第 1 章

Node.js 基础与环境搭建

Node.js 是 JavaScript 运行时环境，是一个基于 Google Chrome V8 引擎设计实现的跨平台兼容的、可以运行在服务器端的脚本开发语言。如今，随着全栈开发技术的日益盛行与不断深入，Node.js 逐渐成为前短和后端设计开发的通用标准框架。例如，大多数读者耳熟能详的 Angular、React 和 Vue.js 这三大渐进式前端开发框架，均与 Node.js 有着密不可分的关联关系。本书的重点就是介绍关于 Node.js 与 Vue.js 前端全栈开发的相关技术。

本章主要对 Node.js 框架进行整体介绍，并对其发展历史和相关版本进行详细说明，同时也介绍后续开发中所涉及的基础知识。

通过本章的学习可以：

- 了解Node.js的发展历史和特点。
- 了解V8引擎与Node.js的关系。
- 掌握Node.js的一些应用场景。

1.1 Node.js 基础

本节将讲解 Node.js 简介、发展历史、组织架构、特点以及具体应用等方面的内容。

1.1.1 Node.js 简介

Node.js 是基于 Google Chrome 浏览器内置的 V8 引擎所开发的 JavaScript 运行时环境。它充分利用了 V8 引擎的强大性能，借鉴了其很多的前沿技术（例如：GC 机制、事件驱动、非阻塞的 I/O 模型，等等），保证了 Node.js 的轻量与高效，进而受到了众多开发者的追捧。

Node.js 最显著的特点就是能够运行在服务器端（区别于其他脚本语言），以及良好的多平台兼容性（支持 Windows、Linux、Mac OS X、SunOS 和 FreeBSD 等多种系统平台），使其成为最重要的脚本程序设计语言。

我们都知道，JavaScript 脚本语言需要在浏览器环境下才可以解释执行。而 Node.js 是服务器端的脚本语言，可以直接在后端进行解释执行。下面就是最基本的 Node.js 命令执行方法。

```
node filename.js    //node 命令直接解释执行 filename.js 脚本文件，得到结果
```

由于 Chrome V8 引擎执行 JavaScript 脚本的速度非常快，因此 Node.js 所开发出来的应用程序性能非常好。Node.js 已经成为全栈开发的首选语言之一，并且从它衍生出众多出色的全栈开发框架。Node.js 在全球已经被众多公司使用，包括创业公司 Voxer、Uber，以及知名公司沃尔玛、微软等。它们每天通过 Node.js 处理的请求数以亿计，可以说对于要求苛刻的服务器系统来说，Node.js 也可以轻松胜任。

Node.js 还包括一个完善的社区。在 Node.js 的官方网站 https://nodejs.org/可以找到大量的帮助文档和示例程序，并且 Node.js 还有一个强大的 npm 包管理器。由于其强大的服务端功能，越来越多的人参与到本项目中来，可用的第三方模块和扩展增长迅猛，而且质量也在不断提升，Node 已是全球较大的开源库生态系统之一。

提示：Node.js 并不是一个 JavaScript 应用，而是一个 JavaScript 的运行时环境，其底层由 C++语言编写而成。

1.1.2　Node.js 的发展历史

任何语言或框架都不是一天形成的，而是经过漫长的测试、发布、再测试、再发布的迭代过程，本节将重点介绍一下 Node.js 的发展历史。

Node.js 的创始人就是大名鼎鼎的 Ryan Dahl。Ryan Dahl 其实是学数学的，在 2008 年年末，一个偶然的机会让他知道了 Google 推出的全新的 Chrome 浏览器及其 V8 引擎。而当他了解到，Chrome V8 是一个为了实现更快的 Web 体验而专门制作的 JavaScript 引擎时，非常希望能找到一种语言能够提供先进的推送功能，并集成到自己的网站中去，从而避免采用传统的不断轮询拉取数据的访问方式。

Ryan Dahl 对 C/C++和系统调用非常熟悉，他使用系统调用（用 C）实现消息推送功能。如果只使用非阻塞式 Socket，每个连接的开销都会非常小。在小规模测试中，它能同时处理几千个闲置连接，并可以实现相当大的吞吐量。但是，他并不想使用 C，他希望能采用另外一种漂亮灵活的动态语言。最初他也想使用 Ruby 来写 Node.js，但是发现 Ruby 虚拟机的性能不能满足要求，后来便尝试采用 V8 引擎，所以选择了 C++语言。

2009 年 2 月，Ryan Dahl 首次在自己的博客上宣布准备基于 V8 创建一个轻量级的 Web 服务器并提供一套库。2009 年 5 月，他正式在 GitHub 上发布最初版本的部分 Node.js 包。随后几个月里，有人开始使用 Node.js 开发应用。实践证明，JavaScript 与非阻塞 Socket 配合得相当完美，只需要简单的几行 JavaScript 代码，就可以构建出非常复杂的非阻塞服务器。到了 2010 年年底，Node.js 获得云计算服务商 Joyent 的资助。创始人 Ryan Dahl 加入 Joyent，全职负责 Node.js 的发展。从此以后 Node.js 迅猛发展，并成为一种流行的开发语言。

在官方网站上，Node.js 的版本号是从 0.1.14 开始的，每个发布版本对应不同的 V8 引擎版本和 npm 包管理器版本。截至笔者写作时，最新的 LTS（Long-Term Support，长期支持）版本为 V18.17.1。当然，这期间 Node.js 发生了很曲折的故事，感兴趣的读者可以自行去了解一下。

总结一下，Node.js 的发展大致可以分为如下 4 个阶段。

1. 发展初期

创始人 Ryan Dahl 带着他的团队开发出以 Web 为中心的 Web.js，此时的一切都非常混乱，API 也大多处于试验阶段。

2. 快速发展时期

Node.js 的核心用户 Isaac Z. Schlueter 开发出了奠定 Node.js 如今地位的重要工具——npm 包管理工具。同时，这也是 Schlueter 未来成为 Ryan 接班人的重要条件。之后 Connect、Express、Socket.io 等库的出现，吸引了一大波爱好者加入 Node.js 开发者阵营。CoffeeScript 的出现更是让不少 Ruby 和 Python 开发者找到了学习的理由。期间，以 Node.js 作为运行环境的 CLI 工具涌现出来，其中不乏用于加速前端开发的优秀工具，如 Babel、Less、Sass、UglifyJS、Browserify、Grunt、Gulp 等。在这个阶段，Node.js 的发展势如破竹。

3. 不稳定时期

经过了一大批一线工程师的探索实践后，Node.js 开始进入时代的更迭期，新模式代替旧模式，新技术代替旧技术，新实践代替旧实践。ECMAScript 6（ECMAScript 2015）也开始出现在 Node.js 世界中。ECMAScript 6 的发展越来越明显，V8 也对 ECMAScript 6 中的部分特性实现了支持，如 Generator 等。

4. 稳步发展时期

随着 ECMAScript 6 的发展和最终定稿，出现了大量利用 ECMAScript 6 特性开发的新模块，如原 Express 核心团队开发的 Koa。Node.js 之父 Ryan Dahl 退出 Node.js 的核心开发，转做其他的研究项目。Ryan Dahl 的接任者 Schlueter 负责将 Node.js 一直开发下去并不断进行完善。

1.1.3　Node.js 组织架构

前面介绍了 Node.js 是一个完整的 JavaScript 开发环境，并且是基于 Google 的 Chrome V8 引擎进行代码解释的。它在设计之初就已经被定位用来解决传统 Web 开发语言所遇到的诸多问题，所以 Node.js 有很多其他开发语言所不具备的优点。下面主要介绍 Node.js 的组织架构，如图 1.1 所示。

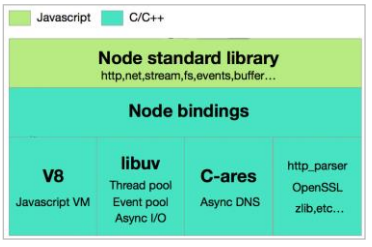

图 1.1　Node.js 系统架构图

从图 1.1 中可以看到，只有最顶层的 Node 标准库（Node standard library）部分是用 JavaScript 语言编写的，其余的底层均是用 C/C++语言编写的。

继续分析图 1.1 中描述的 Node.js 组织架构，关于 Node.js 的结构大致可以分为以下 3 个层次。

1. Node.js标准库

这一层由 JavaScript 编写，是在使用过程中能直接调用的 API。它在 Node.js 源代码中的 lib 目录下可以看到，具体包括 http、net、stream、fs、buffer、events 等模块。

2. Node bindings

这一层是 JavaScript 与底层 C/C++能够沟通的关键，前者通过 bindings 调用后者，相互交换数据。

3. Node基础构件

这一层是支撑 Node.js 运行的基础构件，使用 C/C++语言编写，具体包括以下主要模块。

- V8：Google 推出的 JavaScript VM，也是 Node.js 为什么使用 JavaScript 的关键，它为 JavaScript提供了在非浏览器端运行的环境，它的高性能是Node.js之所以高效的原因之一。
- libuv：为Node.js提供了跨平台、线程池、事件池、异步I/O等能力，是Node.js如此强大的关键。
- C-ares：提供了异步处理DNS相关的能力。
- http_parser、OpenSSL、zlib等：提供了包括HTTP解析、OpenSSL、数据压缩等功能。

1.1.4　Node.js 的特点

Node.js 的强大体现在很多方面，如事件驱动、异步处理、非阻塞 I/O 等。这里将介绍 Node.js 具备的不同于其他框架的特点，包括事件驱动、异步非阻塞 I/O、高性能、单线程等。

1. 事件驱动

在某些传统的网络编程语言中，都会用到回调函数。比如：当 Socket 资源达到某种状态时，注册的回调函数就会执行。在 Node.js 的设计思想中，是以事件驱动为核心的，它提供的绝大多数 API 都是基于事件的、异步的风格。以 net 模块为例，其中的 net.Socket 对象就有 connect、data、end、timeout、drain、error、close 等事件。使用 Node.js 的开发人员，需要根据自己的业务逻辑注册相应的回调函数。这些回调函数都是异步执行的。这意味着虽然在代码结构中这些函数看起来是依次注册的，但是它们并不依赖于自身出现的顺序，而是等待相应的事件触发。

事件驱动的优势在于充分利用了系统资源，执行代码无须阻塞等待某种操作完成，有限的资源可以用于其他的任务。此类设计非常适合后端的网络服务编程，Node.js 的目标也在于此。在服务器开发中，并发的请求处理是一个大问题，阻塞式的函数会导致资源浪费和时间延迟。通过事件注册、异步函数，开发人员可以提高资源的利用率，性能也会得到改善。

2. 异步非阻塞I/O

从 Node.js 提供的支持模块中可以看到，包括文件操作在内的许多函数都是异步执行的，这和传统语言存在区别。为了方便服务器开发，Node.js 的网络模块特别多，包括 http、dsn、net、udp、https、tls 等。开发人员可以在此基础上快速构建 Web 服务器应用。一个异步 I/O 的大致流程如图 1.2 所示。

