和Twitter非常类似。

初始化应用

我们选择一个名为node_redis（https://github.com/ mranney/node_redis）项目作为Redis客户端，其NPM的项目名就叫redis：

package.json

```
{
    "name": "sample-social-graph"
  , "version": "0.0.1"
  , "dependencies": {
        "redis": "0.7.1"
    }
}
```

连接redis

node-redis遵循了和我们在第13章中介绍的MySQL客户端类似的设计。

首先，通过require来引入该模块，然后通过createClient来初始化客户端：

```
/**
 * 模块依赖
 */

var redis = require('redis')

/**
 * 创建客户端
 */

var client = redis.createClient();
```

客户端将所有的Redis命令都以函数的方式提供出来。比如，要使用SET，可以这样：

268

```
client.set('my key', 'my value', function (err) {
  // . . .
});
```

其他如SMEMBERS、HEXISTS命令等也是相同的使用方式。

定义模型

首先，我们需要为每个用户关注的人和粉丝创建不同的Redis数据集，其中ID作为键的一部分：

```
user:<id>:follows
user:<id>:followers
```

当某个用户（id "1"）关注了另一个用户（id "2"）时，我们需要执行如下操作：

```
- Add user id "2" to the user:1:follows
- Add user id "1" to the user:2:followers
```

除此之外，我们将用户信息存储在哈希中：

```
user:<id>:data
```

接下来，从定义基本的模型开始：

```
/**
 * 用户模型
 */

function User (id, data) {
  this.id = id;
  this.data = data;
}
```

每个User实例都会包含一个id，来标识该用户及其数据。

我们也可以提供一个静态方法find，用来从Redis查询结果中构建一个User实例：

```
User.find = function (id, fn) {
  client.hgetall('user:' + id + ':data', function (err, obj) {
    if (err) return fn(err);
    fn(null, new User(id, obj));
  });
};
```

创建及修改用户信息

模型需要有查询一个用户、修改该用户的信息以及重新保存到Redis的能力。我们需要能够运行new User，设置一些数据，然后将其保存到Redis中。

269 幸运的是，通过Node来操作哈希要比在Redis命令行中更简单。hgetall和hmset函数可以直接操作JavaScript原生对象，这就是为什么操作起来更容易的原因：

```
client.hmset('somehash', { a: 'key', another: 'key' });
```

上述代码等同于如下所示的在Redis命令行（CLI）执行的命令：

```
HMSET somehash "key" "value" "anotherkey" "anothervalue"
```

当通过hgetall获取到数据后，回调函数中的第二个参数就是JavaScript对象：

```
client.hgetall('somehash', function (err, obj) {
  // obj.a == 'key'
});
```

所以，我们可以给模型添加一个save方法，该方法执行hmset来创建和修改用户信息：

```
User.prototype.save = function (fn) {
  if (!this.id) {
    this.id = String(Math.random()).substr(3);
  }

  client.hmset('user:' + this.id + ':data', this.data, fn)
};
```

定义图谱方法

对于一个指定用户，我们需要做的操作是：关注另一个用户，以及取消关注另一个用户。

```
User.prototype.follow = function (user_id, fn) {
  client.multi()
    .sadd('user:' + user_id + ':followers', this.id)
    .sadd('user:' + this.id + ':follows', user_id)
    .exec(fn);
};

User.prototype.unfollow = function (user_id, fn) {
  client.multi()
    .srem('user:' + user_id + ':followers', this.id)
    .srem('user:' + this.id + ':follows', user_id)
    .exec(fn);
};
```

注意了，和此前的例子不同，这一次调用的是client.multi。调用了multi就意味着告诉Redis客户端，所有的命令都必须要等到exec执行后才能执行，它们作为事务的一部分，应当需要一起执行。

270 如果修改关注列表的时候，发生了某些事件导致了粉丝列表修改失败，这时就会导致数据的不一致。所以，这两部分的修改需要放到同一个事务中来处理。

最后，还需要定义两个获取关注的人和粉丝的方法：

```
User.prototype.getFollowers = function (fn) {
  client.smembers('user:' + this.id + ':followers', fn);
};

User.prototype.getFollows = function (fn) {
  client.smembers('user:' + this.id + ':follows', fn);
};
```

计算交集

除了关注的人和粉丝之外，我们还需要计算第三种关系：朋友。

我们可以说两者互相关注的是朋友。换句话说，如果某个用户的ID同时出现在另一个用户的关注和粉丝列表中，那么他们就是朋友。

添加一个getFriends很容易，直接使用SINTER命令来计算两个数据集的交集即可：

```
User.prototype.getFriends = function (fn) {
  client.sinter('user:' + this.id + ':follows', 'user:' + this.id + ':followers',
  fn);
};
```

测试

要测试模型，我们需要创建一些对Web应用来说有代表性的用户数据。

第一步要做的就是为了更好地重用，需要把此前我们书写的模型代码放到单独的文件中，这样就可以简单地通过require来引入了。注意了，我添加了一行module.exports代码：

model.js

```
/**
 * 模块依赖
 */

var redis = require('redis')

/**
 * 模块导出
 */

module.exports = User;

/**
 * 创建客户端
```

