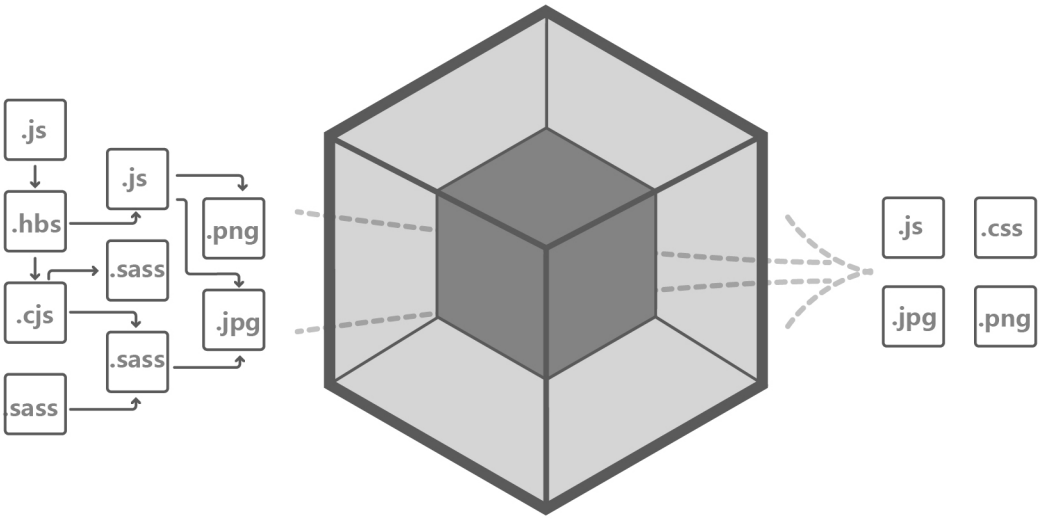

Node.js+Webpack
开发实战

夏 磊 著

清華大學出版社
北 京

内 容 简 介

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境，它用于构建高速、可伸缩的网络应用程序，为前端开发提供了新的机遇。为了让前端开发者更有效地使用 Node.js 进行开发，作者结合自己的开发经验编著了本书，全书提供了丰富的示例代码，详细讲述和演示了如何将所学的知识应用于实际的开发中。

本书分为三部分共 21 章，第一部分 Node.js 基础：Node.js 概述，搭建 Node.js 开发环境，Node.js 编程基础；第二部分后端的 Node.js：Express 框架，Koa 框架，MongoDB 数据库，MySQL 数据库，ORM 框架 Sequelize，微博系统实战项目，高性能内存型数据库 Redis，前端的发展现状；第三部分前端的 Node.js：前端发展状况，Webpack 基础，Webpack 常用配置，Webpack 构建 Vue 应用，Webpack 构建 React 应用，服务端渲染技术和同构应用的开发，Webpack 构建传统多页面 Web 应用，Webpack 性能优化，Webpack 自定义 Loader 的编写，Webpack 自定义 Plugin 的编写。

本书适合 Node.js+Webpack 前端开发工程师作为自学参考书，也适合高等院校和培训学校相关专业的师生作为教学参考书。

图书在版编目（CIP）数据

Node.js+Webpack 开发实战 / 夏磊著.—北京：清华大学出版社，2020.6
ISBN 978-7-302-55595-7

Ⅰ. ①N…　Ⅱ. ①夏…　Ⅲ. ①JAVA 语言－程序设计②网页制作工具－程序设计　Ⅳ. ①TP312.8 ②TP392.092.2

中国版本图书馆 CIP 数据核字（2020）第 089921 号

责任编辑：夏毓彦
封面设计：王　翔
责任校对：闫秀华
责任印制：宋　林

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：三河市君旺印务有限公司
经　　销：全国新华书店
开　　本：190mm×260mm　　**印　　张**：20.25　　**字　　数**：518 千字
版　　次：2020 年 8 月第 1 版　　**印　　次**：2020 年 8 月第 1 次印刷
定　　价：69.00 元

产品编号：076384-01

前　言

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境，用于构建高速、可伸缩的网络应用程序。

事实上，Node.js 不仅仅用来构建网络应用程序，还为前端开发提供了新的机遇。现在拥有 JavaScript 经验的开发人员可以在前端和后端使用 Node.js，降低了语言导致的过渡成本。Node.js 拥有一个巨大的 JavaScript 生态系统，再加上这几年前端的发展，出现了许多新框架和新语言，但是对于初次接触 Node.js 的用户来说是不太友好的，再加上缺乏系统性的指南，导致前端开发者无法有效地学好 Node.js，作者编著本书希望对改变这种情况尽绵薄之力。

关于本书

第一部分　Node.js 基础

第一部分是对 Node.js 的介绍，涵盖了它的原理和基础知识。

第 1 章介绍 Node.js 的原理和应用场景。

第 2 章介绍如何在计算机上安装 Node.js 以及 Visual Studio Code 编辑器。我们将用一个简单的 HTTP 服务器来测试 Node.js 是否成功安装。

第 3 章介绍 Node.js 的编程基础。内容包括 NPM、模块系统、异步编程方式和常用模块。

第二部分　后端的 Node.js

第二部分是对后端 Node.js 的介绍，涵盖了主流的 Web 框架和常用组件，包含数据库、缓存，等等。

第 4 章详细介绍 Express 开发框架，这是最早也是最流行的 Node.js Web 开发框架。内容包括 Express 的请求路由、请求与响应、中间件、错误处理和页面渲染。最后演示如何使用 Express 框架开发留言板系统。

第 5 章详细介绍 Koa 框架。Koa 框架被称为“下一代的 Web 开发框架”，Koa 的“一切皆为中间件”思想被其他 Web 框架广泛地采用。本章内容包括 Koa 的上下文对象、中间件模型、请求路由、错误处理和模板渲染。最后演示如何使用 Koa 开发博客系统。

第 6 章介绍 MongoDB 数据库。MongoDB 通常被称为 Node.js 的“黄金搭档”，因为 MongoDB 采用了“BeJSON”的结构，对 JavaScript 有天然的亲和性。本章内容包括 MongoDB 的安装、基础使用和 Node.js 对 MongoDB 的操作。

第 7 章介绍 MySQL 数据库，这是目前最流行的、开源的关系型数据库系统。内容包括

MySQL 的安装、基础语法、关联关系和事务操作，为后续的实战项目打下基础。

第 8 章介绍 ORM 框架——Sequelize，Sequelize 一个操作 MySQL 的框架，能够通过对象的方式操作数据库。本章内容包括 Sequelize 模型、关联关系、对数据的操作和事务的使用。

第 9 章介绍一个完整的实战项目开发过程。我们将基于 Koa 和 Sequelize 来开发一个微博系统，带领大家学习一个完整项目的研发流程。

第 10 章介绍高性能内存型 NoSQL 数据库 Redis，Redis 常用在高并发场景，比如秒杀活动、抽奖、排行榜等。本章内容包括 Redis 的基础知识，数据结构以及 Node.js 对 Redis 的操作。

第 11 章介绍实时 Web 通信技术 WebSocket，WebSocket 的出现赋予了 Web 应用更多的可能性。本章内容包括传统的实时 Web 技术、WebSocket 协议的原理以及使用 Node.js 实现 WebSocket 服务器，最后演示如何使用 Node.js 来构建一个在线聊天室。

第三部分　前端的 Node.js

第三部分介绍前端的发展以及 Node.js 在前端的应用，重点介绍目前最流行的构建工具——Webpack。

第 12 章介绍前端的发展现状，包括模块系统、新语言、新框架和新的构建工具。

第 13 章介绍 Webpack 的基础使用和核心概念，包括如何一步一步对 Webpack 进行配置以及 Loader 和 Plugin 的使用。

第 14 章详细介绍 Webpack 的常用配置。

第 15 章介绍如何使用 Webpack 构建 Vue 应用，包括构建 Vue 应用需要的模块、相应的配置和导入 TypeScript 支持。

第 16 章介绍如何使用 Webpack 构建 React 应用，包括 JSX 语法、Babel 工具、Webpack 的配置以及导入 TypeScript 支持。

第 17 章介绍服务端渲染技术和同构应用的开发，包括服务端渲染技术的原理以及如何使用 Webpack 构建同构应用，最后演示如何构建一个 React 的同构应用。

第 18 章详细介绍如何使用 Webpack 构建传统多页面 Web 应用。

第 19 章详细介绍 Webpack 性能优化的常用手段，包括优化配置、提取公共代码、多进程编译、按需加载和热更新的知识。

第 20 章介绍 Webpack 自定义 Loader 的编写，包括基本 Loader、Loader 配置、异步 Loader 等知识，最后演示如何编写一个多语言 Loader。

第 21 章介绍 Webpack 自定义 Plugin 的编写，包括 Webpack 构建流程、Compiler 和 Compilation、Tapable 对象和常用 API，最后演示清单文件插件的编写以及将构建结果上传到 CDN 插件的编写。

示例代码下载

本书提供了丰富的示例，演示如何利用每个所学的知识点。本书的源码已经托管到 GitHub 网站，读者可以通过 **https://github.com/nodejs-inaction** 链接进行访问及下载。也可以扫描下面的二维码下载。