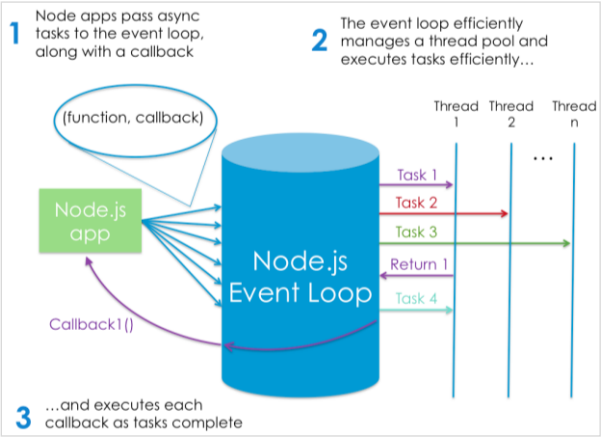

图 1.2　异步 I/O 的流程

异步 I/O 流程主要包括以下过程：

（1）发起 I/O 调用：

① 用户通过 JavaScript 代码调用 Node 核心模块，将参数和回调函数传入核心模块。

② Node 核心模块会将传入的参数和回调函数封装成一个请求对象。

③ 将这个请求对象推入 I/O 线程池等待执行。

④ JavaScript 发起的异步调用结束，JavaScript 线程继续执行后续操作。

（2）执行回调：

① I/O 操作完成后会将结果存储到请求对象的 result 属性上，并发出操作完成的通知。

② 每次事件循环时会检查是否有完成的 I/O 操作，如果有，就将请求对象加入 I/O 观察者队列中，之后当作事件处理。

③ 处理 I/O 观察者事件时，会取出之前封装在请求对象中的回调函数，执行这个回调函数，并将 result 当作参数，以实现 JavaScript 回调的目的。

Node.js 的网络编程非常方便，提供的模块（在这里是 HTTP）开放了容易上手的 API 接口，短短几行代码就可以构建服务器。

3. 性能出众

创始人 Ryan Dahl 在设计的时候就考虑了性能方面的问题，因此选择了 C++和 V8，而不是 Ruby 或者其他的虚拟机。Node.js 在设计上以单进程、单线程模式运行。事件驱动机制是 Node.js 通过内部单线程高效率地维护事件循环队列来实现的，没有多线程的资源占用和上下文切换。这意味着面对大规模的 HTTP 请求，Node.js 是凭借事件驱动来完成的。从大量的测试结果分析来看，Node.js 的处理性能非常出色，在 QPS（每秒查询率）达到 16 700 次时，内存仅占用 30MB（测试环境：RHEL 5.2、CPU 2.2GHz、内存 4GB）。

4. 单线程

Node.js 和大名鼎鼎的 Nginx 一样，都是以单线程为基础的。这正是 Node.js 保持轻量级和高性能的

关键，也是Ryan Dahl设计Node.js的初衷。这里的单线程是指主线程为“单线程”，所有阻塞的部分交给一个线程池处理，然后这个主线程通过一个队列跟线程池协作。我们写的 JavaScript 代码部分不用关心线程问题，代码也主要由一堆回调函数构成，然后主线程在循环过程中适时调用这些代码。

单线程除了保证 Node.js 高性能之外，还保证了绝对的线程安全，使开发者不用担心因为同一变量同时被多个线程读写，而造成的程序崩溃。

1.1.5　Node.js 应用场景

Node.js 可以应用到很多方面，可以说从 Node.js 开始，开发者就可以使用 JavaScript 来开发服务器端的程序了。Node.js 为前端开发者提供了便利，并在各大网站中承担重要角色，成为开发高并发大型网络应用的关键技术。Web 站点早已不局限于内容的呈现，很多交互型和协作型环境也逐渐被搬到了网站上，而且这种需求还在不断增长。这就是所谓的数据密集型实时（data-intensive real-time）应用程序，例如在线协作的白板、多人在线游戏等。这种 Web 应用程序需要一个能够实时响应大量并发用户请求的平台来支撑它们，而这也正是 Node.js 擅长的领域。此外，Node.js 的跨平台特性也是开发人员选择使用 Node.js 语言进行开发的另一大原因。

Node.js 的主要应用场景如下：

- JSON APIs：构建一个Rest/JSON API服务，Node.js可以充分发挥其非阻塞I/O模型以及JavaScript对JSON的功能支持（如JSON.stringfy函数）。
- 单页面、多Ajax请求应用：如Gmail，前端有大量的异步请求，需要服务后端有极高的响应速度。
- 基于Node.js开发UNIX命令行工具：Node.js可以大量生产子进程，并以流的方式输出，这使得它非常适合用作UNIX命令行工具。
- 流式数据：传统的Web应用通常会将HTTP请求和响应看作原子事件，而Node.js会充分利用流式数据的这个特点，构建非常酷的应用，如实时文件上传系统Transloadit。
- 准实时应用系统：如聊天系统、微博系统，但JavaScript是有垃圾回收（GC）机制的，这就意味着系统的响应时间是不平滑的（垃圾回收会导致系统在这一时刻停止工作）。如果想要构建硬实时应用系统，Erlang是一个不错的选择。

例如，实时互动交互比较多的社交网站，像 Twitter 这样的公司，它必须接收 tweets 并将其写入数据库。实际上，几乎每秒就有数千条 tweets 达到，数据库不可能及时处理高峰时段所需的写入数量。Node.js 成为解决这个问题的重要一环。Node.js 能处理数万条入站 tweets。它能快速而又轻松地将它们写入一个内存排队机制（例如 memcached），而另一个单独进程可以在那里将它们写入数据库。Node.js 能处理每个连接而不会阻塞通道，从而能够捕获尽可能多的 tweets。

虽然看起来 Node.js 可以做很多事情，并且拥有很高的性能，但是 Node.js 并不是万能的，有一些类型的应用 Node.js 处理起来可能会比较吃力。例如，CPU 密集型的应用、模板渲染、压缩/解压缩、加/解密等操作都是 Node.js 的软肋。

1.1.6　Node.js 在国内的发展

在 Node.js 初期发展的时候，国内就有大量的开发者开始持续关注了。随着 Node.js 的不断成熟，

很多国内的公司都开始采用这一新技术。Node.js 开发者在国内的数量不断增加，并涌现出很多组织和机构来自发地进行推广和技术分享。

国内的各大视频培训网站上都有 Node.js 开发的培训教程，各大门户网站也都或多或少地采用了 Node.js 的开发技术，比如淘宝、网易、百度等有很多项目就运行在 Node.js 之上。阿里云是这方面比较靠前的公司，它们的云平台率先支持 Node.js 的开发。淘宝也为 Node.js 搭建了国内的 NPM 镜像网站，方便国内的开发者下载各种开发包。

以下是关于 Node.js 中文资源的汇总清单：

（1）Node.js 官方网站：该网站是 Node.js 在国内的官方网站，里面有 Node.js 最新版本的下载资源和丰富的文档资料，是 Node.js 开发爱好者不容错过的网站。网址为 https://nodejs.org/en/。

（2）CNode 社区：该社区由一批热爱 Node.js 技术的工程师发起，已经吸引了很多互联网公司的专业技术人员加入，是目前国内非常具有影响力的 Node.js 开源技术社区。它致力于 Node.js 的技术研究，拥有论坛，并定期组织一些技术交流活动。网址为 https://cnodejs.org。

（3）Node.js 中文网：该网站是一个专业的 Node.js 中文知识分享社区，致力于普及 Node.js 知识，分享 Node.js 研究成果，努力推进 Node.js 在中国的应用和发展。网站中有大量的技术博客和文章，各个级别的开发者都能找到适合自己学习的资料。网址为 https://www.nodejs.cn/。

（4）淘宝 NPM 镜像：是一个完整 npmjs.org 镜像，可以用此代替官方版本，同步频率为每 10 分钟一次，以保证尽量与官方服务同步。网址为 https://npm.taobao.org/。

（5）Node.js：这也是一个学习 Node.js 和前端开发技术非常好的网站，每天都有大量原创文章发布，并且技术问题可以很快被回答。当然，如果你愿意为其他人解答技术问题，或者进行技术分享，也是非常受欢迎的。网址为 http://cnodejs.org/。

每年的 JavaScript 中国开发者大会和各种 Node.js 分享沙龙，都是很好的学习 Node.js 开发技术和交流的机会。一个开发者要时刻保持谦虚的心态，并不断学习最新的技术，这对开发者来说是一种基本能力和素养。

1.2　搭建 Node.js 开发环境

学习任何一门编程语言，第一步都是搭建好该语言的开发环境。Node.js 可以在多个不同的平台稳定运行，并且均具有良好的兼容性。本节主要介绍如何在 Windows 系统平台下搭建 Node.js 的开发环境，至于其他操作系统平台，操作方法大同小异，读者可自行学习。

1.2.1　Windows 10 系统下安装部署 Node.js 开发环境

Node.js 可以在多个版本的 Windows 系统平台（Windows 7、Windows 10、Windows 11 以及 Windows Server 系列等）下稳定运行，本书主要介绍在 Windows 10 系统下 Node.js 开发环境的安装部署过程。

在 Windows 10 系统中进行 Node.js 环境部署相对简单。从 Node.js 的官方网站（https://nodejs.org/en/download/）上下载最新的 Node.js 安装包。如果下载网速较慢，国内用户还可

以通过 Node.js 官方中文网站（http://nodejs.cn/download/）进行下载。中文站点的下载页面与英文版官方网站的下载页面略有不同，中文网站只提供最新的发布版本，而英文官方网站同时提供最新的长期支持（LTS）版本和最新的发布版本。

Node.js 的安装包在 Windows 平台分为 installer 和 binary 两个版本。installer 是常用的安装包发布版本（.msi），binary 为二进制版本（.exe），可以下载后直接运行。这里建议使用后缀为.msi 的安装版本。此外，Node.js 的安装包分为 32 位和 64 位，在下载的时候要查看一下自己系统的具体信息，并选择正确的安装包进行下载和安装。

提示：Node.js 的其他发布版本可以在 https://nodejs.org/dist/中找到，本书以 v18.17.1（LTS）64-bit 版本在 Windows 10 系统下的安装为例进行介绍。

打开 Node.js 官方网站的下载页面（https://nodejs.org/en/download/），如图 1.3 所示。

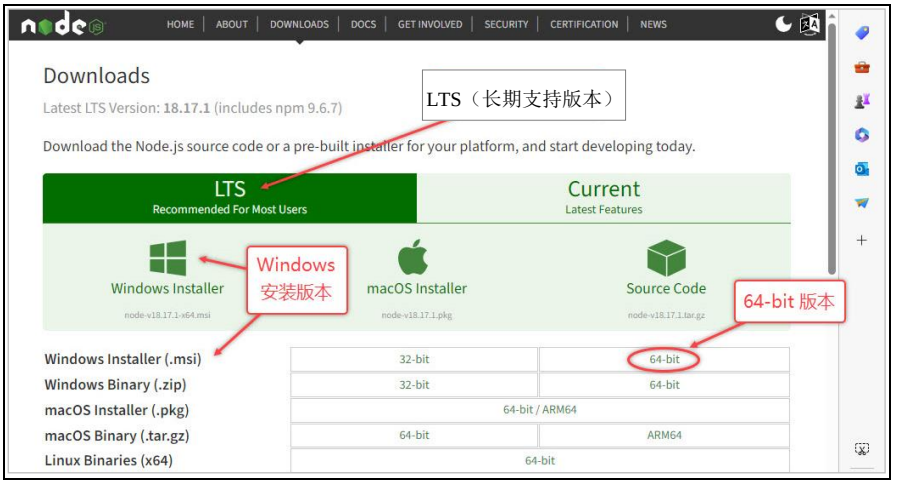

图 1.3　Node.js 官方网站下载页面

如图 1.3 中的标识所示，首先选择 LTS（长期支持版）版本，然后在页面上方选择 Windows Installer 图标，并左边菜单栏上找到 Windows Installer(.msi)菜单项，最后选择 64-bit 版本进行下载。这里具体下载得到的安装包名称为“node-v18.17.1-x64.msi”，之后就可以进行安装了。