271

```
 */

var client = redis.createClient();

/**
 * 用户模型
 */

function User (id, data) {
  this.id = id;
  this.data = data;
}

User.prototype.save = function (fn) {
  if (!this.id) {
    this.id = String(Math.random()).substr(3);
  }

  client.hmset('user:' + this.id + ':data', this.data, fn)
};

User.prototype.follow = function (user_id, fn) {
  client.multi()
    .sadd('user:' + user_id + ':followers', this.id)
    .sadd('user:' + this.id + ':follows', user_id)
    .exec(fn);
};

User.prototype.unfollow = function (user_id, fn) {
  client.multi()
    .srem('user:' + user_id + ':followers', this.id)
    .srem('user:' + this.id + ':follows', user_id)
    .exec(fn);
};

User.prototype.getFollowers = function (fn) {
  client.smembers('user:' + this.id + ':followers', fn);
};

User.prototype.getFollorws = function (fn) {
  client.smembers('user:' + this.id + ':follows', fn);
};

272 User.prototype.getFriends = function (fn) {
  client.sinter('user:' + this.id + ':follows', 'user:' + this.id + ':followers',
 fn);
};

User.find = function (id, fn) {
  client.hgetall('user:' + id + ':data', function (err, obj) {
```

```
    if (err) return fn(err);
    fn(null, new User(id, obj));
  });
};
```

确保要运行npm install redis来安装本例依赖的唯一模块，与此同时通过运行redis-cli来确保Redis服务器已经运行了。

接下来，我们需要创建一个测试脚本并引入模型。在同一目录下，创建一个test.js文件：

test.js

```
/**
 * 模块依赖
 */

var User = require('./model')
```

如果现在通过node test执行上述文件，你会发现它的进程会挂在那里，不会自己退出。这是因为模型正在和Redis建立连接。

接着，我们要创建一些测试用户。为了更好地组织代码，避免过多的嵌套回调函数，我们定义一个create方法，该方法接收一个包含email和用户数据的对象。本例中，email地址会作为唯一的id存储到Redis中。

test.js

```
/**
 * 模块依赖
 */

var User = require('./model')

/**
 * 创建测试用户
 */

var testUsers = {
    'mark@facebook.com': { name: 'Mark Zuckerberg' }
  , 'bill@microsoft.com': { name: 'Bill Gates' }
  , 'jeff@amazon.com': { name: 'Jeff Bezos' }
  , 'fred@fedex.com': { name: 'Fred Smith' }
};

/**
 * 用于创建用户的函数
```

273

```
 */

function create (users, fn) {
  var total = Object.keys(users).length;
  for (var i in users) {
    (function (email, data) {
      var user = new User(email, data);
      user.save(function (err) {
        if (err) throw err;
        --total || fn();
      });
    })(i, users[i]);
  }
}

/**
 * 创建测试用户
 */

create(testUsers, function () {
  console.log('all users created');
  process.exit();
});
```

这个时候运行node test，应当就已经成功添加了四个用户。可以使用redis-cli来检查是否正确：

```
redis-cli
redis 127.0.0.1:6379> KEYS *
1) "user:fred@fedex.com:data"
2) "user:jeff@amazon.com:data"
3) "user:bill@microsoft.com:data"
4) "user:mark@facebook.com:data"
redis 127.0.0.1:6379> HGETALL "user:fred@fedex.com:data"
1) "name"
2) "Fred Smith"
```

注意了，上述HGETALL命令和User.find函数的作用是一样的：它根据id获取对应用户的数据。

274 为了避免混淆，每次执行完node test，都建议将Redis数据库清空以确保由不同测试用例创建的数据不会相互混淆。清空数据库可以使用如下命令：

```
redis-cli FLUSHALL
```

至此，用户已经成功创建好了，接下来我们需要根据email来获取User对象，这样就可以通过此前在模型中定义的方法来建立用户之间的关系。这个处理过程称为水合（hydration），我们要建立一个hydrate方法，并在create回调函数中使用：

test.js

```
/**
 * 用于水合用户的函数
 */

function hydrate (users, fn) {
  var total = Object.keys(users).length;
  for (var i in users) {
    (function (email) {
      User.find(email, function (err, user) {
        if (err) throw err;
        users[email] = user;
        --total || fn();
      });
    })(i);
  }
}

/**
 * 创建测试用户
 */

create(testUsers, function () {
  hydrate(testUsers, function () {
    console.log(testUsers);
  });
});
```

如图14-1所示，若再次运行测试脚本，你会发现testUsers对象包含了User对象（该对象包含在User构造器中传递的id和data属性）。

```
{ 'mark@facebook.com': { id: 'mark@facebook.com', data: { name: 'Mark Zuckerberg' } }
 },
  'bill@microsoft.com': { id: 'bill@microsoft.com', data: { name: 'Bill Gates' } },
  'jeff@amazon.com': { id: 'jeff@amazon.com', data: { name: 'Jeff Bezos' } },
  'fred@fedex.com': { id: 'fred@fedex.com', data: { name: 'Fred Smith' } } }
```

275

图14-1：以JSON格式输出的创建好的用户数据

水合后，就可以使用模型上多种不同的方法了：

test.js

```
create(testUsers, function () {
  hydrate(testUsers, function () {
    testUsers['bill@microsoft.com'].follow('jeff@amazon.com', function (err) {
      if (err) throw err;
      console.log('+ bill followed jeff');

      testUsers['jeff@amazon.com'].getFollowers(function (err, users) {
        if (err) throw err;
        console.log("jeff's followers", users);

        testUsers['jeff@amazon.com'].getFriends(function (err, users) {
          if (err) throw err;
          console.log("jeff's friends", users);

          testUsers['jeff@amazon.com'].follow('bill@microsoft.com'
          , function (err) {
            if (err) throw err;

            console.log('+ jeffed follow bill');
276         testUsers['jeff@amazon.com'].getFriends(function (err, users) {
              if (err) throw err;

              console.log("jeff's friends", users);
              process.exit(0);
            });
          });
        });
      });
    });
  });
});
```

