第 1 章　Node.js 概述

Node.js 是一个 JavaScript 运行环境，由 Ryan Dahl 基于谷歌的 Chrome V8 引擎开发而来。由于 JavaScript 是单线程、事件驱动的语言，Node.js 利用了这个优点，采用事件循环、异步 I/O 的架构，可以编写出高性能的服务器；同时 Node.js 的诞生使得 JavaScript 的运行可以脱离浏览器平台，这极大促进了 JavaScript 语言的发展。

本章首先介绍 Node.js 产生的背景及其高性能的特点和应用场景；接着在不同系统上演示 Node.js 环境的搭建，并指导如何创建第一个 Node.js 程序；最后对开发工具 IDE 进行介绍并演示 Node.js 程序的调试方法。

本章涉及的主要知识点如下：

- Node.js 的发展历程、特点和使用场景；
- 不同操作系统下的 Node.js 环境搭建；
- 创建第一个 Node.js 程序；
- Visual Studio Code 工具的使用及 Node.js 程序的调试方法。

注意：Node.js 本身采用 C++编写，用于提供 JavaScript 程序的运行环境。

1.1　Node.js 简介

本节首先介绍 Node.js 的诞生背景、高性能特点以及 Node.js 的实际应用场景；接着介绍 Node.js 发展历程，以便让读者对 Node.js 有整体的认识。

1.1.1　Node.js 是什么

官方定义：Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行时环境（Runtime Environment）。它是一个能让 JavaScript 脱离浏览器运行在服务器端的开发平台，这使得 JavaScript 可以像 Java、Python 和 PHP 等服务器端语言一样进行服务器端的开发。因此通过学习 JavaScript 和 Node.js 可以快速掌握全栈开发技术。

从定义中得出两点：

- Node.js 是一个 JavaScript 运行时环境；
- Node.js 基于 Chrome V8 引擎。

JavaScript 运行时环境又称 JavaScript 引擎，它将操作系统底层相关操作（如解释编译、内存管理和垃圾回收等）进行抽象封装，使得开发者无须关心底层细节。

也就是说，Node.js 运行时环境提供了诸多模块 API，开发者只需要通过 JavaScript 调用相应的模块 API 即可完成不同的业务需求。JavaScript 与 Node.js 的关系，类比 Java 与 JDK 的关系。Node.js 提供了大量的内置模块，将在第 4 章详细介绍。

注意：运行时的不同必然导致编程 API 的不同。JavaScript 诞生之初主要用于制作网页特效，而网页运行于浏览器中，因此浏览器中的运行时可以使用 JavaScript 包含的 ECMAScript、DOM（文档对象模型）、BOM（浏览器对象模型）三部分 API。但是由于 Node.js 脱离了浏览器，因此 DOM 和 BOM 在 Node.js 环境中无法使用。

清楚了什么是运行时后，再来分析 Node.js 为什么要基于 Chrome V8 引擎进行封装。

这还得从 Node.js 创始人 Ryan Dahl 说起，“大佬”总是惊人的相似，和 Vue（全称为 Vue.js，本书简写为 Vue）框架创始人一样，Ryan Dahl 并非科班出身，他厌倦了代数拓扑学从而放弃攻读博士学位，在弃学旅行的过程中成为一名使用 Ruby 的 Web 开发者。

随着承接项目的增多，Ryan Dahl 在两年之后成为 Web 服务器性能问题专家。他曾尝试使用 C、Ruby、Lua 来解决 Web 服务器高并发的问题，虽然都失败了，但却找到了解决问题的关键：非阻塞、异步 I/O。

2008 年 9 月，谷歌发布了 V8 引擎，Ryan Dahl 仔细研究发现这是一个绝佳的 JavaScript 运行环境，单线程、非堵塞异步 I/O。于是他想到 JavaScript 本身就是单线程，浏览器发起 AJAX 请求也是非阻塞的，基于这个原理如果将 JavaScript 和异步 I/O 以及 HTTP 服务器集合在一起就能解决高并发的问题。根据这个思路，在接下来几个月的时间里，Ryan Dahl 独自完成了 Web.js（后来改名为 Node.js）的开发，并于 2009 年 5 月正式推出 Node.js。Node.js 默认集成了现在被我们熟知的 NPM 模块管理工具，用于简化 Node.js 模块源代码的管理。2010 年，Node.js 获得 Joyent 公司赞助，Ryan Dahl 加入 Joyent 公司，大力推动了 Node.js 的发展。

1.1.2　Node.js 能做什么

由于 Node.js 是一个开放源代码的 JavaScript 运行时环境，采用事件驱动、异步 I/O 的模式，所以非常适合轻量级、快速的实时应用（如协作工具、聊天工具、社交媒体等）以及高并发的网络应用。

据 Stack Overflow 统计，Node.js 是非常受欢迎的技术之一，国内外都有大量公司使用 Node.js 构建应用。例如：

- 雅虎：在 2009 年（Node.js 首次发布不到一年）就开始使用，雅虎博客证实其网络应用中有 75%是基于 Node.js 的。
- 领英：2011 年其平台用户已突破 6300 万，通过从 Ruby on Rails 转向 Node.js，完成从同步系统迁移到异步系统，使得服务器数量从 15 台减少到 4 台，流量服务在原有基础上提升了 1 倍，程序运行速度提升了 2～10 倍。
- Uber：世界著名的网约车平台，其应用程序使用了一些 Node.js 工具，虽然其不断引入新技术，但是 Node.js 仍是其基础。
- PayPal：2013 年从 Java 迁移到 Node.js，使得其页面响应时间缩短为 200ms，每秒处理请求的数量在原有基础上增加了一倍。

- 阿里巴巴：2017 年，优酷除账户模块和土豆部分页面使用 Node.js 外，其他 PC 和 HTML 5 的核心页面仍然采用的是 PHP 模板渲染，经过 Node 改造后，完成了从 PHP 到 Node.js 的迁移，成功地完成了 2019 年的“双 11”重任。阿里巴巴是国内 Node.js 的布道者，开源了基于 Node.js 的框架 Egg，产品大量使用 Node.js，包括语雀、淘宝、阿里云和天猫等。
- 腾讯：每逢体育赛事或重大节日，腾讯视频直播都是每秒数亿次的高并发请求，这充分证明 Node.js 在高并发场景下的出色能力。同时，腾讯 NOW 直播、花样直播等产品也在广泛使用 Node.js。

注意：随着业务需求的变化和新技术的引入，系统架构可能会发生变化。

随着众多公司加入 Node.js 阵营，诞生了非常多的基于 Node.js 的 Web 框架，如 Express、Koa、Meteor 和 Egg 等。除了这些大型框架，还有一些非常优秀的应用或模块也可以提升工作效率，值得我们研究。

- Socket.io 是一个 WebSocket 库，包含客户端 JavaScript 和服务器端的 Node.js，其目标是在不同浏览器和移动设备上构建实时应用，非常适合构建 Web 聊天室。
- Hexo 是一款基于 Node.js 的静态博客框架，依赖少且易安装，可以生成静态页面托管到 GitHub 上，是搭建博客的首选框架之一。
- Node Club 是一个使用 Node.js 和 MongoDB 开发的新型社区软件，界面简洁，功能丰富。Node.js 中文社区就是使用此程序搭建的。

除此之外还有一些非常好用的工具：Bower.js、Browserify 和 Commander 等。

Node.js 之所以得到众多公司和开发者的青睐，主要原因有以下几点。

1．跨平台

基于 Node.js 开发的应用，可以运行在 Windows、Linux 和 macOS 平台上，实现一次编写、处处运行。国内很多开发者都是在 Windows 平台上开发，然后部署到 Linux 服务器上运行。

2．异步I/O

I/O 表示输入/输出（input/output）。软件应用的性能瓶颈往往就在 I/O 上，无论网络 I/O 还是读取文件 I/O 都可能会耗费大量时间。这种情况下如果顺序执行多个耗时的任务，则总任务完成的时间等于各项任务依次执行完成时间之和，显然这是不明智的。为了解决这类问题从而引申出了两个概念：同步、异步。

- 同步：任务一件一件地依次执行，上一个任务完成后才能进入下一个任务。
- 异步：任务一件一件地依次执行，但不用等到任务执行完成便可以进入下一个任务。为了得到上一事件执行的结果，需要额外开辟存储空间将未执行完的任务放入一个事件队列，等任务执行完成后，再将其取出进行处理。

Node.js 采用异步 I/O，不必等到 I/O 完成即可执行其他任务，从而提高性能。这就是典型的用空间换时间。

3．单线程

Node.js 只用一个主线程来接收请求，接收到请求后将其放入事件队列，继续接收下一次请求。当没有请求或主线程空闲时就会通过事件循环来处理这些事件，从而实现异步效果，满足高并发场景。

需要注意的是，主线程空闲时去处理事件队列并非亲自去处理 I/O 操作，仅仅是去取执行结果，根据结果完成后续操作而已。那么这个 I/O 究竟是谁去完成的呢？实际上需要依赖操作系统层面的线程池。

简单理解就是 Node.js 本身是一个多线程平台，只是对 JavaScript 层面的任务处理是单线程的。如果是 I/O 任务则由操作系统底层线程池处理，如果是其他任务则由主线程自己完成，而如数据加密、数据压缩等需要 CPU 耗时处理的任务，依然是由主线程依次执行，同样可能会造成阻塞。

注意：由此得出，Node.js 非常适合 I/O 密集型场景，不适合 CPU 密集型任务。

4．事件驱动

传统的高并发场景中通常是采用多线程模型，为每个任务分配一个线程，通过系统线程切换来弥补同步 I/O 调用时间的开销，这是典型的时间换空间。

Node.js 是单线程模型，避免了频繁的线程间切换。它维护一个事件队列，程序在执行时进入事件循环（Event Loop）等待事件的到来；而每个异步 I/O 请求完成后会被推送到事件队列，等待主线程对其进行处理。这种基于事件的处理模式，才使得 Node.js 中执行任务的单线程变得高效。

事件循环机制如图 1.1 所示。

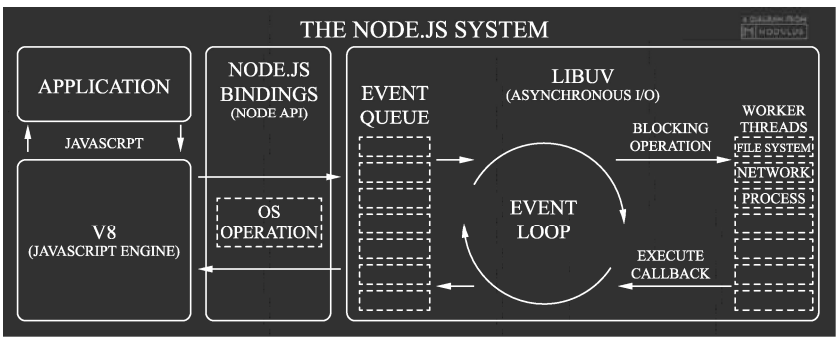

图 1.1　Node.js 事件循环机制

5．支持微服务

微服务（Microservices）是一种软件设计风格，其把复杂业务进行拆分，根据特定功能完成单一服务划分，这些服务可以单独部署并能最小化地集中管理。

Node.js 本身就轻量且跨平台，易于构建 Web 服务，支持从前端到后端数据库的全栈操作，因此非常适合用于创建微服务。

通过 Node.js 内置的 HTTP 等网络模块，能够快速创建微服务应用。除了 Node.js 内置模块外，Node.js 生态中也有很多成熟、开源的微服务框架可以用于创建微服务应用，如 Seneca、腾讯的 Tars.js 等。