Node.js 安装包的具体安装步骤如下：

步骤 01　Node.js 安装包是一个约 31MB 大小的、msi 格式的 Windows 系统安装文件。双击运行该安装包，会弹出如图 1.4 所示的欢迎界面。

步骤 02　如图 1.4 中的箭头所示，通过单击 Next（下一步）按钮进入如图 1.5 所示的 End-User License Agreement（终端用户协议许可）界面。

步骤 03　如图 1.5 中的箭头所示，勾选接受协议许可选项后，Next 按钮会变为如图 1.6 所示的可用状态。

步骤 04　如图 1.6 中的箭头所示，单击 Next 按钮进入下一步，此时会打开选择目标安装目录界面，如图 1.7 所示。默认的安装目录为“C:\Program Files\nodejs\”，可以通过单击 Change…按钮选择自己的目标安装目录，这里建议安装在磁盘（可选任一磁盘，笔者选择的是 D 盘）的根目录下。

步骤 05　单击图 1.7 所示界面中的 Next 按钮，进入如图 1.8 所示的 Custom Setup（自定义安装

选项）界面。

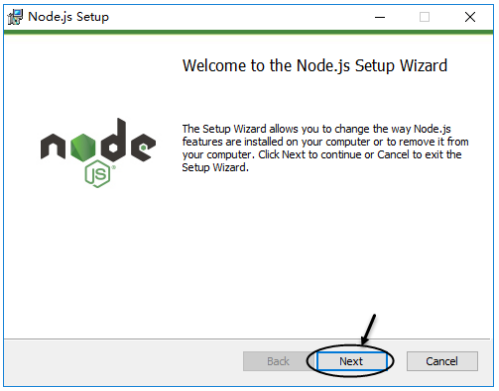

图 1.4　Node.js 安装（1）

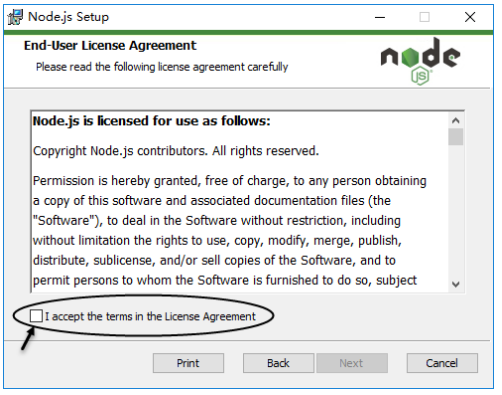

图 1.5　Node.js 安装（2）

图 1.6　Node.js 安装（3）

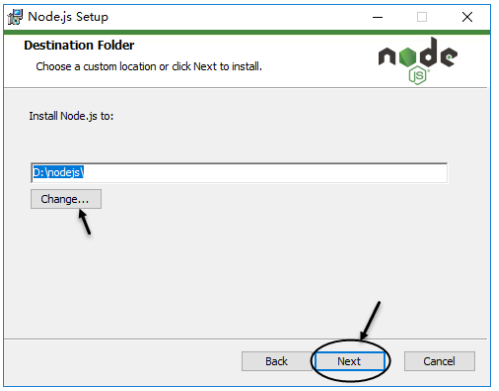

图 1.7　Node.js 安装（4）

步骤 06　如图 1.8 所示，默认会安装全部的“自定义安装选项”。这里，建议读者选择安装全部选项，尤其 Add to PATH 选项是用来设置系统默认的环境变量（PATH）的。另外，在完成安装 Node.js 的时候，也默认安装了 npm（npm package manager），npm 是 Node.js 的包管理工具。

步骤 07　单击图 1.8 中的 Next 按钮，进入如图 1.9 所示的 Ready to install Node.js（准备安装）界面。

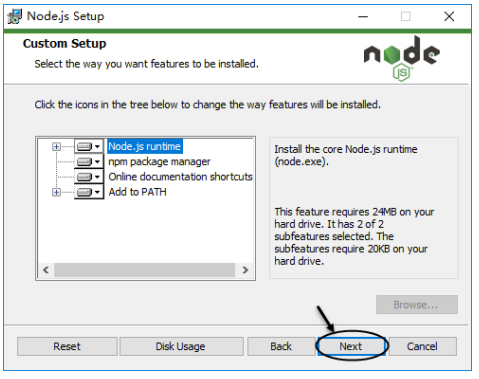

图 1.8　Node.js 安装（5）

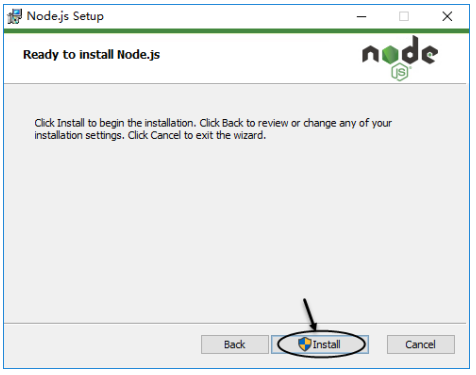

图 1.9　Node.js 安装（6）

步骤 08　如图 1.9 中箭头所示，单击 Install 按钮就会开始安装，如图 1.10 所示。

步骤 09　安装完毕后，会显示如图 1.11 所示的界面，单击 Finish 按钮完成 Node.js 的安装。

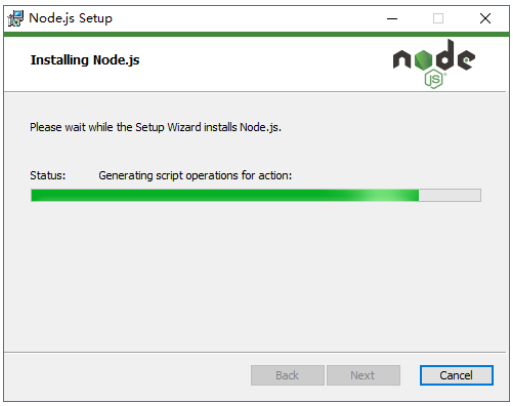

图 1.10　Node.js 安装（7）

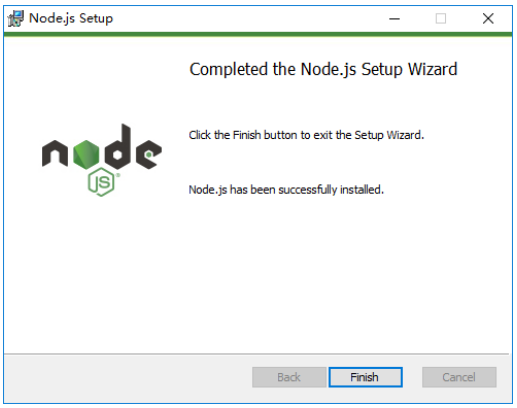

图 1.11　Node.js 安装（8）

1.2.2　测试 Node.js 开发环境

在 Node.js 开发包安装完毕后，要测试 Node.js 的开发环境，以验证 Node.js 是否安装成功。测试方法很简单，在命令行窗口输入以下 node 命令查看输出结果就可以得知。

```
node --version      // 完整命令参数，查询 node 版本号
node -v             // 简化命令参数，查询 node 版本号
```

这里选择 Node.js 自带的命令行工具（Node.js command prompt）进行测试，在系统菜单上找到 Node.js command prompt，点击打开，效果如图 1.12 所示。

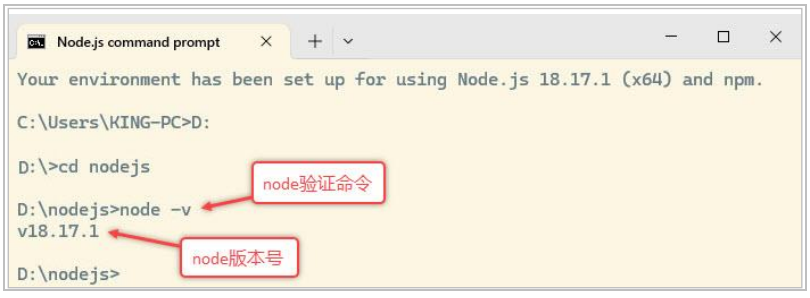

图 1.12　验证 Node.js 环境是否安装成功

如图 1.12 中的箭头和标识所示，通过输入 node -v 命令查询到当前系统安装了 v18.17.1 版本，表明 Node.js 开发环境已经安装成功了。

前文中介绍了安装 Node.js 时会自动安装包管理工具 npm，下面再验证一下 npm 工具是否也安装成功了，方法是通过输入以下 npm 命令来实现。

```
npm --version       // 完整命令参数，查询 npm 版本号
npm -v              // 简化命令参数，查询 npm 版本号
```

效果如图 1.13 所示。

图 1.13　验证 NPM 包管理工具是否安装成功

如图 1.13 中的箭头和标识所示，通过输入 npm -v 命令查询到当前系统安装了 v9.6.1 版本的 npm，表明 npm 包管理工具也同步安装成功了。

那么，Node.js 开发环境能做什么呢？最简单的一项功能就是可以直接在命令行运行 JavaScript 脚本程序，具体可参看如图 1.14 所示的操作过程。

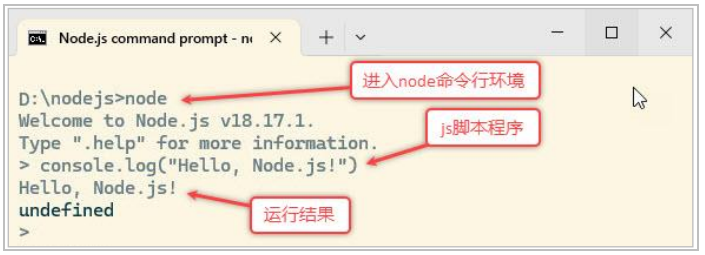

图 1.14　直接在命令行运行 JavaScript 脚本程序（1）

如图 1.14 中的箭头和标识所示，首先要在命令行通过输入“node”进入 Node.js 开发环境，然后就可以输入 JavaScript 脚本代码了。由于 Node 命令行开发环境是交互式的 JavaScript 解释器，因此在输入 JavaScript 代码并按回车键后，直接就可以打印出运行结果。

其实，这种输入 JavaScript 代码并按回车键后直接输出结果的方式不够用户友好，如果想实现一些稍微复杂的 JavaScript 代码就会很困难。好在 JavaScript 代码是通过分号（;）断句的，可以将若干句 JavaScript 代码写在一行中来完成。具体可参看如图 1.15 所示的操作过程，将若干句 JavaScript 代码写在一行中，就可以实现一个简单的求和运算了。

