Node.js开发实战

Node.js 8 the Right Way

[美] Jim R. Wilson 著

梅晴光 杜万智 陈琳 纪清华 段鹏飞 译

华中科技大学出版社
中国 · 武汉

前言

Preface

近年来，软件开发领域发生了两大变革，Node.js 都处在变革的最前沿。

第一，异步编程技术应用越来越广泛。不论是等待大数据计算任务的完成，与客户端交互，操控无人机，还是响应 API 请求，你都会用到异步编程技术。

第二，JavaScript 运行环境已经成为通用的代码执行环境，它无处不在。浏览器、NoSQL 数据库、机器人、服务器中都能运行 JavaScript。

Node.js 已成为这两大变革不可或缺的组成部分，并且发挥了巨大作用。

为什么写这本书

Why Node.js the Right Way

让我们把时间拨回到 2010 年 3 月，我在波士顿 NoSQL 会议上做了一个主题为“全栈 JavaScript”的小分享。那时我就意识到可以使用 JavaScript 实现系统架构中的所有技术栈，同时大幅降低系统的复杂度。

如果系统中所有部分都是使用 JavaScript 实现的，那么你将很容易解决系统匹配问题，轻松实现代码复用。Node.js 将成为连接前端用户界面和数据存储层的重要一环。

本书内容既包含了 Node.js 入门知识，又涵盖了 Node.js 程序开发的深入实践。

学习 Node.js

像所有快速兴起的技术一样，Node.js 也有大量学习资源。遗憾的是，其中大部分都只是针对 web 应用的。

web 应用有非常广泛的影响，但它不是 Node.js 的全部。Node.js 不局限于提供 web 服务，本书将会介绍它在更多领域的应用。

无论你面对什么类型的需求，本书都会提供你所需要的知识，让你成为高效的 Node.js 程序员。

使用 Node.js

我喜欢 JavaScript，它有多种实现方式。开发者有很大的空间可以探索和实验，找到最佳的实现方式。

Node.js 开发社区、Node.js 开发规范，甚至 JavaScript 语言本身都在快速发展，本书的示例代码和开发建议都是基于当前的最佳实践，同时也充分考虑到未来可能发生的变化。

本书内容
What's in This Book

本书针对的读者是想学习 Node.js 异步编程的中级开发者和高级开发者。阅读本书之前最好掌握一些 JavaScript 的知识，但不必精通。全书分为三个部分，在此先简单介绍这三部分的内容。

第一部分：开始接触 Node.js

第一部分（第 1 章至第 4 章）讲解 Node.js 的基础知识。你会学着写一些 Node.js 代码，使用原生的核心模块和外部模块实现一些简单功能，例如：与文件系统进行交互、启动子进程、管理网络连接，等等。

第 1 章介绍 Node.js 中的事件循环以及为什么 Node.js 能够在单线程的前提下支持高并发场景。本章还简要介绍了在后续章节中会提及的 Node.js 开发中的五个方面，以及如何安装 Node.js。

在第 2 章你会写更多的 Node.js 代码。如果你之前开发过服务端程序，那么应该接触过文件的读/写。我们将从这个常见需求出发，使用 Node.js 的文件模块创建异步的、非阻塞的文件处理工具。我们也会使用 Node.js 中非常常用的 `EventEmitter` 和 `Stream` 类处理数据，还会创建子进程并与之进行通信。

第 3 章详细阐述了 Node.js 的网络 I/O 开发。我们会开发 TCP 服务器，并创建一个与服务器通信的客户端。还会开发一个自定义类来发送消息，它的消息遵循基于 JSON 格式的消息通信协议。我们会使用 Node.js 中非常流行的测试框架 Mocha 开发单元测试。

第 4 章将把注意力转向第三方框架。你将会学习使用 npm 引入高效率低延迟的网络应用开发库 ØMQ（读作“Zero-M-Q”）。你将会学习使用 ØMQ 开发基于发布-订阅模式和请求-应答模式的网络应用，创建一系列互相配合的程序，还会学习进程管理工具。

第二部分：数据处理

第二部分（第 5 章、第 6 章）学习如何操作数据，为端到端的应用打好基础。先从处理数据文件开始，使用易于测试的方式处理数据。还会学习使用 Node.js 开发命令行工具，以及如何与 HTTP 服务进行交互。

第 5 章将启动一个贯穿第二部分和第三部分的项目。从一个叫 Project Gutenberg 的电子书在线服务下载代码。使用 Cheerio 模块解析数据文件和提取重要字段。使用 npm、Mocha，以及 Chai 断言库来搭建集成测试环境，还会学习使用 Chrome 浏览器的 DevTools 开展交互式的调试。

第 6 章学习将 Project Gutenberg 数据导入 Elasticsearch 索引。你将会使用到 Commander 模块开发一个叫 `esclu` 的命令行工具来完成数据的导入。由于 Elasticsearch 是基于 JSON 的 RESTful 数据存储，所以你将会使用 Request 模块与它进行交互，还会使用 `jq` 这个非常强大的命令行工具来处理 JSON。

第三部分：从头开始创建应用程序

第三部分（第 7 章至第 9 章）把之前学习的知识综合起来开发一个 web 服务，这个服务处理从 API 调用到后端数据服务之间的逻辑。终端用户不直接发起 API 请求，所以需要实现一个美观的用户界面。最后实现 session 管理和认证，使用它

把界面和 web 服务结合起来。

Node.js 完美支持 HTTP 服务的开发，第 7 章讲解这方面的知识。使用 Express 做路由（Express 是一个非常流行的 Node.js web 框架）。深入阐述 REST 语义，并讲解如何使用 Promise 和 async 函数组织异步代码。最后学习使用 `nconf` 模块配置服务，使用 `nodemon` 模块维持服务运行。

第 8 章学习前端界面的开发。使用 webpack 将前端项目打包（webpack 是基于 Node.js 的构建工具）。使用 TypeScript 将代码转换成可以在浏览器执行的代码。TypeScript 是微软开发的一种语言，具有类型检查的特性，它提供的转译器可以将 TypeScript 转译成 JavaScript。

第 9 章把用户界面和 web 服务连接起来，形成一种端到端的解决方案，使用 Express 中间件实现身份认证 API 和有状态的 session。学习使用 npm 的 `shrinkwrap` 选项保证系统不会受依赖模块更新的影响。

第 10 章讲解 Node-RED 智能可视化编辑器的用法，在 Raspbian 树莓派操作系统上设计和开发基于事件的应用程序。学习使用 Node-RED 快速开发 HTTP API。

附录 A 和附录 B 分别介绍 Angular 开发环境和 React 开发环境的设置，并学习通过 webpack 把它们跟 Express 整合起来。

本书不包含的内容
What This Book Is Not

开始读本书之前，你应该知道哪些内容不会出现在本书里。

Node.js 百科全书

现在 npm 仓库里有超过 528000 个模块，而且以每天 500 个的速度在增加。Node.js 社区非常大，并且在高速增长，本书不可能包含所有这些内容。

书中提到的 Node.js 与非 Node.js 服务之间的调用非常复杂，我们的示例程序会涉及处理多个系统和用户之间的互相调用，涉及许多前后端技术。我们会简要介绍这些技术，让大家了解其全貌，但不会做太深入的探讨。

MEAN

本书不会专门介绍特定的技术栈，例如 MEAN[1]（Mongo、Express、Angular、Node.js）。我们把注意力放在 Node.js 的学习和使用上。

我选择 Elasticsearch 而不选择 MongoDB 做数据库。据 RisingStack 在 2016 年的一份调查报告显示，Elasticsearch 在资深 Node.js 开发者中越来越流行[2]。

本书也不讲解前端框架。目前最流行的两个前端框架是 Facebook 的 React[3]和 Google 的 Angular[4]。大家自己可以多关注这两个框架。

JavaScript 初学者指南

JavaScript 语言可能是当今最被误解的语言。虽然我们偶尔会讨论 JavaScript 语法（尤其是 ES6+语法），但本书不是 JavaScript 初学者指南。你需要掌握 JavaScript 的基本语法，比如你应能读懂下面的代码，能够理解其中的逻辑。

```
const list = [];
for (let i = 1; i <= 100; i++) {
  if (!(i % 15)) {
    list.push('FizzBuzz');
  } else if (!(i % 5)) {
    list.push('Buzz');
  } else if (!(i % 3)) {
    list.push('Fizz');
  } else {
    list.push(i);
  }
}
```

细心的读者会发现，这是 Jeff Atwood 在 2007 年提出的经典编程题 FizzBuzz[5]的答案。下面是另一个版本的答案，其中使用了更高级的 JavaScript 新特性。

```
'use strict';
const list = [...Array(100).keys()]
  .map(n => n + 1)
  .map(n => n % 15 ? n : 'FizzBuzz')
  .map(n => isNaN(n) || n % 5 ? n : 'Buzz')
  .map(n => isNaN(n) || n % 3 ? n : 'Fizz');
```

---

[1] http://www.modulecounts.com/
[2] https://blog.risingstack.com/node-js-developer-survey-results-2016/
[3] https://facebook.github.io/react/
[4] https://angularjs.org/
[5] https://blog.codinghorror.com/why-cant-programmers-program/

如果你看不懂上面这段代码也没关系，本书会讲解如何使用这些新特性。

给 Windows 用户的小提示

本书的示例代码是针对 Unix 操作系统编写的，我们使用标准输入/输出流，标准的数据传输方式，shell 脚本都在 Bash 中测试通过，也能在其他 shell 命令行工具运行。

如果你使用 Windows 操作系统，我建议你安装 Cygwin[6]，或者运行 Linux 虚拟机，这样可以尽量保证示例代码正常运行。

示例代码和格式约定
Code Examples and Conventions

本书包含 JavaScript、shell 脚本和 HTML/XML 代码，大部分情况下，示例代码都可以在运行环境下直接执行。Shell 命令行语句以`$`开头。

本书会讲解如何捕获异常，开发 Node.js 程序应该养成捕获异常的习惯，哪怕只是简单地捕获之后重新抛出，也应该把捕获逻辑写上。但是，为了提高代码的可读性和节省版面，书中有些示例代码没有写捕获异常的逻辑。请务必自己养成处理异常逻辑的习惯。

在线资源
Online Resources

Pragmatic Bookshelf 网站[7]上有本书的相关资源，你可以下载所有示例代码，同时可以在上面找到在线论坛和勘误表。

最后，感谢你选择本书开启 Node.js 开发之路。

Jim R. Wilson
2017 年 12 月

[6] http://cygwin.com/
[7] http://pragprog.com/book/jwnode2/node-js-8-the-right-way

目录

Contents

第一部分

开始接触 Node.js

Getting Up to Speed on Node.js

> Node.js 是强大的 JavaScript 服务端开发平台。
>
> 第一部分将从简单的命令行程序讲起，然后逐渐深入微服务开发。在此过程中，你将学习如何把代码导出为模块，如何使用 npm 中的第三方模块，以及如何使用最新的 ECMAScript 语言特性。