在图14-2中，如控制台输出所示，它展示了Redis中的交集操作的结果。

图14-2：社交图谱中的粉丝和朋友

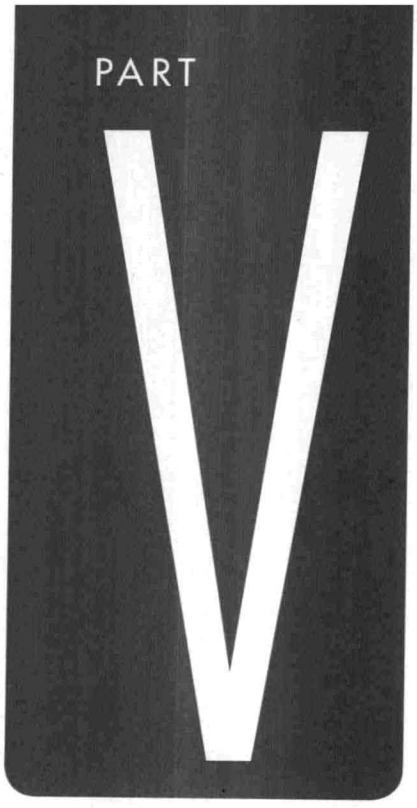
小结

在我看来Redis是最重要的、生机盎然的数据库之一，这也是我在本章一开始花笔墨来介绍它的基本技术点的原因。

因为它就像是结构化的数据服务器，既可以用作普通的数据库，也可作为一种黏合剂来协调多个小程序。

比如，在第9章中，我们介绍了如何使用`connect-redis`来当重启Node进程时保持session数据的持久化。当然了，Node本身也可以将这些数据存储在内存中，不过通过将数据分离到另一个进程，并通过简单的TCP协议来连接操作，就可以获得灵活独立的好处：不管程序本身有没有运行，数据都在那儿。

大量程序和Web应用都有简单的数据模型，数据集也非常适合存储在内存中。对于这些使用场景，我推荐你首先考虑Redis，因为它简单、可靠，而且通过Node.js非常易于使用。本章中的示例程序就是一个很好的例子：我们书写了一个完整的模型，该模型仅通过使用 Redis驱动器就包含了所有当今社交应用所需的重要功能。



测试

CHAPTER

15 代码共享

在本书介绍部分，我就提过，Node.js最重要的特点之一就是：它所使用的语言——JavaScript，是浏览器唯一支持的语言。 279

尽管现在我们已经写了很多独立的JavaScript代码，减少了在开发 Web应用时总要在不同语言之间切换上下文的痛苦。我们还没有享受到一次编码处处运行的好处。

本章分析适合代码共享的最佳用例，以及如何解决常见的语言兼容性问题。本章还会介绍如何书写模块化Node代码的最佳实践，并将其通过`browserbuild`编译后在浏览器中直接运行。

什么样的代码可以共享 280

回答代码是否可以在浏览器和服务器端共享的最简单的方式就是把问题拆为如下两个问题：

- 是否值得在两个环境中运行同一份代码？
- 代码依赖的API在两个环境中是否都有？如果不是，那么是否能够很容易地找到替换它们的方法（称为模拟实现法）。

回答第一个问题比较简单，答案本身取决于程序和项目本身。回答第二个问题就有点困难了。

在第2章中，我们介绍了有些特定的和JavaScript有关联的API并非是语言的一部分，而是浏览器提供的标准API。例如：XMLHttpRequest、WebSocket、DOM、2d画布API，等等。

尽管Node.js核心并未提供XMLHttpRequest API，但是我们可以通过使用Node.js的HTTP客户端API来替代。在NPM有个名为xmlhttprequest的项目就是起这个作用的。

在其他的场景中，某些模块只和原生API交互，这类代码就很容易写成可以在两个环境中运行。

这类例子如下所示。

- 日期操作工具集：它们通常简单地处理原生Date API或者对其做相应扩展。
- 模板引擎：它们通常接收一个字符串，通过正则表达式或者循环来解析，并生成一个（编译）函数，用于输出编译后的字符串。
- 数学及加密库：它们通常处理Number和与数学相关的内容。
- 面向对象框架：它们提供一些在JavaScript中书写类的语法糖。换句话说，提供一些API，使得在JavaScript中能和在其他典型的面向对象语言中一样书写面向对象代码。

书写兼容的JavaScript代码

第一个挑战就是要解决在Node.js模块系统中书写的JavaScript代码无法在浏览器中运行的问题。

导出模块

第一个问题是浏览器环境中没有module全局变量。哪怕文件本身没有依赖，但是，要想让它能为其他Node程序所用，需要通过module.exports或者exports。

281 假设，我们已熟悉一个简单的两数求和的函数：

add.js

```
module.exports = function (a, b) {
  return a + b;
}
```

在Node中，只需简单地从另外一个文件中调用require('./add')就可以了。在浏览器中，我们更希望将其暴露为一个全局变量add。

为了避免修改已有代码，我们可以在浏览器中，通过伪造一个module对象来模拟module.exports：

```
if ('undefined' == typeof module) {
  module = { exports: {} };
}
```

模块代码的最后，对象生成后，就可以将其暴露给window全局对象了。

```
if ('undefined' != typeof window) {
  window.add = module.exports;
}
```

和Node.js不同，在浏览器中，文件都是在全局作用于下执行的。因此，我们需要为程序引入一个全局的module变量。

将模块代码包裹在一个自执行的函数中是个不错的主意。

```
(function (module) {
  module.exports = function (a, b) {
    return a + b;
  }

  if ('undefined' != typeof window) {
    window.add = module.exports;
  }
})('undefined' == typeof module ? { module: { exports: {} } } : module);
```