图 1.15　直接在命令行运行 JavaScript 脚本程序（2）

不过，这种直接在 Node 命令行中编写 JavaScript 脚本代码的方式，仅限于非常简单的场景。如果想完成复杂的代码功能，就需要通过 Node.js 环境运行 JavaScript 脚本文件来实现了。

1.2.3　通过 Node.js 运行 JavaScript 文件

在 Node 命令行环境中，可以直接运行 JavaScript 脚本文件。方法也很简单，通过 Node 命令指定 JavaScript 文件名即可，具体如下：

```
node filename.js        // filename.js 指定具体 JavaScript 脚本文件名
```

下面，我们将 1.2.2 节中测试的两段 JavaScript 脚本代码整合到同一个 JavaScript 脚本文件中，代码如下：

【代码 1-1】（详见源代码 commandline 目录中的 commandline.js 文件）

```
01  console.log("Hello Node.js!");
02  var a = 1;
03  var b = 2;
04  var c = a + b;
05  console.log("c = %d", c);
```

【代码说明】

- 在上面的代码中，将1.2.2节中测试的两段JavaScript脚本代码写在了一个JavaScript脚本文件中，然后通过Node命令行工具运行该JavaScript脚本文件，如图1.16所示。

图 1.16　Node 命令行运行 JavaScript 脚本文件

如图 1.16 中的标识所示，通过 Node 命令行运行 JavaScript 脚本文件（commandline.js），得到了同样的运行结果。

在实际的 Node.js 项目开发中，无论是使用轻量级代码开发工具，还是使用集成式的开发平台工具，且不论项目的 JavaScript 源代码文件有多复杂（数量众多且关系嵌套），在后台均是通过上面的方式运行 JavaScript 脚本文件的。

1.3　通过 Visual Studio Code 开发 Node 应用

本节将介绍如何开发一个完整的 Node 应用程序。这里需要用到 Visual Studio Code（简称 VS

Code）作为 Node 代码开发和管理的工具，同时还需要使用 Webpack 模块打包器作为构建 Node 应用框架的工具。这些都是开发 Web 前端应用的基础工具。

1.3.1　通过 Visual Studio Code 开发和管理代码

Visual Studio Code 可谓是近年来风头日盛的代码开发和管理工具，该工具由微软（Microsoft）公司负责开发与维护，并在 2015 年 4 月 30 日的 Build 开发者大会上第一次正式对外发布。

Visual Studio Code 的设计目标是成为一款能够满足跨平台运行的、轻量级的、可扩展的开发工具，主要用作编写当前流行的 Web 应用和云服务应用的源代码编辑器。虽然，Visual Studio Code 隶属于 Visual Studio 系列开发平台，但与诸如 Visual Studio 2019、Visual Studio 2022 这类重量级的、功能强大的、集成式的开发平台完全不同，其自身仅是一款源代码编辑器，写好的 C 程序无法编译运行，JavaScript 脚本文件也无法解释执行。

不过，如果因此而小看 Visual Studio Code 就大错特错了，Visual Studio Code 的特点就是提供了很强的扩展功能，强大到让设计人员惊叹的程度。Visual Studio Code 的扩展功能是通过安装相应的插件实现的，前面提到的 C 程序和 JavaScript 脚本，甚至包括 Java、Python、Node 程序，以及本书后面将要重点介绍的 Vue.js 程序，都可以通过安装相对应的功能插件而得到支持。因此，Visual Studio Code 在极短的时间内就得到了业内的充分认可，并迅速在源代码编辑器市场占据了一席之地。对此，相信你也只能感叹微软研发能力的强大了。

在 Windows 平台下安装 Visual Studio Code 的操作非常简单，自家的操作系统平台自然会提供最好的支持。首先，进入 Visual Studio Code 官方网址的下载页面（https://code.visualstudio.com/Download），如图 1.17 所示。

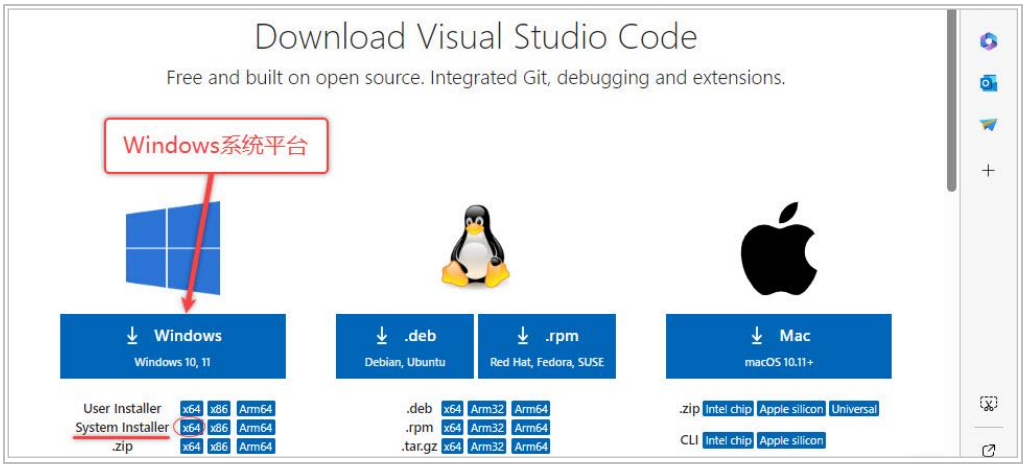

图 1.17　Visual Studio Code 官方下载页面

在 Windows 系统平台区域中的“System Installer”类别中选择 64-bit 版本的安装包（VSCodeSetup-x64-1.81.1.exe）进行下载，该版本是为系统用户配置的。另外，还有一种“User Installer”类别是为个人用户配置的版本。在图 1.17 中还可以看到 Visual Studio Code 的 Linux 版本和 Mac 版本，这表明 Visual Studio Code 是一款跨平台的开发工具。

在 Windows 操作系统中安装 VS Code，直接双击运行安装包，按照安装步骤操作就可以了。

步骤 01　首先会进入安装程序的“许可协议”界面，如图 1.18 所示。

步骤 02　单击“我接受协议”单选按钮，激活“下一步”按钮，然后单击“下一步”按钮进入“选择目标位置”界面，如图 1.19 所示。在这里可以选择用户自定义的 Visual Studio Code 安装路径。

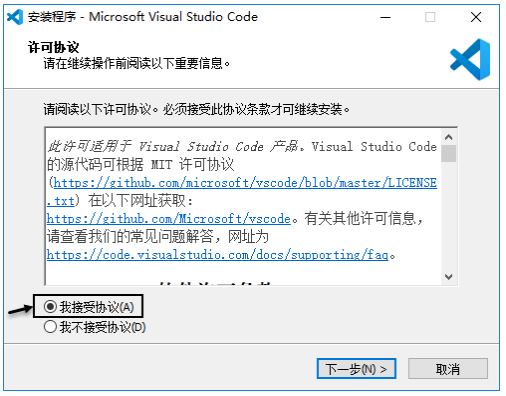

图 1.18　Visual Studio Code 安装过程（1）

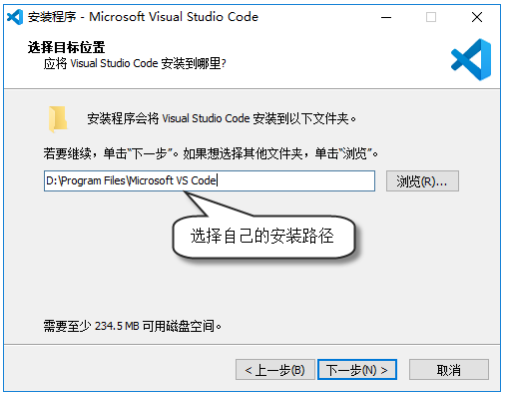

图 1.19　Visual Studio Code 安装过程（2）

步骤 03　继续单击“下一步”按钮进入实际安装过程，直到安装程序执行完毕，如图 1.20 所示。

图 1.20　Visual Studio Code 安装过程（3）

步骤 04　如果勾选了“启动 Visual Studio Code”复选框，在单击“完成”按钮时就会退出安装程序，同时自动运行 Visual Studio Code 开发工具。初次运行 Visual Studio Code 的界面如图 1.21 所示。Visual Studio Code 开发工具的界面非常简洁（轻量级），窗口的顶部是一个主菜单栏，窗口的左侧是一列快捷工具按钮列表（可以安装扩展功能），窗口的主体是文档编辑区域（上面显示的内容包括一些开发功能的快捷方式）。

步骤 05　单击左侧快捷工具按钮列表中最上面的“资源管理器（Ctrl+Shift+E）”按钮，界面效果如图 1.22 所示。

图 1.21　Visual Studio Code 程序初次运行界面

图 1.22　Visual Studio Code 资源管理器

如图 1.22 中的箭头和标识所示，打开“资源管理器”后，会在窗口左侧显示一个区域，里面包括了工程目录、大纲和时间线等项目。

在 VS Code 资源管理器中，设计人员可以新建或引入自己的工程目录进行源代码的有效管理。本书的工程名称为“vueprojects”（注意，VS Code 默认会将工程目录名称全部显示为大写）。

下面尝试通过 Visual Studio Code 测试运行【代码 1-1】所创建的 commandline.js 文件。首先，在 Visual Studio Code 中通过“文件”菜单中的“打开文件夹…”菜单项来打开工程目录，效果如图 1.23 所示。

图 1.23　Visual Studio Code 打开工程目录（1）

如图 1.23 中的箭头所示，单击“打开文件...”后会打开一个“文件选择对话框”，选中之前创建好的 vueprojects 工程目录就可以了，效果如图 1.24 所示。

图 1.24　Visual Studio Code 打开工程目录（2）

如图 1.24 中的箭头所示，已经能看到之前【代码 1-1】所创建的 commandline.js 文件。那么，如何通过 Visual Studio Code 工具运行该 JavaScript 脚本文件呢？

这里，需要在 Visual Studio Code 中安装一个名称为“Code Runner”的插件，之后就能运行 JavaScript 脚本文件，具体安装方法如下：