第 1 章

入门

Getting Started

有一句编程名言：功能是资产，代码是负债[1]。

在学习本书的 Node.js 程序时，请牢记这句名言，避免不必要的代码，尽量使用已有的成果。

不过，本书会教你自己实现一些功能，而不管这些功能是否已经被其他人实现过。因为只有自己动手做过，并且了解其中的原理，才能更好地利用它。

我们会循序渐进地学习。为了熟悉开发环境、语言特性、基础 API，我们会使用最基础的方式进行开发。等你打好基础后，再使用第三方模块、库、服务来代替之前写的代码。

你会发现使用第三方库很方便，而自己开发实现这些功能也是很有意义的事情。一旦你认识到这一点，就说明你已经成功脱离了初学者的行列。

1.1 不限于 Web

Thinking Beyond the web

围绕 Node.js 的讨论大多局限于 web，人们常常忽略了它在其他领域的作用。我们使用图 1.1 来展示 Node.js 的应用范围。

[1] http://c2.com/cgi/wiki?SoftwareAsLiability

把所有计算机程序想象成栖息在大海里的生物，功能相似的程序栖息在相近的地方，功能不同的程序则相隔较远。图 1.1 是程序大海里的一个小岛，叫 I/O 密集之岛。

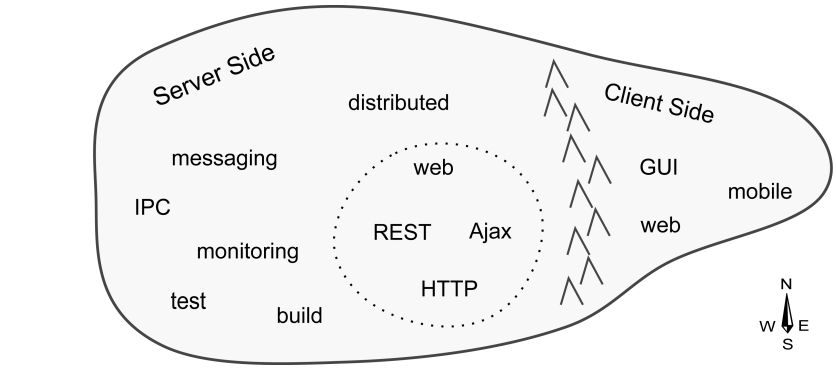

图 1.1 I/O 密集之岛

客户端程序在岛的东侧，包括各种 GUI 工具、购物应用程序、手机端应用程序和各种 web 应用程序。客户端程序直接与用户进行交互，它们通常需要等待用户输入。

服务端程序在岛的西侧，这片广袤的地区都是 Node.js 的领域。

在服务端领域深处，有一片叫 web 的区域，其中包含传统的 HTTP、Ajax、REST。那些吸引了大家注意力的网站、应用程序和 API 栖息在这里。

现在的情况是人们过分强调 Node.js 在 web 开发领域中的作用，而实际上 Node.js 的应用范围比这广得多，本书会带你慢慢探索。

1.2 Node.js 的应用范围
Node.js's Niche

自从 JavaScript 1995 年发明以来，它一直被用于解决前端到后端的各种问题。图 1.2 展示了 Node.js 的应用范围。

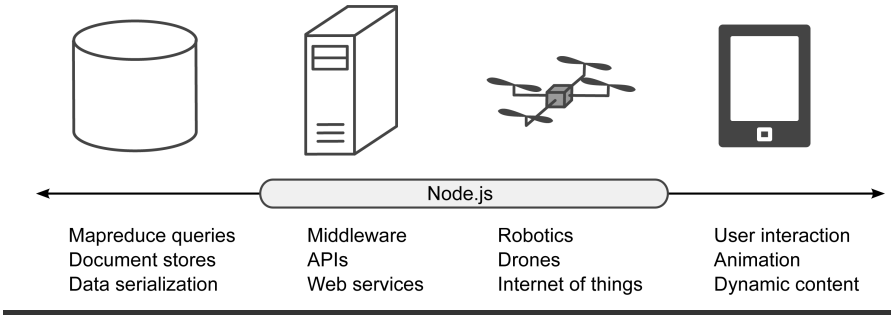

图 1.2 Node.js 的应用范围

在图 1.2 的右侧，浏览器运行的大量脚本都与用户交互有关，如点击、拖曳、选择文件等。JavaScript 在这些方面的应用已经非常成功。

在图 1.2 的左侧，后端数据库也大量用到 JavaScript。面向文档的数据库（如 MongoDB 和 CouchDB），从数据修改到 ad-hoc 查询，以及 mapreduce 任务，都大量用到 JavaScript。像 Elasticsearch 和 Neo4j 这样的 NoSQL 数据库也使用 JavaScript 对象标记语言（JSON）来展示数据。现在甚至可以使用 JavaScript 插件给 Postgres 写 SQL 函数。

大量中间件任务也像客户端脚本和数据库一样，属于 I/O 密集型应用。服务端程序往往要等待某些事情，比如数据库查询结果、第三方 web 服务的响应、网络连接请求等。Node.js 正是为了解决这些问题而生的。

Node.js 也进入了自治系统领域，用于开发物联网平台，例如树莓派的操作系统 Raspbian[1]，以及基于 Node.js 构建的 Tesse[2]。还有 Johnny-Five 和 CylonJS，这是两个机器人开发平台，可帮助你为各种硬件组件开发 Node.js 应用程序[3,4]。

机器人技术和物联网应用往往依赖于特定的硬件环境，因此它们不在本书的讨论范围之中。但是，如果你将来决定转向这些领域，本书介绍的 Node.js 开发技巧也能派上用场。

[1] https://www.raspberrypi.org/downloads/raspbian/
[2] https://tessel.io/
[3] http://johnny-five.io/
[4] https://cylonjs.com/

1.3 Node.js 的工作原理
How Node.js Applications Work

Node.js 通过事件循环机制快速分发处理事件，这是 Node.js 最核心的特性。

Node.js 的理念是给用户提供事件和操作系统资源的底层访问权限，用 Node.js 核心贡献者 Felix Geisendörfe 的话说，在 Node.js 中“除了你的代码，一切都是并行的”。[1]

我们可以通过图 1.3 理解事件循环的工作原理。

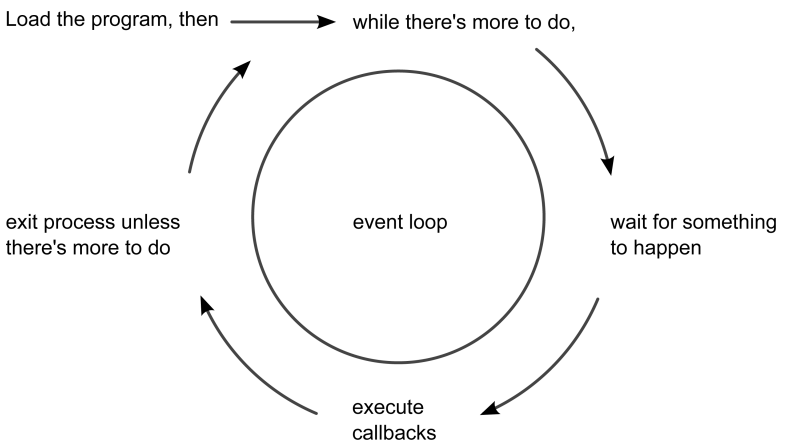

图 1.3 Node.js 事件循环

Node.js 线程会持续进行事件循环，直到所有任务都完成后才退出。当有事件发生时，Node.js 会触发相应的回调函数（事件处理器）。

Node.js 开发者的工作本质上就是编写事件处理回调函数，多个事件会多次触发回调函数，但同一时刻只有一个回调函数被执行。

应用程序所做的大多数事情，在 Node.js 里都是异步处理的，例如读取数据文件，处理 HTTP 请求等。一旦它开始执行，就会占据 JavaScript 引擎的整个执行线程，绝不会跟另一个应用程序同时执行。

[1] http://www.debuggable.com/posts/understanding-node-js:4bd98440-45e4-4a9a-8ef7-0f7ecbdd56cb

1.3.1 单线程与高并发
Single-Threaded and Highly Parallel

在实现并发的方式上，Node.js 与众不同。大部分并发方案都是利用多线程同时运行多份代码，但 Node.js 是单线程的，任何时刻都只有一份代码在执行。

Node.js 采用非阻塞方式处理大部分 I/O 任务，发生 I/O 成功或失败情况时会触发相应的回调函数，而不是阻塞在那里等待 I/O 操作完成。

代码块执行完成后，控制权交还给事件循环，Node.js 会利用空闲时间执行其他任务。第 2 章将举例说明 Node.js 的并发和事件循环原理。

Node.js 在任何时刻只执行一份代码，但能处理并发请求，看起来似乎没有道理。我把这称为 Node.js 的反向主义。

1.3.2 Node.js 的反向主义
Backwardisms in Node.js

反向主义的意思是，事情朝着表面方向相反的方向运行。编程过程中，大家已经接触了很多反向主义的例子，只是你可能没有留意。

以变量的概念为例，`7x + 3 = 24` 在代数中很常见。其中 `x` 是变量，它有一个确定的值，只要解开方程式，就能得到 `x` 的值。

软件编程有 `x = x + 7` 这样的表达式。这里的 `x` 也叫变量，但你可以赋予它不同的值，并且每次执行后 `x` 的值都可能不一样。

从代数的角度看编程中的表达式就是典型的反向主义，`x = x + 7` 这个方程式是毫无意义的。软件中的变量和代数中的变量是两个完全相反的概念，这就是反向主义。学习了编程中的赋值，你就不难理解 `x = x + 7` 中变量的概念了。

Node.js 的事件循环也是这样，从多线程的角度看，同一时刻只执行一份代码是很愚蠢的做法。如果你理解了基于非阻塞 API 的事件驱动编程，那么你就不难理解事件循环的做法了。

软件世界中有许多这样的反向主义，Node.js 就是一个例子。本书中有很多这样的代码，它们看起来应该这样运行，但实际上并非如此。

编写过一些简单的 Node.js 程序后，你会更容易理解 Node.js 的反向主义。

1.4 Node.js 开发的 5 个方面
Aspects of Node.js Development

Node.js 开发是一个很大的话题，甚至可以根据不同的 JavaScript 版本划分内容。本书主要关注以下 5 个方面。

- 开发实战。
- Node.js 核心。
- 开发模式。
- JavaScript 语言特性。
- 支持代码。

下面逐一进行介绍。

1.4.1 开发实战
Practical Programming

开发实战指的是实用的编程技巧，如读/写文件、创建 socket 连接、提供 web 服务。

本书后续每一章都会选择一个领域进行实战开发。虽然这些例子是限定在某个领域的，但同时会介绍 Node.js 核心、开发模式、JavaScript 语言特性和支持代码。

1.4.2 Node.js 核心
Node.js Core

学习 Node.js 核心模块有助于理解 Node.js 代码的执行特点，也能有效避免错误。例如，Node.js 中的事件循环处理逻辑是使用 C 语言编写的，但 Node.js 是在 JavaScript 环境下运行的。后面你会理解如何在这两种语言之间传递消息。

1.4.3 开发模式
Patterns

Node.js 拥有非常成功的开发模式。其中一些模式应用于 Node.js 核心代码内

部，还有一些模式大量出现在第三方 Node.js 库中。比如回调函数、错误处理和在事件驱动编程中广泛应用的 `Emitter` 和 `Stream`。

在不同领域进行开发的过程中，我们会发现很多类似的开发模式。你会逐步了解为什么要按这些模式进行开发，以及如何有效地使用这些模式。

1.4.4 JavaScript 语言特性
JavaScriptisms

本书会尽量使用 Node.js 支持最新的 JavaScript 语言特性，即使你以前有过 JavaScript 开发经验，也会发现示例代码中可能有你读不懂的新特性。书中会用到如箭头函数、延展参数、解构赋值等最新的 JavaScript 语言特性。

1.4.5 支持代码
Supporting Code

任何程序都不是独立运行的，它们依赖大量的支持代码。从单元测试到部署脚本，都需要额外的辅助程序来支撑。本书中的支持代码可以让程序更健壮，易于扩展，易于维护。

完成以上 5 个方面的学习后，你就能使用 Node.js 平台的大部分功能开发出符合规范的 Node.js 应用。此外，示例代码在本书的主要作用是清晰地阐释 Node.js 中的概念，它们往往都比较功能化且简短。