太好了！模块还是可以通过require获得：

```
$ node
> require('./add')(1,2)
3
```

同时，还能在DOM中引入：

282

add.html

```
<script src="add.js"><script>
<script>
  console.log(add(1,2));
</script>
```

模拟实现ECMA API

下一个挑战就是，一些主流浏览器中的特性在其他浏览器及JavaScript引擎中都没有。

有时，Node中的v8引擎甚至比诸多电脑所用的Google Chrome浏览器的引擎还要新。然而，对于Function#bind这样的API，一些主流引擎，如Safari引擎JavaScriptCore VM就还不支持。所以，要想让代码处处运行，对于使用了哪些功能要特别小心。

对于缺失的这类功能，有两个解决办法：通过扩展原型来提供这类功能的实现或者使用工具函数。

扩展原型

假设我们书写了一个同时在Node和浏览器中运行的模块，它用到了Function#bind方法。

```
var myfn = fn.bind(this);
```

由于bind方法并不一定在所有环境中都存在，所以当它不存在时，我们要创建一个出来：

```
if (!Function.prototype.bind) {
  Function.prototype.bind = function () {
    // code that replicates bind behavior
  }
}
```

有一个名为es5-shim的项目做了类似上面的事情，它把所有浏览器中缺少的ECMA标准API都实现了。要了解该项目详情，可以参考https://github.com/kriskowal/es5-shim。

这种技术有个很明显的好处就是在加入填补的方法后，几乎不需要修改源代码。

283 缺点就是这么做会破坏原型对象，影响其他使用者，可能还会更糟。

工具函数

另一种解决方法就是定义简单的函数，接收原生对象作为参数，如果该对象上的函数已经实现了，就直接使用，否则就实现一次。

例如，Object.keys就是个很好的例子，在Node中会经常使用此函数，但是在很多浏览器中却还没有提供其实现。

接着，我们就通过如下方式来定义一个工具函数：

```
var keys = Object.keys || function (obj) {
  var ret = [];
  for (var i in obj) {
    if (Object.prototype.hasOwnProperty.call(obj, i)) {
      ret.push(i);
    }
  }
  return ret;
};
```

这种技术的好处就是随处都能用，没有隐患，并且对于开发者来说，能够一眼就看明白哪些方法是模拟的，哪些是原生的，这对于在某些浏览器中检测代码性能很有帮助。

缺点就是，我们要记住是用工具函数而不是用原生的。

除此之外，有的时候，最终书写出来的代码会有些冗长。比如，旧版本的IE不支持Array#forEach、Array#map及Array#filter，要想写出兼容的代码，可能要写成如下形式：

```
arr.filter(function () {}).map(function(){ }).forEach(function () {})
```

不过，上述代码远没有使用工具函数来得清晰：

```
each(map(filter(arr, function() {}), function () {}), function (){})
```

模拟实现Node API

有些如EventEmitter这样的Node API一般会在自定义的类中使用。幸运的是，Node社区书写了可以在所有环境下运行的Node API。

EventEmitter

284

对node EventEmitter的实现可以参考https://github.com/Wolfy87/EventEmitter和https://github.com/tmpvar/node-eventemitter。

assert

支持浏览器的assert模块可以在这里找到： https://github.com/Jxck/assert。

模拟实现浏览器端API

很多情况下，我们希望在Node中也能运行浏览器支持的方法。

XMLHttpRequest

node-XMLHttpRequest项目（https://github.com/driverdan/node-XMLHttpRequest）为Node提供了XMLHttpRequest的模拟实现，通过如下方式就可以引入该模块了：

```
var XMLHttpRequest = require('xmlhttprequest')
```

DOM

一个完整并且测试完全的DOM I、DOM II以及 DOM III的实现在这里：https://github.com/tmpvar/jsdom。

WebSocket

Node中也有WebSocket客户端的API实现（https://github.com/einaros/ws）：

```
var WebSocket = require('ws')
```

node-canvas

用于图片操作的2D画布上下文也有在Node中的实现，参考node-canvas：https://github.com/learnboost/node-canvas。

跨浏览器的继承实现

在模块中，经常需要将一个类继承自另一个类。第2章中介绍的.__proto__可以用于实现继承，但却很难模拟实现。

一个简单的解决方法是通过如下方式定义一个工具函数：

```
/**
 * 用于实现继承的工具函数
 *
 * @param {Function} 构造器
285 * @param {Function} 父类构造器
 * @api private
 */

function inherits (a, b) {
  function c () {};
  c.prototype = b.prototype;
  a.prototype = new c;
};
```

然后就可以在Node和浏览器中使用了。

```
function A () {}
function B () {}

inherits(A, B);
// instead of A.prototype.__proto__ = B.prototype
```

集成到一起：browserbuild

通过自执行函数将模块代码包裹起来，同时到处都得执行typeof检测多少有点麻烦。

我创建了一个名为browserbuild的项目，它的作用就是将以Node风格书写的代码（即使用了require、module.exports以及exports，并且代码都分属不同文件），通过运行简单的命令，就编译为浏览器端可执行的版本。

browserbuild可以允许你以Node风格书写此前的求和模块，通过编译后，就会生成一个可以直接在浏览器端通过<script>嵌入执行的版本。

browserbuild同时还是个NPM模块，它提供了browserbuild命令行脚本。要全局安装该模块，可以采用如下方式：

```
npm install -g browserbuild
browserbuild --version
```

上述第二行命令用于展示所安装browserbuild的版本号。这里所用的版本号是0.4.8。

还可以直接安装到工作目录中，通过.bin目录来访问：

```
npm install browserbuild
./node_modules/browserbuild/.bin/browserbuild --version
```