在左侧快捷工具按钮列表中找到“扩展（Ctrl+Shift+X）”按钮，界面效果如图 1.25 所示。

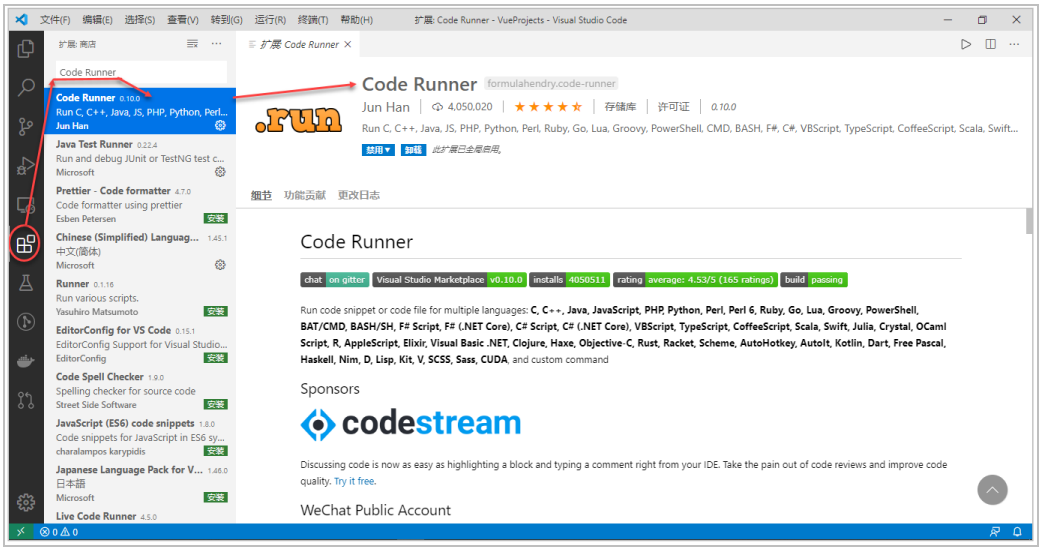

图 1.25　Visual Studio Code 安装扩展

如图 1.25 中的箭头和标识所示，打开“扩展”界面后在搜索栏输入“Code Runner”字符串，下面扩展列表中自动筛选出来的第一项就是 Code Runner 插件，直接单击进行“安装”即可。由于笔者已经安装过该插件，因此图 1.25 中显示的是已安装状态，从右侧主窗口可以看到关于 Code Runner 插件的功能介绍，该插件除了支持 JavaScript 脚本语言外，还支持多种编程语言（C、C++、Java、PHP、Python、Go 等），功能十分强大。

安装好 Code Runner 插件后，就可以直接在 VS Code 中测试运行 JavaScript 脚本文件了。返回如图 1.24 所示的界面，选中并右击 commandline.js 文件，会弹出一个快捷菜单，如图 1.26 所示。

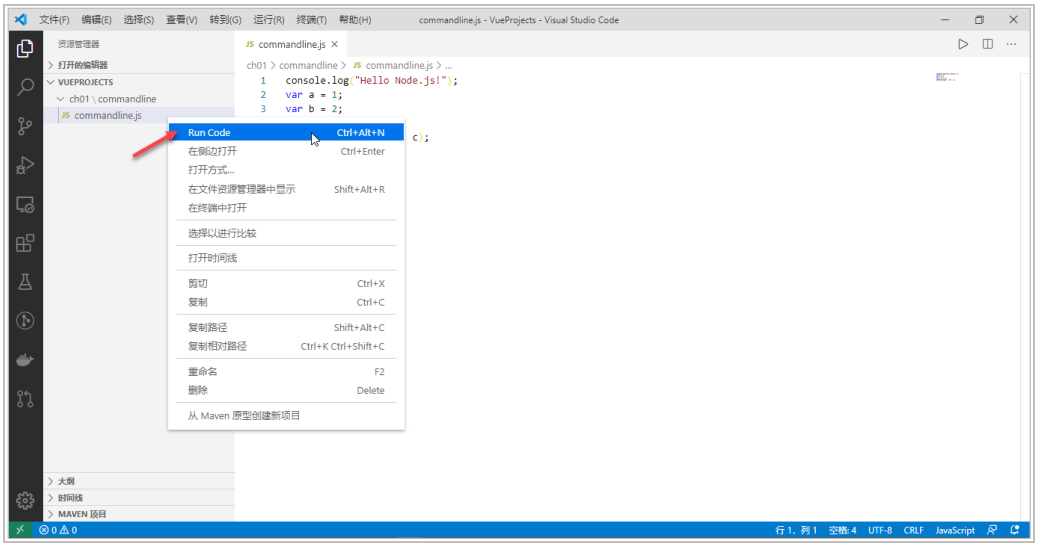

图 1.26　通过 Code Runner 运行 JavaScript 脚本文件（1）

如图 1.26 中的箭头所示，在快捷菜单中单击 Run Code 菜单项，Visual Studio Code 开发工具会弹出一个“输出”界面窗口，显示 JavaScript 代码调试运行的输出结果，效果如图 1.27 所示。

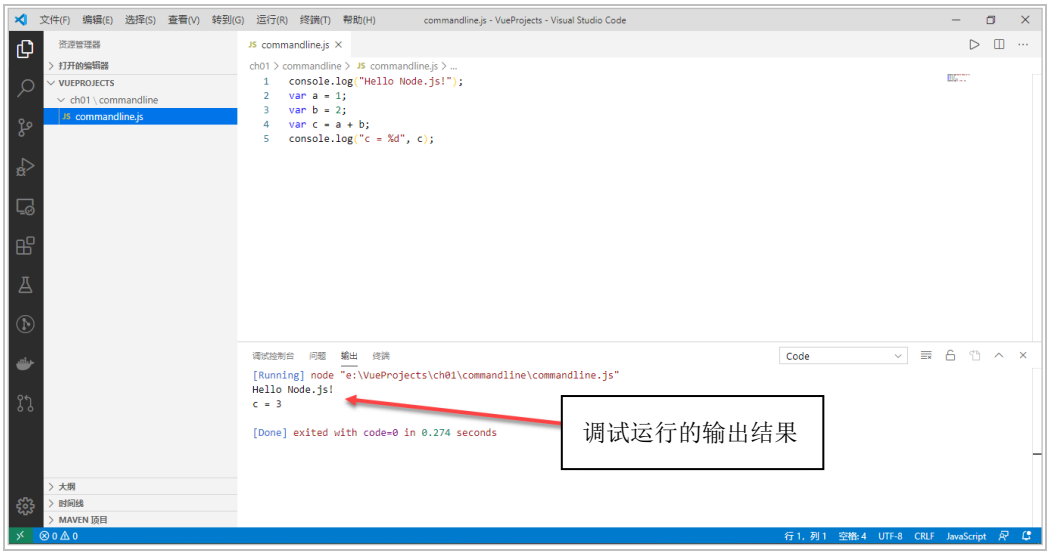

图 1.27　通过 Code Runner 运行 JavaScript 脚本文件（2）

如图 1.27 中的标识所示，调试运行的输出结果与图 1.16 所示的结果是一致的，说明 Visual Studio Code 开发工具的 Code Runner 插件运行方式，可以完美替代 Node 程序的命令行终端运行方式。

1.3.2　通过 Webpack 构建 Node 应用程序架构

前文中介绍了如何通过 Visual Studio Code 的插件运行单个 JavaScript 脚本文件的 Node 程序。不过，这种开发方式在近年来已经被一种全新的、以自动化方式构建 Web 前端应用的方式取代了。

所谓“自动化”方式，其实就是通过一种或几种自动化工具来构建 Web 前端应用的开发框架。这类开发框架基本都是通过一种或几种工具自动生成的，生成后的框架会包含大部分 Web 前端应用所需的基本类库、第三方插件和支撑配置文件，等等。可以说，这种全新的开发方式将 Web 前端应用开发提升到了一个全新的高度，并且符合将前后端分离开来进行独立设计的大趋势。

Web 前端的自动化构建工具有很多种，其中最著名的就是 Webpack 模块化打包器工具，它也是目前 Web 前端开发中最流行的工具之一。Webpack 的功能十分强大，设计了“入口（entry）”“输出（output）”“加载器（loader）”和“插件（plugins）”这四个概念，以递归方式构建出一个应用程序主要资源的依赖关系图，并将 JavaScript 模块打包成一个或多个“包（bundle）”。

由于本书不是专门介绍 Webpack 工具的（感兴趣的读者可以参考 Webpack 官网的内容），这里仅就如何使用 Webpack 工具构建 Node 应用进行简单介绍，过程如下：

（1）创建 Node 应用程序目录，并进入该目录，命令行如下：

```
mkdir vueprojects && cd vueprojects
```

（2）通过 npm 命令进行初始化项目的操作，命令行如下：

```
npm init -y
```

npm 就是包管理工具命令，参数 init 表示进行 Node 应用项目的初始化操作。该命令参数会生

成一个 package.json 配置文件，用于描述该项目的详细配置信息。如果再添加-y 参数，则表示项目使用默认的配置参数（省去了人工配置的过程），效果如图 1.28 所示。

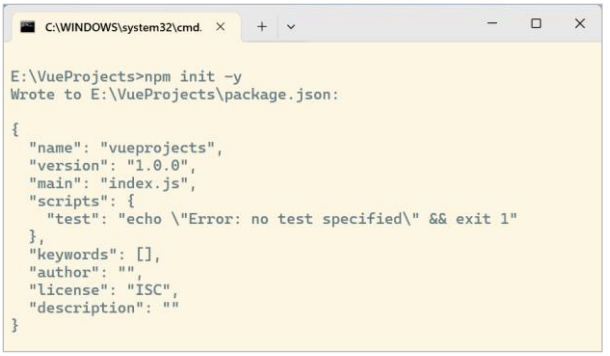

图 1.28　通过“npm init -y”命令初始化项目

图 1.28 所示的命令行中显示出生成的 package.json 文件内容，这些信息都是默认生成的，后期用户可以自行修改。

（3）通过 npm 命令在本地安装 Webpack 开发包，命令如下：

```
npm install webpack webpack-cli --save-dev
```

在 npm 命令中使用参数 install 表示安装第三方开发包，后面指定了开发包的名称“webpack”和“webpack-cli”。其中，webpack-cli 是 webpack 的命令行工具，也就是在命令行中可以支持使用 Webpack。另外，使用参数“--save-dev”表示该安装包为调试开发时的依赖项，信息会记录到 package.json 文件中的 devDependencies 子项中，后期项目发布时不需要这些安装包。安装过程需要一些时间，请耐心等待。安装效果如图 1.29 所示。

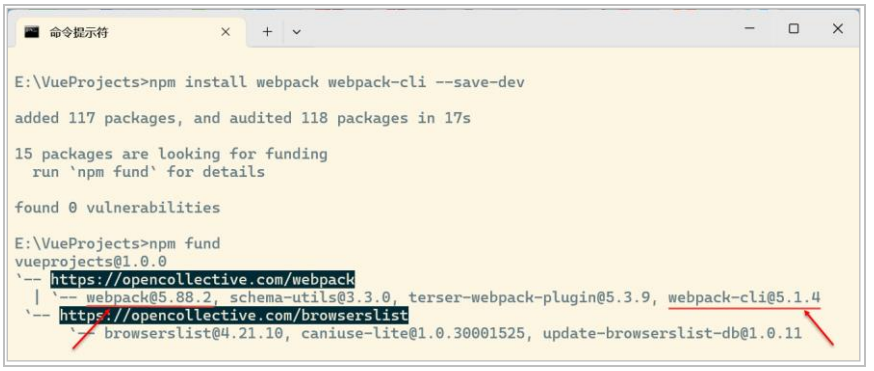

图 1.29　通过 npm 命令安装 Webpack

如图 1.29 中的箭头和标识所示，已经成功安装了 webpack@5.88.2 和 webpack-cli@5.1.4 开发包，其中“@”符号后面标识的是开发包的版本号。