如果你在下载过程中遇到问题，可发送邮件至 booksaga@163.com 获得帮助，邮件标题为“Node.js+Webpack 开发实战”。

关于作者

夏磊，毕业于湖南工业大学网络工程专业，拥有多年研发经验。在过去的几年里，他还是很多开源项目的贡献者。精通 PHP 脚本开发、Node.js/Golang 服务端开发以及 JavaScript 开发，善于把握与应用新技术，博客和公众号上有大量的 Web 相关技术文章，深受读者好评。著有图书《ThinkPHP 实战》和《ThinkPHP5 实战》。

著者

2020 年 5 月

目　录

第一部分　Node.js 基础篇

第 1 章　Node.js 概述 3

1.1　Node.js 是什么 3
1.2　Node.js 的运行原理 5
1.3　Node.js 的应用场景 6
1.3.1　Node.js 优缺点 6
1.3.2　应用场景 7
1.4　本章小结 7

第 2 章　搭建 Node.js 开发环境 8

2.1　安装 Node.js 8
2.1.1　Windows 上安装 Node.js 8
2.1.2　Linux 安装 Node.js 9
2.1.3　Ubuntu 安装 Node.js 9
2.1.4　CentOS 安装 Node.js 9
2.1.5　macOS 安装 Node.js 10
2.2　安装 VSCode 编辑器 10
2.3　编写 HTTP 服务器 11
2.4　本章小结 12

第 3 章　Node.js 编程基础 13

3.1　NPM 包管理器介绍 13
3.1.1　更换 NPM 镜像源 13
3.1.2　初始化项目 14
3.1.3　使用 npm 命令安装模块 14
3.1.4　本地安装与全局安装 14
3.1.5　生产依赖和开发依赖 15
3.1.6　其他 npm 命令 15
3.2　Yarn 包管理器介绍 16
3.2.1　安装 Yarn 16

3.2.2 Yarn 常用命令 16
3.3 解读 package.json 文件 16
3.3.1 package.json 字段说明 17
3.3.2 版本号说明 18
3.3.3 常见的版本号限定符 18
3.4 Node.js 的模块系统 18
3.4.1 module 和 exports 19
3.4.2 require 20
3.4.3 开发一个自定义模块 21
3.5 Node.js 的异步编程风格 22
3.5.1 回调函数 22
3.5.2 Promise 23
3.5.3 async/await 26
3.6 Node.js 常用核心模块 28
3.6.1 events 模块 28
3.6.2 fs 模块 29
3.6.3 stream 接口 32
3.6.4 http 模块 35
3.7 本章小结 36

第二部分 后端的 Node.js

第 4 章 最流行的 Web 框架——Express 39

4.1 框架简介 39
4.2 快速开始 40
4.2.1 初始化项目 40
4.2.2 开始编码 40
4.2.3 运行应用 41
4.2.4 小结 41
4.3 路由 41
4.3.1 路由方法 42
4.3.2 路由路径 42
4.3.3 路由参数 44
4.3.4 路由函数 45
4.4 请求对象 47
4.4.1 获取请求 Cookie 49
4.4.2 获取请求体 50
4.5 响应对象 50
4.6 中间件 54

4.6.1 全局中间件 54
4.6.2 路由中间件 55
4.6.3 可配置的中间件 55
4.6.4 Cookie 中间件 57
4.6.5 响应时长中间件 57
4.6.6 静态资源中间件 58
4.7 错误处理 59
4.7.1 同步错误 59
4.7.2 异步错误 60
4.7.3 自定义错误处理函数 60
4.7.4 多个错误处理函数 61
4.8 模板渲染 62
4.8.1 使用 ejs 模板 62
4.8.2 ejs 语法 63
4.9 留言板项目开发 65
4.9.1 开始编码 65
4.9.2 运行项目 67
4.10 本章小结 68

第 5 章 下一代 Web 开发框架——Koa 70

5.1 Koa 简介 70
5.2 Bluebird 71
5.3 Koa 快速开始 72
5.3.1 初始化项目 72
5.3.2 开始编码 73
5.4 Context 73
5.5 Cookie 操作 75
5.5.1 Cookie 签名 75
5.5.2 写入 Cookie 75
5.5.3 读取 Cookie 76
5.5.4 中间件 76
5.5.5 请求日志中间件 78
5.5.6 可配置的中间件 79
5.5.7 Cookie 解析中间件 80
5.5.8 路由函数 81
5.5.9 多个路由函数 81
5.5.10 错误处理 82
5.5.11 多个错误处理器 83
5.6 路由系统 84

5.6.1 快速开始 85
5.6.2 路由对象 85
5.6.3 路由路径 86
5.6.4 路由函数 86
5.6.5 路由级别中间件 87
5.6.6 路由前缀 87
5.6.7 模块化路由 88
5.7 模板渲染 89
5.7.1 快速开始 89
5.7.2 模板布局 90
5.8 博客项目实战 92
5.8.1 功能梳理 92
5.8.2 项目代码 93
5.8.3 效果展示 100
5.8.4 项目小结 102
5.9 本章小结 102

第 6 章 文档型 NoSQL 数据库——MongoDB 103

6.1 简介 103
6.1.1 主要特点 103
6.1.2 概念 104
6.1.3 数据库 104
6.1.4 集合 105
6.1.5 文档 105
6.2 安装 106
6.2.1 Windows 106
6.2.2 Linux 107
6.2.3 macOS 108
6.3 常用操作 109
6.3.1 创建数据库 109
6.3.2 删除数据库 109
6.3.3 创建集合 110
6.3.4 查看集合 110
6.3.5 删除集合 110
6.3.6 索引 111
6.3.7 插入文档 112
6.3.8 更新文档 112
6.3.9 删除文档 113
6.3.10 查询文档 113

6.3.11 其他查询语法 115
6.4 Node.js 集成 116
6.4.1 初始化项目 116
6.4.2 连接数据库 116
6.4.3 mongoose 的关键概念 117
6.4.4 Schema 117
6.4.5 Model 120
6.5 本章小结 121

第 7 章 最流行的关系型数据库——MySQL 123

7.1 简介 123
7.2 安装 123
7.2.1 Windows 124
7.2.2 Linux 124
7.2.3 macOS 126
7.3 术语 126
7.4 索引 127
7.4.1 普通索引 127
7.4.2 唯一索引 127
7.4.3 联合索引 128
7.5 事务 128
7.5.1 ACID 原则 128
7.5.2 事务并发问题 129
7.5.3 隔离级别 129
7.5.4 事务控制语句 129
7.6 关联关系 130
7.6.1 一对多关联 130
7.6.2 一对一关联 131
7.6.3 多对多关联 131
7.7 数据库操作 132
7.8 数据类型 133
7.9 数据表操作 135
7.9.1 创建数据表 135
7.9.2 删除数据表 136
7.9.3 添加字段 136
7.9.4 删除字段 137
7.9.5 修改字段 137
7.10 数据操作 137
7.10.1 插入数据 137

7.10.2 查询数据 138
7.10.3 修改数据 139
7.10.4 删除数据 139
7.11 本章小结 140

第 8 章 ORM 框架——Sequelize 141

8.1 ORM 141
8.2 Sequelize 简介 142
8.3 快速开始 142
8.4 构造方法 143
8.5 数据类型 144
8.6 模型定义 146
8.6.1 字段设置 146
8.6.2 模型选项 147
8.6.3 Hooks 148
8.6.4 生命周期函数 149
8.6.5 模型验证器 150
8.6.6 模型方法 155
8.6.7 索引 156
8.6.8 数据库同步 157
8.7 模型使用 157
8.7.1 插入数据 158
8.7.2 更新数据 159
8.7.3 删除数据 160
8.7.4 查询数据 161
8.7.5 查询语法 164
8.7.6 事务 165
8.8 关联 167
8.8.1 hasOne 167
8.8.2 belongsTo 169
8.8.3 hasMany 171
8.8.4 belongsToMany 173
8.9 本章小结 175

第 9 章 微博项目开发 176

9.1 功能分析 176
9.2 数据模型 177
9.3 开始编码 177
9.3.1 初始化项目 177

9.3.2 项目目录 178
9.3.3 路由设计 178
9.3.4 共享组件 178
9.3.5 中间件 179
9.3.6 模型代码 180
9.3.7 生成数据表 183
9.3.8 业务代码 184
9.3.9 路由代码 188
9.3.10 视图文件 192
9.3.11 Web 应用引导文件 193
9.4 效果展示 194
9.5 项目代码 196
9.6 本章小结 196

第 10 章 高性能内存型 NoSQL 数据库——Redis 197

10.1 Redis 简介 197
10.1.1 特点 197
10.1.2 应用场景 198
10.2 Redis 安装 198
10.2.1 在 Windows 下安装 Redis 198
10.2.2 在 Linux 下安装 Redis 199
10.2.3 在 macOS 下安装 Redis 199
10.3 Redis 支持的数据结构 200
10.3.1 String（字符串） 200
10.3.2 哈希表（Hash） 201
10.3.3 列表（List） 202
10.3.4 集合（Set） 203
10.3.5 有序集合（ZSet） 203
10.3.6 发布订阅 204
10.4 Node.js 集成 Redis 205
10.4.1 快速开始 205
10.4.2 Promise 206
10.5 本章小结 207

第 11 章 实时双向 Web 技术——WebSocket 208

11.1 传统的实时 Web 技术 208
11.1.1 Ajax 轮询（Ajax Polling） 208
11.1.2 服务器推送（Comet） 209
11.2 WebSocket 209

11.3 实现 WebSocket 握手协议 210
11.3.1 握手协议过程 211
11.3.2 服务端代码 211
11.3.3 客户端代码 212
11.4 使用 ws 模块开发聊天室 212
11.4.1 安装依赖 213
11.4.2 服务端代码 213
11.4.3 客户端代码 214
11.5 本章小结 215