基础案例

286

本例中，我们要书写一个模块，该模块依赖于日志工具。

首先，书写一个main.js文件：

main.js

```
var log = require('./log'):

module.exports = function () {
log('Executed my module');
}
```

在使用特定模块时，main文件不管是在NPM模块系统中，还是在单独的文件中，都可以通过require来引入。

log.js

```
module.exports = function (str) {
  return console.log(str);
}
```

在Node中，可以直接写成；

node.js

```
var mymodule = require('./main')
mymodule();
```

对于浏览器端，我们需要其导出成全局的mymodule变量。所以，我们需要在工作目录下运行browserbuild命令，同时提供全局变量名以及main文件。

```
browserbuild --main main --global mymodule main.js log.js > out.js.
```

上述代码的含义是告诉browserbuild：编译main.js和log.js，并将main模块导出为mymodule全局变量。

除此之外，注意，命令行最后我们使用了>compiled.js。这是将输出的结果保存到compiled.js文件中。

要将其作为库引入到浏览器中，只需添加一个<script>标签，并指向compiled.js脚本即可：

```
<script src='compiled.js">
```

287 生成的脚本如下所示：

compiled.js

```
(function(){var global = this;function debug(){return debug};function require(p,
 parent){ var path = require.resolve(p) , mod = require.modules[path]; if (!mod)
 throw new Error('failed to require "' + p + '" from ' + parent); if (!mod.exports)
 { mod.exports = {}; mod.call(mod.exports, mod, mod.exports, require.relative(path),
 global); } return mod.exports;}require.modules = {};require.resolve =
 function(path){ var orig = path , reg = path + '.js' , index = path + '/index.js';
 return require.modules[reg] && reg || require.modules[index] && index ||
 orig;};require.register = function(path, fn){ require.modules[path] = fn;};require.
 relative = function(parent) { return function(p){ if ('debug' == p) return debug;
 if ('.' != p.charAt(0)) return require(p); var path = parent.split('/') , segs =
 p.split('/'); path.pop(); for (var i = 0; i < segs.length; i++) { var seg =
 segs[i]; if ('..' == seg) path.pop(); else if ('.' != seg) path.push(seg); } return
 require(path.join('/'), parent); };};require.register("main.js", function(module,
 exports, require, global){

var log = require('./log');

module.exports = function () {
  log('Executed my module');
}

});require.register("log.js", function(module, exports, require, global){
module.exports = function (str) {
  return console.log(str);
}

});mymodule = require('main');
})();
```

上述脚本中，第一部分实现了一个require函数供后续代码使用。编译后的脚本全部压缩在一行代码中，这样文件长度对于编译过程的影响可以减到最小。

另一部分比较有意思的代码是，require.register函数提供了一个global参数。这样做是为了避免你去检测window对象是否存在，以此来提供另一个global变量。书写代码时可以直接看作是Node.js的global对象，要是在浏览器中，该对象会变成window对象。

最后，上述代码的最后一部分是导出了全局变量（mymodule），这也解释了运行browserbuild的时候要指定是main模块的原因：

```
mymodule = require('main');
```

browserbuild另一个有意思的特性是if node代码块，它允许你使用JavaScript注释来告诉编译器，该代码块中的代码不需要编译到浏览器端版本中： 288

nodeonly.js

```
// if node
process.exit(1);
// end

console.log('browser and node');
```

如果运行如下命令编译上述代码，if node代码块中的代码就不会被编译进去：

```
$ browserbuild --main nodeonly nodeonly.js
```

你会注意到process.exit代码行不在里面。

```
// …
require.register("nodeonly.js", function(module, exports, require, global){
console.log('browser and node');
});nodeonly = require('nodeonly');
})();
```

要获取更多browserbuild命令的选项，可以执行browserbuild --help，或者参看项目主页：http://github.com/learnboost/browserbuild。

小结

正如全书一直在强调的，Node.js做了一项很伟大的工作就是将 JavaScript带到了服务器端。

这一创新依赖于其模块系统，而浏览器端则没有对应的默认模块系统。

本章从介绍如何利用运行时的方法检测来书写同时能在服务器端和浏览器端运行的代码。通过执行typeof检查，可以实现模块系统的特性检查，并为浏览器端也提供一个导出机制，比如通过全局对象。

然而，在所有的文件中都要进行手动地通过自执行的函数将代码包裹起来，并执行typeof进行检查，这显然有悖于Node中require系统的简洁性，browserbuild就是为了解决这类问题而生的。有了它，就可以很轻松地书写Node.js模块，并编译到浏览器端进行运行了。

这种方法的显著的好处就是，模块是完全以全局变量的方式在浏览器端导出的，就和jQuery和IO一样，也就是说，并没有引入一个特定的模块系统API来让终端用户在浏览器环境中使用。

CHAPTER

16 测试

至此，每次书写完Node程序，都要通过执行程序来检测是否结果和我们预期的一样，从而判断工作是否正常。随着时间越来越长，这种确保程序工作正常并且没有引入新bug的测试方法是非常低效的。 289

自动化测试是一个过程，它通过执行一系列程序来检验函数工作是否正确。本章首先要介绍的自动化测试的方法是为每个测试创建一段小的 node程序，然后使用原生的`assert`模块来检验。

接着，我们会通过使用一个名为expect.js的项目来优化书写断言（assertion）的过程。紧接着，我们还会介绍如何使用测试框架 Mocha来组织测试代码。

最后，对于在Node和浏览器端都要运行的代码，本章会介绍如何将已有的测试也用到浏览器端。

简单测试 290

开始前，我们先要确定测试目标。换句话说，我们得确定哪些脚本或者功能需要测试。

测试目标

本章的测试目标是我们在第9章中书写的查询推文的应用。

这里，我们书写一个程序，来断言提交查询时，看看是否能够找到对应的HTML代码来证明查询的推文内容返回了。本例子，推文列表是通过一个或者多个<li>元素来构建的。通过断言查询关键字是否存在，以及<li>字符串是否为HTTP响应内容的一部分来判断应用程序是否工作正常，应当已经很充分了。