至此，通过 Webpack 工具构建 Node.js 应用程序框架就基本完成了。

1.3.3　通过 Visual Studio Code 开发调试 Node 应用

目前，有多种开发工具可以支持 Node.js 应用的开发，比如 jetBrains WebStorm、Eclipse、Visual Studio Code 等。这些开发工具原则上是“条条大路通罗马”的，相互间各有优势，并无优劣之分。在本书中，我们选择 Visual Studio Code 开发工具，其中一个原因也是为了配合后面讲解 Vue.js 的相关内容。

下面，通过 VS Code 打开刚刚创建的 Node 应用目录（vueprojects），效果如图 1.30 所示。

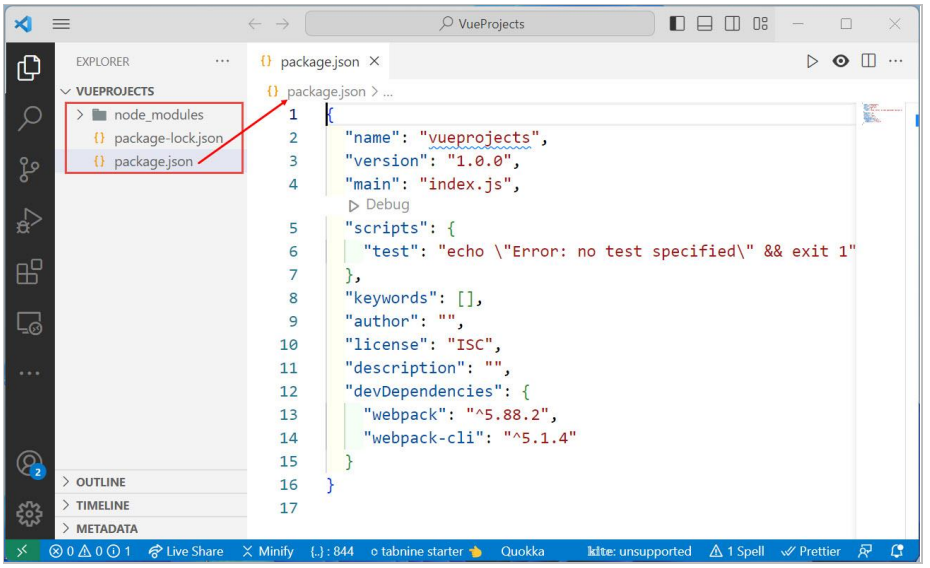

图 1.30　通过 Visual Studio Code 管理 Node 应用（1）

查看图 1.30 左上方的方框标识，里面显示了 Node 应用的目录结构，具体内容如图 1.31 所示。

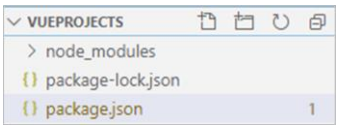

图 1.31　通过 Visual Studio Code 管理 Node 应用（2）

node_modules 目录存放了通过 npm 命令安装的各种开发包，里面不仅有刚刚安装的 webpack 和 webpack_cli 开发包，还包括了整个 Node 生态系统必要的依赖项。因此，node_modules 目录的体积通常有一个较大的数量级，这点也算是 Node 生态系统的不足之处。

再次返回图 1.30 所示的窗口，打开的是 package.json 配置文件，内容如下：

【代码 1-2】

```
{
  "name": "vueprojects",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
```

```
10    "author": "",
11    "license": "ISC",
12    "devDependencies": {
13      "webpack": "^5.88.2",
14      "webpack-cli": "^5.1.4"
15    }
16  }
```

【代码说明】

- 第02行代码中的“name”字段标识的是该项目的名称（vueprojects）。
- 第03行代码中的“version”字段标识的是该项目的版本号（1.0.0）。
- 第05行代码中的“main”字段标识的是该项目的主入口脚本文件。
- 第06行代码中的“scripts”字段用于执行自定义脚本命令。这里定义“test”参数，就相当于执行“npm run test”脚本命令。
- 第12～15行代码中的“devDependencies”字段用于定义开发调试阶段安装的依赖项，其中第13、14行代码定义的依赖项印证了之前的安装命令（npm install webpack webpack-cli --save-dev）。

以上关于 package.json 配置文件的内容都是最基本的。功能越复杂，安装依赖项越多，package.json 配置文件的信息也会随之增加。

另外，还有一个 package-lock.json 配置文件，这个文件是在 npm v5 版本以后新增加的功能。在通过 npm install 命令安装开发包后会自动生成该文件，该文件锁定了当前安装的小版本号。因此，当用户再次使用 npm install 命令后，就可以避免通过 package.json 配置文件将开发包升级到最新版本，从而有效地避免了因为版本升级带来的各种依赖冲突。但是，若用户真想升级到最新版本或某个指定版本，则必须在开发包名称后使用“@latest”或“@版本号”来执行指定版本号的升级。

下面具体介绍如何开发一个基本的 Node 应用程序，该程序实现了一个简单的、基于 Node 框架的 Web 服务器。

步骤 01　在应用程序根目录下创建一个 HTML 5 页面文件（index.html），我们通过 Visual Studio Code 开发工具来实现该操作。在应用程序根目录（vueprojects）上单击“新建文件”图标，效果如图 1.32 所示。

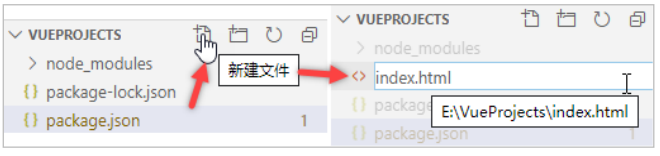

图 1.32　通过 Visual Studio Code 新建 HTML 页面文件（1）

单击“新建文件”图标后会自动创建一个文件。注意，这个文件没有名称，也没有文件类型后缀，全部需要设计人员手动输入（index.html），这点可能与其他开发平台差异比较大。因此，该新建文件是一个空白的 HTML5 网页，如图 1.33 所示。

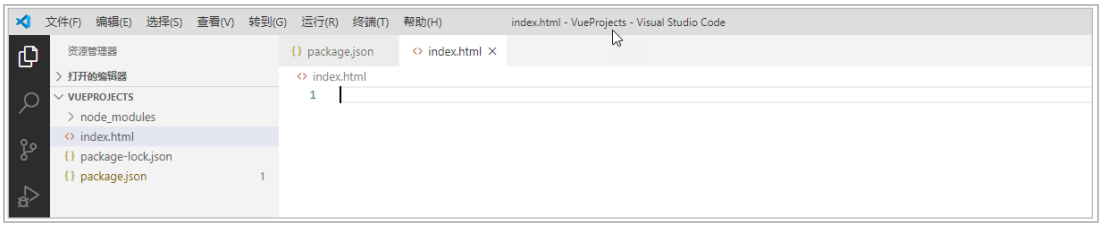

图 1.33　通过 Visual Studio Code 新建 HTML 页面文件（2）

新建的 index.html 网页全部空白，没有一行默认生成的代码。此时，读者如果质疑 Visual Studio Code 开发工具，那就大错特错了。Visual Studio Code 为设计人员提供了很强大的自动代码生成功能，下面演示如何操作。

在空白页面最开始处输入字符“!”，会弹出一个快捷输入方式，如图 1.34 所示。

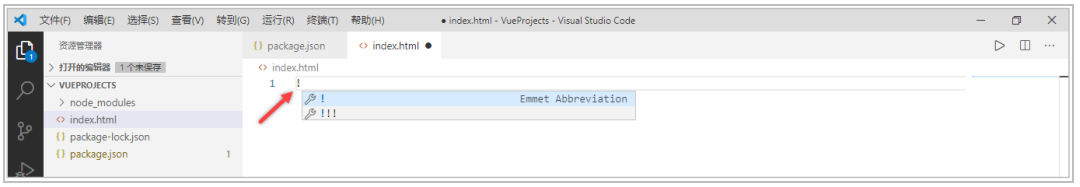

图 1.34　通过 Visual Studio Code 新建 HTML 页面文件（3）

如图 1.34 中箭头所示，此时会弹出一个快捷输入方式，里面的列表菜单中有一个“!”选项。注意，后面的提示信息为“Emmet Abbreviation”，说明该功能是由 Emmet 插件所提供的。这个 Emmet 插件可谓是大名鼎鼎，在很多集成开发工具或轻量级代码编辑器中都有其身影，是一款非常好用的代码自动生成工具。因此，Visual Studio Code 就将 Emmet 插件内置其中，用户不用再单独安装该插件了。

下面看一下 Emmet 插件的使用方法，选中“!”选项后直接按回车键或 Tab 键，效果如图 1.35 所示，Emmet 插件直接在 index.html 文件中自动生成了一个 HTML 5 网页模板。

图 1.35　通过 Visual Studio Code 新建 HTML 页面文件（4）

步骤 02　在应用程序根目录下创建一个 src 子目录，在该子目录下新建一个 JavaScript 脚本文件（index.js），方法同图 1.32 所示。在 index.js 脚本文件中输入如下代码：

【代码 1-3】

```
01  document.write('Hello Node!');
```

【代码说明】

- 第01行代码通过调用document对象的write()方法向页面中写入一行信息。

步骤 03　对 index.html 页面代码进行适当修改，将 index.js 脚本文件在 index.html 页面文件中进行引用，代码如下：

【代码 1-4】

```
01 <!DOCTYPE html>
02 <html lang="en">
03 <head>
04    <meta charset="UTF-8">
05   <meta name="viewport" content="width=device-width, initial-scale=1.0">
06    <title>Document</title>
07 </head>
08 <body>
09    <script src="./src/index.js"></script>
10 </body>
11 </html>
```

【代码说明】

- 在第09行代码中通过<script>标签引入了index.js脚本文件。

步骤 04　更新 package.json 配置文件的内容，代码如下：

【代码 1-5】

```
01 {
02   "name": "vueprojects",
03   "version": "1.0.0",
04   "description": "",
05   "private": true,
06   "scripts": {
07     "test": "echo \"Error: no test specified\" && exit 1"
08   },
09   "keywords": [],
10   "author": "",
11   "license": "ISC",
12   "devDependencies": {
13     "webpack": "^5.88.2",
14     "webpack-cli": "^5.1.4"
15   }
16 }
```

【代码说明】

- 在第05行代码中添加private参数，确保安装包是私有的（private）；同时，移除main入口，防止意外发布应用代码。

步骤 05　此时就可以简单地测试一下该 Node 应用了，最直接的方法就是在浏览器中运行

index.html 页面文件，效果如图 1.36 所示。

如图中的箭头所示，浏览器页面中显示了 index.js 脚本代码执行的结果。不过，虽然 Node 应用被正确执行了，但远没有发挥出 Webpack 工具的能力。下面，我们通过 Webpack 重构一下这个 Node 应用。

步骤 06　创建“分发（dist）”目录结构。

这里需要调整一下原始的目录结构，将“源（src）”代码从“分发（dist）”代码中分离出来。根据 Webpack 定义的规范，“源（src）”代码是开发时使用的代码，“分发（src）”代码是构建过程产生的代码最小化和优化后的“输出（output）”目录，用于在浏览器中展示给终端用户。