1.1.3　Node.js 架构原理

Node.js 是基于 Chrome V8 引擎封装的 JavaScript 运行时环境，采用 C++编写，组成结构如图 1.2 所示。

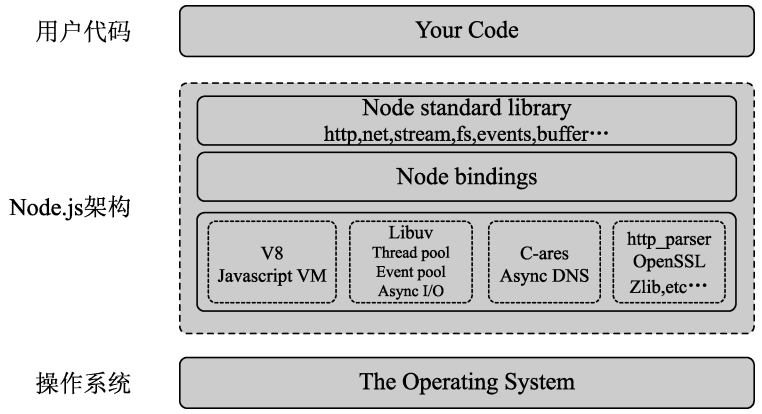

图 1.2　Node.js 的组成结构

从图 1.2 中可以看出，Node.js 由 3 层组成：

- 核心模块层（Core Modules）：也称 Node standard library 或 Node.js API，是 Node.js 专门提供给开发人员使用的标准库，包含 HTTP、Buffer 和 fs 等模块，此层采用 JavaScript 编写，可以使用 JavaScript 直接调用，其下各层采用 C++/C 编写。
- Node.js 绑定层（Node bindings）：是连接 JavaScript 和 C++的桥梁，封装了 Chrome V8 引擎和 Libuv 等其他功能模块的细节，向上层提供基础的 API 服务。
- JavaScript 运行时及功能库：此层是支撑 Node.js 运行的关键，由 C++/C 实现，包含 Chrome V8 引擎及 Libuv、C-ares、http_parser、OpenSSL 和 zlib 等库。

Chrome V8 引擎由谷歌开源，Node.js 将其封装提供给 JavaScript 运行时环境，其可以说是 Node.js 的发动机；Libuv 是为解决 Node.js 跨平台提供异步 I/O 功能而封装的一个库；C-ares 是封装了异步处理 DNS 相关功能的库；http_parser、OpenSSL 和 zlib 等库则提供了包括 HTTP 解析、SSL、数据压缩等功能。

接下来再看看程序代码在 Node.js 上的执行流程。

（1）采用 JavaScript 语言调用 Core Modules 模块完成相应的业务功能。

（2）JavaScipt 运行在 Chrome V8 引擎上，Core Modules 核心模块通过 Bindings 调用下层的 Libuv、C-ares 和 HTTP 等 C++/C 类库，进而调用操作系统提供的底层平台功能。

注意：Node.js 的核心是 Chrome V8 和 Libuv，Ryan Dahl 对部分特殊用例进行了优化，提供了替代的 API，使得 Chrome V8 在非浏览器环境中运行得更好。

1.1.4　Node.js 的发展历程

Node.js 在 2009 年由 Ryan Dahl 封装并开发，发展至今版本更加趋于稳定，社区和平台也更加趋于成熟，这离不开广大开发者的无私奉献。如前面所述，目前很多高流量网站都采用了 Node.js 进行开发，或者在原本项目中加入 Node.js 作为中间层进行优化。

Node.js 的官网地址为 https://nodejs.org/，如图 1.3 所示。

图 1.3　Node.js 的官网

从图 1.3 中可以看出，Node.js 分为两个版本：LTS（长期维护版）和 Current（尝鲜版），截至写书时，LTS 的版本为 20.11.1，Current 的版本为 21.6.2。

注意：由于 Current 的版本是根据开发进度实时更新的，所以可能存在 bug，生产环境建议使用 LTS 版本。

Node.js 并不是 JavaScript 框架，但从命名上可以看出，其官方开发语言是 JavaScript，这使得前端人员也可以快速转向全栈开发，JavaScript 也因此由前端脚本语言迅速拓展到服务器端开发行列，很多公司专门设置了 Node.js 工程师这个岗位。

Node.js 能迅速成为热门，除了功能强大之外，还有一个很重要的原因，那就是它提供了一个 NPM 包管理工具，它可以轻松管理项目依赖，这个包管理工具中有成千上万的模块或工具，可以快速提高开发人员的效率。很多前端人员可能就是因为 NPM 才知道 Node.js 的。

以下列举 Node.js 发展过程中的一些大事件。

- 2008 年 9 月，谷歌发布 Chrome V8 引擎。
- 2009 年 5 月，Ryan Dahl 正式推出 Node.js。
- 2010 年 3 月，基于 Node.js 的 Web 框架 Express 发布；Socket.io 诞生。
- 2010 年 8 月，Node.js 0.2.0 发布。
- 2010 年 12 月，Joyent 公司赞助 Node.js，因此 Ryan Dahl 加入 Joyent 公司进行全职开发。

- 2012 年 1 月，Ryan Dahl 辞职，不再负责 Node.js 项目，但这并没有影响 Node.js 的发展。
- 2013 年 12 月，Node.js 另外一个重量级框架 Koa 诞生。
- 2014 年 11 月，多位 Node.js 重量级开发人员不满 Joyent 公司的管理，辞职后创建了 Node.js 分支项 io.js，并于 2015 年 1 月发布了 io.js 1.0.0 版本。
- 2015 年 2 月，Joyent 公司携手各大公司和 Linux 基金会成立了 Node.js 基金会，并提议与 io.js 和解，同年 5 月和解达成，Node.js 与 io.js 合并，隶属于 Node.js 基金会。
- 2016 年 10 月，YARN 包管理器发布。
- 2019 年 3 月，Node.js 基金会和 JS 基金会合并成 OpenJS 基金会，以促进 JavaScript 和 Web 生态系统的发展。
- 2020 年 4 月，Node.js 发布 14.0.0 版本。
- 2021 年 4 月，Node.js 发布 16.0.0 版本。

从时间点可以看出，Node.js 发布不到一年就产生了 Express 框架，在其后两年多的时间里，如 LinkedIn、Uber、eBay 和沃尔玛等大公司先后加入 Node.js 阵营，极大推进了 Node.js 的应用和发展。

1.2　Node.js 的安装配置

对 Node.js 有了初步了解之后，进行 Node.js 开发之前，本节先分别演示如何在 Windows 和 Linux 系统中安装配置 Node.js。

1.2.1　在 Windows 中安装 Node.js

如果读者的计算机中还未安装 Node.js，需要先从其官网上下载对应的版本进行安装，下载地址为 https://nodejs.org/zh-cn/download/。相比于 Linux 平台，Node.js 对 Windows 平台的适配较晚，在微软的支持下，Node.js 的安装非常简单，只需要下载后缀为 msi 的安装包，然后双击运行并按提示进行操作即可。

（1）下载安装包。如果要下载最新的版本，那么直接在官网首页上下载即可。这里下载的 Node.js 版本为 V12.18.2，后面将以该版本为例展开介绍，其下载地址为 https://nodejs.org/download/release/v12.18.2/node-v12.18.2-x64.msi。

（2）双击安装包开始安装，在弹出的对话框中单击 Next 按钮，如图 1.4 所示。

（3）在弹出的对话框中勾选复选框协议后，单击 Next 按钮，如图 1.5 所示。

（4）在弹出的对话框中选择安装目录，单击 Next 按钮，如图 1.6 所示。

（5）在弹出的对话框中选择要安装的功能，这里保持默认即可，然后单击 Next 按钮，如图 1.7 所示。

（6）在弹出的对话框中保持默认选项，单击 Next 按钮，如图 1.8 所示。

（7）在弹出的对话框中单击 Install 按钮开始安装，如图 1.9 所示。

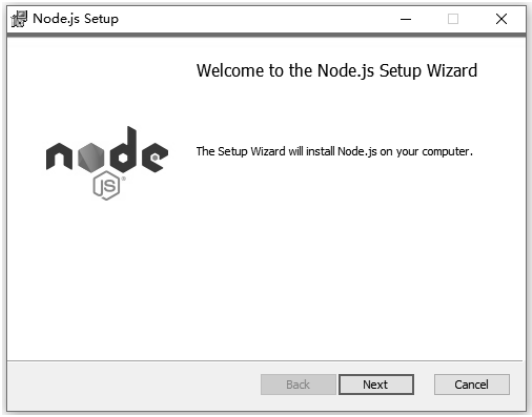

图 1.4　安装 Node.js

图 1.5　同意协议

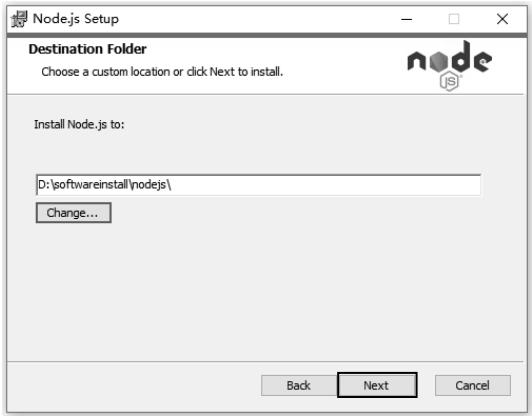

图 1.6　选择安装目录

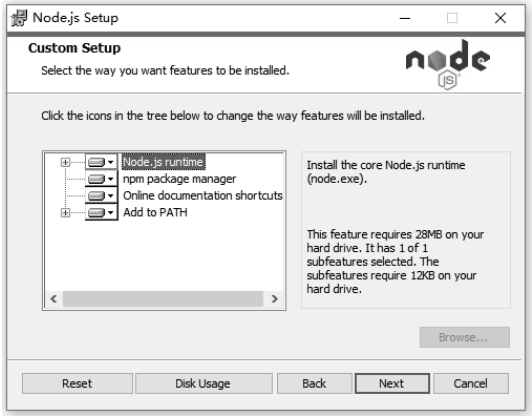

图 1.7　选择安装功能

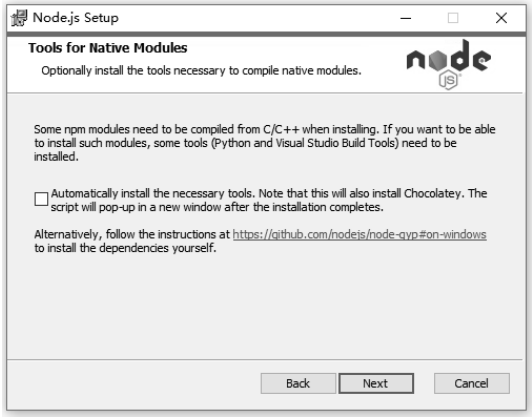

图 1.8　安装工具选项

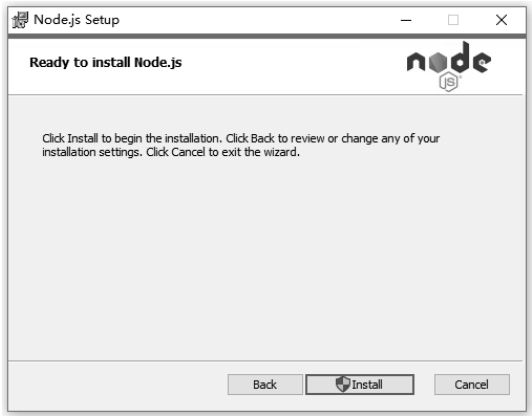

图 1.9　准备安装

（8）此时，将弹出一个安装进度对话框，如图 1.10 所示。安装完成后，弹出一个安装完成对话框，单击 Finish 按钮完成 Node.js 的安装，如图 1.11 所示。