运行示例代码之前请务必安装 Node.js 运行环境。

1.5 安装 Node.js
Installing Node.js

请根据操作系统来选择 Node.js 安装包，如果你喜欢自己动手，也可以选择源码安装。

本书代码要求使用 Node.js 8 的稳定版本。如果你安装的是其他版本，例如，根据最新源码安装的版本，那可能会导致示例代码运行失败。在命令行中，执行 `node-version` 可以查看当前安装的 Node.js 版本：

```
$ node-version
v8.0.0
```

最简单的安装方法是从 nodejs.org[1]官网下载安装包。

另一种常见的选择是使用 Node.js 版本管理器（nvm）[2]。如果你的操作系统是类 Unix 系统（如 Mac OS X 或 Linux），可以这样安装：

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.8/install.sh
| bash
```

安装完 nvm 后再安装指定版本的 Node.js

```
$ nvm install v8.0.0
```

如果安装过程中遇到问题，则可以通过 Node.js 的维护者邮件列表或者 IRC 频道寻求帮助[3]。

准备好了吗？接下来进入大家熟悉的领域：文件操作。

[1] http://nodejs.org/download/
[2] https://github.com/creationix/nvm
[3] http://nodejs.org/community/

第 2 章

文件操作

Wrangling the File System

你在工作中肯定会遇到文件操作，比如，读文件、写文件、文件重命名、删除文件等。我们就以文件操作作为学习 Node.js 的起点，接下来会创建一些实用的异步操作文件的工具，在这个过程中，你会接触到以下几个方面的内容。

Node.js 核心

从架构层面讲解事件循环的原理，以及它是如何参与到程序的运行流程中的。我们将学习 Node.js 的 JavaScript 引擎和底层原生模块之间如何通过 `Buffer` 传输数据，还会学习如何用 Node.js 的模块系统在代码中引入核心模块。

开发模式

使用 Node.js 的常用开发模式进行开发，比如，使用回调函数处理异步事件。还会用到 `EventEmitter` 和 `Stream` 这两个工具类进行数据传输。

JavaScript 语言特性

学习 JavaScript 语言的一些特性和最佳实践，如块级作用域和箭头函数。

支持代码

学习如何创建子进程，如何在子进程之间通信，如何获取子进程的输出结果，以及如何探测子进程的状态变化等。

我们最终会开发一个工具来监听文件内容的变化，在此过程中，你不但会熟悉

Node.js 文件系统 API，还会对事件循环的原理有更深入的理解。

2.1 Node.js 事件循环编程
Programming for the Node.js Event Loop

从文件操作程序入手，这个程序的功能是从命令行读取参数并监听文件变化。虽然代码很简单，但它可以让我们了解 Node.js 基于事件的编程架构。

2.1.1 监听文件变化
Watching a File for Changes

讲解 Node.js 概念时，通常会以监听文件为例，因为它会用到异步编程的思想，并且有非常多的应用场景，比如自动部署和自动执行单元测试。

打开命令行终端，创建一个目录，命名为 `filesystem`，然后进入这个目录。

```
$ mkdir filesystem
$ cd filesystem
```

本章所有示例代码都会在这个文件夹中运行。使用 `touch` 命令新建 `target.txt` 文件。

```
$ touch target.txt
```

如果你的系统没有 `touch` 命令（比如 Windows），那么可以换成 `echo` 命令。

```
$ echo > target.txt
```

接下来试试这个监听文件。打开你常用的编辑器，输入如下代码：

filesystem/watcher.js

```
'use strict';
const fs = require('fs');
fs.watch('target.txt', () => console.log('File changed!'));
console.log('Now watching target.txt for changes...');
```

将以上代码保存为 `watcher.js` 文件并和刚才的 `target.txt` 文件放在同一级目录下。别看这短短几行代码，它使用了很多 JavaScript 和 Node.js 特性。下面来一行一行地学习。

第一行的 `'use strict'` 表示让代码在严格模式下运行。严格模式是

ECMAScript 5 的新特性。在这种模式下，一些不合理、不严谨的行为将被禁止，并且会抛出错误。开启严格模式是一个很好的习惯，本书代码都会在严格模式下运行。

注意看 `const` 关键词，它声明 `fs` 为常量。使用 `const` 声明的常量必须在声明时赋值，而且禁止任何形式的再次赋值（会触发运行时报错）。

你可能想问为什么不能重新赋值。在很多场景中，变量的值是不需要变化的，这时候使用 `const` 赋值就是很好的选择。还有一种代替 `const` 的声明方式是 `let`，后面会介绍。

`require()`函数用于引入 Node.js 模块并且把这个模块作为返回值。在本例中，执行 `require('fs')`是为了引入 Node.js 内置的文件模块[1]。

在Node.js中，模块是一段独立的JavaScript代码，它提供的功能可以用于其他地方。`require()`的返回值通常是 JavaScript 对象或函数。模块还可以依赖别的模块，类似于其他编程语言中库的概念，其他编程语言中的库也可以是 `import` 或 `#include` 其他库。

然后是调用 `fs` 模块的 `watch()`方法，这个方法接收两个参数，一个是文件路径，另一个是当文件变化时需要执行的回调函数。在 JavaScript 中，函数是一等公民，也就是说，函数可以被赋值给变量，或者作为参数传递给别的参数。现在仔细看看这个回调函数：

```
() => console.log('File changed!')
```

这是箭头函数表达式，也称胖箭头函数或简称为箭头函数。开头空括号的意思是这个函数不需要任何参数。函数体中使用 `console.log` 把一段消息输出到标准输出。

箭头函数是ECMAScript 2015的新特性，本书会大量用到它。在我介绍箭头函数之前，你一定用过另一种更冗长的写法 `function(){}`。

```
function() {
    console.log('File changed!');
}
```

除了简洁的语法，箭头函数还有更大的优点：不会创建新的 `this` 作用域。一直以来，如何正确理解 `this` 作用域都是 JavaScript 开发者心中的痛，有了箭头函

[1] http://nodejs.org/api/fs.html

数，this 作用域的问题变得简单很多。就像 const 关键词是变量声明的第一选择，大家也应该把箭头函数作为函数声明的第一选择（例如回调函数）。

最后一行代码只是简单地输出一行文字，告诉调用者一切准备就绪。

现在在命令行试着运行，使用 node 启动这个监听程序：

```
$ node watcher.js
Now watching target.txt for changes...
```

程序启动之后，Node.js 会安静地等待目标文件内容的变化。打开另一个命令行终端窗口，进入刚才的文件夹，然后使用 touch 命令触发 target.txt 文件内容的变化。这时就能在 watcher.js 监听程序的命令行看到输出 File changed!，然后监听程序会继续等待文件内容的变化。

如果你看到几条重复的输出消息，并不是代码出了 bug，出现这种状况的原因与操作系统对文件变化的处理方式有关，这种情况主要出现在 Mac OS 和 Windows 操作系统上。

本章会常用到 touch 命令来触发文件内容的变化，下面这条语句可以用 watch 命令来实现自动地执行 touch：

```
$ watch -n 1 touch target.txt
```

这条语句每秒钟会触发一次目标文件，直到手动退出。如果操作系统不支持 watch 命令也没有关系，通过任何形式修改 target.txt 文件都可以。

2.1.2 看得见的事件循环
Visualizing the Event Loop

上一节的例子展示了 Node.js 事件循环的工作。正如图 1.3 所示，我们的文件监听程序按照图中的流程一步一步地运行着。

Node.js 按照如下方式运行：

- 加载代码，从开始执行到最后一行，在命令行输出 Now watching 消息。
- 由于调用了 fs.watch，所以 Node.js 不会退出。
- 它等待着 fs 模块监听目标文件的变化。
- 当目标文件发生变化时，执行回调函数。

- 程序继续等待，继续监听，还不能退出。

事件循环会一直持续下去，直到没有任何代码需要执行、没有任何事件需要等待，或者程序由于其他因素退出。比如程序运行时发生错误抛出异常，而异常又没有被正确捕获到，通常会导致进程退出。

2.1.3 接收命令行参数
Reading Command-Line Arguments

接下来改进我们的监听程序，让它能够接收参数，在参数中指定我们要监听哪个文件。你会在这段程序中用到 process 全局对象，还会学到如何捕获异常。

打开编辑器，输入如下代码：

filesystem/watcher-argv.js
```
const fs = require('fs');
const filename = process.argv[2];
if (!filename) {
  throw Error('A file to watch must be specified!');
}
fs.watch(filename, () => console.log(`File ${filename} changed!`));
console.log(`Now watching ${filename} for changes...`);
```

保存并命名为 watcher-argv.js，然后按如下方式运行：

```
$ node watcher-argv.js target.txt
Now watching target.txt for changes...
```

输出的内容与之前的 watcher.js 一模一样，在输出 *Now watching target.txt for changes...*之后，也开始等待目标文件内容发生变化。

通过 process.argv 访问命令行输入的参数。argv 是 *argument vector* 的简写，它的值是数组，其中数组的前两项分别是 node 和 watcher-argv.js 的绝对路径，数组的第三项（下标为 2）就是目标文件的文件名 target.txt。

注意输出信息是由反引号（`）包裹起来的字符串，称为模板字符串：

```
`File ${filename} changed!`
```

模板字符串支持多行文本，也支持插值表达式。利用插值表达式可以在${}占位符内写一个表达式，最终会用表达式的字符串值替换模板字符串中的占位符。

如果没有提供目标文件名参数，那么 `watcher-argv.js` 会抛出异常。把刚才那条命令最后的参数 `target.txt` 去掉，再运行一次就能看到如下错误信息：

```
$ node watcher-argv.js
/full/path/to/script/watcher-argv.js:4
throw Error('A file to watch must be specified!');
^

Error: A file to watch must be specified!
```

所有未捕获的异常都会导致 Node.js 执行进程退出。错误信息一般包含抛错的文件名、抛错的行数和具体的错误位置。

进程是 Node.js 中非常重要的概念，在开发中最常见的做法是把不同的工作放在不同的独立进程中执行，而不是所有代码都塞进一个巨无霸 Node.js 程序里。下一节学习如何在 Node.js 中创建进程。

2.2 创建子进程
Spawning a Child Process

下面继续优化监听程序，让它在监听到文件变化后创建一个子进程，再用这个子进程执行系统命令。在此过程中，我们会接触到 `child-process` 模块、Node.js 的开发模式和一些内置类，还会学习如何用流进行数据传送。

为了方便，我们的代码会执行 `ls` 命令并加上 `-l` 和 `-h` 参数，这样就能看到目标文件的修改时间。同样也可以用这种方法执行其他命令。

打开编辑器，并输入如下代码：

filesystem/watcher-spawn.js

```
'use strict';
const fs = require('fs');
const spawn = require('child_process').spawn;
const filename = process.argv[2];

if (!filename) {
  throw Error('A file to watch must be specified!');
}

fs.watch(filename, () => {
  const ls = spawn('ls', ['-l', '-h', filename]);
  ls.stdout.pipe(process.stdout);
});
console.log(`Now watching ${filename} for changes...`);
```

将文件保存为 watcher-spawn.js，然后用之前的方式运行它：

```
$ node watcher-spawn.js target.txt
Now watching target.txt for changes...
```

然后打开另外的命令行窗口，并且 touch 目标文件，监听程序将会输出类似这样的信息：