要开始自动化测试前，我们先运行应用，确保它能正常工作。然后，通过浏览器访问http://localhost:3000。

测试策略

最基本的测试方法就是书写一个新的node程序，当测试通过时就以状态码0退出程序，失败则以状态码1退出。

如果有未捕获的异常（uncaught exception）抛出，Node会自动以失败错误码退出程序，和我们的设计相符。除此之外，还能获得堆栈踪迹来帮助调试错误原因。因此，书写测试代码时，目标就是要断言条件是否通过，不通过就抛出异常。Node自带了一个assert模块，顾名思义，它的作用就是检查判断条件是否通过，如果不通过，就抛出一个AssertionError异常。

作为例子，我们书写一个测试程序，当时间戳是偶数时成功，是奇数时失败，同时抛出堆栈信息。

```
/**
 * 模块依赖
 */

var assert = require('assert');

/**
 * 断言条件
 */

var now = Date.now();
console.log(now);
assert.ok(now % 2 == 0);
```

291 assert.ok可以用来判断提供的值是否为真（哪怕真是值不是true，我们还是要断言它是true）。当数字除以2没有余数时，那么该数字就是偶数。

现在，我们多运行几次上述程序，看看时间戳：

```
 ∞ simple-testing node assert-example.js
1325520251830
 ∞ simple-testing node assert-example.js
1325520252742
 ∞ simple-testing node assert-example.js
1325520253637

node.js:134
        throw e; // process.nextTick error, or 'error' event on first tick
        ^
AssertionError: true == false
    at Object.<anonymous> (assert-example.js:14:8)
    at Module._compile (module.js:411:26)
    at Object..js (module.js:417:10)
    at Module.load (module.js:343:31)
    at Function._load (module.js:302:12)
    at Array.<anonymous> (module.js:430:10)
    at EventEmitter._tickCallback (node.js:126:26)
```

前两次中，由于时间戳是偶数，所以测试通过了。第三次，时间戳变成了奇数，所以抛出了异常，输出了堆栈信息。

测试程序

现在，我们使用superagent，发送GET请求来查询bieber关键字，然后分析响应结果：

```
/**
 * 模块依赖
 */

var request = require('superagent')
  , assert = require('assert')

/**
 * 测试 /search?q=<tweet>
 */

request.get('http://localhost:3000')
  .data({ q: 'bieber' })
  .exec(function (res) {

  // 断言响应状态码是否正确
  assert.ok(200 == res.status);

  // 断言关键字是否存在
  assert.ok(~res.text.toLowerCase().indexOf('bieber'));

  //断言列表项是否存在
  assert.ok(~res.text.indexOf('<li>'));
});
```

292

- throwException：断言Function在调用时是否会抛出异常。

```
expect(fn).to.throwException();
expect(fn2).to.not.throwException();
```

- within：断言数组是否在某个区间内。

```
expect(1).to.be.within(0, Infinity);
```

- greaterThan/above：断言 >。

```
expect(3).to.be.above(0); expect(5).to.be.greaterThan(3);
```

- lessThan/below：断言 <。

```
expect(0).to.be.below(3); expect(1).to.be.lessThan(3);
```

改进了断言风格，接下来，我们要通过一个名为Mocha的框架重构测试代码的组织方式。

Mocha

Mocha是一个测试框架，简化了书写测试代码的过程，提供划分测试集、运行并同时输出有助于开发者理解的结果页。

相比把一份份测试代码写在不同文件中，会导致的文件系统中存有大量无序文件，通过Mocha，可以书写出如下形式的测试代码：

test.js

```
describe('a topic', function () {

  it('should test something', function () {

  });

  describe('another topic', function () {

    it('should test something else', function () {

    });

  });

});
```

295

与expect.js类似，在test.js中描述和组织测试代码的方式非常自然直观。

要运行测试代码只需通过mocha命令即可。在这之前，需要确保运行npm install -g mocha安装了mocha。然后，就通过如下方式来运行：

- be/equal：类似===。

```
expect(1).to.be(1);
expect(NaN).not.to.equal(NaN);
expect(1).not.to.be(true);
expect('1').to.not.be(1);
```

- eql：断言非严格相等，支持对象。

```
expect({ a: 'b' }).to.eql({ a: 'b' });
expect(1).to.eql('1');
```

- a/an：断言所属类型，支持数组和instanceof。

```
// typeof with optional array
expect(5).to.be.a('number');
expect([]).to.be.an('array'); // works
expect([]).to.be.an('object'); // works too, since it uses typeof`
// constructors
expect(5).to.be.a(Number);
expect([]).to.be.an(Array);
expect(tobi).to.be.a(Ferret);
expect(person).to.be.a(Mammal);
```

- match：断言字符串是否匹配一段正则表达式。

```
expect(program.version).to.match(/[0-9]+\.[0-9]+\.[0-9]+/);
```

- contain：断言字符串是否包含另一个字符串，内部使用indexOf方法。

```
expect([1, 2]).to.contain(1);
expect('hello world').to.contain('world');
```

- length：断言数组长度，内部使用.length方法。

```
expect([]).to.have.length(0);
expect([1,2,3]).to.have.length(3);
```

- empty：断言数组是否为空。

```
expect([]).to.be.empty();
expect([1,2,3]).to.not.be.empty();
```