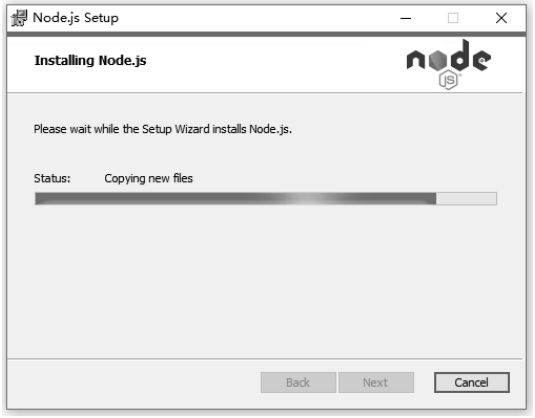

图 1.10　等待安装完成

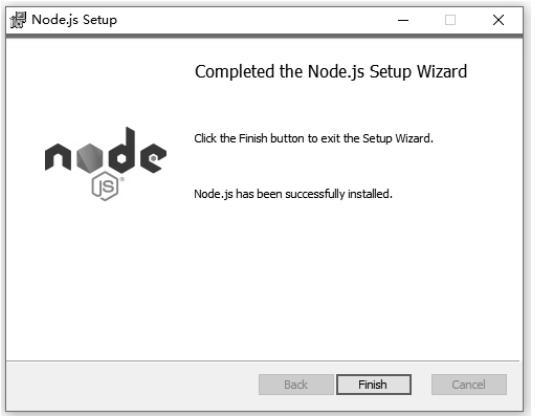

图 1.11　安装完成

1.2.2　在 Linux 中安装 Node.js

大多数情况下，我们的日常工作是在 Windows 中完成的，但上线时我们通常使用的服务器为 Linux 系统。众所周知，Linux 系统是免费和开源的，在服务器中的应用非常广泛，因此 Node.js 也提供了对应的安装包。

注意：Linux 系统有非常多的发行版（如 RedHat、Centos 和 Ubuntu 等），不同的发行版有自己的命令和格式，因此不同系统的安装略有区别。下面以 Centos 7.0 为例进行演示。

在 Linux 中安装软件比较简单，只需要下载对应的文件并解压运行即可。

（1）登录 Linux 系统。

登录 Linux 系统后，通过如下命令可以查看系统的版本，如图 1.12 所示。

```
cat /etc/redhat-release
```

图 1.12　查看 Linux 的版本

（2）下载 Node.js。

在终端中输入如下下载命令，等待下载完成，如图 1.13 所示。

```
wget https://nodejs.org/dist/v12.18.2/node-v12.18.2-linux-x64.tar.xz
```

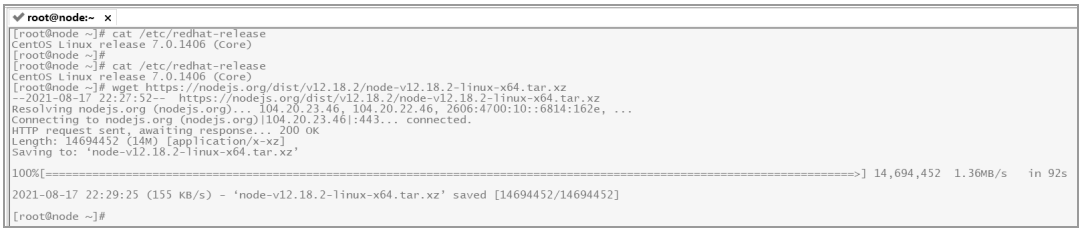

图 1.13　下载 Node.js

（3）解压文件。

通过如下命令解压下载的 node 文件，如图 1.14 所示。

```
cat /etc/redhat-release
```

```
[root@node ~]# ls
anaconda-ks.cfg  Desktop  Documents  Downloads  initial-setup-ks.cfg  Music  node-v12.18.2-linux-x64.tar.xz
Pictures  Public  Templates  Videos
[root@node ~]# tar -Jxf node-v12.18.2-linux-x64.tar.xz
[root@node ~]# ls
anaconda-ks.cfg  Documents  initial-setup-ks.cfg  node-v12.18.2-linux-x64         Pictures  Templates
Desktop          Downloads  Music                 node-v12.18.2-linux-x64.tar.xz  Public    Videos
[root@node ~]#
```

图 1.14　解压文件

（4）运行并测试。

切换到刚才解压后的文件夹 node-v12.18.2-linux-x64 的 bin 目录下，通过 ll 命令查看 node 文件并执行命令 node -v 测试安装是否成功，如图 1.15 所示。

```
[root@node ~]# cd node-v12.18.2-linux-x64/bin/
[root@node bin]# ll
total 47496
-rwxr-xr-x. 1 1001 1001 48634352 Jun 30  2020 node
lrwxrwxrwx. 1 1001 1001       38 Jun 30  2020 npm -> ../lib/node_modules/npm/bin/npm-cli.js
lrwxrwxrwx. 1 1001 1001       38 Jun 30  2020 npx -> ../lib/node_modules/npm/bin/npx-cli.js
[root@node bin]# ./node -v
v12.18.2
[root@node bin]#
```

图 1.15　测试是否安装成功

（5）添加环境变量。

通过以下命令，将 Node.js 和 NPM 添加到环境变量中，如图 1.16 所示。

```
//将 Node.js 放入环境变量
ln -s /root/node-v12.18.2-linux-x64/bin/node /usr/local/bin/node
//将 NPM 放入环境变量
ln -s /root/node-v12.18.2-linux-x64/bin/npm /usr/local/bin/npm
```

```
[root@node bin]# pwd
/root/node-v12.18.2-linux-x64/bin
[root@node bin]# ln -s /root/node-v12.18.2-linux-x64/bin/node /usr/local/bin/node
[root@node bin]# node -v
v12.18.2
[root@node bin]# ln -s /root/node-v12.18.2-linux-x64/bin/npm /usr/local/bin/npm
[root@node bin]# npm -v
6.14.5
[root@node bin]#
```

图 1.16　添加环境变量

1.3　编写第一个 Node.js 程序

【本节示例参考：\源代码\C1\Example_HelloWorld】

1.2 节已经搭建好 Node.js 的执行环境，由于 Node.js 可以直接运行 JavaScript 代码，所以本节将创建一个普通的 JavaScript 程序并演示如何在 Node.js 中执行这个程序。

1.3.1　创建 Node.js 应用

在本地磁盘工作目录下（笔者的本地目录为 E:\node book\C1\Example_HelloWorld）创建名为 HelloWorld.js 的 JavaScript 文件，采用任何一个文本编辑器（记事本、Notepad++、HBuilder、Visual Studio Code 等）打开并编辑文件，输入以下内容并保存文件。

代码 1.1　第一个Node.js程序：HelloWorld.js

```
var hello='hello world';
console.log(hello)
```

以上代码采用 JavaScript 语法书写，先通过 var 关键字声明 hello 变量并赋值为 hello world 字符串；接着通过 console 对象的 log 方法将对象值打印到控制台，这样 JavaScript 文件就创建好了。

1.3.2　运行 Node.js 应用

前面创建 JavaScript 的方法与以前在网页里书写 JavaScript 并没有区别，有一定 Web 基础的读者一定知道，如果想要在浏览器的控制台里打印输出结果，则需要先将 JavaScript 代码放入一个 HTML 文件中，然后在浏览器里打开即可。

与以往不同的是，这次我们希望这段代码在 Node.js 环境中运行，而非在浏览器中运行。因此需要打开 CMD 命令行工具，切换到 HelloWorld.js 文件对应的目录，然后执行 node 命令，观察输出结果。

注意：笔者的操作系统是 Windows 10，因此使用 CMD 命令行工具，如果是其他操作系统，则打开对应的终端即可。

（1）打开终端并切换到目标目录。按键盘上的 Windows+R 组合键，弹出的对话框如图 1.17 所示。

（2）单击“确定”按钮，在终端切换到 HelloWorld.js 文件所在的目录，如图 1.18 所示。

图 1.17　运行终端

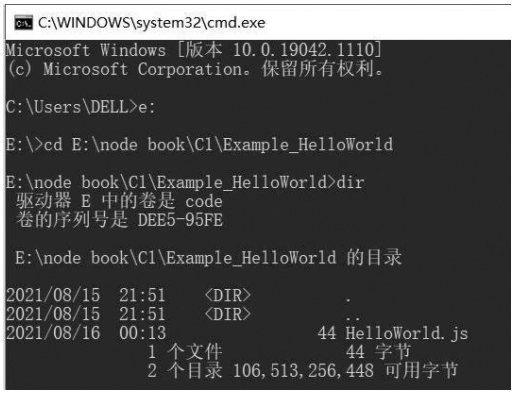

图 1.18　切换目录

在图 1.18 中，命令 e:表示切换到 E 盘，接着通过 cd 命令切换到 HelloWorld.js 文件所在的目录，然后通过 dir 命令查看文件信息。

注意：需要掌握 cd、dir 等常见的 cmd 命令。

（3）通过 node 命令执行文件。通过命令“node 文件名”直接在 Node.js 中执行 JavaScript 程序，如图 1.19 所示。

图 1.19　执行结果

可以看到，Node.js 打印输出了预期的 hello world 值。文件名可以不带后缀，Node.js 默认会自动查找后缀为.js 的文件。

通过以上操作，我们就实现了在 Node.js 中运行一个最简单的 JavaScript 程序的目标。

1.4　开发工具及其调试

【本节示例参考：\源代码\C1\Example_HelloWorld】

工欲善其事，必先利其器。在 1.3.2 节的示例中我们直接通过记事本使用 JavaScript 编写了一个 Hello World 文件，并直接在 CMD 终端中执行程序。在实际进行项目开发的过程中，往往需要借助 IDE 工具来提高开发效率，这类工具非常多，常用的有 Visual Studio Code、WebStorm、HBuilderX、Atom 和 EditPlus 等。其中，Visual Studio Code 深受好评，它是微软在 2015 年发布的一款跨平台、开源、免费的编辑器，下面就来学习如何使用它编写并调试 Node.js 程序。

1.4.1　安装 Visual Studio Code

可以从 Visual Studio Code 官网上下载最新的稳定版本，下载地址为 https://code.visualstudio.com/，其官网下载主界面如图 1.20 所示。

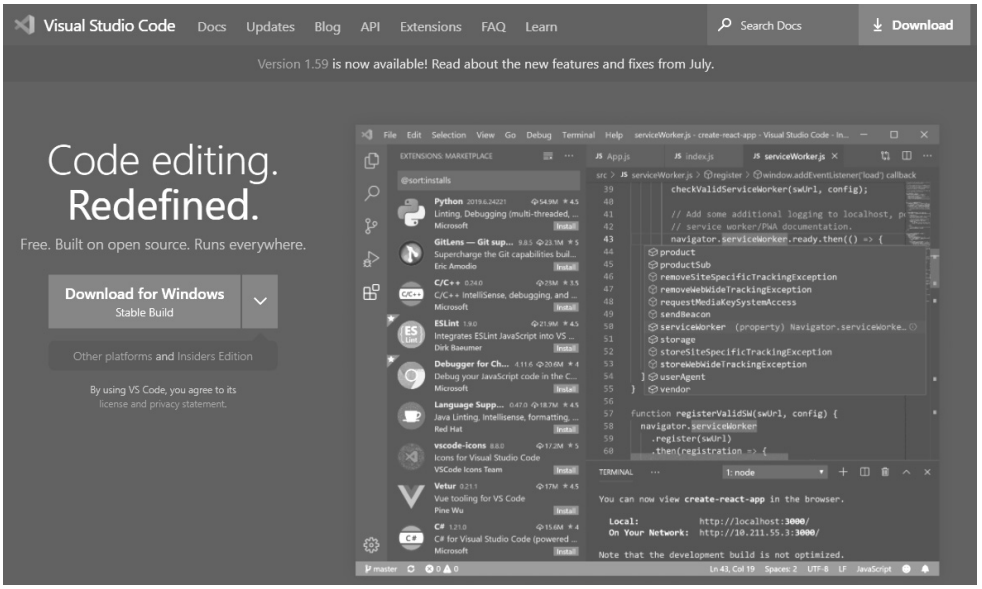

图 1.20　Visual Studio Code 官网界面

笔者是 Windows 10 系统，下载最新的稳定版，得到文件 VSCodeUserSetup-x64-1.54.3.exe。双击该文件进行安装，在弹出的对话框中选择同意协议，单击“下一步”按钮，如图 1.21 所示。