```
-rw-rw-r— 1 jimbo jimbo 6 Dec 8 05:19 target.txt
```

注意你自己运行的输出结果会跟上面的不太一样，比如用户名、用户组、文件属性会不同，但格式是一样的。

代码的开始部分有新的 require()语句，require('child_process')语句将返回 child process 模块。[1]目前我们只关心其中的 spawn()方法，所以把 spawn()方法赋值给一个常量且暂时忽略模块中的其他功能。

```
const spawn = require('child_process').spawn;
```

记住，函数在 JavaScript 中是一等公民，可以直接赋值给另一个变量。

接下来看看传给 fs.watch()的回调函数：

```
() => {
  const ls = spawn('ls', ['-l', '-h', filename]);
  ls.stdout.pipe(process.stdout);
}
```

与之前的示例不同，这个箭头函数的函数体不只一行，因此，需要用大括号{}包裹起来。

spawn()的第一个参数是需要执行命令的名称，在本例中就是 ls。第二个参数是命令行的参数数组，包括 ls 命令本身的参数和目标文件名。

spawn()返回的对象是 ChildProcess。它的 stdin、stdout、stderr 属性都是 Stream，可以用作输入和输出。使用 pipe()方法把子进程的输出内容直接传送到标准输出流。

有些场景下，我们需要读取输出的数据而不是直接传送，那该怎么做呢？

[1] https://nodejs.org/api/child_process.htm

2.3 使用 EventEmitter 获取数据
Capturing Data from an EventEmitter

EventEmitter 是 Node.js 中非常重要的一个类，可以通过它触发事件或者响应事件。Node.js 中的很多对象都继承自 EventEmitter，例如上一节提到的 Stream 类。

现在修改刚才的例子，通过监听 stream 的事件来获取子进程的输出内容。在编辑器中打开上一节的 watcher-spawn.js 文件，找到 fs.watch()语句，替换为如下代码：

filesystem/watcher-spawn-parse.js
```
fs.watch(filename, () => {
  const ls = spawn('ls', ['-l', '-h', filename]);
  let output = '';

  ls.stdout.on('data', chunk => output += chunk);

  ls.on('close', () => {
    const parts = output.split(/\s+/);
    console.log([parts[0], parts[4], parts[8]]);
  });
});
```

把修改后的代码保存为新文件 watcher-spawn-parse.js，然后像之前一样运行，再打开新命令行窗口，使用 touch 修改目标文件。你会看到如下输出：

```
$ node watcher-spawn-parse.js target.txt
Now watching target.txt for changes...
[ '-rw-rw-r—', '0', 'target.txt' ]
```

这个新的回调函数会像之前一样被调用，它会创建一个子进程并把子进程赋值给 ls 变量。函数内也会声明 output 变量，用于把子进程输出的内容暂存起来。

注意 output 变量是用 let 关键词声明的，let 和 const 都可以用于声明变量，但它声明的变量能够被多次赋值。通常我们会选 const 关键词声明变量，除非明确知道这个变量的值在运行时会修改。

可以用 var 声明变量吗？

在引入 const 和 let 之前，在 JavaScript 中都是用 var 关键词声明变量的。var 和 let 类似，不同之处在于它声明的变量具有函数级作用域，而不是像 let 一样的块级作用域。

这里有一个例子能说明为什么我们应该用 const 和 let 代替 var:

```
if (true) {
  var myVar = "hello";
  let myLet = "world";
}

console.log(myVar); // Logs "hello".
console.log(myLet); // throws ReferenceError
```

myVar 变量在 if 语句之外也能够访问，看起来有点费解。这一奇怪的现象在 JavaScript 中叫变量提升。var 语句声明的变量，会被提升到所在的函数或者模块作用域顶部。

我的建议是，如果想要这个变量在整个函数或模块内有效，那就应该在函数内直接用 const 或者 let 关键词声明。

接下来添加事件监听函数。当特定类型的事件发生时，这个监听函数就会被调用。Stream 类继承自 EventEmitter，所以能够监听到子进程标准输出流的事件：

```
ls.stdout.on('data', chunk => output += chunk);
```

这行代码的信息量比较大，我们拆开来看。

这里的箭头函数接收一个 chunk 参数，当箭头函数只需要一个参数时，可以省去参数两端的括号。

on()方法用于给指定事件添加事件监听函数，本例中监听的是 data 事件，因为我们要获得输出流的数据。

事件发生后，可以通过回调函数的参数获取跟事件相关的信息，比如本例的 data 事件会将 Buffer 对象作为参数传给回调函数，然后每拿到一部分数据，我们就把这个参数里的数据添加到 output 变量。

Node.js 中使用 Buffer 描述二进制数据[1]。它指向一段内存中的数据，这个数据由 Node.js 内核管理，而不在 JavaScript 引擎中。Buffer 不能修改，并且需要编码和解码的过程才能转换成 JavaScript 字符串。

在 JavaScript 里，把非 string 值添加到 string 中（像上例中的 chunk 那样），都会隐式调用对象的 toString()方法。具体到 Buffer 对象，当它跟一个 string 相加时，会把这个二进制数据复制到 Node.js 堆栈中，然后使用默认方式（UTF-8）编

[1] https://nodejs.org/api/buffer.html

码。

把数据从二进制复制到 Node.js 的操作非常耗时，所以尽管 string 操作更加便捷，但还是应该尽可能直接操作 Buffer。在这个例子中，由于数据量很小，相应的耗时也微乎其微，对整个程序影响不大。但希望大家今后在使用 Buffer 时，脑子里有这个印象，尽可能直接操作 Buffer。

ChildProcess 类也继承自 EventEmitter，也可以给它添加事件监听函数。

```
ls.on('close', () => {
  const parts = output.split(/\s+/);
  console.log([parts[0], parts[4], parts[8]]);
});
```

当子进程退出时，会触发 close 事件。回调函数将数据按空白符切割（正则表达式/\s+/），然后用 console.log 打印出第 1、5、9 个字段（下标分别为 0、4、8），这三个字段分别对应权限、大小、文件名。

本节通过文件监听程序学习了很多 Node.js 特性，包括使用 EventEmitter、Stream、ChildProcess 和 Buffer 这些内置类。也初步体验了异步编程和事件循环。

下面通过读/写文件继续学习这些 Node.js 概念。

2.4 异步读/写文件
Reading and Writing Files Asynchronously

我们在本章的前面章节开发了监听文件内容的程序，接下来学习 Node.js 的文件读/写操作。你会学习如下两种处理异常的方式：EventEmitter 的 error 事件和回调函数的 err 参数。

Node.js 有多种读/写文件的方式，其中最简单直接的是一次性读取或写入整个文件，这种方式对小文件很有效。另外的方式是通过 Stream 读/写流和使用 Buffer 存储内容。下面是一次性读/写整个文件的例子：

filesystem/read-simple.js

```
'use strict'
const fs = require('fs');
fs.readFile('target.txt', (err, data) => {
  if (err) {
    throw err;
  }
```

```
  console.log(data.toString());
});
```

保存为 read-simple.js 文件并按如下方式运行：

```
$ node read-simple.js
```

target.txt 文件的内容会输出到命令行，如果文件是空的，那么会显示一个空白行。

注意 readFile()的回调函数的第一个参数是 err，如果 readFile()执行成功，则 err 的值为 null。如果 readFile()执行失败，则 err 会是一个 Error 对象。这是 Node.js 中统一的错误处理方式，尤其是内置模块一定会按这种方式处理错误。在本例中，如果有错误，我们直接就将错误抛出，未捕获的异常会导致 Node.js 直接中断退出。

回调函数的第二个参数 data 是 Buffer 对象，就像上一节的例子展示的那样。

一次写入整个文件的做法也是类似的，如下：

filesystem/write-simple.js

```
'use strict';
const fs = require('fs');
fs.writeFile('target.txt', 'hello world', (err) => {
  if (err) {
    throw err;
  }
  console.log('File saved!');
});
```

这段代码的功能是将 *hello world* 写入 target.txt（如果这个文件不存在，则创建一个新的；如果已经存在，则覆盖它）。如果有任何因素导致写入失败，则 err 参数会包含一个 Error 实例对象。

2.4.1 创建读/写流
Creating Read and Write Streams

分别用 fs.createReadStream()和 fs.createWriteStream()来创建读/写流。如下面这段程序所示，在 cat.js 里使用文件流把数据传送到标准输出。

filesystem/cat.js

```
#!/usr/bin/env node
'use strict';
require('fs').createReadStream(process.argv[2]).pipe(process.stdout);
```

第一行代码以#!开头，因此这段程序可以在类 Unix 系统中直接运行，不必用 node 来启动它（当然也可以用 node）。

使用 chmod 命令给它赋予可执行权限。

```
$ chmod +x cat.js
```

然后直接运行，并且在后面跟上目标文件参数：

```
$ ./cat.js target.txt
hello world
```

在 cat.js 中没有将 fs 模块赋值给变量，因为 require()函数直接返回这个模块，可以直接调用这个模块的方法。

也可通过监听文件流的 data 事件来达到同样效果，如下面的 read-stream.js 所示：

filesystem/read-stream.js

```
'use strict';
require('fs').createReadStream(process.argv[2])
  .on('data', chunk => process.stdout.write(chunk))
  .on('error', err => process.stderr.write(`ERROR: ${err.message}\n`));
```

这里使用 process.stdout.write()输出数据，替换原来的 console.log。输入数据 chunk 中已经包含文件中的所有换行符，因此不再需要 console.log 来增加换行。

更方便的是，on()返回的也是 emitter 对象，因此可以直接在后面链式地添加事件处理函数。

当使用 EventEmitter 时，最方便的错误处理方式就是直接监听它的 error 事件。现在人为地触发一个错误，看看它会输出什么。执行这段代码并给它传入一个不存在的文件作为参数：

```
$ node read-stream.js no-such-file
ERROR: ENOENT: no such file or directory, open 'no-such-file'
```

由于监听了 error 事件，所以 Node.js 调用了错误监听函数（并且正常退出）。如果没有监听 error 事件，并且恰好发生了运行错误，那么 Node.js 会直接抛出这个异常，然后会导致进程异常退出，就像之前的例子介绍的那样。

2.4.2 使用同步文件操作阻塞事件循环
Blocking the Event Loop with Synchronous File Access

到目前为止，我们讨论的文件操作方法都是异步的，它们都是默默地在后台履行 I/O 职责，只有事件发生时才会调用回调函数，这是较妥当的 I/O 处理方式。

同时，`fs` 模块中的很多方法也有相应的同步版本，这些同步方法大多以**Sync* 结尾，比如 `readFileSync`。如果你以往没有异步开发的经验，那以同步的方式操作文件可能对你来说会更熟悉，但这种方式会消耗更多的资源。

当调用**Sync* 方法时，Node.js 进程会被阻塞，只到 I/O 处理完毕。也就是说，在这个时候 Node.js 不会执行其他代码，不会调用任何回调函数，不会触发任何事件，也不会建立任何网络连接。它会完全停止下来，等待 I/O 操作结束。

虽然会有阻塞的问题，但同步的方式使用起来更简单，不必关心回调函数的问题。同步方法要么执行成功要么抛出异常，不必在回调函数中处理。有些场景适合用同步方式写，在后面的章节中会有讨论。

下面是用 `readFileSync()`读文件的例子：