- property：断言某个自身属性（或值）是否存在。

```
expect(window).to.have.property('expect');expect(window).to.have.
  property('expect', expect)
expect({a: 'b'}).to.have.property('a');
```

- key/keys：断言键是否存在，支持only修饰符。 294

```
js expect({ a: 'b' }).to.have.key('a');
expect({ a: 'b', c: 'd' }).to.only.have.keys('a', 'c');
expect({ a: 'b', c: 'd' }).to.only.have.keys(['a', 'c']);
expect({ a: 'b', c: 'd' }).to.not.only.have.key('a');
```

注意了，如果请求抛出异常，我们直接不做处理，往上抛即可，最终它会变成一个未捕获的异常，导致程序失败并退出。

记住在运行测试前要先安装superagent：

```
npm install superagent@0.4.1
```

expect.js

在此前的例子中，我们使用了assert.ok以及基本的JavaScript表达式。

在看测试代码时，你也许会发现很难看懂到底在测试什么。比如，断言一个字符串是否包含另一个字符串时，最简单的方法就是使用indexOf方法和~操作符，正如此前例子中处理的那样。

expect.js提供了一个简单的expect函数，可以将：

```
assert.ok(~res.text.indexOf('<li>'));
```

改写成：

```
expect(res.text).to.contain('<li>');
```

通过近自然语言的表达，使得书写和理解测试代码变得更加容易。

expect.js可以通过NPM获取，其名为expect.js，同时，它的官方文档是https://github.com/learnboost/expect.js。

接下来，会介绍一些expect.js的基础API。

API一览

通过引入expect.js模块就可以使用expect函数：

```
var expect = require('expect.js')
```

293

expect.js可以和任何模块一起使用，如此前用到的assert。与所有assert模块提供的函数类似，在expect.js中，当断言失败时会抛出AssertionError。

下面是一些expect.js 0.1.2中最有用的方法。

- ok：断言值是否为真。

```
expect(1).to.be.ok();
expect(true).to.be.ok();
expect({}).to.be.ok();
expect(0).to.not.be.ok();
```

```
mocha test.js
```

Mocha使用多种报告形式来展示测试通过和运行结果。

```
mocha test.js

  ..

    2 tests complete (0ms)
```

比如，有种报告形式是列表：

```
mocha -R list test.js

    a topic should test something: 0ms
    a topic another topic should test something else: 0ms

    2 tests complete (1ms)
```

正如下面将要介绍的，Mocha还有一种HTML报告形式，可以让你直接在浏览器查看运行。

测试异步代码

Mocha默认会在一个测试用例执行完毕后立刻执行另一个。然而，很多时候，当有异步事件发生时，我们希望能够延缓下一个测试用例的执行。

考虑如下例子：

```
it('should not throw', function () {
  setTimeout(function () {
    throw new Error('An error!');
  }, 100);
});
```

上述测试代码总会通过。原因就在于设置了计时器后代码就执行完毕了，Mocha会继续去执行下一个。由于在计时器设置后，没有立刻抛出异常，所以测试通过。

对于这类异常要在将来才会抛出（异步的行为）的测试，我们需要告诉Mocha，等到我们通知它说完成了才能认为该测试完成了。

要解决这个问题，我们只要简单地在回调函数中添加一个参数就可以了。也就是说，正如第2章中介绍的，我们将函数的参数数量（或者是`function#length`）设置为1：

296

```
it('should not throw', function (done) {
  setTimeout(function () {
    assert.ok(1 == 1);
  }, 100);
});
```

现在，测试代码看起来就像中间件一样。Mocha会一直等到done函数调用之后才会认为该测试已经结束。如果该函数在两秒内（默认）没有调用，就会抛出一个超时异常来告诉你哪个测试卡住了。你也可以通过mocha命令的-t选项来修改这个超时时长。还可以通过如下方式来自定义超时时长：

```
it('will fail', function () {
  this.timeout(100);
  setTimeout(function () {
    // the test will timeout before this occurs
  }, 1000);
});
```

为了让这个测试用例能够工作，我们在断言结束后调用done方法：

```
it('should not throw', function (done) {
  setTimeout(function () {
    assert.ok(1 == 1);
    done();
  }, 100);
});
```

我们可以使用Mocha将此前测试查询Bieber推文应用的代码改写为如下形式：

```
it('should find bieber tweets', function (done) {
  request.get('http://localhost:3000')
    .data({ q: 'bieber' })
    .exec(function (res) {

      //断言状态码是否正确
      assert.ok(200 == res.status);

      //断言查询关键字是否存在
      assert.ok(~res.text.toLowerCase().indexOf('bieber'));

      //断言列表项是否存在
      assert.ok(~res.text.indexOf('<li>'));

      done();
    });
});
```

297 有时，一个测试用例只有在当多个异步操作一起完成时才算通过。

这时我们就可以使用一个计数器来解决这个问题：

```
it('should complete three requests', function (done) {
  var total = 3;
  request.get('http://localhost:3000/1', function (res) {
    if (200 != res.status) throw new Error('Request error');--total || done();
  });
```

```
  request.get('http://localhost:3000/2', function (res) {
    if (200 != res.status) throw new Error('Request error');--total || done();
  });
  request.get('http://localhost:3000/3', function (res) {
    if (200 != res.status) throw new Error('Request error');;
    --total || done();
  });
});
```

注意了，Mocha非常“聪明”，它能够识别出未捕获的异常属于哪个测试用例。这是因为Mocha任何时候只执行一个测试用例，所以它能够准确地将通过process.on('uncaughtException')处理器捕获的未捕获的错误（uncaught Exception）链接到正确的测试用例上。