具体操作还是在 Visual Studio Code 工具下完成，效果如图 1.37 所示。

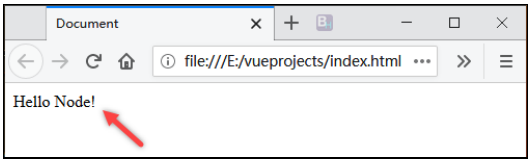

图 1.36　在浏览器中运行 HTML 页面

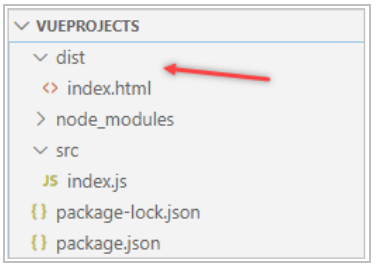

图 1.37　创建“分发（dist）”目录结构

如上图中的箭头所示，我们还需要将 index.html 页面文件移入“分发（dist）”目录，以便于后期调试运行。

步骤 07　修改完善 index.js 脚本文件和重新更新 index.html 页面文件。

将直接在页面文档写入（document.write()方法）文本的方式，替换为通过分区（div）标签插入文本的方式。另外，这里需要借助 Lodash 插件来完成该任务。Lodash 是一个高性能的、模块化的 JavaScript 实用工具库，它简化了数组、字符串和对象等类型的操作难度，在业内十分受欢迎。Lodash 插件的使用方法有以下 3 种：

① 直接引用方式：

```
<script src="lodash.js"></script>
```

② CDN 方式：

```
<script src="https://unpkg.com/lodash@4.17.5"></script>
```

③ 本地安装方式：

```
npm install --save lodash       // --save 表示为生产环境
```

以上 3 种方式均可行，但从 Node 应用的角度看，推荐使用第 3 种方式。Lodash 本地安装的过程如图 1.38 所示。

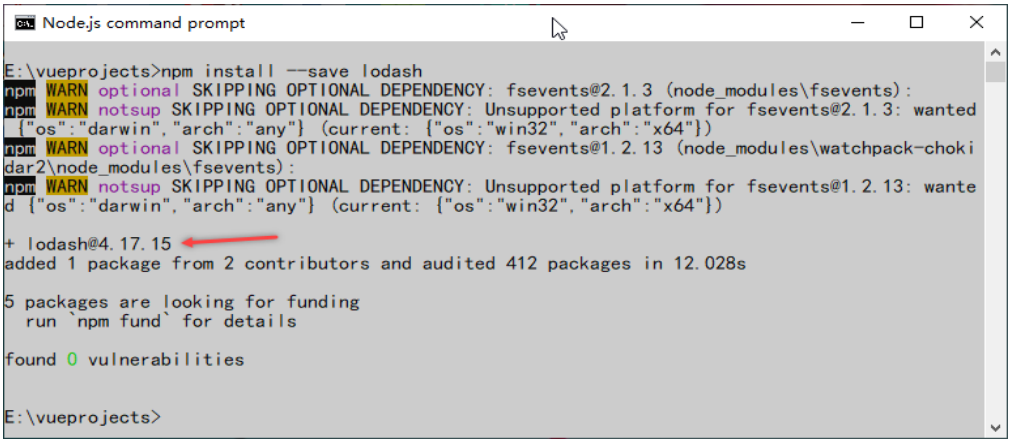

图 1.38　安装 Lodash 库

如图 1.38 中的箭头所示，默认安装的是 Lodash 库的最新稳定版（4.17.15）。另外，在第 2 种“CDN 方式”中，安装一般都是最新的稳定版。

接下来，修改一下 index.js 脚本文件，修改后的代码如下：

【代码 1-6】

```
01  import _ from 'lodash';
02  /**
03   * func - create div component
04   */
05  function divComp() {
06      var eleDiv = document.createElement('div');
07      // TODO: use Lodash '_' to join string.
08      eleDiv.innerHTML = _.join(['Hello', 'Webpack', '!'], ' ');
09      // TODO: return element div
10      return eleDiv;
11  }
12  // TODO: append div to body
13  document.body.appendChild(divComp());
```

【代码说明】

- 在第01行代码中，通过import方式导入已安装的Lodash模块，并将模块命名为“_”（Lodash通用命名方式）。
- 第05～11行代码定义一个divComp()方法，用于创建一个分区（div）标签元素，然后在该标签元素内插入文本信息。具体内容如下：
  - 在第06行代码中，通过document对象调用createElement()方法创建一个分区（div）标签元素（eleDiv）。
  - 在第08行代码中，通过“_”调用join()方法连接一组字符串，并定义为分区（eleDiv）标签元素的innerHTML属性。
  - 在第10行代码中，返回创建好的分区（eleDiv）标签元素。
- 在第13行代码中，通过document对象的appendChild()方法，将刚刚创建的分区（eleDiv）元

素追加到body元素中，实现在页面中输入文本的效果。

在修改好 index.js 脚本文件后，还要修改一下 index.html 页面文件。因为现在需要通过 Webpack 工具打包合成脚本文件，所以需要加载分发（dist）目录中的 bundle（包）脚本文件（main.js），这里就不再需要加载 src 目录的原始脚本文件了。index.html 页面文件修改后的代码如下：

【代码 1-7】

```
01  <!DOCTYPE html>
02  <html lang="en">
03  <head>
04      <meta charset="UTF-8">
05    <meta name="viewport" content="width=device-width, initial-scale=1.0">
06      <title>Document</title>
07  </head>
08  <body>
09      <!--<script src="./src/index.js"></script>-->
10      <script src="main.js"></script>
11  </body>
12  </html>
```

【代码说明】

- 在第09行代码中，注释了通过<script>标签引入的index.js脚本文件。
- 在第10行代码中，通过<script>标签引入了main.js脚本文件。

步骤 08　通过 Webpack 工具打包 Node 应用。

通过执行 Webpack 打包命令，将 index.js 脚本作为入口起点，打包合并输出 main.js 脚本文件。命令如下：

```
npx webpack
```

npx 命令是从 npm 5.2+版本开始提供的命令，可以自动执行 npm 包内的二进制安装文件。npx webpack 命令的执行过程，如图 1.39 所示。

如上图中的箭头所示，通过 Webpack 打包后自动生成了 main.js 脚本文件。至于后面的警告信息先不用理会，后续会通过相关配置消除该警告信息。

接下来，返回 VS Code 的项目源代码目录（dist）看一下有没有什么变化，效果如图 1.40 所示。

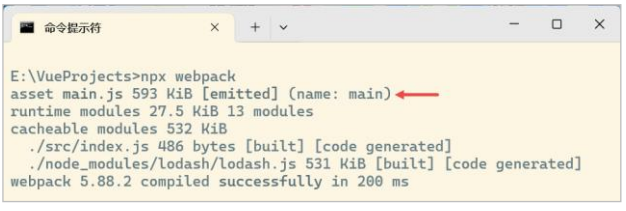

图 1.39　通过 Webpack 打包 Node 应用

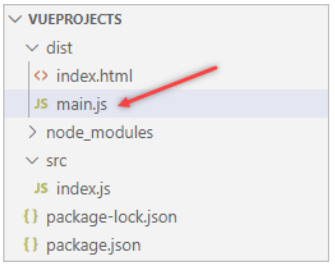

图 1.40　更新“分发（dist）”目录结构

如图 1.40 中的箭头所示，在“分发（dist）”目录下自动生成了一个 main.js 脚本文件，这就是

通过 Webpack 工具打包生成的 bundle 包文件。下面，让我们见识一下这个 main.js 脚本文件的庐山真面目，如图 1.41 所示。

图 1.41　在浏览器中运行 HTML 页面

main.js 脚本文件是一个经过代码压缩的打包文件（提高运行速度），但代码阅读起来是相当有难度的。另外，还可以看到 main.js 脚本文件将 Lodash 包也一起合并压缩进去了。

步骤 09　再次测试这个 Node 应用，在浏览器中打开 index.html 页面文件，效果如图 1.42 所示。

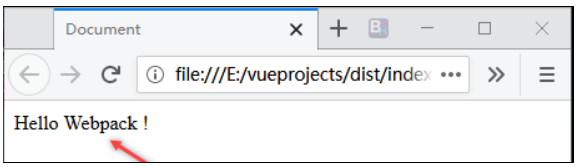

图 1.42　在浏览器中运行 HTML 页面

如图 1.42 中的箭头所示，浏览器页面中显示了 index.js 脚本代码第二次更新修改后的执行结果（Hello Webpack!），说明经过 Webpack 工具打包后生成的 main.js 脚本文件被正确运行了。

步骤 10　使用 Webpack 配置文件。

细心的读者可能会发现上面的“步骤 08”有些奇怪，在 Webpack 打包过程中似乎隐藏了一些细节，在 main.js 脚本文件中看不到具体配置信息。其实，这是通过 Webpack 工具的默认配置完成的，而它对于简单的应用通常不需要配置。

但是，Webpack 工具有一个配置文件（webpack.config.js），在应对较为复杂的 Node 应用时就可以派上用场。在应用 Webpack 工具时，通过使用配置文件（webpack.config.js）可以有效地避免在命令行终端中手动输入大量烦琐的命令，通过自动化的方式完成压缩、合并和打包等复杂操作。

Webpack 配置文件（webpack.config.js）一般放置在 Node 应用的根目录下，与 Node 应用配置

文件（package.json）处于同一级，效果如图 1.43 所示。

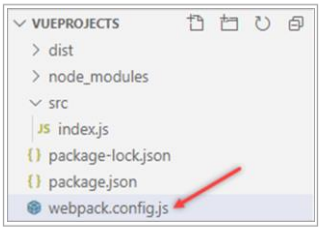

图 1.43　Webpack 配置文件——webpack.config.js

如图 1.43 中的箭头所示，Webpack 配置文件（webpack.config.js）放置于 Node 应用（vueprojects）根目录下。

下面简单配置一下 Webpack 配置文件（webpack.config.js），代码如下：

【代码 1-8】

```
01 const path = require('path');
02 /**
03  * module : exports
04  */
05 module.exports = {
06   entry: './src/index.js',
07   output: {
08     filename: 'bundle.js',
09     path: path.resolve(__dirname, 'dist')
10   },
11   mode: 'development'
12 };
```

【代码说明】

- 在第06行代码中，通过entry参数配置Webpack工具打包的入口（index.js脚本文件的路径）。
- 在第07行代码中，通过output参数配置Webpack工具打包的出口，具体内容如下：
  - 在第08行代码中，通过filename参数配置打包后的出口脚本文件的名称（bundle.js）。
  - 在第09行代码中，通过path参数解析打包后的出口文件路径。
- 在第11行代码中，通过mode参数配置打包模式（development表示开发模式，该模式会对脚本代码进行压缩）。

步骤 11　重新配置 package.json 配置文件。

在原始 package.json 配置文件中的 scripts 节点下添加一个 build 参数，具体脚本代码如下：

【代码 1-9】

```
01 {
02   "name": "vueprojects",
03   "version": "1.0.0",
04   "description": "",
05   "private": true,
06   "scripts": {
07     "test": "echo \"Error: no test specified\" && exit 1",
08     "build": "webpack"
09   },
10   "keywords": [],
11   "author": "",
```

```
  "license": "ISC",
  "devDependencies": {
    "webpack": "^4.43.0",
    "webpack-cli": "^3.3.11"
  },
  "dependencies": {
    "lodash": "^4.17.15"
  }
}
```