在弹出的对话框中，单击“浏览”按钮，选择合适的安装位置，然后单击“下一步”按钮，如图 1.22 所示。

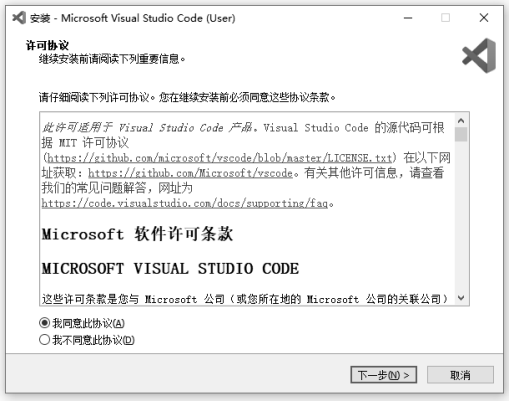

图 1.21　同意协议

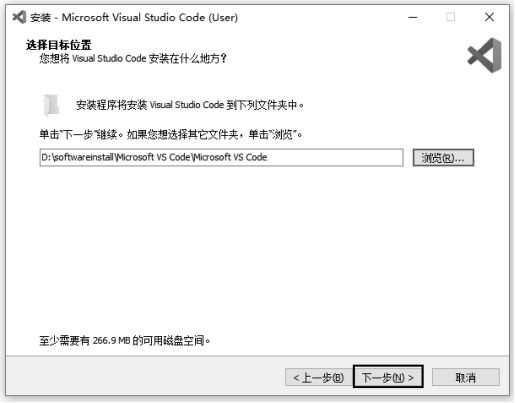

图 1.22　选择安装位置

后面的安装保持默认选项，直接单击“下一步”，如图 1.23 到图 1.27 所示。

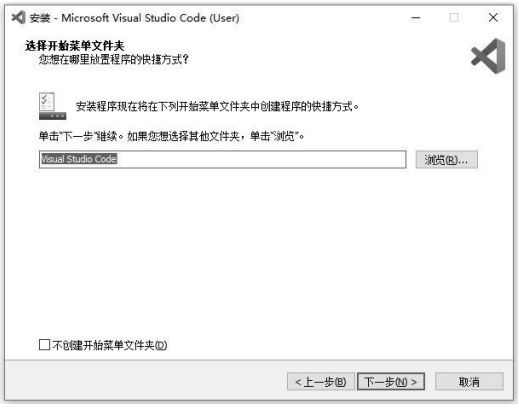

图 1.23　创建开始菜单

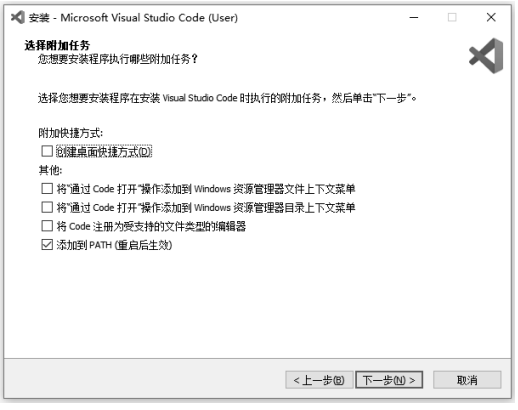

图 1.24　创建快捷方式

图 1.25　开始安装

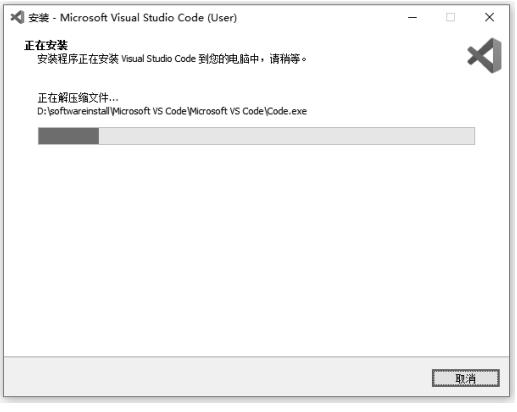

图 1.26　安装进度展示

图 1.27　安装完成

安装完成后，运行 Visual Studio Code，主界面如图 1.28 所示。

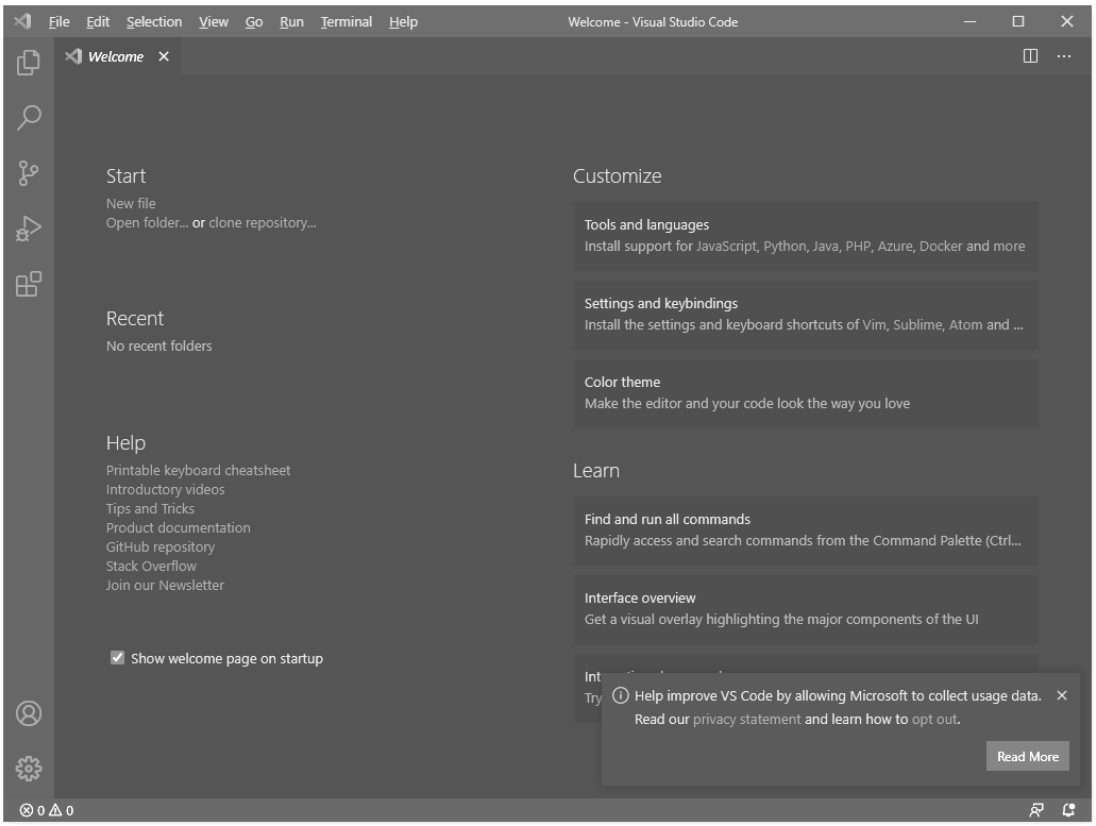

图 1.28　Visual Studio Code 主界面

将 1.3 节创建的应用对应的目录 Example_HelloWorld 拖曳到 Visual Studio Code 中，打开之前创建的程序，在 Visual Studio Code 中依然可以运行，如图 1.29 所示。

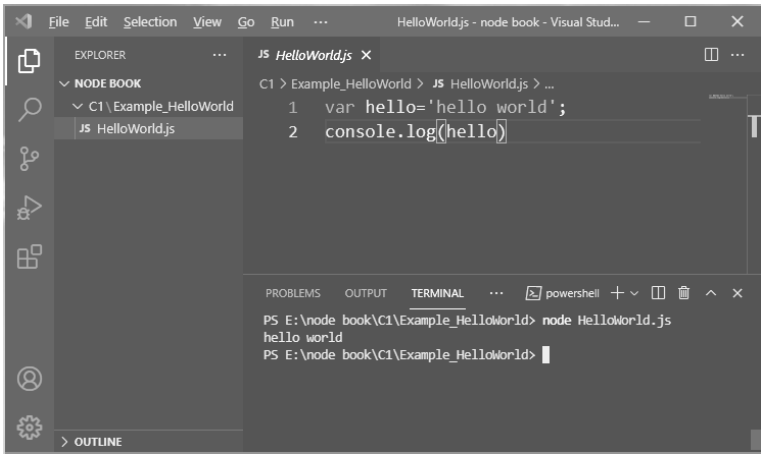

图 1.29　在 Visual Studio Code 中运行 HelloWorld 程序

1.4.2 调试 Node.js 程序

在网页里编写 JavaScript 程序，可以借助浏览器的功能进行调试，Node.js 并不运行在浏览器中，怎么进行调试呢？本节演示如何在 Visual Studio Code 中创建并调试 Node.js 程序。

（1）创建程序。

在 Visual Studio Code 中新建文件，如图 1.30 和图 1.31 所示。

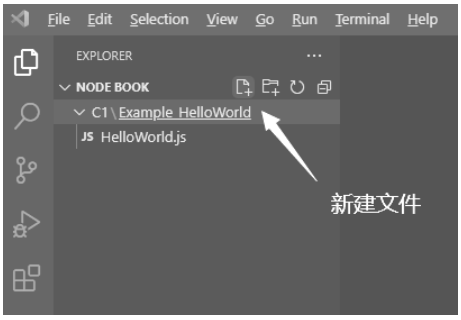

图 1.30　Visual Studio Code 新建文件

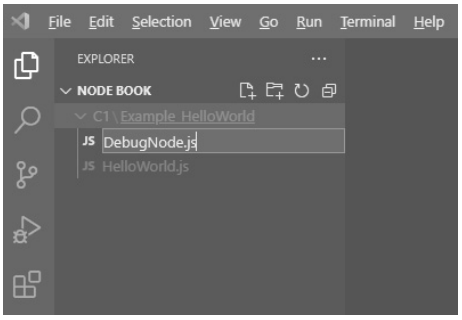

图 1.31　在 Visual Studio Code 中新建文件

输入内容见代码 1.2。

代码 1.2　第一个 Node.js 程序：HelloWorld.js

```
var username = 'heimatengyun';
var age = '18';
console.log(username, age)
```