第三部分 前端中的 Node.js

第 12 章 迅速发展的前端技术 219

12.1 模块系统 219
12.1.1 CommonJS 220
12.1.2 AMD 220
12.1.3 CMD 221
12.1.4 ES6 模块化 221
12.2 新语言 222
12.2.1 ES6 222
12.2.2 TypeScript 222
12.2.3 Less 223
12.2.4 SCSS 223
12.3 新框架 224
12.3.1 AngularJS 224
12.3.2 React 224
12.3.3 Vue 224
12.3.4 Angular 225
12.4 构建工具 225
12.4.1 Grunt 226
12.4.2 Gulp 226
12.4.3 Webpack 227
12.5 本章小结 228

第 13 章 Webpack 起步 229

13.1 安装 229
13.2 示例项目 230
13.3 Loader 231
13.3.1 CSS 处理 231

13.3.2 图片处理 232
13.4 Plugin 234
13.4.1 提取 CSS 234
13.4.2 自动更新 HTML 中的资源引用 236
13.5 开发服务器 237
13.6 核心概念 239
13.7 本章小结 239

第 14 章 Webpack 配置 240

14.1 Mode 241
14.2 Entry 和 Context 241
14.2.1 不配置 Context 的情况 241
14.2.2 配置 Context 的情况 242
14.3 Output 242
14.3.1 chunkFilename 243
14.3.2 path 243
14.3.3 publicPath 244
14.3.4 libraryTarget 和 library 244
14.4 Module 246
14.4.1 noParse 246
14.4.2 rules 247
14.5 Resolve 248
14.5.1 alias 248
14.5.2 extensions 249
14.5.3 mainFields 249
14.5.4 modules 250
14.6 devtool 250
14.7 externals 250
14.8 DevServer 251
14.9 Plugins 252
14.10 完整示例 252
14.11 本章小结 254

第 15 章 Vue 实战 255

15.1 Hello World 255
15.2 配置 Webpack 257
15.2.1 Loader 和 Plugin 257
15.2.2 安装依赖模块 257
15.2.3 编写配置文件 258

15.2.4 执行构建 259
15.3 生产构建 259
15.3.1 Webpack 配置 259
15.3.2 package.json 修改 260
15.4 TypeScript 支持 261
15.4.1 TypeScript 配置 261
15.4.2 Webpack 配置 262
15.4.3 App.vue 263
15.5 本章小结 264

第 16 章 React 实战 265

16.1 JSX 265
16.2 Babel 266
16.3 TypeScript 268
16.4 本章小结 271

第 17 章 服务端渲染 272

17.1 SSR 原理 273
17.2 添加 SSR 的 webpack.config.js 273
17.3 添加 SSR 的入口文件 274
17.4 添加 SSR 打包命令 275
17.5 执行构建 275
17.6 添加 Node.js HTTP 服务器 275
17.7 目录结构 276
17.8 运行应用 276
17.9 本章小结 277

第 18 章 多页应用脚手架 278

18.1 项目结构 278
18.2 开发步骤 279
18.2.1 初始化项目与安装依赖 279
18.2.2 配置 280
18.3 业务代码 282
18.4 本章小结 283

第 19 章 性能优化 284

19.1 限定 Webpack 处理文件范围 284
19.2 DllPlugin 285
19.3 HappyPack 287

19.4 Tree-Shaking 288
19.5 按需加载 289
19.6 提取公共代码 289
19.7 热更新 290
19.8 本章小结 290

第 20 章 编写自定义 Loader 291

20.1 基本 Loader 291
20.2 Loader 选项 293
20.3 异步 Loader 294
20.4 "Raw" Loader 295
20.5 读取 Loader 配置文件 295
20.5.1 项目结构 295
20.5.2 执行构建 297
20.6 本章小结 297

第 21 章 编写自定义插件 298

21.1 基本构建流程 298
21.2 插件示例 299
21.3 Compiler 与 Compilation 对象 299
21.4 Tapable 300
21.5 常用操作 301
21.5.1 读取输出资源、模块及依赖 301
21.5.2 修改输出资源 302
21.6 插件编写实例 302
21.6.1 生成清单文件 303
21.6.2 构建结果上传到 CDN 304
21.7 本章小结 306

第一部分　Node.js基础篇

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境，用于构建高速、可伸缩的网络应用程序。

在这一部分，我们将学习 Node.js 的原理、应用场景、开发环境的搭建和 Node.js 的基础知识。

第 1 章

Node.js 概述

本章内容

- Node.js 是什么
- Node.js 的模块架构
- Node.js 的运行原理
- Node.js 的应用场景

1.1 Node.js 是什么

你很有可能已经听说过 Node.js，甚至已经用上了。的确，近年来不管是 Web/服务端还是桌面/移动应用的开发都可以看到它的身影。它在 GitHub 上拥有 64000 多颗星（Star），拥有 2500 多位贡献者，官方的包管理平台 NPM 下拥有高达 100 多万的软件包（Package）。所有这些都表明 Node.js 的强大活力。

官网上（https://nodejs.org）给 Node.js 的定义是：

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境，用于构建高速、可伸缩的网络应用程序。Node.js 采用的事件驱动、非阻塞 I/O 模型，使它既轻量又高效，是构建可扩展的网络应用程序的完美选择。

1. JavaScript 引擎

JavaScript 引擎是一个专门处理 JavaScript 脚本的虚拟机，一般会附带在网页浏览器之中。目前为止一些知名的 JavaScript 引擎如下：

- V8 引擎：Chrome 浏览器采用的 JavaScript 引擎，也是第一个使用 JIT 技术的引擎。

- JavaScriptCore：Safari 浏览器采用的 JavaScript 引擎。
- Chakra：IE/Edge 浏览器采用的 JavaScript 引擎。
- SpiderMonkey：Mozilla Firefox 浏览器采用的 JavaScript 引擎。

为什么会有这么多的 JavaScript 引擎呢？

主要是因为历史的原因，1995 年，网景（NetScape）公司开发了一种运行在浏览器的脚本语言，最初命名为 Mocha，后来改名为 LiveScript，临近发布的时候为了蹭一蹭 Java 的热度，最终命名为 JavaScript。直到 1997 年，JavaScript 才被欧洲计算机制造商协会（ECMA）进行了标准化，标准编号为 ECMA-262，标准化后的名称为 ECMAScript。

ECMAScript 只是一种语言规范，满足该规范的语言都可以称为“ECMAScript 兼容”。现在比较出名的是 JavaScript 语言，早些年还有 ActionScript（Flash 使用的脚本语言）。

2. V8 引擎

V8 是一个由 Google 公司开发的开源 JavaScript 引擎，使用 C++编写，用于 Google Chrome 浏览器和 Google Chromium 浏览器。

V8 在运行之前将 JavaScript 编译成了机器代码，而非字节码或是解释执行它，以此提升性能。此外，V8 还使用了如内联缓存（Inline Caching）等技术来提高性能。有了这些功能，JavaScript 程序在 V8 引擎上的速度可以媲美二进制编译程序的执行速度。

Node.js 也因为采用了 V8 引擎才有如此高的 JavaScript 执行效率。

3. 事件驱动

事件驱动程序设计（Event-Driven Programming）是一种计算机程序设计模型。这种模型的程序运行流程是由用户的操作（如鼠标的按键、键盘的按键操作）或者是由其他程序的消息来驱动的。

事件驱动的程序至少会有一个事件队列，当有新的请求进来时会被插入到队列中，然后通过循环来检测队列中的事件，当发现有一个事件发生时就会调用回调函数。

Node.js 的 JavaScript（JS）线程是单线程运行的，通过一个事件循环（Event Loop）来循环取出消息队列（Event Queue）中的消息进行处理，处理过程基本上就是去调用该消息对应的回调函数。

4. 非阻塞 I/O

I/O 模型的一种，与之相对的还有阻塞 I/O。

简单来说，阻塞 I/O 就是进行 I/O 操作的时候，进程会被阻塞，直到 I/O 操作完成后进程才会解除阻塞状态继续执行。而非阻塞 I/O 在进行 I/O 操作时，进程不会被阻塞，进程继续执行；如果需要知道 I/O 操作的结果，可以通过轮询的方式。

5. Node.js 的模块架构

Node.js 核心基于 libuv 框架，由 C/C++编写。图 1-1 所示是 Node.js 的模块架构图。

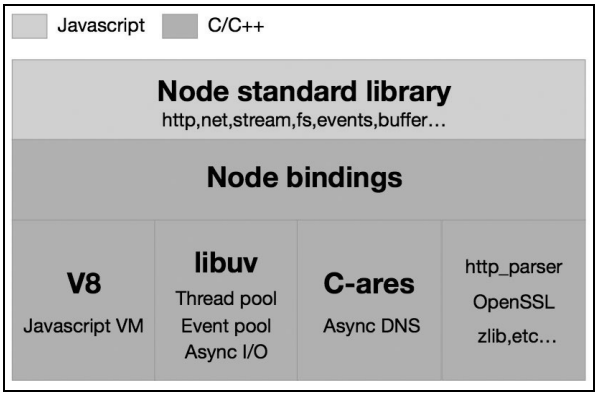

图 1-1

6. Node standard library（Node 标准库）

Node.js 标准库由 JavaScript 编写而成，它提供网络、文件、事件等操作，可以认为是一层比较薄的 API 封装层，实际的操作还是由底层来完成。

7. Node bindings（Node 绑定层）

Node.js 底层使用了大量的 C/C++代码，是高性能的保证。不过，JavaScript 代码是怎么和这些 C/C++代码相互调用的呢？这里不是使用了好几种语言吗？