```
const fs = require('fs');
const data = fs.readFileSync('target.txt');
process.stdout.write(data.toString());
```

`readFileSync()`返回的是 `Buffer` 对象，这个对象跟之前 `readFile()`异步方法的回调函数接收的参数是相同的。

2.4.3 文件操作的其他方法
Performing Other File-System Operations

Node.js 的 `fs` 模块还有很多其他方法，这些方法都遵循 POSIX 标准。（POSIX[1] 是一系列用于规范操作系统之间协作性的规范，其中就包括文件系统工具。）举几个简单的例子，可以使用 `copy()`方法复制文件，可以使用 `unlink()`方法删除文件，可以使用 `chmod()`方法变更权限，可以使用 `mkdir()`方法创建文件夹。

这些函数的用法类似，它们的回调函数都接受相同的参数，就像本节介绍的

[1] http://pubs.opengroup.org/onlinepubs/9699919799/utilities/contents.html

那样。这些方法默认是异步的，同时会提供*Sync* 形式的同步版本。

2.5 Node.js 程序运行的两个阶段
The Two Phases of a Node.js Program

之前说到同步方法会阻塞 Node.js 的事件循环，你可能会认为使用这种方式操作文件不太好。了解 Node.js 运行的两个阶段后，你就会知道什么时候适合使用同步操作文件的方法了。

第一个阶段是初始化阶段，代码会做一些准备工作，导入依赖的库、读取配置参数等。如果在这个阶段发生了错误，我们没有太多办法，最好是尽早抛出错误并退出。因此在初始化阶段，可以考虑同步的文件操作。

第二个阶段是代码执行阶段，事件循环机制开始工作。相当多的 Node.js 应用是网络应用，也就是会建立连接、发送请求或者等待其他类型的 I/O 事件。在这个阶段不要使用同步的文件操作，否则会阻塞其他的事件。

`require()`函数就是上述原则的最好例子，它同步地加载执行目标模块代码，返回整个模块对象。模块的加载要么成功，要么失败导致整个程序退出。

一般来说，如果文件操作失败会导致整个程序无法运行，就可以使用同步的方式。不管文件是否操作成功，程序都可以继续运行，最好使用异步的方式。

2.6 小结与练习
Wrapping Up

本章使用 Node.js 开发了操作文件的程序，用到了 Node.js 的事件监听、异步操作、回调函数。学习了如何监听文件变化、读/写文件内容、创建子进程、获取命令行参数。

学习了 `EventEmitter` 类，使用 `on()`方法监听事件，并在回调函数中处理事件。讲解了流的概念，它是一种特殊的`EventEmitter`，用于获取`Buffer`数据或直接传输到其他流。

还讲解了异常处理。Node.js 约定回调函数的第一个参数是 `err` 参数。还可以通过 `EventEmitter` 的 `error` 事件捕获异常。

请牢记上述捕获异常的方式，也许有第三方库的风格不同，但它是 Node.js 最

常用的方式。

第 3 章将学习另一种服务端 I/O：网络连接。下面会逐步探索网络服务应用，并在本章知识的基础上继续开发。

下面是一些扩展思考题。

2.6.1 加固代码
Fortifying Code

本章的示例代码都缺少安全检查，想想如何修改代码来应对这两种情形。

- 在文件内容监听的例子中，如果文件不存在，会发生什么？
- 如果监听的目标文件被删除，又会发生什么？

2.6.2 扩展功能
Expanding Functionality

在监听程序示例中，我们是从 `process.argv` 获取监听的目标文件名。请考虑如下两个问题。

- 如何从 `process.argv` 获取需要创建的目标子进程？
- 如何从 `process.argv` 传送任意数量的参数给子进程（例如 `node watcher-spawn-cmd.js target.txt ls -l -h`）？

第 3 章

Socket 网络编程

Networking with Sockets

Node.js 天生就是用来做网络编程的，本章将讲解 Node.js 对 socket 连接的底层内置支持。TCP socket 是当今网络应用的重要部分，充分理解 socket，将有助于学习本书后面更复杂的内容。

开发基于 socket 的服务端和客户端程序的过程中，将学习以下 Node.js 知识。

Node.js 核心

第 2 章的异步编程在本章大有用处，你会学习如何扩展 `EventEmitter` 这样的 Node.js 类，还会创建自定义模块来实现代码复用。

开发模式

网络连接至少有两个节点，通常是一个服务端和一个客户端。本章会实现基于 JSON 的客户端-服务端通信。

JavaScript 语言特性

JavaScript 的继承模型非常独特，你将学习使用 Node.js 内置工具创建类结构。

辅助配套程序

为了保证代码按我们期望的方式运行，需要引入测试。本章会从 npm 安装 Mocha 测试框架，并用它开发单元测试。

我们会从最简单的 TCP 服务器开始，逐步提升代码的健壮性、模块化程度和易测性。

3.1 监听 Socket 连接
Listening for Socket Connections

网络服务通常要做两件事情：建立连接和传输信息。不论最终需要传输的是什么信息，都必须先把连接建立起来。

首先，你将学习如何在 Node.js 中搭建基于 socket 的网络服务，会开发一个示例应用，使用命令行工具连接服务端；然后服务端发送数据给客户端；最后会学习 Node.js 中客户端-服务端的经典开发模式。

3.1.1 给服务端绑定 TCP 端口
Binding a Server to a TCP Port

TCP socket 连接包含两个端点（endpoint），其中一个端点绑定到操作系统的端口，另一个端点则连接到这个端口。

有点类似电话系统，一个电话绑定了电话号码，另一个电话拨打这个号码建立通信。一旦电话接通，双方就可以互相发送信息（声音）。

在 Node.js 中，由 `net` 模块提供绑定端口和建立连接的能力，下面是绑定端口的例子：

```
'use strict'
const
  net = require('net'),
  server = net.createServer(connection => {
    // Use the connection object for data transfer.
  });
server.listen(60300);
```

`net.createServer` 方法接收一个回调函数作为参数，返回 `Server` 对象。连接建立成功后，Node.js 会执行回调函数，并传递 `Socket` 对象给 `connection` 参数，可以用这个对象接收和发送数据。

这里的回调函数是箭头函数，就像我们在第 2 章中用到的一样。

执行 `server.listen` 方法绑定指定的端口，在本例中绑定的 TCP 端口是

60300。图 3.1 展示了这个过程。

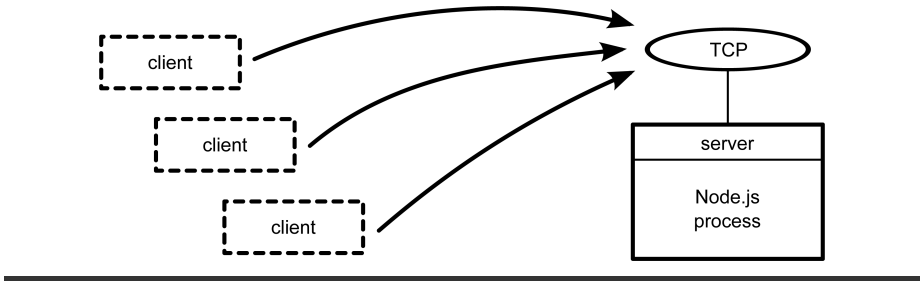

图 3.1 客户端连接 TCP 端口

Node.js 服务端进程绑定了 TCP 端口，多个客户端都可以连接到这个端口上，这些客户端可能是 Node.js 进程，也可能是其他进程。

这个服务端程序还不能做任何事，下面来添加给客户端发送数据的功能。

3.1.2 向 Socket 连接发送数据
Writing Data to a Socket

第 2 章我们开发了一些简单的文件操作程序，能够监听文件内容的变化。现在复用这段监听程序，把它作为网络服务向 socket 连接发送数据的数据源。

新建一个名称为 `networking` 的文件夹，用你常用的编辑器输入如下代码：

networking/net-watcher.js

```
'use strict';
const fs = require('fs');
const net = require('net');
const filename = process.argv[2];

if (!filename) {
  throw Error('Error: No filename specified.');
}

net.createServer(connection => {
  // Reporting.
  console.log('Subscriber connected.');
  connection.write(`Now watching "${filename}" for changes...\n`);

  // Watcher setup.
  const watcher =
    fs.watch(filename, () => connection.write(`File changed: ${new Date()}\n`));

  // Cleanup.
```

```
  connection.on('close', () => {
    console.log('Subscriber disconnected.');
    watcher.close();
  });
}).listen(60300, () => console.log('Listening for subscribers...'));
```

保存到刚才的 networking 文件夹中并命名为 net-watcher.js。在最前面两行，引入了两个 Node.js 核心模块：fs 模块和 net 模块。

监听的目标文件名通过 process.argv 的第三个（下标为 2）参数获取，但如果没有传入文件名，则将会直接抛出自定义的 Error。未捕获的错误将导致 Node.js 进程中断退出，并发送异常堆栈信息给标准错误接口。

再来看看 createServer 的回调函数，这个函数做了下面三件事情。

- 建立连接时打印通知（通过 connection.write 发送给客户端，也通过 console.log 发给控制台）。
- 开始监听目标文件内容发生变化，并把 watcher 对象保存在内存中，通过 connection.write 将变化的文件内容发送给客户端。
- 监听 connection 的 close 事件，在控制台打印通知，并用 watcher.close 停止监听文件。

最后一行，server.listen 也有一个回调函数作为参数，成功绑定到 60300 端口准备好接收客户端的连接后，Node.js 会调用这个回调函数并在控制台打印通知。

3.1.3 使用 Netcat 连接 TCP Socket 服务器

Connecting to a TCP Socket Server with Netcat

接下来运行 net-watcher 程序，看它是否按我们预期的方式运行。

我们需要三个命令行会话来运行和测试 net-watcher 程序：一个用于服务端，一个用于客户端，还有一个用来触发文件变化。

在第一个命令行会话窗口执行下面的 watch 命令，每一秒钟触发一次目标文件的变化：

```
$ watch -n 1 touch target.txt
```

在第二个命令行会话窗口执行 net-watcher 程序：

```
$ node net-watcher.js target.txt
Listening for subscribers...
```

这段程序创建了一个 TCP 服务器，并监听 60300 端口。netcat 是 socket 工具库，我们用它向服务端发起连接。打开第三个命令行窗口，执行以下 nc 命令：

```
$ nc localhost 60300
Now watching “target.txt” for changes...
File changed: Wed Dec 16 2015 05:56:14 GMT-0500(EST)
File changed: Wed Dec 16 2015 05:56:19 GMT-0500(EST)
```

如果操作系统没有 nc 命令，则可以使用 telnet：

```
$ telnet localhost 60300
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Now watching “target.txt” for changes...
File changed: Wed Dec 16 2015 05:56:14 GMT-0500(EST)
File changed: Wed Dec 16 2015 05:56:19 GMT-0500(EST)
^]
telnet> quit
Connection closed.
```

回到 net-watcher 命令行窗口，可以看到这样的信息：

```
Subscriber connected.
```

使用 Ctrl-C 关闭 nc 会话。如果是 telnet，则使用 Ctrl-]，输入 *quit* 后回车。再回到 net-watcher 的命令行窗口，将看到以下信息：