BDD风格

前面小节中使用的测试代码书写风格称为：行为驱动开发（BDD）。

在接下来的例子中，我们给jade提供一个包含段落的模板来测试Jade的处理行为。过程中还会用到expect.js。

首先，安装Jade、Mocha和expect.js：

```
$ npm install expect.js jade mocha
```

bdd.js

```
var expect = require('expect.js')
  , jade = require('jade');

describe('jade.render', function () {
  it('should render a paragraph', function () {
    expect(jade.render('p A paragraph')).to.be('<p>A paragraph</p>');
  });
});
```

由于Mocha安装在项目路径中时并非是全局安装，所以，要在./node_modules/bin找到它并执行：

```
$ ./node_modules/.bin/mocha bdd.js
```

TDD风格

298

接下来要介绍的是测试驱动开发（TDD）。它和BDD类似，但是组织方式是使用测试集（suite）和测试（test）。

每个测试集都有setup和teardown函数。这些方法会在测试集中的测试执行前执行，它们的作用是为了避免代码重复以及最大限度使得测试之间相互独立。

现在，测试代码看起来就像中间件一样。Mocha会一直等到done函数调用之后才会认为该测试已经结束。如果该函数在两秒内（默认）没有调用，就会抛出一个超时异常来告诉你哪个测试卡住了。你也可以通过mocha命令的-t选项来修改这个超时时长。还可以通过如下方式来自定义超时时长：

```
it('will fail', function () {
  this.timeout(100);
  setTimeout(function () {
    // the test will timeout before this occurs
  }, 1000);
});
```

为了让这个测试用例能够工作，我们在断言结束后调用done方法：

```
it('should not throw', function (done) {
  setTimeout(function () {
    assert.ok(1 == 1);
    done();
  }, 100);
});
```

我们可以使用Mocha将此前测试查询Bieber推文应用的代码改写为如下形式：

```
it('should find bieber tweets', function (done) {
  request.get('http://localhost:3000')
    .data({ q: 'bieber' })
    .exec(function (res) {

      //断言状态码是否正确
      assert.ok(200 == res.status);

      //断言查询关键字是否存在
      assert.ok(~res.text.toLowerCase().indexOf('bieber'));

      //断言列表项是否存在
      assert.ok(~res.text.indexOf('<li>'));

      done();
    });
});
```

297 有时，一个测试用例只有在当多个异步操作一起完成时才算通过。

这时我们就可以使用一个计数器来解决这个问题：

```
it('should complete three requests', function (done) {
  var total = 3;
  request.get('http://localhost:3000/1', function (res) {
    if (200 != res.status) throw new Error('Request error');--total || done();
  });
```

2 告诉Mocha采用什么样的测试风格（TDD、BDD或者export）。

3 载入测试代码。

4 运行Mocha。

expect.js可以在所有浏览器中运行，它可以很好地和Mocha配合使用。

建立项目

我们从创建一个test/文件夹开始，该文件夹包含Mocha在浏览器端运行所需文件（mocha.css和mocha.js）。

我们可以从node_modules/mocha目录中将这两个文件复制过来，或者从git仓库中直接下载。要了解如何下载这些文件，可以参考 http://mochajs.com。

我们还需要载入jQuery、expect.js以及测试代码，然后调用mocha.setup来设置测试风格。

最终，test/index.html代码如下所示：

test/index.html

```
<!doctype html>
<html>
  <head>
    <title>my tests</title>
    <link href="/mocha.css" rel="stylesheet" media="screen" />
    <script src="/jquery.js"></script>
    <script src="/mocha.js"></script>
    <script src="/expect.js"></script>
    <script>mocha.setup('bdd');</script>
    <script src="/my-test.js"></script>
    <script>window.onload = function () { mocha.run(); };</script>
  </head>
  <body>
    <div id="mocha"></div>
  </body>
</html>
```

300

如上述代码所示，Mocha会执行my-test文件，该文件中以BDD的风格做了一些简单的测试：

my-test.js

```
describe('my tests', function () {
  it('should not throw', function () {
    expect(1 + 1).to.be(2);
  });
});
```

托管目录

最简单的以网站的形式托管整个目录的方式是使用serve(1)命令。这是一个简单的命令行程序，serve内部使用connect的static中间件来托管指定的目录：

```
$ serve .
```

之后，简单地通过浏览器访问http://localhost:3000就可以了。

小结

本章一开始介绍了如何以最简单的方式书写测试程序：运行一个简单的测试脚本并查看其运行结果是否成功。

接着，为了验证在测试脚本中，某段条件判断是否通过，本章介绍了一个Node.js核心模块assert。

在这基础上，本章又介绍了使用expect.js和Mocha来提升测试代码风格以及组织形式。Expect可以让书写的测试代码清晰易懂，而Mocha提供了很好的组织测试代码的方法，同时还允许将测试代码在浏览器端运行。除此之外，对于测试异步代码，也游刃有余。

索引

D

E

F

G

H

I

S

T

U

V

W

X

JS精品图书

《基于MVC的JavaScript Web富应用开发》

Alex MacCaw 著
李晶 张散集 译
2012年05月出版
ISBN：
978-7-121-10956-0
定 价：59.00元

《JavaScript语言精髓与编程实践》

周爱民 著
2012年03月出版
ISBN：
978-7-121-15640-3
定 价：79.00元

《JavaScript语言精粹（修订版）》

Douglas Crockford 著
赵泽欣 鄢学鹃 译
2012年09月出版
ISBN：
978-7-121-17740-8
定 价：49.00元

《用AngularJS开发下一代Web应用》

Green,B. Seshadri,S. 著
大漠穷秋 译
2013年10月出版
ISBN：
978-7-121-21574-2
定 价：55.00元