【代码说明】

- 在第08行代码中，新添加了一个build参数，参数值指向webpack命令。

步骤 12　再次修改 index.js 脚本文件和 index.html 页面文件。

修改后的 index.js 脚本文件代码如下：

【代码 1-10】

```
import _ from 'lodash';
//func - create div component
function divComp() {
    var eleDiv = document.createElement('div');
    // TODO: use Lodash '_' to join string.
    eleDiv.innerHTML = _.join(['Hello','Webpack','&','NodeJS','!'],' ');
    // TODO: return element div
    return eleDiv;
}
// TODO: append div to body
document.body.appendChild(divComp());
```

【代码说明】

- 在第06行代码中，通过Lodash插件库重新连接了一组字符串，用于在页面中进行显示。

由于在 Webpack 配置文件（webpack.config.js）中重新定义出口脚本文件（bundle.js），因此还要再修改一下 index.html 页面文件，修改后的代码如下：

【代码 1-11】

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <!--<script src="./src/index.js"></script>-->
    <!--<script src="main.js"></script>-->
    <script src="bundle.js"></script>
</body>
```

```
13  </html>
```

【代码说明】

- 在第10行代码中，再次注释了通过<script>标签引入的main.js脚本文件。
- 在第11行代码中，通过<script>标签引入了bundle.js脚本文件。

步骤 13　再次通过 Webpack 工具打包 Node 应用。

由于在 package.json 配置文件的 script 节点中重新定义脚本执行命令，因此需要输入新的命令来执行 Webpack 打包操作，命令如下：

```
npm run build       // build parameter is defined in package.json
```

上面命令行中的 build 参数是在 package.json 配置文件中定义的，读者可以返回去参考一下。

关于 npm run build 命令的执行过程，如图 1.44 所示。

如图 1.44 中的箭头所示，通过 Webpack 打包后自动重新生成了 bundle.js 脚本文件。另外可以注意到，警告信息已经不见了，这是因为在【代码 1-8】中的第 11 行代码中添加了 mode 参数。

下面，我们再次返回 Visual Studio Code 开发工具的源代码目录（dist）看一下有没有什么变化，效果如图 1.45 所示。

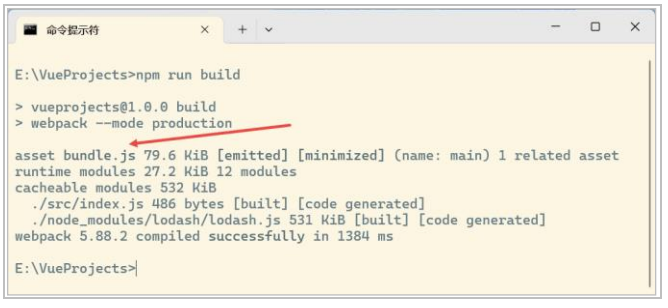

图 1.44　通过 Webpack 配置文件打包 Node 应用

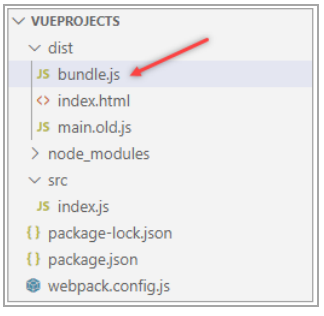

图 1.45　再次更新“分发（dist）”目录结构

如图 1.45 中的箭头所示，在“分发（dist）”目录下自动生成了一个 bundle.js 脚本文件，这就是通过 Webpack 工具配置文件重新打包生成的 bundle 包文件。这个重新生成的 bundle.js 脚本文件是符合 ES 6 标准规范的，由于代码比较长，这里就不做演示了。

步骤 14　此时可以再次测试一下这个 Node 应用，仍旧是在浏览器中运行 index.html 页面文件，效果如图 1.46 所示。

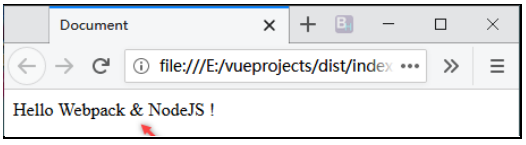

图 1.46　在浏览器中运行 HTML 页面

如图 1.46 中的箭头所示，浏览器页面中显示了 index.js 脚本代码第三次更新修改后的执行结果（Hello Webpack & NodeJS!），说明通过 Webpack 工具配置文件打包后生成的 bundle.js 脚本文件被正确运行了。

步骤 15　通过 webpack-dev-server 插件实现 Node 应用热加载。

Webpack 工具的打包功能固然十分强大，不过相信读者也发现了每次的打包操作都需要手动进行，之后的页面测试过程也很粗放（直接运行 HTML 页面）。于是，研发人员为了配合 Webpack 工具就设计了一个 webpack-dev-server 插件，可以实现 Node 应用的自动打包和热加载功能。webpack-dev-server 插件的安装方法与 webpack 方式一致，命令如下：

```
npm install webpack-dev-server --save-dev
```

安装完毕后，还需要在 package.json 配置文件的 script 节点中，新增一个 dev 参数，配置信息为插件名称（webpack-dev-server）。

然后，通过直接运行 npm run dev 命令就可以进行打包操作了，效果如图 1.47 所示。

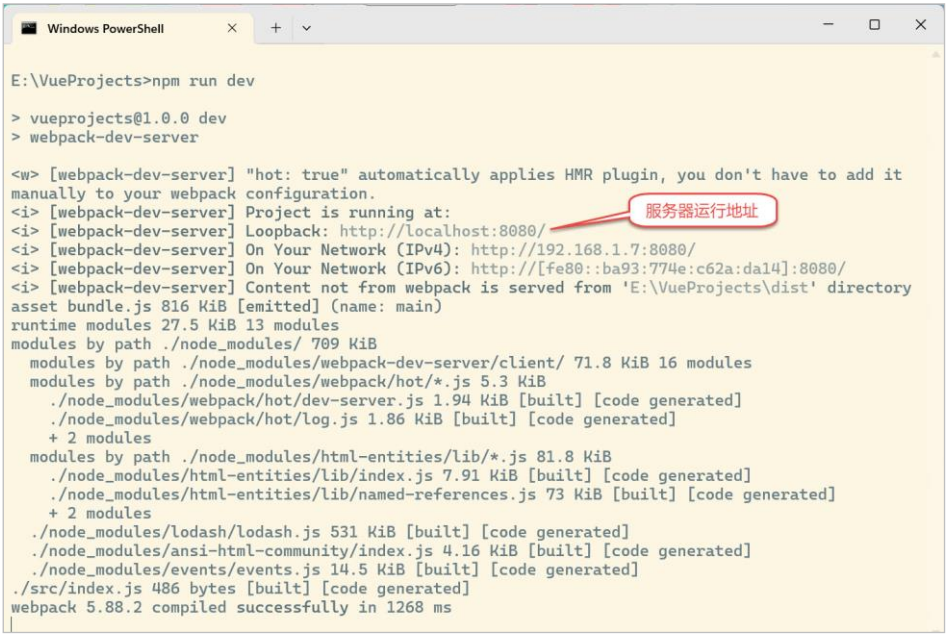

图 1.47　再次运行 npm run dev 命令进行打包

如图 1.47 中的标识所示，命令行终端中给出了 Node 应用服务端运行地址的提示信息（http://localhost:8080/）。通过浏览器直接访问该地址就能测试该项目了，效果如图 1.48 所示。

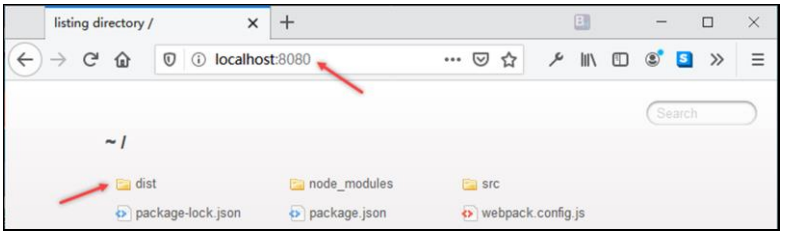

图 1.48　在浏览器中测试 Node 应用

如图 1.48 中的箭头所示，页面打开后显示的并不是预期的运行效果，而是项目的目录结构。这里，需要再次单击“分发（dist）”目录才可以运行 index.html 页面。

下面，我们体验一下 webpack-dev-server 插件的热加载功能。具体方法就是修改更新一下源文件（index.js）代码，然后注意观察一下命令行终端的变化，如图 1.49 所示。

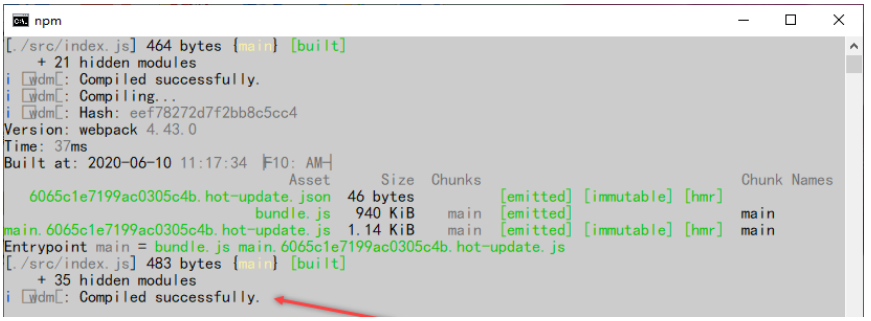

图 1.49　通过 webpack-dev-server 实现热加载

如图 1.49 中的箭头所示，在没有经过手动命令操作的情况下，webpack-dev-server 插件重新编译了源文件，这就是所谓的“热加载”功能。不过，比较遗憾的是浏览器页面并不会随之更新，需要手动操作来完成更新显示。

步骤 16　配合 HotModuleReplacementPlugin 插件实现 Node 应用热更新。

为了解决 webpack-dev-server 插件无法实现热更新操作的问题，可以使用 Webpack 工具自带的 HotModuleReplacementPlugin 插件，来配合 webpack-dev-server 插件完成热更新功能。

使用 HotModuleReplacementPlugin 插件时，需要在 Webpack 配置文件（webpack.config.js）中增加 plugins 节点的配置，代码如下：

```
plugins: [
    new webpack.HotModuleReplacementPlugin()
  ]
```

然后，再次尝试修改更新一下源文件（index.js）代码，注意观察浏览器页面的变化，如图 1.50 所示。

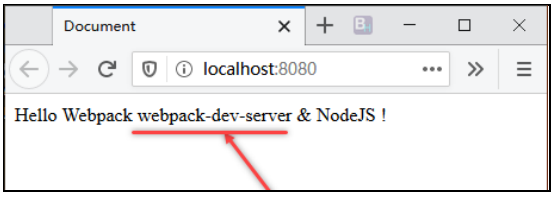

图 1.50　浏览器页面热更新

如图 1.50 中的箭头所示，浏览器页面的内容通过“热更新”功能进行了自动更新。至此，关于使用 Visual Studio Code 开发工具通过 Webpack 开发 Node 应用的基本方法，这里就基本介绍完成了，希望读者能熟练掌握这个集成开发工具。