```
Subscriber disconnected.
```

使用 Ctrl-C 结束 net-watcher 程序。

图 3.2 描绘了刚才的程序。net-wathcer 进程（方形）绑定了 TCP 端口并能监听文件，图中的椭圆形表示资源。

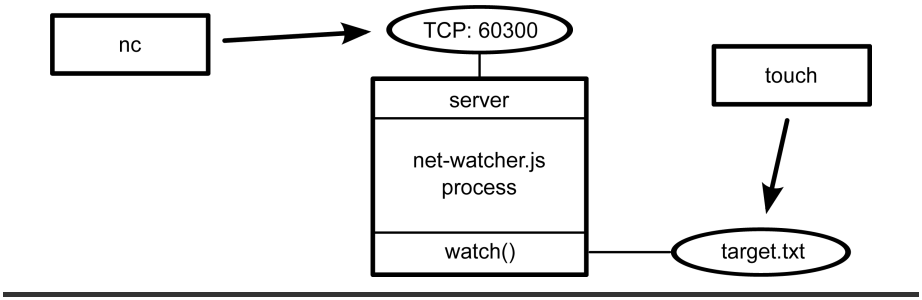

图 3.2 Socket 服务端和客户端连接

多个订阅者可以同时连接和接收信息，如果再打开一个命令行，并使用 nc 连接到 60300 端口，它将和之前的 nc 客户端一样接收文件内容更新的消息。

使用 TCP socket 在两个网络计算机之间进行通信非常方便，但如果是在同一台计算机上的两个进程之间通信，则 Unix socket 有更好的替代方案。net 模块也可以创建这样的 socket 连接，后面还会做进一步介绍。

3.1.4 监听 Unix Socket
Listening on Unix Sockets

我们将 net-watcher 程序改成 Unix socket 方式，看看 net 模块如何使用这种通信方式。要注意 Unix socket 只能在类 Unix 操作系统运行。

将 net-watcher.js 的最后一行按如下方式修改：

```
.listen('/tmp/watcher.sock', () => console.log('Listening for subscribers...'));
```

另存为 net-watcher-unix.js 文件，然后像上次一样运行：

```
$ node net-watcher-unix.js target.txt
Listening for subscribers...
```

如果发生了 EADDRINUSE 错误，则必须删除 watcher.sock 文件后再次运行。

像上次一样使用 nc 建立连接，但这次要在 socket 文件名前面加上-U 参数。

```
$ nc -U /tmp/watcher.sock
Now watching target.txt for changes...
```

由于跟网络设备没有关系，Unix socket 的速度比 TCP socket 的速度快很多，但这种方式限制在机器内部。

以上就是在 Node.js 创建 socket 服务器的内容。我们学习了如何创建 socket 服务器，如何使用 nc 连接服务器。后续章节将使用这个结构进行开发。

接下来让传输的数据格式更容易解析，就可以开发自定义的客户端了。

3.2 实现消息协议
Implementing a Messaging Protocol

上一节探索了在 Node.js 中如何创建 socket 服务器并监听连接。到目前为止，示例程序发送的消息都是纯文本，都是给人看的消息。本节我们来设计和实现一套更好的消息协议。

这里的协议是指一系列能定义系统中多个终端之间如何通信的规则。开发 Node.js 网络应用至少会用到一个协议。我们将设计一个通过 TCP 传输的 JSON 格式的协议。

本书将 JSON 用于数据序列化和配置数据。JSON 在 Node.js 开发中很常用，并且也具有较好的可读性。

我们会使用新的基于 JSON 的协议来实现服务端和客户端，这样可以为将来的测试开发和模块化开发提供便利。

3.2.1 JSON 结构的消息
Serializing Messages with JSON

现在开始开发 JSON 消息协议，每条消息都使用 JSON 格式对象表示，这些对象包含一些键值对。下面是有两个键值对的对象的例子：

```
{"key":"value","anotherKey":"anotherValue"}
```

前面开发的 net-watcher 服务会发送两种类型的消息，我们要把这两种消息转换成 JSON 格式。

- 成功建立连接后，客户端收到的消息是：*Now watching "target.txt" for changes...*
- 当目标文件发生变化时，客户端收到的消息是：*File changed: Fri Dec 18 2015 05:4400 GMT-0500 (EST)*。

第一种消息这样表示：

```
{"type":"watching","file":"target.txt"}
```

这里的 `type` 字段表示这个消息的类型是 `watching`，`file` 字段表示当前正在被监听的目标文件。

第二种消息这样表示：

```
{"type":"changed","timestamp":1358175733785}
```

这里 `type` 字段的值表示文件内容发生了变化，`timestamp` 字段的值是整数，这个数是从 1970 年 1 月 1 日零点开始计算的毫秒数。这种格式的时间在 JavaScript 中非常好用，比如，可以使用 `Date.now()`方法直接得到当前时刻的毫秒数。

注意这些 JSON 消息不能包含换行，虽然 JSON 格式本身会忽略所有空白符，也允许换行，但这个协议只允许在消息之间添加换行符，也就是以换行为分隔的 JSON 协议（line-delimited JSON, LDJ）。

3.2.2 切换到 JSON 格式
Switching to JSON Messages

我们已经定义了一个计算机可处理的协议，现在用这个协议改造 net-watcher 程序。然后再开发接收和处理这些消息的客户端程序。

使用 `JSON.stringify` 函数对消息对象进行序列化，然后使用 `connection.write` 发送出去。`JSON.stringify` 函数接收 JavaScript 对象，返回这个对象转换成的 JSON 格式字符串。

在编辑器中打开 `net-watcher.js` 程序。找到这一行：

```
connection.write(`Now watching "${filename}" for changes...\n`);
```

替换为下面的内容：

```
connection.write(JSON.stringify({type: 'watching', file: filename}) + '\n');
```

然后找到 watcher 里调用 connection.write 的地方：

```
const watcher =
  fs.watch(filename, () => connection.write(`File changed: ${new Date()}\n`));
```

替换为下面的内容：

```
const watcher = fs.watch(filename, () => connection.write(
  JSON.stringify({type: 'changed', timestamp: Date.now()}) + '\n'));
```

把新文件保存为 net-watcher-json-service.js，然后还是像之前一样运行，不要忘了加上目标文件参数：

```
$ node net-watcher-json-service.js target.txt
Listening for subscribers...
```

然后在第二个命令行使用 netcat 连接：

```
$ nc localhost 60300
{"type":"watching","file":"target.txt"}
```

使用 touch 命令修改文件时，你会在客户端看到这样的输出信息：

```
{"type":"changed","timestamp":1450437616760}
```

好了，接下来可以开始开发客户端来处理这些消息了。

3.3 建立 Socket 客户端连接
Creating Socket Client Connections

我们已经搞定了服务端程序，现在可以开始开发 socket 客户端了，这个客户端用于接收 net-watcher-json-service 发出的 JSON 消息。先实现一个简单版的客户端，然后在本章的后续章节中逐步完善。

在编辑器中输入如下代码：

networking/net-watcher-json-client.js

```
'use strict';
const net = require('net');
const client = net.connect({port: 60300});
client.on('data', data => {
  const message = JSON.parse(data);
  if (message.type === 'watching') {
    console.log(`Now watching: ${message.file}`);
  } else if (message.type === 'changed') {
    const date = new Date(message.timestamp);
    console.log(`File changed: ${date}`);
  } else {
    console.log(`Unrecognized message type: ${message.type}`);
  }
});
```

保存为 networking/net-watcher-json-client.js 文件。

上面这段代码中使用 net.connect 创建了从客户端到 localhost 端口 60300 的连接，然后开始等待数据。其中 client 是 Socket 对象，跟服务端的回调函数中的 connection 对象类似。

data 事件触发时，回调函数接收 buffer 对象，并解析成 JSON 数据，然后输出到控制台。

启动 net-watcher-json-service 服务，在另一个命令行窗口执行客户端程序：

```
$ node net-watcher-json-client.js
Now watching: target.txt
```

使用 touch 命令修改目标文件时，在命令行可以看到如下信息：

```
File changed: Mon Dec 21 2015 05:34:19 GMT-0500 (EST)
```

成功！程序运行起来了，但还有待改善。

想想连接中断或者第一次建立连接时失败的情况。这段代码只监听了 data 事件，没有监听 end 和 error 事件。我们应该监听这些事件并进行相应的处理。

我们的代码中有一个隐藏较深的问题，这是由我们对消息边界想当然的处理方式造成的。第 3.4 节开发的测试程序可以发现并修复这个问题。

3.4 网络应用功能测试
Testing Network Application Functionality

功能测试能确保代码按预期的方式运行。本节要开发一个测试程序，用于测试文件监控网络服务和客户端程序。我们还会开发一个 mock 服务器，它遵循之前的 LDJ 协议，使用这个 mock 服务器可以发现客户端的缺陷。

完成测试程序之后，我们要修复客户端的问题，这样才能通过测试。我们会接触到很多新的 Node.js 知识，包括扩展核心类、创建和使用自定义模块、基于 `EventEmitter` 的开发。在此之前，我们来看看客户端和服务端程序中隐藏的问题。

3.4.1 理解消息边界问题
Understanding the Message-Boundary Problem

使用 Node.js 开发网络应用时，经常会涉及消息通信。理想情况下，每条消息都一次接收成功，但有时情况不理想，消息有可能被切分成几块数据，由多个 `data` 事件接收。这种情况下，程序要能正确处理这种分块消息。

之前设计的 LDJ 协议使用换行符分隔消息，两条消息之间以换行符为边界，下面的例子将换行符标记出来，这样更容易理解多条消息之间如何分界。

```
{"type":"watching","file":"target.txt"}\n
{"type":"changed","timestamp":1450694370094}\n
{"type":"changed","timestamp":1450694375099}\n
```

回头看我们刚才开发的服务，每当有文件变化时，都会向 socket 连接发送一条消息，这条消息尾部包含一个换行符。在客户端，接收到的每行消息对应一个事件，或者说事件的分隔规则和消息的分隔规则是一致的。

客户端程序目前正是依赖这种方式运行的，直接把 `data` 事件得到的内容传给 `JSON.parse()`：

```
client.on('data', data => {
  const message = JSON.parse(data);
```

现在考虑这种情况，消息被分成两份并以两个独立的 `data` 事件传输。实践中

经常这样拆分消息，尤其是当消息内容很多时。图 3.3 展示了消息拆分的细节。

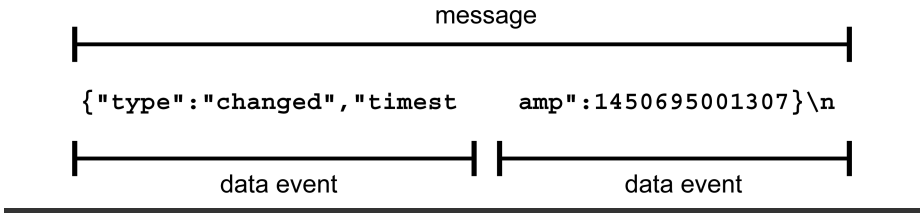

图 3.3 消息拆分

下面创建测试服务器，测试一条消息拆分成多条发送的情况，看看会发生什么。

3.4.2 实现测试服务器
Implementing a Test Service

健壮的 Node.js 应用要求能够优雅地处理各种情况，例如被拆分成多份的输入数据、中断的网络连接，还有不符合要求的数据。我们现在来实现一个测试服务器，刻意把一条消息拆分成多次发送。

打开编辑器并输入以下代码：

networking/test-json-service.js

```
'use strict';
const server = require('net').createServer(connection => {
  console.log('Subscriber connected.');

  // Two message chunks that together make a whole message.
  const firstChunk = '{"type":"changed","timesta';
  const secondChunk = 'mp":1450694370094}\n';

  // Send the first chunk immediately.
  connection.write(firstChunk);

  // After a short delay, send the other chunk.
  const timer = setTimeout(() => {
    connection.write(secondChunk);
    connection.end();
  }, 100);

  // Clear timer when the connection ends.
  connection.on('end', () => {
    clearTimeout(timer);
    console.log('Subscriber disconnected.');
  });
});
```

```
server.listen(60300, function() {
  console.log('Test server listening for subscribers...');
});
```

保存为 test-json-service.js 文件，然后运行以下代码：