（2）添加断点。

在 Visual Studio Code 中单击行号前的小红点设置断点，如图 1.32 所示。

（3）调试设置。

单击 Visual Studio Code 主界面左侧的调试按钮设置调试信息，如图 1.33 至图 1.35 所示。

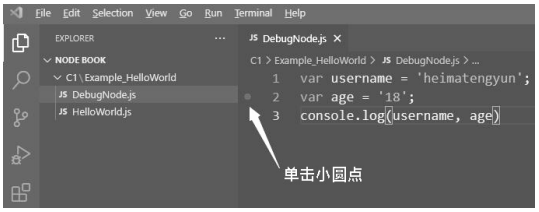

图 1.32　设置断点

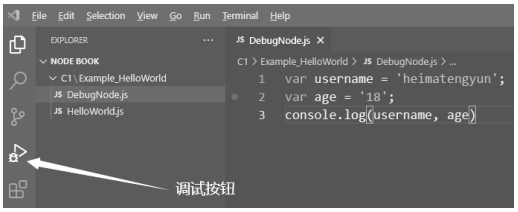

图 1.33　单击调试按钮

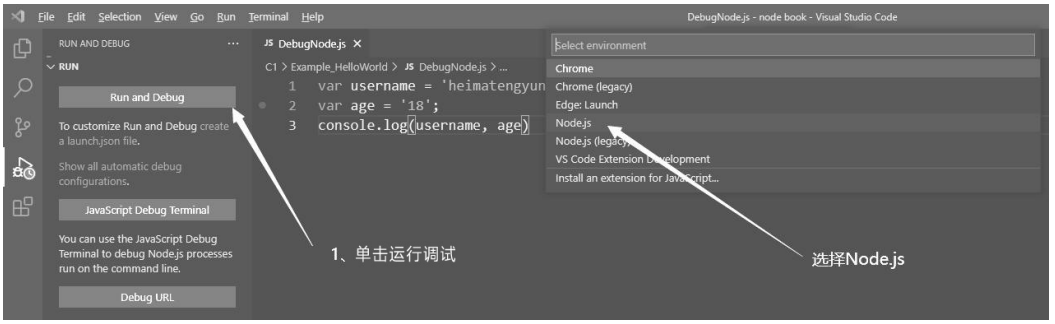

图 1.34　设置调试目标

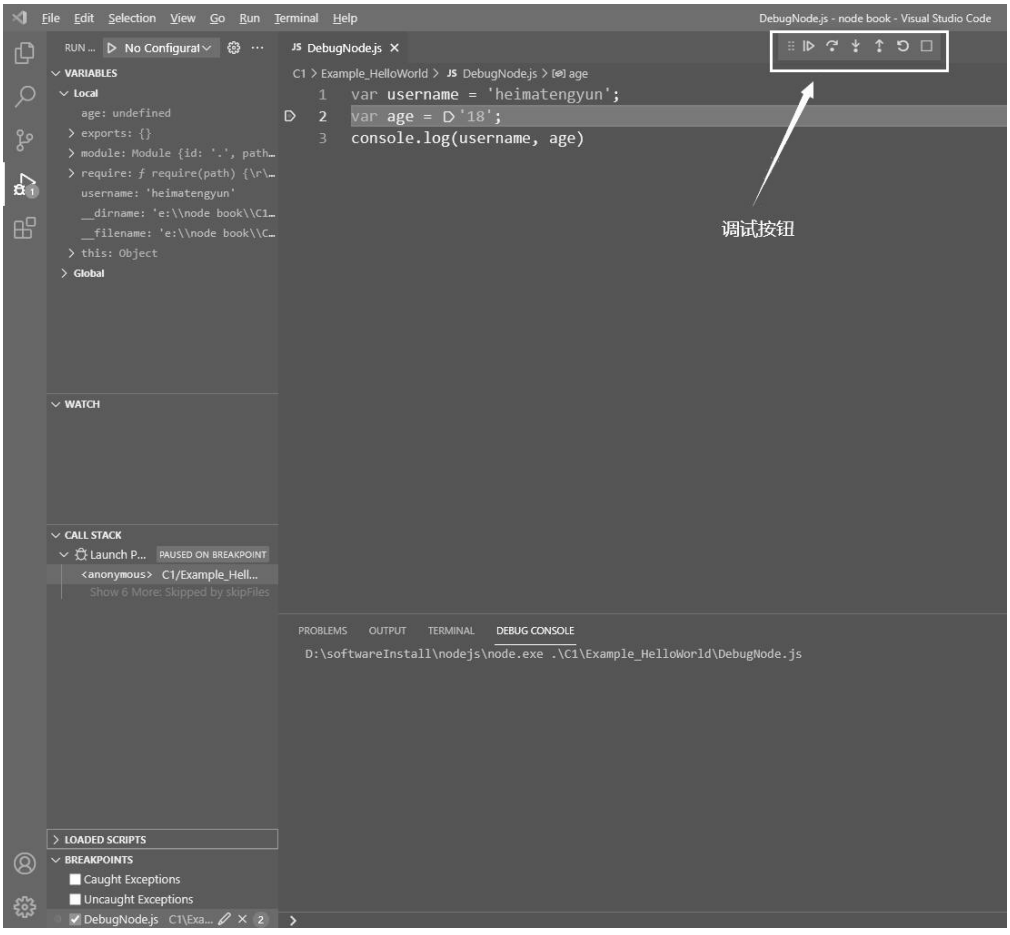

图 1.35　开启调试功能

调试功能开启后，程序就会停止在断点处，单击下一步按钮，程序将运行下一步，如图 1.36 所示。

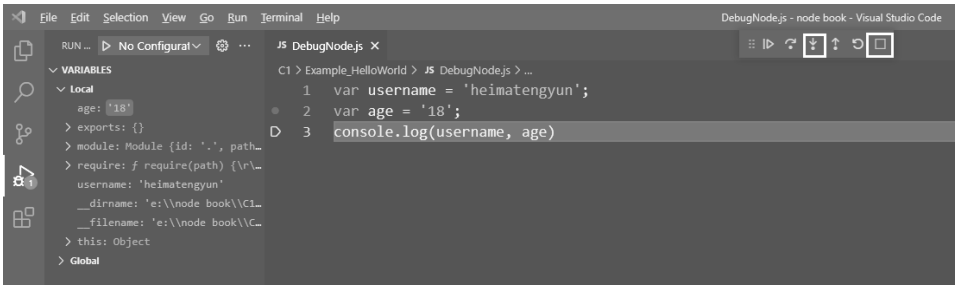

图 1.36　断点调试

如果想结束程序，单击最右边的停止按钮即可。

1.5　创建 Web 服务器案例

【本节示例参考：\源代码\C1\Example_HelloWorld】

Node.js 提供了 HTTP 模块，因此只需要引入该模块并调用 API 即可完成 HTTP 的相关功能。本示例采用 Node.js 搭建简单的 HTTP 服务器，实现用户访问不同的地址给出不同的响应内容的效果。

代码 1.3　创建Web服务器程序：HttpServer.js

```
//1. 引入 HTTP 模块
const http = require('http');
//2. 创建 HTTP 服务器
const server = http.createServer(function (request, response) {
    const url = request.url;                    //获取请求地址
    console.log(url)
    var answer = '';                            //设置响应内容
    switch (url) {
        case '/':
            answer = '欢迎访问首页';
            break;
        case '/login':
            answer = '欢迎来到登录页';
            break;
        default:
            answer = '非法闯入';
            break;
    }
    //设置响应头的编码格式为 UTF-8，避免中文乱码
    response.setHeader('Content-Type', 'text/plain;charset=utf-8');
    response.end(answer);
});
//3. 启动服务器监听 8888 端口
server.listen('8888', function () {
    console.log("服务器启动成功，访问：http://127.0.0.1:8888")
})
```

代码首先通过 require 引入 Node.js 内置的 HTTP 模块；接着通过 HTTP 对象调用 createServer 方法创建 HTTP 服务器，该方法可以接收一个回调函数，其包含两个参数，第一个参数包含用户请求的相关 URL 等信息，第二个参数用于设置向用户返回的响应信息；最后在 8888 端口上启动 HTTP 服务器。

在 Visual Studio Code 终端上通过 node 命令（node HttpServer）启动服务器，如果看到控制台输出如下信息则表示启动成功，如图 1.37 所示。

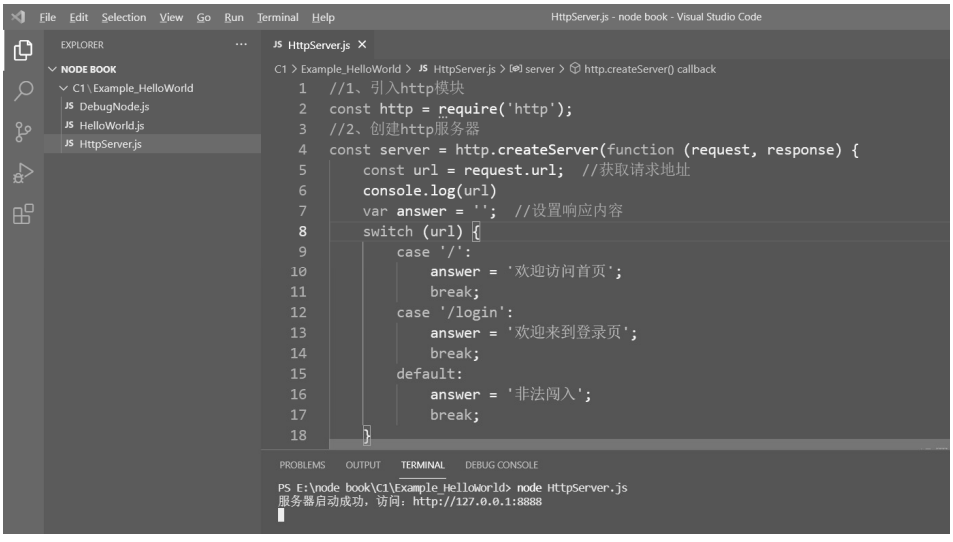

图 1.37　服务器启动成功

注意：选择菜单栏上的 Terminal | New Terminal 命令，可以打开终端面板。

当用户访问 http://127.0.0.1:8888 时，请求会进入回调函数并向用户返回“欢迎访问首页”的信息。当用户访问不同的地址时将会得到不同的结果，如图 1.38 所示。

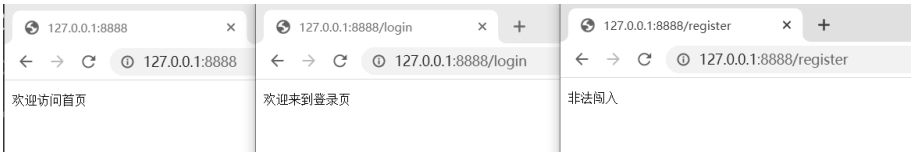

图 1.38　服务器的运行效果

1.6　本 章 小 结

本章先介绍了 Node.js 产生的背景、高并发的特点、架构组成及其发展历程；接着演示了在 Windows 和 Linux 中如何安装配置 Node.js 环境；最后讲解了在 Visual Studio Code 中创建和调试 Node.js 程序的技巧，并通过一个“创建 Web 服务器”实例演示了 Node.js 程序的创建过程和执行流程。通过本章的学习，期望读者能对 Node.js 有一个初步的认识。

第 2 章　Node.js 模块化管理

在 Node.js 运行环境中采用 JavaScript 语言编写代码，应当把一些通用的功能封装为模块以便复用。随着功能的完善和 bug 的修复，应该由版本号来标识这些模块的特定版本，并且模块之间可能存在依赖关系。这就需要用到模块管理工具来对模块的版本和依赖关系进行管理。

在 Node.js 中，通过 NPM 包管理器对模块进行管理，NPM 在 Node.js 的发展过程中起到了非常重要的作用。本章就介绍模块化的相关规范、Node.js 的模块以及如何通过 NPM 进行模块管理。

本章涉及的主要知识点如下：

- 了解 JavaScript 常见的规范及其发展历程；
- 掌握 CommonJS 规范与 ECMAScript 6（后面简写为 ES 6）模块规范的异同；
- 掌握 Node.js 内置的核心模块；
- 掌握如何自定义模块；
- 掌握如何通过 NPM 和 YARN 管理 Node.js 模块。

注意：JavaScript 语言由三部分组成，分别是核心部分、DOM 和 BOM。在 Node.js 中，只能使用核心部分，其中针对浏览器的 DOM 和 BOM 不再适用。

2.1　JavaScript 模块化

在进行大型系统开发时，需要采用模块化思维，将系统拆分为“高内聚、低耦合”的子模块，各个子模块专注处理各自的业务逻辑，模块之间相互协作，共同完成特定的功能。本节介绍 JavaScript 常见的模块化规范，需要掌握 CommonJS 规范及 ES 6 的模块规范。