一般来说，不同语言之间的规范不同，所以写出来的代码无法直接沟通，这时候就需要 Bindings 层，它是一些胶水代码，能够把不同的语言绑定在一起使其能够相互沟通。在 Node.js 中，Bindings 层所做的就是把底层的 C/C++接口暴露给 JavaScript 环境，从而打通 JavaScript 与 C/C++之间的相互调用。

8. libuv

libuv 是提供异步功能的 C 库。它在运行时负责一个事件循环（Event Loop）、一个线程池、文件系统 I/O、DNS 相关的 I/O 和网络 I/O，以及一些其他重要功能。

9. 其他的 C/C++组件和库

如 C-ares、http_parser、OpenSSL 以及 zlib 等，这些依赖提供了对系统底层功能的访问，比如网络、压缩、加解密，等等。

1.2　Node.js 的运行原理

我们已经了解了 Node.js 组成部分各自的面貌。现在来看看它们是如何相互协作以提供强大的网络服务功能。Node.js 系统如图 1-2 所示。

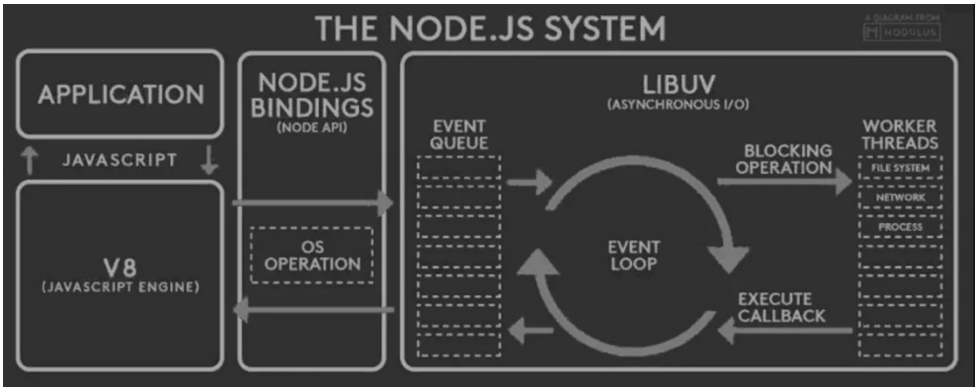

图 1-2

Node.js 应用启动时，会开启 JS 线程（主线程）、由 libuv 提供的线程池（Worker Threads）和一个事件循环（Event Loop）。JS 线程负责执行应用代码，当发现有 I/O 操作时，直接提交给 libuv 的线程池并注册回调函数，不会等待 I/O 结束后再继续运行，而是拿到一个状态后继续执行，这就是“单线程非阻塞 I/O”。

I/O 操作结束之后会有一个事件，该事件会放在事件队列（Event Queue）中，事件循环每次都会检查是否有事件需要处理，如果有就处理，否则进行下一轮轮询；如果没有任何事件需要处理则退出进程。这就是“事件驱动”。

在 I/O 密集型应用中，主线程只负责提交任务，轮询结果，耗时的任务执行部分会提交给底层执行，这就是 Node.js 为什么会有如此高性能的原因。

1.3 Node.js 的应用场景

1.3.1 Node.js 优缺点

在介绍应用场景之前，需要了解一下 Node.js 的利弊。只有在合适的场景下使用 Node.js，才能达到高性能。

Node.js 优点：

- 事件驱动、异步编程，在 I/O 密集型场景下有着极高的性能。
- 轻量高效，资源占用率低。
- 使用 JavaScript（JS）作为应用层语言，语言门槛低，对于拥有 JS 基础的人来说，几乎没有门槛。

Node.js 缺点：

- 单进程，一旦 JS 线程出现未处理的错误，进程会退出，服务会终止。
- 单线程（特指 JS 线程），一旦 JS 线程上出现耗时的 CPU 计算（加解密之类的计算），JS

线程将出现阻塞，会拖慢事件轮询。

1.3.2　应用场景

由于较低的语言门槛以及强大的 I/O 处理能力，大致有以下场景是 Node.js 能够胜任的。

1. Restful API

这是 Node.js 最理想的应用场景，可以处理数万条连接，本身没有太多的逻辑，只需要调用请求 API、组织数据进行返回即可。它本质上只是从某个数据库或缓存中查找一些值并将它们组成一个响应。由于响应是少量的文本，入站请求也是少量的文本，因此流量不高，甚至一台机器也可以处理最繁忙的 API 需求。社区有 Koa、Express、hapi 等框架提供该功能。

2. 实时 WebSocket 应用

大量用户同时在线，互相收发数据，但是几乎不需要对数据进行处理，Node.js 只需要接收数据然后转发。社区有著名的 socket.io 库来提供 WebSocket 功能。

3. 前端工具链

由于采用的语言是 JavaScript，并且拥有大量的第三方模块，在前端工程师手里可以开发一整套前端工具链，包括压缩、混淆、模块化等。比如业界著名的 Webpack。

4. 桌面开发

基于开源的 Chromium 和 Node.js，开发者可以通过 HTML/CSS/JS 来构建桌面端应用程序，业界著名的有 Electron 和 node-webkit。

1.4　本章小结

Node.js 并不是银弹（Silver Bullet），只能解决特定场景下的问题。Node.js 比较有意思的一点是，很多进入 Node.js 世界的是客户端 JavaScript 程序员，此外还有一些使用其他语言进行服务端开发的程序员。不管你是做什么的，我们都希望你能够知道 Node.js 到底适不适合你，适合你的哪些业务。

回顾一下本章所学：

- Node.js 是事件触发和非阻塞的。
- Node.js 专为 I/O 密集型应用而设计。
- Node.js 的模块架构和运行原理。

第 2 章将介绍如何在 Windows/Linux/macOS 系统下安装 Node.js 及其开发环境。

第 2 章

搭建 Node.js 开发环境

本章内容

- Node.js 的安装
- 使用 VSCode 搭建 Node.js 开发环境
- 编写 HTTP 服务器

2.1 安装 Node.js

为了保证下载速度，本文所用的下载地址为淘宝 NPM 镜像源中的 Node.js 二进制镜像。

本书写作时最新的 Node.js 稳定版的版本号为 12.12.0，各操作系统采用的安装都以该版本作为示例。

一般来说，按照书中的步骤来安装不会出现问题，如果出现问题，请把截图反馈给作者公众号或者 GitHub。

2.1.1 Windows 上安装 Node.js

32 位下载地址：https://npm.taobao.org/mirrors/node/latest-v12.x/node-v12.12.0-x86.msi。

64 位下载地址：https://npm.taobao.org/mirrors/node/latest-v12.x/node-v12.12.0-x64.msi。

大家可以根据系统位数来选择对应的安装包进行下载与安装。

2.1.2　Linux 安装 Node.js

Node.js 官方发布的 Linux 通用二进制文件，支持主流的 Linux 发行版（Ubuntu、CentOS 等等）。新版本的 Node.js 发行版不再提供 32 位版本。

64 位下载地址：https://npm.taobao.org/mirrors/node/latest-v12.x/node-v12.0.0-linux-x64.tar.gz。

推荐使用/opt 目录来部署已经编译好的二进制包，安装步骤如下：

```
# 下载文件
wget https://npm.taobao.org/mirrors/node/latest-v12.x/node-v12.12.0-linux-x64.
    tar.gz
# 解压并移动到指定目录
mv node-v12.12.0-linux-x64.tar.gz /opt
tar xf node-v12.12.0-linux-x64.tar.gz
mv node-v12.12.0-linux-x64 /opt/nodejs
# 编辑 ~/.bashrc 文件，添加 Node.js 到 PATH 环境变量中
echo 'PATH=/opt/nodejs/bin:$PATH' >> ~/.bashrc
# 更新环境变量
source ~/.bashrc
# 测试安装结果
node -v
npm -v
# 设置 NPM 包镜像源为 taobao
npm config set registry https://registry.npm.taobao.org
```

2.1.3　Ubuntu 安装 Node.js

Ubuntu 官方仓库中提供了 Node.js 和 NPM 包，我们可以直接使用 apt-get 来进行安装。安装步骤如下：

```
sudo apt-get install nodejs npm
# 测试安装结果
node -v
npm -v
# 设置 NPM 包镜像源为 taobao
npm config set registry https://registry.npm.taobao.org
```

2.1.4　CentOS 安装 Node.js

CentOS 官方仓库中也提供了 Node.js 和 NPM 包，但是版本一般比较旧，建议使用上节中 Linux 安装 Node.js 的方式进行安装。安装步骤如下：

```
yum install nodejs -y
# 测试安装结果
node -v
npm -v
```

```
# 设置 NPM 包镜像源为 taobao
npm config set registry https://registry.npm.taobao.org
```

2.1.5 macOS 安装 Node.js

1. pkg 方式

Node.js 官方提供了适用于 macOS 系统的 pkg 安装包，我们直接下载进行安装即可。

64 位下载地址：https://npm.taobao.org/mirrors/node/latest-v12.x/node-v12.12.0.pkg。

安装完毕后需要设置以下 NPM 包的镜像源为淘宝：

```
npm config set registry https://registry.npm.taobao.org
```