```
$ node test-json-service.js
Test server listening for subscribers...
```

这个测试服务与我们之前开发的 net-watcher-json-service.js 有些不同，它没有像之前那样启动文件监听，而是直接把第一个数据发送出去了。

然后我设置了计时器，在短暂的时间间隔后发送第二条消息。JavaScript 中的 setTimeout()函数接收两个参数：回调函数和以毫秒为单位的时间间隔。在指定的时间间隔之后，将会执行回调函数。

当连接中断时，使用 clearTimeout()函数取消回调函数的调用。连接中断后再使用 connection.write 发送消息会造成异常，所以取消回调函数是必不可少的。

最后，看看当我们用客户端连接这个测试服务时会发生什么：

```
$ node net-watcher-json-client.js
undefined:1
{"type":"changed","timesta
                  ^

SyntaxError: Unexpected token t
    at Object.parse (native)
    at Socket.<anonymous> (./net-watcher-json-client.js:6:22)
    at emitOne (events.js:77:13)
    at Socket.emit (events.js:169:7)
    at readableAddChunk (_stream_readable.js:146:16)
    at Socket.Readable.push (_stream_readable.js:110:10)
    at TCP.onread (net.js:523:20)
```

抛出的异常是 *Unexpected token t*，说明 JSON 格式不合法。客户端以为接收到的是一整条符合 JSON 格式的消息并尝试使用 JSON.parse()解析它。

我们成功模拟了将一条消息拆分成多条的情况，现在试试在客户端修复它。

3.5 在自定义模块中扩展 Node.js 核心类
Extending Core Classes in Custom Modules

第 3.4 节开发的 Node.js 客户端程序有一个缺陷，任何分块发送的消息都会导致程序退出，原因是没有把接收到的数据缓存起来，并合并成原来的消息。

所以客户端实际上需要做两件事情：一是把数据缓存起来并加工成消息；二是对接收到的消息进行处理和响应。

不要把这两部分的代码塞进同一个 Node.js 程序里，应该把其中一部分封装成模块。接下来创建一个模块处理接收到的数据，并把数据合并成有效的消息，这样主程序就可以从这个模块获取可靠的 JSON 消息。在此之前，先要介绍如何创建自定义模块，以及如何扩展 Node.js 核心类。

3.5.1 扩展 EventEmitter
Extending EventEmitter

我们要实现一个 LDJ 缓存模块来解决 JSON 分块消息的问题，然后把它集成到 network-watcher 客户端。

Node 中的继承

我们先看看如何在 Node.js 中实现继承，以下代码中的 `LDJClient` 类继承自 `EventEmitter` 类。

networking/lib/ldj-client.js
```
const EventEmitter = require('events').EventEmitter;
class LDJClient extends EventEmitter {
  constructor(stream) {
    super();
  }
}
```

`LDJClient` 是一个类，所以可以通过 `new LDJClient(stream)`得到它的实例。`stream` 参数是可以接收 `data` 事件的对象，就像 `Socket` 连接那样。

构造函数通过 `super()`调用父类 `EventEmitter` 的构造函数。今后实现类的继承时，都要记得在第一行执行 `super()`并传入适当的参数。

在 JavaScript 内部，实际上是基于原型式继承建立 `LDJClient` 和 `EventEmitter` 之间的继承关系。原型式继承非常强大，不仅能用于构造类，还能用于其他方面，不过现在直接使用原型的场景越来越少了。`LDJClient` 类的用法如下：

```
const client = new LDJClient(networkStream);
client.on('message', message => {
  // Take action for this message.
});
```

类的结构出来了，但还没有实现任何逻辑。接下来要在 Node 中缓存数据。

缓存数据

使用 LDJClient 中的 stream 参数来获取和缓存收到的数据。我们要做的是把从 stream 获得的原始数据缓存下来，并解析成 message 对象，然后通过 message 事件转发出去。

看看下面的代码，将收到的数据块添加到缓存字符串中，然后查找换行符的位置（也就是 JSON 消息的结尾）。

networking/lib/ldj-client.js

```
constructor(stream) {
  super();
  let buffer = '';
  stream.on('data', data => {
    buffer += data;
    let boundary = buffer.indexOf('\n');
    while (boundary !== -1) {
      const input = buffer.substring(0, boundary);
      buffer = buffer.substring(boundary + 1);
      this.emit('message', JSON.parse(input));
      boundary = buffer.indexOf('\n');
    }
  });
}
```

原型式继承

回忆刚才创建的 LDJClient 类，在 JavaScript 语言引入 class、constructor、super 关键词之前，通常会使用下面的代码来实现类的继承。

```
const EventEmitter = require('events').EventEmitter;
const util = require('util');

function LDJClient(stream) {
  EventEmitter.call(this);
}
util.inherits(LDJClient, EventEmitter);
```

LDJClient 是构造函数，它与 class、constructor 的效果类似。把 EventEmitter 的构造函数指向 this，也达到了类似 super()的效果。

最后，使用 util.inherits 把 LDJClient 的原型链指向 EventEmitter，也就是说，如果查找不到 LDJClient 下的属性，则会接着去 EventEmitter 下查找。

假设有一个 LDJClient 的实例变量 client，当执行 client.on 方法时，即使 client 对象和 LDJClient 原型对象都没有这个 on 方法，JavaScript 也会找到 EventEmitter 的 on 方法并执行它。

同理，当执行 client.toString 方法时，JavaScript 引擎会一直找到 EventEmitter 原型链上游的 Object 对象的 toString 方法。

通常情况下，我们不需要关心这个层面的逻辑，框架开发者可能会利用这些特性做些事情，但普通开发者可以直接使用 class 关键词构造类。

回到上面的例子，先像之前一样执行 super，再设置 buffer 变量用于存储收到的数据，然后用 stream.on 处理 data 事件。

data 事件处理函数内的逻辑很多，但都是必要的。先把接收到的原始数据添加到 buffer 变量中，然后从前往后查找消息结束符，每条消息都用 JSON.parse 解析，最后用 this.emit 把消息发送出去。

代码写到这里，我们已经解决了刚开始时提出的问题，可以完美处理拆分成多次传输的消息。不管是一次传输多条消息，还是一条消息拆分成多次传输，我们的代码都能正确处理并触发 LDJClient 实例上的 message 事件。

回到上面的例子，先像之前一样执行 super，再设置 buffer 变量用于存储收到的数据，然后使用 stream.on 处理 data 事件。

data 事件处理函数内的逻辑很多，但都是必要的。先把接收到的原始数据添加到 buffer 变量中，然后从前往后查找消息结束符，每条消息都使用 JSON.parse 解析，最后使用 this.emit 把消息发送出去。

代码写到这里，我们已经解决了刚开始时提出的问题，可以完美处理拆分成多次传输的消息。不管是一次传输多条消息，还是一条消息拆分成多次传输，我们的代码都能正确处理并触发 LDJClient 实例上的 message 事件。

接下来把这个 class 整合进 Node.js 模块里，给上层的客户端代码调用。

对外暴露模块功能

现在把前面的示例代码整合起来，将 LDJClient 作为一个模块对外暴露出

来。先新建 lib 文件夹，当然你也可以给文件夹取别的名称，但 Node.js 社区约定把支撑性代码放在 lib 文件夹里。

打开编辑器并输入如下代码：

networking/lib/ldj-client.js

```
'use strict';
const EventEmitter = require('events').EventEmitter;
class LDJClient extends EventEmitter {
  constructor(stream) {
    super();
    let buffer = '';
    stream.on('data', data => {
      buffer += data;
      let boundary = buffer.indexOf('\n');
      while (boundary !== -1) {
        const input = buffer.substring(0, boundary);
        buffer = buffer.substring(boundary + 1);
        this.emit('message', JSON.parse(input));
        boundary = buffer.indexOf('\n');
      }
    });
  }

  static connect(stream) {
    return new LDJClient(stream);
  }
}

module.exports = LDJClient;
```

保存为 lib/ldj-client.js 文件。这段代码只是把之前的示例结合起来，另外还增加了静态方法和最后的 module.exports 语句。

在 class 的定义中，constructor 方法之后加了一个静态方法 connect。静态方法是添加在 class 上而不是实例上。connect 方法只不过是给使用者提供便利，让他们能够简单快速地创建 LDJClient 实例。

module.exports 对象是 Node.js 模块和外界的桥梁，任何添加在 exports 上的属性都能被外部访问到。在上面的示例中，我们把 LDJClient 类整个暴露出去了。

外部使用 LDJ 模块的代码大致像下面这样：

```
const LDJClient = require('./lib/ldj-client.js');
const client = new LDJClient(networkStream);
```

也可以使用 connect 方法，如下所示：

```
const client = require('./lib/ldj-client.js').connect(networkStream);
```

注意上面的代码中我们给 require()函数传入了真实文件路径，而不像以前那样传入 fs、net、util 这样的模块名。当 require()函数的参数是路径时，它会以当前文件为起点去查找相对路径中的文件。

模块完成！现在去 network-watching 客户端调用这个模块，把所有功能都整合起来。

3.5.2 导入自定义模块
Importing a Custom Node.js Module

现在修改客户端代码。原来是直接从 TCP 流读取数据，现在要把刚开发的模块利用起来。

打开编辑器并输入如下代码：

networking/net-watcher-ldj-client.js
```
'use strict';
const netClient = require('net').connect({port: 60300});
const ldjClient = require('./lib/ldj-client.js').connect(netClient);

ldjClient.on('message', message => {
  if (message.type === 'watching') {
    console.log(`Now watching: ${message.file}`);
  } else if (message.type === 'changed') {
    console.log(`File changed: ${new Date(message.timestamp)}`);
  } else {
    throw Error(`Unrecognized message type: ${message.type}`);
  }
});
```

保存为 net-watcher-ldj-client.js 文件。它与第 3.3 节开发的 net-watcher- json-client 文件很相似，不同之处是它没有直接使用 JSON.parse()解析数据，而是从刚开发的 ldj-client 模块的 message 事件获取消息。

我们来运行一下测试服务，看看分块消息的问题是否解决了。

```
$ node test-json-service.js
Test server listening for subscribers...
```

在另一个窗口运行客户端：

```
$ node net-watcher-ldj-client.js
File changed: Tue Jan 26 2016 05:54:59 GMT-0500 (EST)
```

成功！现在服务端和客户端可以通过自定义的消息格式进行可靠的通信了。最后介绍流行的测试框架 Mocha，用它编写单元测试。

3.6 使用 Mocha 编写单元测试
Developing Unit Tests with Mocha

Mocha 是非常流行的多范式 Node.js 测试框架，它可以通过多种方式开发测试，这里我们用到行为驱动的开发方式（behavior-driven development，BDD）。

首先通过 npm 安装 Mocha，npm 是 Node.js 内置的包管理器。然后为 `LDJClient` 类开发单元测试，最后使用 npm 运行测试套件。

3.6.1 通过 npm 安装 Mocha
Installing Mocha with npm

使用 npm 安装 Node.js 模块非常方便。除了安装方法，你还需要了解安装过程中发生了什么，这样才能更好地管理依赖的模块。

先创建 `package.json` 文件，npm 非常依赖这个配置文件。在 `networking` 项目目录下打开终端，执行如下代码：

```
$ npm init -y
```

执行 `npm init` 命令创建 `package.json` 文件。后续章节会详细介绍这个文件，目前先把注意力放在如何安装 Mocha 上，执行 `npm install`：

```
$ npm install –save-dev –save-exact mocha@3.4.2
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN networking@1.0.0 No description
npm WARN networking@1.0.0 No repository field.