2.1.1　什么是模块化

【本节示例参考：\源代码\C2\mutilFile】

众所周知，JavaScript 诞生之初仅是为了实现网页特效、完成网页表单数据校验。JavaScript 创始人仅用 10 天就完成了该语言的设计，因此并没有模块化地规范设计。

随着计算机硬件的发展、浏览器性能的提升，很多页面逻辑都可以在客户端完成。Web 2.0 时代的到来，使 AJAX 技术得到广泛应用，各种 RIA（富客户端应用）层出不穷，其间诞生了非常多的框架，如 ExtJS、EasyUI、Bootstrap、jQuery、React 和 Vue 等。随着业务逐渐赋值，前端代码日益膨胀，在这种情况下就要考虑使用模块化规范地进行管理。

那么，什么是模块化呢？JavaScript 模块化就是指 JavaScript 代码分为不同的模块，模块内部定义的变量作用域只属于模块内部，模块之间的变量名互不冲突；各个模块既相互独立，又可以通过某种方式相互引用协作。

注意：随着前端技术的发展，如 React、Vue 等这类新的 MVVM 框架逐步取代了以前的 ExtJS、jQuery 等框架。

模块化，简单说就是将代码文件拆分为不同的模块。那么模块又是什么呢？简单理解模块就是文件。因此模块化就是将一个复杂程序依据一定规则拆分并封装成几个文件块，然后通过一定规则将各个块文件组合在一起，块内部的数据和函数是私有的，只向外部暴露部分接口或函数与外部的其他模块通信。

这里说的模块化规则即接下来要介绍的模块化规范。在此之前，我们先思考一下，拆分代码会带来什么问题？

以计算工资为例，假设工资的计算公式为：工资=当月奖金+当月实际出勤工资，实际出勤工资=每日工资×出勤天数，每日工资=基本工资/22 天。按上述思路，将计算工资的功能分别拆分到 4 个文件 a.js、b.js、c.js 和 man.html 中，实现运行 man.html 文件后在浏览器控制台中输出工资，具体实现见代码 2.1 至代码 2.4。

代码 2.1　a.js文件

```
//a.js
var bonus = 50000;                    //当月奖金
var monthSalary = salary + bonus;  //当月工资=实际出勤所得的基本工资+当月奖金
```

在 a.js 文件中计算当月工资，当月工资（monthSalary）=当月出勤工资（salary）+当月奖金（bonus）。在该文件中定义了 bonus，而 salary 没在该文件中定义，而是定义在 b.js 文件中。

代码 2.2　b.js文件

```
//b.js
var day = 30;                                 //实际出勤天数
var salary = (basicSalary / 22) * day; //当月基本工资=每日工资×实际出勤天数
```

在 b.js 文件中计算当月出勤工资（salary），该项计算依赖于每月基本工资 basicSalary，而此变量定义在 c.js 文件中。

代码 2.3　c.js文件

```
//c.js
var basicSalary = 2000;                    //基本工资
```

在 c.js 文件中定义了基本工资 basicSalary 变量。

代码 2.4　main.html文件

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
    <title>计算工资</title>
</head>
<body>
    <!-- 严格按照此顺序引入文件 -->
    <script src='c.js'></script>
    <script src='b.js'></script>
    <script src='a.js'></script>
    <script>
        // var monthSalary=100;  //由于 a.js 文件存在 monthSalary,所以此处会覆盖
        console.log(monthSalary)
    </script>
</body>
</html>
```

main.html 文件分别引入了 3 个 js 文件，然后在控制台打印当月工资。需要注意的是文件之间存在相互依赖关系（a.js 依赖 b.js，b.js 依赖 c.js），也就是说，在上述代码中必须严格按照这个顺序引入这 3 个 js 文件，否则就会报错。假设 4 个文件都由不同人员开发，在 mian.html 中不小心又定义了一个与 a.js 文件同名的变量 monthSalary，则会导致同名被覆盖。

注意：这是一个比较致命的错误，当一个项目很大、文件很多并由很多人协同开发时，难免会出现同名的情况。同名覆盖导致的错误将难以排查。

因此可以看出，模块化不是简单地将文件拆分即可，拆分之后需要避免变量被全局污染、需要解决私有空间问题、需要维护模块与模块之间的依赖关系等。随着前端工程业务的复杂化，除了 JavaScript 代码模块化，还要考虑如图片和 CSS 文件等静态资源的模块化、代码开发完成后对其的压缩、合并优化等。

在前端模块化这条路上，前辈们做了非常多的尝试，从文件拆分、命名空间、立即执行函数、CommonJS 规范、ES 6 模块规范等，再到 Webpack 工具的诞生，每个新工具的诞生都是为了解决一些特定的问题，正是这些尝试促进了前端技术的飞跃发展。

2.1.2　模块化的发展史

在 JavaScript 的发展过程中，模块化规范也经历了不同的阶段，从诞生时间上看，大致如图 2.1 所示。

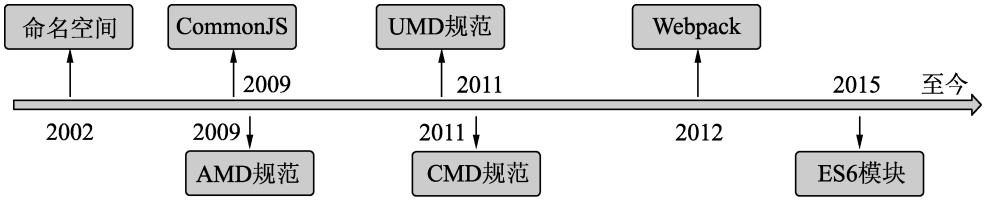

图 2.1　模块化的发展史

1．CommonJS规范

CommonJS 规范规定：一个单独的文件就是一个模块，每个模块都是一个单独的作用域。在一个文件中定义的变量、函数等成员都是私有的，对其他文件不可见。

CommonJS 有个显著的特点，即加载模块是同步的，只有等模块加载完成后才能执行后续的操作。Node.js 采用了此规范，由于 Node.js 主要用于服务器端编程，加载的模块文件都是存储在服务器本地的，加载速度较快，所以采用 CommonJS 规范比较合适。但是，如果 JavaScript 代码运行在浏览器环境，要从服务器加载模块，则必须采用异步加载模式，由此诞生了后续的 AMD 规范。

注意：CommonJS 是规范，Node.js 实现了其中的部分规范。

2．AMD规范

AMD 是 Asynchromous Module Definition 的缩写，AMD 和 CommonJS 不同，它是异步加载模块的，采用 define 函数定义模块。AMD 是 RequireJS 在推广的过程中对模块定义的规范，简单说就是 AMD 是规范，RequireJS 是对应的实现。

3．CMD规范

CMD 是 SeaJS 在推广过程中对模块定义的规范，其和 AMD 的主要区别是 CMD 推崇依赖就近原则，而 AMD 推崇依赖前置。

4．UMD规范

随着 CommonJS 规范和 AMD 规范的流行，产生了新需求，希望有一种更加通用的方案可以兼容这两种规范。于是 UMD（Universal Module Definition）通用模块规范就诞生了。

UMD 规范可以通过运行时或者编译时让同一个代码模块在使用 CommonJs、CMD 甚至 AMD 的项目中运行。未来，同一个 JavaScript 包运行在浏览器端、服务器端甚至 App 端只需要遵守同一个写法就可以了。它没有自己专有的规范，集结了 CommonJs、CMD 和 AMD 规范于一身。

UMD 规范的出现也是前端技术发展的产物，前端在实现跨平台的道路上不断地前进，UMD 规范将浏览器端、服务器端甚至 App 端都统一了，当然它或许不是未来最好的模块化方式，未来，ES 6+、TypeScript、Dart 这些拥有高级语法的语言可能会代替这些方案。

5．ES 6模块规范

前面介绍的这些模块规范都出自各大公司或社区，都是民间的解决方法。直到 2015 年，ECMA 发布了 ES 6 规范，正式成为 JavaScript 的标准。与 CommonJS 和 AMD 规范不同，ES 6 模块设计的实现是尽量静态化，使得编译时就能确定模块的依赖关系，而 CommonJS 和 AMD 模块只能在运行时才能确定。

CommonJS、AMD、CMD 和 UMD 模块规范的区别如表 2.1 所示。

表 2.1　前端模块规范的区别

规 范 名 称	说　　明
CommonJS（同步模块规范）	主要用于服务器端，典型实现Node.js
AMD（异步模块规范）	主要用于浏览器端，典型实现RequireJS

续表

规 范 名 称	说　　明
CMD（普通模块规范）	主要用于浏览器端，典型实现SeaJS
UMD（通用模块规范）	通用模块系统，兼容CommonJS、AMD、CMD规范
ES 6模块规范	JavaScript官方标准

6．Webpack打包工具

针对在浏览器端运行的 JavaScript 来说，模块的拆分意味着客户端需要向服务器端多次请求才能拿到页面运行所需的模块文件，这将占用更多的带宽，当网速较慢时甚至会影响用户体验。

针对这种情况，有人会想：有没有办法在编写程序的时候按模块拆分，在程序打包运行的时候把相关的文件打包组合在一起，以达到减少客户端与服务器端的传输，同时又不改变模块拆分的编程习惯呢？示意图如图 2.2 所示。

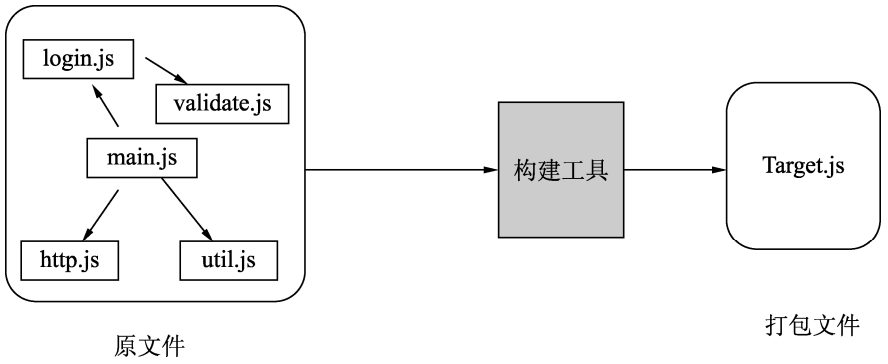

图 2.2　源文件打包合并思想示意

在这种背景下，产生了非常多的模块打包工具，其中，Webpack 应用非常广泛，其官方地址为 https://webpack.js.org/，官方功能介绍如图 2.3 所示。

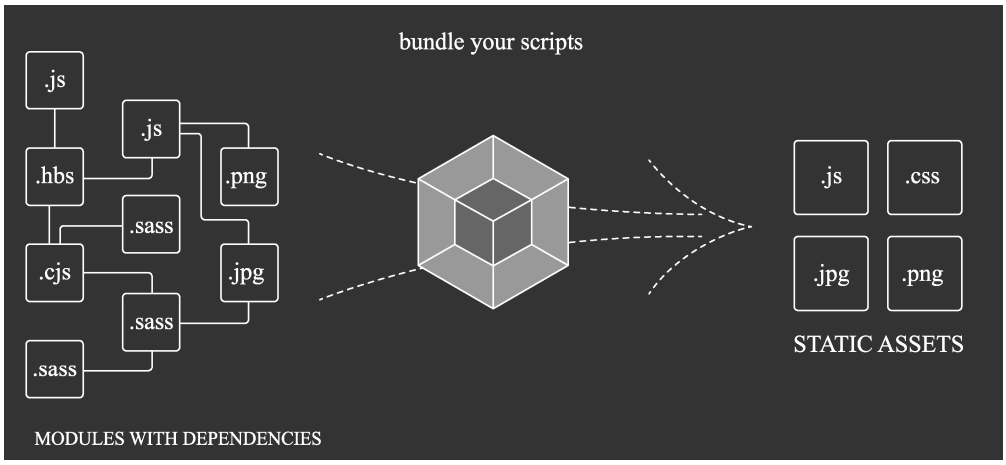

图 2.3　Webpack 功能示意

当然，除了目前流行的 Webpack 打包工具，还有 Grunt 和 Gulp 等其他构建工具。

注意：在 Node.js 中支持 CommonJS 和 ES 6 模块化规范，因此接下来介绍这两种方法的实现。

2.1.3　CommonJS 规范

【本节示例参考：\源代码\C2\CommonJS】

Ryan Dahl 在 2009 年开发 Node.js 时还没有 ES 6 的模块规范，他采用了 CommonJS 规范来实现，CommonJS 成为 Node.js 标准的模块化管理工具。同时 Node.js 还推出了 NPM 包管理工具，NPM 平台上的包均满足 CommonJS 规范。随着 Node.js 和 NPM 的发展，CommonJS 的影响越来越大，极大促进了后续的模块化工具的发展。

Node.js 应用由模块组成，采用 CommonJS 模块规范。每个文件就是一个模块，有自己的作用域，在一个文件里定义的变量、函数和类等都是私有的，对其他文件不可见。CommonJS 规范有如下特点：

- 每个文件都是一个 Module 实例。
- 所有文件加载都是同步完成的。
- 每个模块加载一次之后就会被缓存，以后再加载时直接读取缓存结果，要想让模块再次运行，必须清除缓存。
- 模块通过关键字 module 对外暴露内容。
- 文件内通过 require 对象引入指定的模块。
- 模块按照其在代码中出现的顺序加载。

CommonJS 的基本语法：

暴露模块：module.exports.xxx=value（xxx 表示导出的变量）或 module.exports=value。

引入模块：require(xxx)，如果是第三方模块，则 xxx 为模块名；如果是自定义模块，则 xxx 为模块文件路径。

既然一个文件就是一个模块，那么模块究竟暴露的是什么呢？CommonJS 规范规定，在每个模块内部，module 变量代表当前模块，这个变量有一个 exports 属性，即对外的接口。因此，加载一个模块，实际就是加载该模块的 module.exprots 属性。

下面通过一个例子来演示模块对外暴露的内容，创建模块文件 salaryModule.js，代码如下。

代码 2.5　薪资计算模块：salaryModule.js