2. Homebrew 方式

如果你的计算机中安装了 Homebrew 环境，直接使用以下命令安装即可：

```
brew install node
# 设置 NPM 包镜像源为 taobao
npm config set registry https://registry.npm.taobao.org
```

2.2 安装 VSCode 编辑器

Visual Studio Code（简称 VSCode）是由微软开发的、同时支持 Windows/Linux/macOS 操作系统的开源编辑器。它支持测试功能，并且内置了 git 功能，提供了丰富的语言支持与常用编程工具。

本节将介绍如何在本地计算机上搭建 VSCode 开发环境，以下步骤使用 macOS 版本作为示例，其他操作系统类似。

（1）打开官方网站 https://code.visualstudio.com/，单击蓝色按钮下载即可。

（2）新版本的 VSCode 不再内置中文语言包，需要安装语言包扩展。安装 VSCode 后打开 VSCode 编辑器，在扩展窗口中搜索“Chinese”，安装第一个即可，如图 2-1 所示。

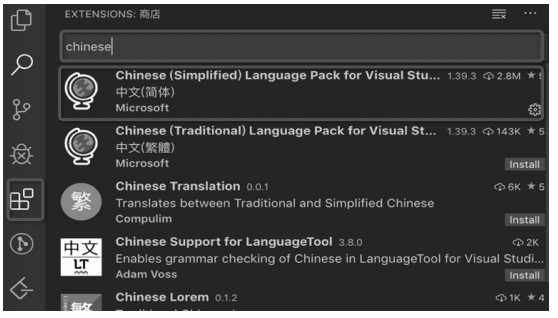

图 2-1

（3）VSCode 内置 Typescript 支持，所以我们开发的时候可以选择使用 JavaScript 语言开

发，同时导入 Node.js 的 Typescript definitions 声明文件（类似于 C 语言的头文件），可以获得代码自动提示的功能。

至此，基于 VSCode 的 Node.js 开发环境已经搭建完毕，如果你遇到疑问或者有搭建上的问题，可以在作者公众号或者 GitHub 上进行咨询。

2.3　编写 HTTP 服务器

在编写第一个 Node.js 应用前，先了解一下 Node.js 应用（本节指服务端的应用）是由哪些部分组成的：

（1）通过 require 或 import 导入依赖模块。
（2）创建服务器并设置好事件回调。
（3）启动服务器。

使用 VSCode 编辑器新建 app.js 文件并写入以下代码：

```
const http = require('http');  // 导入 http 模块

const hostname = '127.0.0.1';  // HTTP 服务器监听地址
const port = 3000;             // HTTP 服务器监听端口

const server = http.createServer((req, res) => {   // 事件回调
  res.statusCode = 200;        // 设置 HTTP 响应状态码
  res.setHeader('Content-Type', 'text/plain');     // 设置响应内容格式
  res.end('Hello World\n');    // 输出内容并结束此次响应
});

server.listen(port, hostname, () => {              // 运行服务器
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

单击如图 2-2 所示的菜单选项打开 VSCode 内置的终端：

图 2-2

输入 node app.js 即可启动 HTTP 服务器，如果出现启动失败，一般是端口被占用的原因，只要更换监听的端口即可，如图 2-3 所示。

图 2-3

打开浏览器访问 http://127.0.0.1:3000，即可看到如图 2-4 所示的界面。

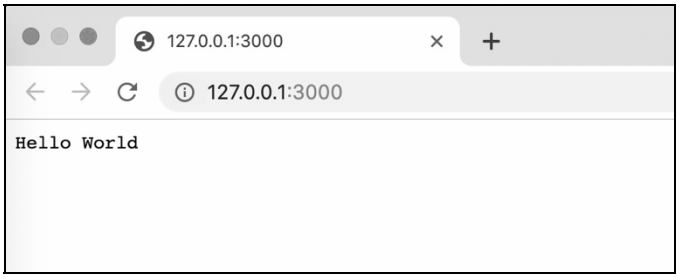

图 2-4

2.4 本章小结

Node.js 的服务器开发过程比传统的 Web 服务器开发要简洁很多，底层为了处理大量关于 HTTP 协议的实现细节，只提供了基本且必要的 API 给 JS 层，这样我们就可以专注业务的开发，而不用消耗时间和精力来处理协议层。

回顾一下本章所学：

- 主流操作系统下 Node.js 的安装。
- VSCode 开发环境的搭建流程。
- Node.js 应用的开发流程。

有的读者看到本章的 HTTP 模块、事件的编写方式可能会存在疑问，没关系，有疑问是正常的，因为 Node.js 的大门才刚刚开启。

第 3 章将重点学习 Node.js 编程基础。

第 3 章

Node.js 编程基础

本章内容

- NPM 包管理器介绍
- package.json 文件介绍
- Node.js 的模块系统
- Node.js 的异步编程方式
- Node.js 常用模块

3.1 NPM 包管理器介绍

Node.js 中包管理器是用来管理 Node.js 软件包的工具，类似于 Java 的 Maven 或者 PHP 的 Composer。

NPM（Node.js Package Manager）是 Node.js 默认的包管理工具，能够解决 Node.js 开发和部署中软件包依赖的问题。常见的使用场景有以下几种：

- 从 NPM 服务器下载别人编写的第三方包到本地进行使用。
- 将自己编写的软件包上传到 NPM 服务器供他人使用。

3.1.1 更换 NPM 镜像源

在国内访问 NPM 默认的中央仓库速度比较慢，可以更换为淘宝提供的 NPM 镜像源以加快软件包的安装。

在终端按需执行以下命令：

```
# 设置镜像源为淘宝
npm config set registry https://registry.npm.taobao.org
# 还原官方镜像源
npm config delete registry
```

使用淘宝镜像之后无法使用 publish 和 unpublish 命令，如果需要发布软件包和撤销发布的软件包，则需要还原为官方镜像。

3.1.2 初始化项目

在项目目录下执行 npm init 命令，依照提示输入问题的答案之后即可创建一个标准的 npm 模块，同时会生成一个 package.json 文件，其中记录了当前的模块名、版本、依赖等信息。

3.1.3 使用 npm 命令安装模块

安装 Node.js 模块的 npm 命令语法如下：

```
npm install <模块名称>
```

如果需要使用常用的 Koa 框架进行开发，则可以使用如下命令进行安装：

```
npm install koa
```

该命令执行完毕之后，Koa 模块就会出现在当前目录下的 node_modules 文件夹中，使用如下代码即可使用该模块：

```
const koa = require('koa');
```

3.1.4 本地安装与全局安装

npm 的模块安装分为全局安装和本地安装，默认为本地安装，如果需要全局安装，则要使用-g 参数：

```
npm install express-generator -g
```

1. 本地安装

- 第三方模块将被安装到当前目录的 node_modules 下（如果没有该目录则会自动新建）。
- 通过 require（'模块名'）即可导入本地模块。

2. 全局安装

- 第三方模块将被安装到/usr/local/lib/node_modules 目录或者安装 Node.js 的目录。
- 可以直接在命令行使用。
- 不可以使用 require（'模块名'）来引用。

3.1.5　生产依赖和开发依赖

有一些软件包是开发环境和生产环境都需要的，而有一些只在开发环境使用，比如测试框架。

1. 开发依赖

使用--save-dev 选项即可将软件包安装为开发依赖，依赖信息将被保存到 package.json 的 devDependencies 中。

```
npm install eslint --save-dev
```

2. 生产依赖

使用--save 选项即可将软件包安装为生产依赖，开发环境和生产环境都需要使用，依赖信息将被保存在 package.json 的 dependencies 中。

```
npm install koa --save
```

3.1.6　其他 npm 命令

其他 npm 命令如表 3-1 所示。

表 3-1　其他 npm 命令

命令	说明
npm uninstall <模块名> [-g]	卸载本地/全局模块
npm list [-g]	查看已安装的本地/全局模块
npm update <模块名> [-g]	更新本地/全局模块
npm search <模块名>	搜索模块
npm publish	发布本地模块到 NPM 仓库
npm unpublish <模块名>@<版本号>	撤销指定版本的模块发布
npm config get <config_name>	读取配置
npm config set <config_name>	设置配置
npm config delete <config_name>	删除配置
npm run <命令>	执行 package.json 中 scripts 定义的命令

3.2 Yarn 包管理器介绍

Yarn 是 Facebook 发布的一款 NPM 包管理工具，解决了 NPM 包下载速度慢、每次都要全量下载、版本号依赖混乱等问题。

3.2.1 安装 Yarn

Yarn 是命令行工具，需要安装到全局模块中，在终端执行以下命令即可：

```
npm install yarn -g
```

3.2.2 Yarn 常用命令

Yarn 常用命令如表 3-2 所示。

表 3-2 Yarn 常用命令

命令	说明
yarn [global] remove <模块名>	卸载本地/全局模块
Yarn [global] upgrade <模块名>	更新本地/全局模块
yarn [global] add <模块名>	安装本地/全局模块
yarn config get <config_name>	读取配置
yarn config set <config_name>	设置配置
yarn config delete <config_name>	删除配置
yarn run <命令>	执行 package.json 中 scripts 定义的命令

笔者推荐各位读者使用 Yarn 来进行软件包依赖的管理和下载。

3.3 解读 package.json 文件

package.json 是 Node.js 软件包的元数据（Meta）描述文件，一般由 npm/yarn 命令创建，不需要手动维护。一个典型的 package.json 如下：