+ mocha@3.4.2
added 34 packages in 2.348s
```

先忽略警告信息，npm 提示给 package.json 添加描述性字段。

安装完毕后，项目目录下多了一个 node_modules 文件夹，其中包含 Mocha 模块和它依赖的其他模块。打开 package.json 文件，其中有如下代码：

```
"devDependencies":{
  "mocha": "3.4.2"
}
```

Node.js 中有几种不同类型的依赖，常规依赖是在代码运行时会用到的模块，它们用 require 语句引入，开发依赖是只有在开发时需要的模块。Mocha 属于后者，所以安装时使用--save-dev 参数（也可以简写成-D）告诉 npm 把模块添加到 devDependencies 列表中去。

需要注意的是，无论是开发依赖还是常规依赖，都会在不带任何参数运行 npm install 时安装到项目中。如果只想安装常规依赖，则可以在运行 npm install 时使用--production 参数，或者把 NODE_ENV 环境变量的值设为 *production*。

同时，npm 也会创建 package-lock.json 文件，这个文件记录了本次安装的 Mocha 模块的版本和其他模块的版本。我们暂时把这个文件放一放，先学习语义化版本号的概念，以及 npm 是如何解决模块版本问题的。

3.6.2 模块的语义化版本号
Semantic Versioning of Packages

上节例子中的--save-exact 告诉 npm 我们需要记录精确的版本，也就是例子中的 3.4.2。默认情况下，npm 会通过语义化版本号（semantic versioning，也叫 SemVer）找到最合适的可用版本[1]。

语义化版本号在 Node.js 社区是大家广泛遵守的约定，你给自己的模块设定版本号时也要遵守这个约定。版本号由三部分组成（用点号链接）：主版本号，次版本号，修订版本号。

修改版本号时，必须根据语义化版本号的约定修改特定的部分。

[1] http://semver.org/

- 如果本次修改代码没有新增或删除任何功能（比如修复 bug），则应该增加修订版本号。
- 如果本次修改代码新增了功能，但没有删除或者修改已有的功能，则应该增加次版本号，并重置修订版本号。
- 如果本次修改代码会对现有功能产生影响，则应该增加主版本号，并重置次版本号和修订版本号。

如果去掉`--save-exact` 参数，则安装符合条件的最新版本。如果直接把版本号也去掉，则直接安装这个模块最近发布的版本。

本书为了保证所有示例代码都能正确运行，同时避免依赖模块违反语义化版本号约定带来的风险，会使用严格指定的版本号。

如果安装 npm 模块时去掉`--save-exact` 参数，则会在 `package.json` 中添加带^的版本号。例如，添加的版本号是`^3.4.2` 而不是之前的 `3.4.2`。其中的^表示 npm 会安装与你指定版本相同或者更新的次版本。

举个例子，如果依赖版本设置为`^1.5.7`，并且该模块存在`1.6.0`版本和`2.0.0`版本，那么其他开发者安装后会得到最新的 `1.6.0` 版本。但不会安装到 `2.0.0` 版本，因为主版本更新意味着发生了不能向下兼容的变更。

如果想要更严格一点，可以用~做前缀。还是使用刚才的例子，如果依赖版本设置为`~1.5.7`，并且该模块存在`~1.5.8` 版本和 `1.6.0` 版本，那么开发者安装的是 `1.5.8` 而不是 `1.6.0`。前缀~比前缀^更安全，因为大部分人不会在修复缺陷时引入不兼容的变更。

虽然 Node.js 社区广泛采用了语义化版本号，但模块作者有时候会在主版本达到 1 之前做一些不兼容的代码变更。比如，有些项目可能从 `0.0.1` 开始计算版本号，然后在 `0.0.2` 和 `0.0.3` 引入不兼容变更。同样，有些项目可能从 `0.1.0` 到 `0.3.0` 都会引入不兼容变更。所以在匹配^前缀和~前缀版本时，npm 会有意忽略版本号前面的 0。

我的建议是：安装模块时始终加上`--save-exact` 参数。这么做的缺点是当你想要安装新版本时必须手动更新，但这至少是在你的掌控下的更新，而不会出现

你无法控制的意外。

即使你小心翼翼地使用`--save-exact`管理直接依赖，还是有可能出现问题，因为由依赖模块引入的间接依赖的模块可能没有严格遵守版本号约定。这时候需要用到`package-lock.json`文件，它记录了整个依赖树的所有模块版本号和校验码。

如果想在不同机器上安装完全一致的依赖文件，就必须把`package-lock.json`提交到代码仓库里。可以使用`npm outdated`命令生成待更新报告，里面包含所有可以更新的模块版本。一旦在项目里安装了新版本的模块，`package-lock.json`里的版本记录也会同步更新。

在开发过程中提交`package-lock.json`是一个好习惯，这样就能记录任意历史时刻代码的依赖，以确保即使回到过去的某一时刻，代码也能正常运行。这在跟踪bug时非常有用，因为有时bug不一定来自你自己的代码，有可能是依赖的第三方模块引入的。

语义化版本号就介绍到这里，接下来动手写测试吧！

3.6.3 使用 Mocha 开发单元测试
Writing Mocha Unit Tests

安装Mocha后，现在可以开始开发单元测试了。

新建`test`目录，把所有与测试有关的代码都放到里面。这是Node.js约定的做法，Mocha会自动在这个目录下查找测试代码。

接着在`test`目录下新建`ldj-client-test.js`文件并输入如下代码：

networking/test/ldj-client-test.js
```
'use strict';
const assert = require('assert');
const EventEmitter = require('events').EventEmitter;
const LDJClient = require('../lib/ldj-client.js');

describe('LDJClient', () => {
  let stream = null;
  let client = null;

  beforeEach(() => {
```

```
    stream = new EventEmitter();
    client = new LDJClient(stream);
  });

  it('should emit a message event from a single data event', done => {
    client.on('message', message => {
      assert.deepEqual(message, {foo: 'bar'});
      done();
    });
    stream.emit('data', '{"foo":"bar"}\n');
  });
});
```

下面来逐行看看这段代码，首先引入所有需要的模块，其中包括 Node.js 内置的 `assert` 模块，它包含很多实用的比较函数。

然后使用 Mocha 的 `describe()`方法创建一个测试 `LDJClient` 的上下文环境，其中 `describe()`的第二个参数是函数，这个函数包含测试的具体内容。

在测试函数内，我们先声明两个变量，一个是 `LDJClient` 实例，另一个是 `EventEmitter` 实例。在 `beforeEach` 中，将新的实例赋值给上面创建的两个变量。

最后调用 `it()`函数进行实际测试。由于我们的代码是异步的，所以需要通过 Mocha 提供的 `done()`函数告诉 Mocha 测试什么时候结束。

在测试代码中，给 `client` 的 `message` 事件设置了监听函数。监听函数使用 `deepEqual` 方法对测试数据和正确数据进行比较。最后触发 `stream` 的 `data` 事件，这会引发 `message` 事件的回调函数执行。

测试代码写完了，试着运行一下吧！

3.6.4 使用 npm 脚本运行 Mocha 测试代码
Running Mocha Tests from npm

使用 npm 运行 Mocha 测试代码，先要在 `package.json` 文件添加一行配置。打开 `package.json` 文件，在 `scripts` 区块添加一行：

```
"script": {
  "test": "mocha"
},
```

在 scripts 区块下的命令都可以通过 npm run 在命令行调用。例如，如果在命令行执行 npm run test，实际上会执行 mocha 命令。

test 命令还可以将 run 省略掉，简写成 npm test。打开命令行，执行如下命令：

```
$ npm test

> @ test ./code/networking
> mocha


  LDJClient
    ✓ should emit a message event from single data event

  1 passing (9ms)
```

测试通过！接下来把 test-json-service.js 中的代码转换成 Mocha 测试代码。

3.6.5 添加异步测试代码
Adding More Asynchronous Tests

这个代码结构很容易添加新的测试，把 test-json-service.js 改造成基于 Mocha 的测试代码。打开 test/ldj-client-test.js 文件，在 describe()代码块中增加如下代码：

networking/test/ldj-client-test.js
```
it('should emit a message event from split data events', done => {
  client.on('message', message => {
    assert.deepEqual(message, {foo: 'bar'});
    done();
  });
  stream.emit('data', '{"foo":');
  process.nextTick(() => stream.emit('data', '"bar"}\n'));
});
```

这个测试将消息拆分成两部分并依次发出。注意这里用到了 process.nextTick()函数，这是 Node.js 的内置方法，它能让回调函数里的代码在当前代码执行结束后立即执行。

在前端开发中常常会使用 setTimeout 延迟 0 毫秒来达到类似的目的。但 setTimeout(callback,0)和 process.nextTick(callback)还是有区别的，后者

会在下一次事件循环开始之前执行，而 `setTimeout` 会等一次事件循环结束后再执行。

不管用哪种方法，只要延迟的时间小于 Mocha 的超时时间，测试都能通过。Mocha 的默认超时时间为 2 秒。超时时间可以全局修改，也可以为单个测试修改。

使用`--timeout` 参数可以指定 Mocha 本次测试的超时时间，如果设为 0，则禁用超时。

Mocha 的`it()`方法的返回值对象包含了`timeout()`方法，调用这个`timeout()`方法可以为一个特定的测试设置超时时间。如下所示：

```
it('should finish within 5 seconds', done => {
 setTimeout(done, 4500); // Call done after 4.5 seconds.
}).timeout(5000);
```

`describe()`方法返回的对象也包含`timeout()`方法，`timeout()`方法可以设置`describe()`内的一组测试的超时时间。

好了，开始下一章之前，我们来回顾一下本章内容吧！

3.7 小结与练习
Wrapping Up

本章讲解使用 Node.js 开发基于 socket 的网络应用，开发了客户端和服务端逻辑，还定义了基于 JSON 的通信协议。

开发测试用例用于发现代码中的潜在问题，扩展 Node.js 内置的`EventEmitter`类写出了第一个自定义模块。同时学习了如何将流数据缓存下来，如何查找消息分隔符。

使用 npm 安装了测试框架 Mocha，并用它开发了单元测试。

像本章介绍的这样，开发简单网络应用不需要很多代码。只需简单几行，就能创建出功能完善的服务端和客户端应用。

然而，开发健壮的应用程序要比这困难得多，需要考虑各种可能导致出错的情况。第 4 章介绍高性能消息模块和框架，以达到更高的要求。

3.7.1 易测性
Testability

本章使用 Mocha 开发了一个测试用例，但它仅测试了 `LDJClient` 类的最基本行为，也就是当 data 事件触发发送消息的情况。

试着思考下面的问题并开发测试用例。

- 如果一条消息被拆分成数据流的多个 data 事件，如何测试？
- 如果 `LDJClient` 构造函数接收的参数是 `null`，则需要抛出异常，这种情况如何测试？

3.7.2 鲁棒性
Robustness

本章开发的 `LDJClient` 还不完善，试着思考下面的问题并改进代码。

- `LDJClient` 类能够处理一条 JSON 格式的消息被拆分成多次发送的情况，但如果接收到的数据不符合 JSON 格式呢，会发生什么？
- 开发一个测试用例，其中 data 事件发送的不是 JSON 数据。这种情况应该如何编写测试？
- 如果最后一个 data 事件是完整的 JSON 消息，但却没有换行符作为结尾，会发生什么？
- 开发一个测试用例，其中 `stream` 对象发送一个 JSON 数据但结尾没有换行符，后面再跟一个 `close` 事件。实际的 `Stream` 实例会在结束前触发 `close` 事件，修改 `LDJClient` 类的代码，监听 `close` 事件并处理剩余的缓存数据。
- `LDJClient` 要给它的监听者发送 `close` 事件吗？什么情况下需要？