```
//salaryModule.js：薪资计算模块
var basicSalary = 20000;                    //基本工资
var allSalary = function (bonus) {
    return basicSalary + bonus;             //工资=基本工资+奖金
}
module.exports.basicSalary = basicSalary;
module.exports.allSalary = allSalary;
```

在代码 2.5 中定义了一个变量 basicSalary 表示基本工资，allSalary 变量指向计算工资的匿名函数，该函数的作用是将传入的奖金加上基本工资从而得到本月的工资；最后通过 module.exports 分别将变量和函数导出以供其他模块使用。

模块定义好后，再创建一个 getSalary.js 文件用于读取模块的内容，代码如下。

代码 2.6　获取薪资模块的内容：getSalary.js

```
//getSalary.js 用于获取模块的内容
var salaryModule = require('./salaryModule.js');                    //引入模块
console.log("基本工资：" + salaryModule.basicSalary);               //20000
console.log("本月工资：" + salaryModule.allSalary(30000));          //50000
```

在上面的代码中，通过 require 命令引入 salaryModule 模块，并分别访问模块暴露出的变量和方法。在 Visual Studio Code 终端中通过 node getSalary 命令运行文件，可以看到得到了模块内暴露的 basicSalary 和 allSalary 的值，结果如图 2.4 所示。

图 2.4　获取模块的内容

从上面的示例中可以看出，require 命令的基本功能是读取并执行一个 JavaScript 文件，然后返回该模块的 exports 对象。

接下来深入研究 CommonJS 模块的加载机制，在模块被加载后，模块内部对变量的操作是否会影响导出的变量值呢？来看下面的例子。创建计数器模块 counter.js 文件，代码如下。

代码 2.7　计数器模块：counter.js

```
//counter.js
var counter = 1;                                    //计数器初始值为 1
function add() {
    counter++;                                      //计数器自增
    console.log("add 后的 counter 值：" + counter)
}
// function getCounter() {                          //获取模块内的值
//     return counter;
// }
module.exports = {                                  //导出模块
    counter: counter,
    add: add,
}
```

在模块中创建一个计数器 counter 并初始化为 1；同时定义一个 add 方法，在函数内对计数器自增并打印；最后分别将 counter 和 add 导出供外部模块调用。

创建 getCounter.js 文件，通过 require 引入 counter 模块，先打印模块内的 counter 值；接着调用 add 方法使模块内的 counter 自增；然后打印导出模块的 counter 的值，观察值的变化情况。

代码 2.8　调用计数器模块：getCounter.js

```
//getCounter.js
var counter = require('./counter');
console.log(counter.counter);//1
counter.add();//2
console.log(counter.counter);                    //1 获取的是模块导出值的备份
//console.log(counter.getCounter());             //2 //得到模块内的值
```

通过 node 命令运行 getCounter.js 文件，输出结果如图 2.5 所示。

图 2.5　计数器的运行结果

从运行结果中可以看到 counter 的初始输出值为 1；接着调用模块内的 add 方法，在方法内打印 counter 为 2；再次打印 counter 的值依然为 1。可以看出，调用 add 方法后，只是改变了模块内部的 counter 值，并没有影响模块导出的 counter 的值。

这是为什么呢？这就是 CommonJS 模块的加载机制，require 引入的是被导出值的备份，如果在模块内输出一个值，则模块内部的变化也不会影响这个值。可以通过模块暴露的函数获取模块内部变化后的值，将 counter 模块的 getCounter 方法取消注释，在 getCounter 中调用该方法即可获取模块内部发生变化的值，这一点与 ES 6 的模块化有重大差异。

2.1.4　ES 6 模块化规范

【本节示例参考：\源代码\C2\ES6】

ES 6 于 2015 年 6 月发布，它是 JavaScript 的官方规范，虽然它的诞生比 Node.js 晚了很多，但是 Node.js 毕竟是采用 JavaScript 语言编写的程序，因此随着发展，Node.js 也支持 ES 6 规范的模块。

ES 6 模块基本语法：

导出模块：export 变量/函数/类声明。通过 export 关键字将部分代码公开给其他模块。

引入模块：import {标识符} from 模块名。import 语句由两部分组成，一是需要导入的标识符，二是需要导入的模块文件。import 之后的花括号指明从给定模块导入对应的内容，from 关键字指明需要导入的模块文件。

将 2.1.3 节中的示例用 ES 6 的模块进行实现，创建 ES 6 目录并创建模块文件 salaryModule.js，代码如下。

代码 2.9　计算薪资模块：salaryModule.js

```
//salaryModule.js：ES 6 版本的模块导出
export var basicSalary = 20000;                    //基本工资，支持声明时导出
export function allSalary(bonus) {                 //具名函数
    return basicSalary + bonus;                    //月工资=基本工资+奖金
}
var yearendBonus=80000;                            //年终奖
export {yearendBonus};                             //支持先声明再集中导出
```

在上面的代码中：定义一个变量 basicSalary 表示基本工资；定义一个 allSalary 具名函数用于计算工资，该函数的作用是将传入的奖金加上基本工资得到本月的工资；定义 yearendBonus 变量表示年终奖。basicSalary 变量和 allSalary 函数在声明时通过 export 关键字导出，yearendBonus 变量先声明再通过 export 导出。这里分别演示了导出模块的两种方式。

注意：对比之前的 CommonJS 版本会发现，ES 6 版本不支持直接通过 export 导出匿名函数，除非使用 default 关键字。

模块定义好之后，再创建一个 getSalary.js 文件用于读取模块的内容，代码如下。

代码 2.10　获取薪资模块的内容：getSalary.js

```
//getSalary.js：ES 6 版本的模块导入
//导入模块
import { basicSalary,allSalary,yearendBonus } from "./salaryModule.js";
console.log("基本工资：" + basicSalary);              //20000
console.log("本月工资：" + allSalary(30000));         //50000
console.log("年终奖：" + yearendBonus);               //80000
```

在上面的代码中通过 import 关键字导入 salaryModule 模块，并分别访问模块暴露出的变量和方法。在 Visual Studio Code 终端中，切换到文件对应的目录，通过 node getSalary 命令运行文件，发现报错信息如图 2.6 所示。

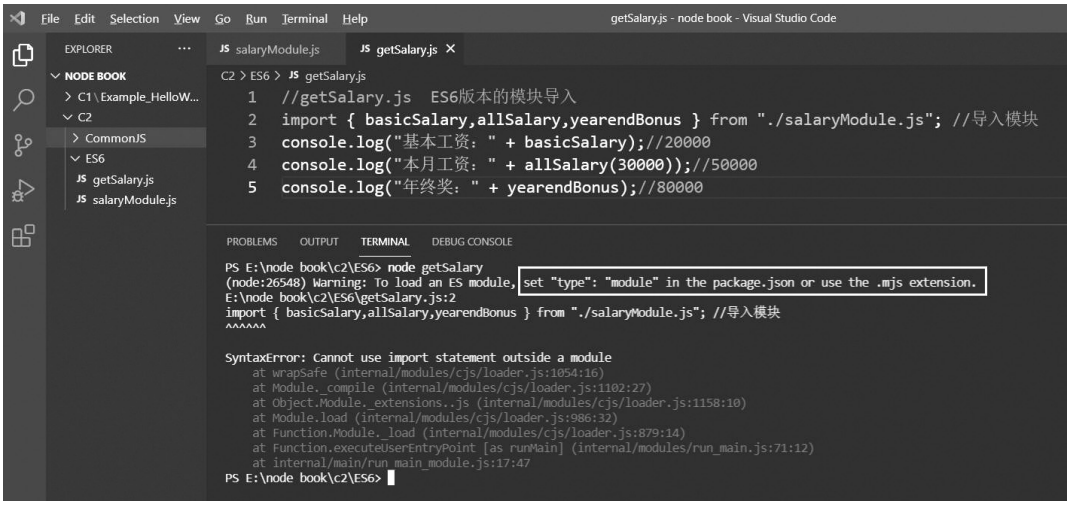

图 2.6　获取模块内容

根据提示信息，在文件目录下新建 package.json 文件并指定 type 类型为 module，代码如下。

代码 2.11　配置文件内容：package.json

```
{
    "type": "module"
}
```

再次在终端中运行 node getSalary 命令，可以正确得到模块里的值，如图 2.7 所示。

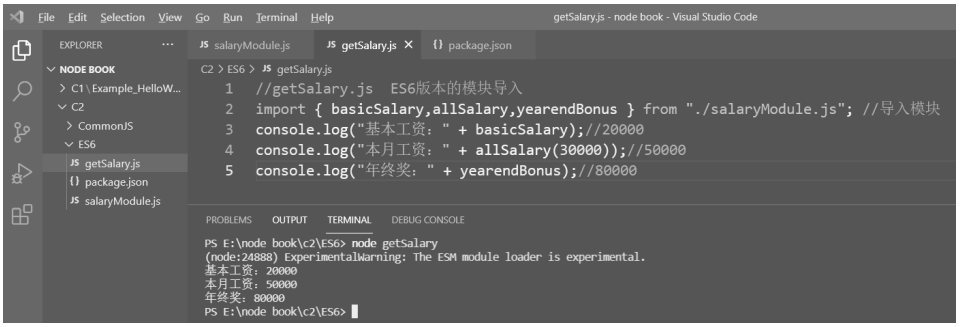

图 2.7　ES 6 模块输出结果

从上面的示例中可以看出，ES 6 模块通过 export 关键字导出模块，并在新模块中通过 import 关键字导入模块内容，这样就能成功访问模块导出的内容了。

ES 6 除了 import、export 和 from 关键字外，还有两个常用的关键字，即 as 和 default。as 关键字可以用于在导入或导出时指定别名；default 关键字用于设置默认导出的内容，可以导出匿名函数，每个文件只能有一个 default 关键字。

2.1.3 节提到 CommonJS 模块的加载机制与 ES 6 不同，接下来采用之前的计数器例子，使用 ES 6 模块导出，分析其不同之处。创建计数器模块 counter.js 文件，内容如下。

代码 2.12　计数器模块：counter.js