```
{
  "name": "example",
  "version": "1.0.0",
  "description": "example Node.js package",
  "main": "index.js",
  "scripts": {
    "test": "NODE_ENV=test mocha -t 25000"
  },
  "repository": {
    "type": "git",
    "url": "git://github.com/xxx/xxx.git"
  },
  "bugs": {
    "url": "https://github.com/xxx/issues"
  },
  "keywords": [
    "storage"
  ],
  "author": "xialeistudio@gmail.com",
  "contributors": [
  ],
  "dependencies": {
    "urllib": "^2.34.1"
  },
  "devDependencies": {
    "mocha": "^6.2.1",
    "should": "^13.2.3"
  },
  "license": "MIT"
}
```

3.3.1　package.json 字段说明

package.json 字段说明如表 3-3 所示。

表 3-3　package.json 字段及其说明

字段名称	字段说明
name	软件包名称
version	软件包版本
description	软件包简介
main	软件包入口
scripts	项目命令
repository	仓库地址
bugs	bug 反馈页面

（续表）

字段名称	字段说明
keywords	软件包关键字
author	软件包作者
contributors	软件包贡献者
dependencies	生产依赖
devDependencies	开发依赖
license	开源协议

3.3.2 版本号说明

NPM 包使用语义化的版本号来管理代码，版本号格式为 X.Y.Z，分别代表主版本号、次版本号和补丁版本号。当代码有修改时，需要按照以下规则执行版本号的变更：

- 只是修复 bug，更新 Z 位。
- 只是新增功能，但是向下兼容（旧 API 不受影响），更新 Y 位。
- 向下不兼容的改动，更新 X 位。

3.3.3 常见的版本号限定符

在 package.json 中会见到类似^0.1.0，~0.1.0,0.1.0 和>=0.1.0 之类具有不同限定符的版本号，为了避免混淆，这里做一下说明。

- ^0.1.0：支持 0.1.0～1.0.0（不含）之内的所有版本。
- ~0.1.0：支持 0.1.0～0.2.0（不含）之内的所有版本。
- 0.1.0：只能使用 0.1.0 版本。
- >= 0.1.0：支持大于等于 0.1.0 之后的所有版本。
- *：任意版本。

3.4 Node.js 的模块系统

Node.js 应用由模块组成，采用 CommonJS 模块规范。

每个 JS 文件就是一个模块，有独立的作用域。在一个文件中定义的变量、函数、类在其他文件都不可见。

```
// example.js
function sum(a, b) {
```

```
  return a + b;
}
```

上述代码中 sum 函数只有 example.js 中能调用，其他文件则不可以调用。

3.4.1　module 和 exports

1. 基本用法

CommonJS 规范规定，每个模块内部的变量 module 代表当前模块。

module 是一个对象，module.exports 是对外的接口。加载模块实际上是读取该模块的 module.exports 属性。

module.exports 也是一个对象，所有需要导出的变量、函数、类都需要挂载到该对象上才能实现导出。

```
// example.js
function sum(a, b) {
  return a + b;
}

module.exports.sum = sum;
```

为了方便起见，Node.js 为每个模块提供了一个 exports 变量，在同一个模块中，module.exports 和 exports 是恒等的(类型和值都相等)。因此在实际开发中，建议通过 exports.xxx 的形式导出变量、函数、类。

不建议更改 exports 的指向，否则模块将不能正常导出。

下列代码无法导出 sum 函数，因为 exports 由于重新赋值导致指向被更改。

```
// example.js
exports.sum = function(a, b) {
  return a + b;
}
exports = 'Hello World';
```

如果模块只需要导出一个变量、函数、类，只能对 module.exports 进行赋值，对 exports 进行赋值达不到如期作用。

上面讲到 module.exports 是恒等于 exports 的，为什么对 exports 进行赋值达不到如期作用呢？

```
// a.js
exports = 'hello world';
// b.js
module.exports = 'hello world';
// c.js
const hello = require('./a');
console.log(hello); // {}

const hello2 = require('./b');
```

```
console.log(hello2); // hello world
```

2. 原理解读

exports 只是一个别名，类似于下面的代码：

```
var exports = module.exports;
```

当不对 exports 重新赋值时，exports 指向不变，exports.xxx 也会如期地添加到 module.exports 中。

当对 exports 重新赋值时，exports 和 module.exports 关联就不存在了，修改 exports 不会对 module.exports 产生作用。

3.4.2 require

1. 基本用法

CommonJS 规定 require 用于加载模块文件。

require 读取并执行一个 JS 模块，然后返回该模块的 exports 对象。如果模块未找到，则会抛出错误。

```
// math.js
exports.sum = function(a, b) {
  return a + b;
}

// index.js
const math = require('./math');
math.sum(1, 1);
```

2. 加载规则

加载模块时模块扩展名为.js。也就是说下列代码是一样的：

```
const math = require('./math');
const math2 = require('./math2');
```

根据传入的参数，require 会有不同的规则（以下规则无先后顺序）：

（1）如果参数以'/'开头，则表示需要加载的是一个绝对路径的 JS 文件。如 require('/home/xialei/math')将加载/home/xialei/math.js。

（2）如果参数字符串是以'./'开头，则表示需要加载一个相对路径的模块文件。如 require('./math')将加载位于当前模块同目录下的 math.js 文件。

（3）如果参数不以'./'或'/'开头，则需要加载的是核心模块或者当前工作目录中 node_modules 下的模块。

（4）如果没有找到指定的模块文件，Node.js 会尝试自动添加.js、.json、.node（编译后的二进制模块）后再去搜索。

（5）如果传入的参数解析之后是一个目录，Node.js 会自动读取该目录下的 package.json 文件，根据 main 字段来加载真正的入口文件。如果该目录下没有 package.json 文件，则尝试

加载 index.js 或 index.node。

3.4.3　开发一个自定义模块

我们将开发一个与时间操作相关的函数模块来巩固本节所学，示例代码如下。

date.js:

```
exports.formatTime = function (timestamp) {
    const duration = parseInt(Date.now() / 1000, 10) - timestamp;
    if (duration < 600) {               // 不足 10 分钟
        return '刚刚';
    }
    if (duration < 3600) {              // 不足 1 小时
        return '1 小时内';
    }
    if (duration < 3600 * 3) {          // 不足 3 小时
        return '3 小时内';
    }
    if (duration < 3600 * 24) {         // 不足 24 小时
        return '今天';
    }
    if (duration < 3600 * 24 * 2) {     // 不足两天
        return '1 天前';
    }
    return new Date(timestamp * 1000).toString();
}
```

index.js:

```
const date = require('./date');

const now = parseInt(Date.now() / 1000, 10);

console.log(date.formatTime(now - 60));
console.log(date.formatTime(now - 600));
console.log(date.formatTime(now - 5400));
console.log(date.formatTime(now - 3600 * 23));
console.log(date.formatTime(now - 3600 * 24));
console.log(date.formatTime(now - 3600 * 24 * 3));
在终端输入以下命令执行 index.js：
node index.js
```

输出如下：

```
刚刚
1 小时内
3 小时内
今天
1 天前
Fri Oct 23 2019 18:07:45 GMT+0800 (中国标准时间)
```

3.5 Node.js 的异步编程风格

在本书的第 1 章介绍过 Node.js 是一个异步运行环境，异步意味着调用函数后结果不是立即返回，而是在未来的时刻再通知给调用者。Node.js 使用最广泛的异步编程风格是基于回调函数来实现的。

截止本书出版时，Node.js 可以使用以下几种异步编程风格：

- 回调函数
- Promise
- async/await

回调函数是最早出现的，也是最烦琐的，实际应用中不推荐以回调函数的形式进行异步调用。原因是多个异步操作有顺序依赖时，会产生如下代码：

```
func1.success(function() {
  func2.success(function() {
    func3.success(function() {

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

3.5.1 回调函数

Node.js 异步编程是通过回调函数来实现的，但不能说使用回调函数就异步了。

如下代码是基于回调函数的实现，但不是异步的：

```
function test(callback) {
  callback(1);
}

test(function(data) {
  console.log(data);
});
```

回调函数在完成任务后就会被调用，Node.js 使用了大量回调函数，几乎所有的 API 都支持回调函数。

由于调用接口存在成功或失败的情况，而基于回调函数的编程无法使用标准 JS 中的抛出错误和捕获错误的方法。因此只能将错误对象作为回调参数来调用回调函数。

Node.js 中回调函数的风格是统一的，这样给我们编程带来了很大的方便。异步函数的签名如下：

```
func(param..., callback(Error, data))
```

- param 调用 API 的参数。如读取文件时传递的文件路径，支持多个参数。
- callback(Error, data...) 回调函数。Node.js 中回调函数的第一个参数永远是 Error 对象，之后才是调用成功的结果，如果没有出错，第一个参数为 null。

例如，我们读取位于桌面的 data.txt 文件：

```
const fs = require('fs');  // fs 是 Node.js 核心模块

fs.readFile('~/Desktop/data.txt', function(err, data) {
  if(err) {
    console.log('读取出错', err);
    return;
  }
  console.log(data);
});
```

3.5.2　Promise

1. 基本知识

Promise 对象用于表示一个异步操作的最终完成（或失败）及其结果值。

一个 Promise 有以下几种状态：

- pending: 初始状态
- fulfilled: 操作成功
- rejected: 操作失败

Promise 只会从 pending 转换为 fulfilled 或者 rejected，整个转换只发生一次。

Promise 构造函数接收一个执行函数，该函数接收 resolve 和 reject 两个回调函数，当执行函数运行成功时需调用 resolve，执行错误时需调用 reject。

Promise 构造函数的签名如下：

```
function Promise(function(resolve, reject): Promise {
   // 原来的异步逻辑
});
```

一旦 Promise 发生状态变化，就会触发 then 方法，then 方法签名如下：

```
Promise.prototype.then = function(onFulfilled[, onRejected]): Promise
```

- onFulfilled Promise: 执行成功时回调。
- onRejected Promise: 执行出错时回调，该参数是可选的。
- then 方法: 返回一个新的 Promise 对象，因此 Promise 支持链式调用。

由于 then 的第二个参数 onRejected 参数是可选的，因此 Promise 的原型上提供了 catch 方法来捕获异步错误，catch 方法签名如下：

```
Promise.prototype.catch = function(onRejected): Promise
```

- onRejected Promise：执行出错时回调。
- catch 方法：返回一个新的 Promise 对象。

2. 基本使用

Promise 是为了解决异步编程问题而出现的，因此可以基于 Promise 来优化上文中读取文件的例子：

```
function readFileAsync(path) {
  return new Promise(function(resolve, reject) {   // 实例化 Promise
    fs.readFile(path, function(err, data) {        // 执行原定的异步逻辑
      if(err) {
        reject(err);
        return;
      }
      resolve(data);
    });
  });
}

readFileAsync('~/Desktop/data.txt')
  .then(function(data) {
  console.log(data);
}).catch(function(err) {
  console.error(err);
});
```

3. 链式调用

Promise 的 then 或 catch 回调函数的返回值会作为下一个 then/catch 的输入参数，因此可以通过链式 Promise 来扁平化嵌套的回调函数。

```
// callback.js 回调风格
const fs = require('fs');

fs.readFile('~/Desktop/data.txt', function(err, data) {
  if(err) {
    console.log('读取 data.txt 出错', err);
    return;
  }
  console.log('data.txt 读取成功', data);

  fs.readFile('~/Desktop/data2.txt', function(err, data2) {
    if(err) {
      console.log('读取 data2.txt 出错', err);
      return;
    }
    console.log('data2.txt 读取成功', data2);
  });
});
```

如果需要依次读取两个文件，那么就需要嵌套一层回调函数。如果依赖的异步操作越多，响应的嵌套层级也会越大，给代码的可读性和可维护性带来困难。

```
// promise.js Promise 风格
const fs = require('fs');

function readFileAsync(path) {
  return new Promise(function(resolve,reject) {
    fs.readFile(path, function(err, data) {
      if(err) {
        reject(err);
        return;
      }
      resolve(data);
    });
  });
}

readFileAsync('~/Desktop/data.txt')
.then(function(data) {
  console.log('读取 data.txt 成功', data);
  return readFileAsync('~/Desktop/data2.txt');     // 返回 Promise
})
.then(function(data2) {
  console.log('读取 data2.txt 成功', data2);
})
.catch(function(err) {
  console.log('读取失败', err)
});
```

多个 Promise 链式调用时一旦有一个出错，整个调用链就会终止，然后回调 catch 函数。通过链式调用，困扰多年的 Node.js 回调嵌套问题终于得到了第一次解决。

4. 其他操作

```
Promise.resolve(value)
```

返回一个状态由 value 决定的 Promise 对象。value 有以下几种取值：

- Promise。value 本身是 Promise 的情况下，返回的 Promise 值由 value 这个 Promise 决定。
- 基本类型/空/或不带 then 方法的对象。返回的 Promise 值为 value，状态为 fulfilled。

```
Promise.reject(reason)
```

返回一个状态为失败的 Promise，通常情况下会传递 Error 对象作为 reason。

```
Promise.all(promises)
```

接收一个 Promise 数组，返回一个新的 Promise 对象。

- Promise 数组中所有 Promise 都成功执行的情况下，返回的 Promise 最终会触发成功。

- Promise 数组中只要有一个 Promise 执行失败，返回的 Promise 最终会执行失败。

```
Promise.race(promises)
```

接收一个 Promise 数组，返回一个新的 Promise 对象。

当 Promise 数组中任意一个 Promise 执行成功或失败，返回的 Promise 则立即成功或失败。

3.5.3 async/await

async 和 await 关键字是 ES2017 中新添加的关键字，本质上是 Promise 的语法糖，使得能够像同步代码一样编写异步代码。

async/await 本质是语法糖，因此尽管编程风格与同步类似，但是不会阻塞 JS 线程。

下面使用 Promise 小节中依次读取两个文件的需求为例：

```
// async.js
const fs = require('fs');

function readFileAsync(path) {
  return new Promise(function(resolve,reject) {
    fs.readFile(path, function(err, data) {
      if(err) {
        reject(err);
        return;
      }
      resolve(data);
    });
  });
}

// 依次读取文件
async function readFiles() {
  try {
    const data1 = await readFileAsync('~/Desktop/data.txt');
    const data2 = await readFileAsync('~/Desktop/data2.txt');
    console.log('文件 1 内容', data1);
    console.log('文件 2 内容', data2);
  } catch(e) {
    console.error('读取失败', e);
  }
}

readFiles();
```

可以看到 readFiles 函数中的代码跟同步编程风格一致（忽略 async/await 关键字）。

针对 Promise 调用链太长的问题，async/await 提供了一个优美的解决方案，确实让异步编程变得简单了。

基本语法

（1）async

async 只能放在函数声明之前，支持普通函数、箭头函数和类函数。被修饰的函数不管返回什么值，最终都会返回 Promise。

- 函数返回基本值/空/或不带 then 方法的对象时，Promise 的结果为该值，状态 fulfilled。
- 函数抛出错误时，Promise 状态为 rejected，reason 为抛出的错误对象。
- 函数本身返回一个 Promise 时，最终的 Promise 结果为该 Promise 的结果。

```
// 普通函数
async function doSomething(a, b) {
  return a + b;
}
// 箭头函数
const doSomething2 = async (a, b) => {
  return a + b;
};
// 类函数
class Demo {
 async test(a, b) {
   return a + b;
 }
}

const data = doSomething(1, 2); // [object Promise]
doSomething(1, 2).then(function(result) {
  console.log(result); // 3
});
```

由于被 async 修饰的函数最终都会返回 Promise，因此需要使用 then 才可以获得 Promise 的执行结果，直接调用函数只能得到一个 Promise。

（2）await

await 只能在被 async 修饰的函数内部调用，await 可以放在任何返回 Promise 的函数前，Promise 执行成功的情况下，await 语句将返回 Promise 的成功值，Promise 执行错误的情况下，await 语句将抛出错误，通过 try/catch 捕获即可。

示例：包装 XMLHttpRequest

```
function ajaxGet(url, timeout) {
    return new Promise(function (resolve, reject) {
        const xhr = new XMLHttpRequest();
        xhr.open('GET', url, true);
        xhr.timeout = timeout;

        xhr.onload = function () {
            resolve(xhr.responseText);
        }
        xhr.onerror = function (e) {
```

```
            reject(new Error('请求失败'));
        };
        xhr.ontimeout = function () {
            reject(new Error('timeout'));
        };

        xhr.send(null);
    });
}

async function getData() {
    try {
        const data = await ajaxGet('xx://a.json', 1000); // 1 秒超时
        console.log(data);
    } catch (e) {
        console.error(e);
    }
}
getData();
```

几乎所有 callback 类型的异步函数都可以包装为 Promise（特殊情况就是 callback 有多个返回值的情况，Promise 不能直接处理，但是可以将多个返回值包装为一个数组）。

3.6 Node.js 常用核心模块

学习 Node.js 必须掌握其核心模块，就像学习 JavaScript 必须掌握函数、对象、数据类型、DOM、BOM 等知识一样。

Node.js 的核心模块很多，这里介绍几个比较常用的：events、fs、stream、http 模块。

3.6.1 events 模块

事件驱动、非阻塞 IO 是 Node.js 的特点，所以事件模块是非常重要的模块。Node.js 绝大多数模块都继承了 events。

基于事件的编程是典型的发布-订阅模式的实现，有效地解耦了发布者和订阅者，发布者只需要关心事件的触发，不需要关心触发之后的逻辑，而订阅者只关注订阅，不需要关注事件是由谁来触发。

1. 基本使用

使用事件模块有以下几个步骤：

（1）实例化事件监听器实例。

（2）注册事件。

（3）触发事件。

```
const EventEmitter = require('events');
// 实例化事件监听器
const emitter = new EventEmitter();

// 注册事件
emitter.on('click', function (param1) {
    console.log('触发了点击', param1);
});

// 触发事件
emitter.emit('click', 'demo');
// 再次触发
emitter.emit('click', 'demo');
```

以上例程输出如下：

```
触发了点击 demo
触发了点击 demo
```

2. 一次性监听

Node.js 中可以对同一个事件进行多次监听，也可以多次触发同一事件。如果需要在事件触发监听函数一次之后不再继续触发，可以调用 once()来进行一次性监听。

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

const emitter = new EventEmitter();

emitter.once('click', function (param1) {
    console.log('触发了点击', param1);
});

emitter.emit('click', 'demo');
emitter.emit('click', 'demo');
```

以上例程输出如下：

```
触发了点击 demo
```

调用 once 函数监听的事件不管触发了多少次事件，都只会执行一次回调函数。

3.6.2　fs 模块

fs 是 File System 的缩写，也就是 Node.js 用来操作系统文件的模块。

1. 读取文件

调用 fs.readFile()函数即可读取文件，函数签名如下：

```
fs.readFile(path[, options], callback)
```

- path：文件路径。
- options：选项。
- callback：回调函数。

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

fs.readFile('./a.js', 'utf8', (err, data) => {
    if (err) {
        console.log('读取失败', err);
        return;
    }
    console.log('读取成功', data);
});
```

2. 写入文件

调用 fs.writeFile()可以写入文件，函数签名如下：

```
fs.writeFile(path, content[, options], callback)
```

- path：文件路径。
- content：文件内容。
- options：选项。
- callback：回调函数。

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

fs.writeFile('./app.log', (new Date()).toString(), { encoding: 'utf8' }, (err) =>
    {
    if (err) {
        console.log('写入失败', err);
        return;
    }
    console.log('写入成功');
});
```

3. 追加内容

如果文件存在时不需要覆盖，可以调用 fs.appendFile()来追加文件内容，函数签名如下：

```
fs.appendFile(path, content[, options], callback)
```

- path：文件路径。
- content：文件内容。
- options：选项。
- callback：回调函数。

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

fs.appendFile('./app.log', (new Date()).toString(), { encoding: 'utf8' }, (err)
    => {
    if (err) {
```

```
        console.log('写入失败', err);
        return;
    }
    console.log('写入成功');
});
```

4. 删除文件

调用 fs.unlink()可以删除文件，函数签名如下：

```
fs.unlink(path, callback)
```

- path：文件路径。
- callback：回调函数。

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

fs.unlink('./aaa', (err) => {
    if (err) {
        console.log('删除失败', err);
        return;
    }
    console.log('删除成功');
});
```

本函数不能删除文件夹，需要调用 fs.rmdir()才可以删除文件夹。

5. 创建文件夹

调用 fs.mkdir()可以创建文件夹，函数签名如下：

```
fs.mkdir(path[, options], callback)
```

- path：文件夹路径。
- options：选项。
- callback：回调函数。

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

fs.mkdir('aaa/bbb', { recursive: true }, (err) => { // recursive 选项在创建多级目录
    时使用
    if (err) {
        console.log('创建失败', err);
        return;
    }
    console.log('创建成功');
});
```

6. 读取文件夹内容

调用 fs.readdir()可以读取文件夹内容，函数签名如下：

```
fs.readdir(path[, options], callback)
```

- path：文件夹路径。
- options：选项。
- callback：回调函数。

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

fs.readdir('.', (err, files) => {
    if (err) {
        console.log('遍历失败', err);
        return;
    }
    console.log(files);
});
```

7. 删除文件夹

调用 fs.rmdir()可以删除文件夹，函数签名如下：

```
fs.rmdir(path, callback)
```

- path：文件夹路径。
- callback：回调函数。

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

fs.rmdir('./aaa', (err) => {
    if (err) {
        console.log('删除失败', err);
        return;
    }
    console.log('删除成功');
});
```

3.6.3 stream 接口

stream 是 Node.js 的一个抽象接口，很多对象都实现了这个接口。比如请求 HTTP 接口时的 request 对象就是一个 stream。

stream 是 EventEmitter 的实例。常用的事件有：

- data：当有数据可读时触发。
- end：当没有更多数据可读时触发。
- error：在读取或写入时发生错误时触发。
- finish：在所有数据写入系统底层时触发。

Node.js 中 stream 有 4 种类型：

- Readable：可读。
- Writable：可写。
- Duplex：可读可写。

- Transform：*数据转换*。

1. 读取流

读取流都可以通过监听 data 和 end 事件来接收数据。比如文件读取流、HTTP 请求读取流都可以用这个方法来读取，不需要关心读取流的类型。

通过流的方式读取文件有个很大的优点：内存占用低，可以以小块的形式处理数据。而直接调用 fs.readFile()之类的函数，如果文件过大，可能会撑爆内存。

```
// a.js
const fs = require('fs');

let data = '';
const stream = fs.createReadStream('./a.js', { encoding: 'utf8' });

stream.on('data', (chunk) => {      // chunk 是本次读取到的数据
    data += chunk;
});

stream.on('end', () => {            // 读取文件完毕
    console.log(data);
});

stream.on('error', (err) => {
    console.error('读取错误', err);
});

console.log('执行完毕');
```

2. 写入流

通过调用写入流的 write()方法来写入数据，调用写入流的 end()来完成写入。当数据写入完毕时会触发 finish 事件。

```
// a.js
const fs = require('fs');

const readStream = fs.createReadStream('./a.js', { encoding: 'utf8' });
const writeStream = fs.createWriteStream('./b.js', { encoding: 'utf8' });

readStream.on('data', (chunk) => { // 读取到小块数据就写入，内存占用低
    writeStream.write(chunk);
});

readStream.on('end', () => {        // 文件读取完毕
    writeStream.end();              // 写入完成
    console.log('读取完成');
});

writeStream.on('finish', () => {
    console.log('写入完成');
```

```
});

readStream.on('error', (err) => {
   console.log('读取错误', err);
});
writeStream.on('error', (err) => {
   console.log('写入错误', err);
});
```

3. 管道流

管道提供了一个读取流到写入流的机制。我们通常用管道从一个流中获取数据并将数据传递到另外一个流中。多个管道可以串行处理，数据会依次流经每个管道。

实际上在上面的写入流示例中，已经体现了管道流的思想。只不过通过调用读取流的 pipe() 方法可以简化该步骤。

```
// a.js
const fs = require('fs');




// 创建读取流
const readStream = fs.createReadStream('./a.js', { encoding: 'utf8' });

// 创建写入流
const writeStream = fs.createWriteStream('./b.js', { encoding: 'utf8' });

readStream.pipe(writeStream);      // 管道操作

writeStream.on('finish', () => {   // 写入完毕
   console.log('写入完成');
});
```

4. 数据转换流

数据转换流本质还是利用管道进行处理，将读取流通过 pipe()操作传入转换流，以达到数据转换的目的。

比如我们需要将一个文本文件通过 gzip 压缩后保存到磁盘，可以使用下面的代码。

```
// a.js
var fs = require("fs");
var zlib = require('zlib');

fs.createReadStream('./a.js')  // 创建读取流
  .pipe(zlib.createGzip())     // 创建 gzip Transform 流
  .pipe(fs.createWriteStream('a.js.gz'));  // 创建写入流

console.log("文件压缩完成。");
```

3.6.4　http 模块

http 模块主要用于创建 HTTP 服务器处理请求、创建 HTTP 客户端发出请求。

1. http 客户端

http 客户端可以向指定的 URL 发出请求，回调一个读取流，可以通过操作读取流进行数据输出或者保存等操作。

下面以保存笔者博客所在的头像文件为例：

```
// 头像链接：https://www.ddhigh.com/images/logo.svg

// 博客是 https 协议，使用 https 模块。如果是 http 协议，则使用 http 模块，其他代码相同
const https = require('https');
const fs = require('fs');

const req = https.request('https://static.ddhigh.com/blog/2019-09-18-094336.jpg',
    (response) => {
   console.log('响应状态码', response.statusCode);
   response.pipe(fs.createWriteStream('logo.jpg')); // 通过管道流保存图片
});

req.end(); // 发出请求
```

2. http 服务器

http 服务可以说是 Node.js 最为常见的一种服务类型，本节中只简单做个示例。在实际开发中，很少直接使用 Node.js 内置的 http 模块进行开发，一般都是使用 Web 框架。

```
const http = require('http'); // 导入 HTTP 模块




// 创建 HTTP 服务器，req 为请求对象，resp 为响应对象
const server = http.createServer((req, resp) => {
   resp.end(JSON.stringify(req.headers));
});

server.listen(8080, () => console.log('listen on 8080')); // 监听端口
```

执行该 JS 之后会监听本机 8080 端口，通过浏览器访问可以得到如图 3-1 所示的输出。

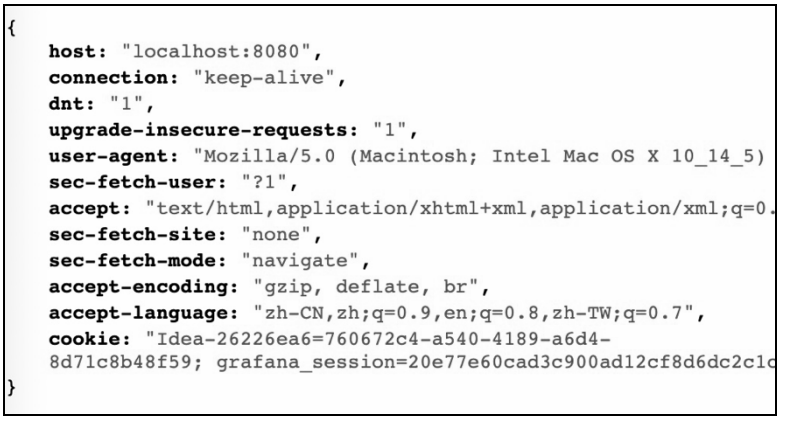

图 3-1

3.7 本章小结

本章介绍了 Node.js 编程基础，这些内容偏基础一点，主要是为了让初学者了解并熟悉 Node.js 应用的开发流程和核心模块的使用。

回顾一下本章所学：

- Node.js 包管理器 NPM 和 Yarn 基本功能的使用。
- Node.js 的模块系统。
- Node.js 异步编程的三种形式：callback/Promise/async && await。
- Node.js 常用核心模块 events/fs/stream/http。