```
//counter.js
var counter = 1;                                    //计数器初始值为 1
function add() {
    counter++;                                      //计数器自增
    console.log("add 后的 counter 值：" + counter)
}
export { counter, add }
```

在模块中创建一个计数器 counter 并初始化为 1；同时定义一个 add 方法，在方法内对计数器自增并打印；最后通过 export 分别将 counter 和 add 导出供外部模块调用。

创建 getCounter.js 文件，通过 import 引入 counter 模块，先打印模块内的 counter 值；接着调用 add 方法使模块内的 counter 自增；然后打印导出模块的 counter 的值，观察值的变化情况。

代码 2.13　调用计数器模块：getCounter.js

```
//getCounter.js
import { counter, add } from "./counter.js";    //自定义模块需要写文件后缀名
console.log(counter);                           //1
add();                                          //2
console.log(counter);                           //2  //注意与 CommonJS 区别
```

通过 node 命令运行 getCounter.js 文件，输出结果如图 2.8 所示。

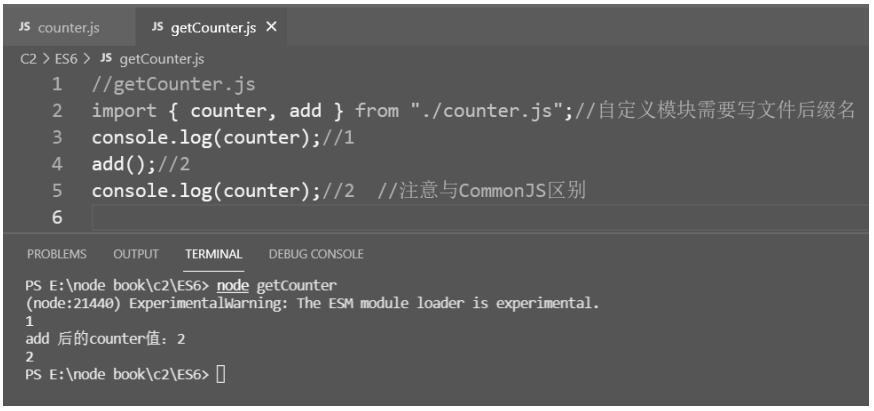

图 2.8　计数器的运行结果

从运行结果中可以看到：counter 初始输出的值为 1；接着调用了模块内的 add 方法，在方法内打印 counter 为 2；再次打印 counter 的值为 2。对比 CommonJS 模块，在这里可以看出二者的不同，这一点与 CommonJS 的模块化有重大差异。

2.2　Node.js 模块分类

Node.js 模块可以分为：核心模块、自定义模块和第三方模块。Node.js 自身提供的模块称为核心模块；在 NPM 仓库中有很多功能强大的第三方模块，这些模块可以极大提高开发效率；开发者也可以按照模块规范封装自定义模块。

2.2.1　核心模块

核心模块是 Node.js 为开发者提供的底层功能，包括一些常用的全局变量和 API。核心模块部分在 Node.js 源码编译过程中被编译为二进制执行文件，当 Node.js 启动时，核心模块被直接载入内存，因此当这部分模块被引用时，加载速度非常快。

注意：API 随着 Node.js 版本的发布而变更，不同版本的 API 不完全一致，使用时需要考虑这一点。

全局变量是无须进行引入，任何时候都可以访问的变量。需要注意的是，在 Node.js 中可以全局访问的变量不一定都是全局变量，全局可以访问的对象实际上包含几个部分，分别是 JavaScript 本身内置的对象、Node.js 的全局对象、Node.js 部分作用在模块作用域的变量（exports、module、__dirname、__filename、require 方法）。

Node.js 提供了一系列全局变量和模块，部分内容如表 2.2 所示。

表 2.2　Node.js的部分全局变量和模块

变量或模块	功 能 说 明
console	console是Node.js提供的简单控制台模块，类似于浏览器提供的JavaScript控制台，其提供了log、error和warn等方法，方便调试
global	global全局命名空间对象
process	进程对象，提供当前Node.js的进程信息并能实现控制
timer模块	timer模块定义了Immediate类、Timeout类用于实现定时器，常用于调度定时器和取消定时器的方法有setTimeout、clearTimeout、setInterval和clearInterval

除了上面的全局变量，Node.js 还提供了一系列核心模块，这些模块需要通过 require 关键字导入文件中进行使用，常用的模块如下：

- fs 文件系统模块：用于与文件系统交互。
- path 路径模块：用于处理文件和目录的路径。
- net 网络模块：提供异步网络 API，用于创建基于流的 TCP 或 IPC 服务器和客户端。
- HTTP 模块：用于创建 HTTP 服务器和客户端。
- events 事件触发器模块：用于事件处理，Node.js 的大部分重要的 API 都是围绕异步事件驱动架构构建的。
- TLS 安全传输层模块：提供基于 OpenSSL 构建的安全传输层协议（TLS）和安全套接字层（SSL）协议实现。

除了以上核心模块之外，还有一些重要的模块如 Buffer、dgram 和 DNS 等，这部分内容将在第 4 章详细介绍。

注意：更多核心模块，可参阅 Node.js 官方的 API 文档，网址为 https://nodejs.org/en/download/releases/。需要注意，不同版本的 API 不同，在上述页面中需要根据使用的版本，选择对应的文档进行查看。

2.2.2　自定义模块

【本节示例参考：\源代码\C2\CustomModule】

如果把所有代码都写在一个文件中，那么后期维护将变得非常困难，因此应该根据业务功能将文件进行拆分，这些被拆分的文件就是用户自定义模块。自定义模块可以采用 2.1 节中讲解的 CommonJS 规范，也可以采用 ES 6 的模块规范。

自定义模块可以导出变量、函数、对象，接下来演示采用 CommonJS 规范自定义模块的过程。创建 CustomMode 目录，新建文件 customModule.js 文件，代码如下。

代码 2.14　自定义模块：customModule.js

```
//自定义模块
var fileInfo = '这是一个自定义模块';
function showFileInfo() {
    return this.fileInfo
};
class authorInfo {
    constructor() {
        this.name = '黑马腾云';
```

```
        this.age = '18';
    }
    sayHello() {
        return `您好! 我是${this.name}`;
    }
}
module.exports = {                                    //可简写为 exports
    fileInfo,
    showFileInfo,
    authorInfo
};
```

在自定义模块中定义变量 fileInfo、函数 showFileInfo 和类 authorInfo，然后通过 module.exports 导出模块供外部调用。其中，module.exports 可以简写为 exports。

注意：在上述代码中定义类采用了 class 关键字，这是 ES 6 新增的语法，如果读者对 JavaScript 及 ES 6 不太熟悉，可以参考第 3 章的内容。

模块定义好之后，新建 useCustomModule.js 文件，在文件中使用模块，代码如下。

代码 2.15　自定义模块：customModule.js

```
//使用自定义模块
//模块文件后缀.js 可以省略；./不可省略
var customModule = require('./customModule.js');
console.log(customModule.fileInfo);                   //使用导出的变量
console.log(customModule.showFileInfo());             //调用导出的函数
var author = new customModule.authorInfo();
console.log(author.sayHello());
```

在上述代码中通过 require 关键字导入自定义模块，参数为模块路径，后缀名可以省略不写。在终端中执行代码，运行结果如图 2.9 所示。

图 2.9　使用自定义模块

可以看到模块运行成功了，说明模块可以导出变量、函数和类。在实际自定义模块时需要根据自身业务进行拆分，拆分的原则是模块之间应该“高内聚、低耦合”。

2.2.3　第三方模块

NPM 仓库中有非常多优秀的模块，如 Moment 和 Marked 等，这些模块可以通过 npm install

命令安装到项目中，这些模块的使用非常简单，只需要按照对应的文档即可轻易调用对应的功能。

注意：包是在模块基础上的进一步封装，这些第三方模块也称为包，NPM 则是 Node.js 提供的管理这些包的工具。

Moment 是一个 JavaScript 日期处理类库，官网地址为 https://momentjs.com/。通过 Moment 可以很方便地实现日期格式化。

Marked 是一个用 JavaScript 编写的 Markdown 解析和编译器，最初只能在 Node.js 中使用，目前已完全兼容客户端浏览器，其官网地址为 https://marked.js.org/。

2.3　NPM 包管理器

NPM（Node Package Manager）是随同 Node.js 一起安装的包管理工具，也是 Node.js 官方推出的默认的包管理工具，用于管理模块的安装、卸载和依赖性。NPM 仓库中有非常多功能成熟的模块和资源，对于模块复用非常便利，极大地提高了开发效率。随着 Node.js 的热门，以谷歌和 Facebook 为首的几家公司一起开发了一款性能更高的包管理工具 YARN。虽然 YARN 的性能更高，但是它并没有完全取代 NPM，Node.js 也支持使用 YARN 对模块进行管理。

2.3.1　NPM 简介

NPM 是一个采用 JavaScirpt 编写的 JavaScript 模块管理工具，遵循 CommonJS 规范，简单理解，NPM 就是一个远程软件仓库。需要注意的是，NPM 并不是由 Node.js 的作者 Ryan Dahl 开发的，他在 2009 年开发出 Node.js 之后，发现还缺少一个包管理器，碰巧的是，当时 NPM 的作者 Isaac Z.Schlueter 在推广 NPM 时遇到瓶颈，于是给当时很热门的 jQuery、Bootstrap、Underscore 等作者发了邮件，希望他们能将这些框架放到仓库中，但均没得到回应。

天下英雄总是惺惺相惜，Ryan Dahl 和 Isaac Z.Schlueter 二人一拍即合，决定抱团取暖，后来 Node.js 内置了 NPM。随着 Node.js 成为热门，大家都开始使用 NPM 来共享 JavaScript 代码了，于是 jQuery 作者也将 jQuery 放到了 NPM 中，采用 NPM 来分享代码已成为前端标配。有趣的是，Node.js 得到 Joyent 公司赞助后，Ryan Dahl 与 Issac Z.Schlueter 成为同事，两年后，Ryan Dahl 离职，Issac Z.Schlueter 成为第二任 Gatekeeper，Ryan Dahl 目前在谷歌从事 AI 研究方面的工作。

注意：代码仓库的概念相信大家并不陌生，其实 NPM 的实现思路与 Maven、Gradle、GitHub、DockerHub 等还是相通的。

NPM 的官网地址为 https://www.npmjs.com/，开发者可以在其官网上下载和上传模块。NPM 是随 Node.js 一起安装的，因此可以通过 npm -v 命令查看其版本，如图 2.10 所示。

NPM 实际由三部分组成，即网站（官网）、注册表（Registry）和命令行工具（CLI）。通过官网可以对包进行查询和管理；注册表记录保存了所有的包信息；命令行工具通过命令行或终端运行，开发者通过命令行工具与 NPM 进行交互。

图 2.10　查看 NPM 版本

2.3.2　使用 NPM 管理模块

本节讲解 NPM 的常用命令，并通过实际案例演示这些命令的使用方法。

1．NPM的常用命令

要使用 NPM 管理模块，就得先熟悉 NPM 的相关命令，这些命令可以在 CLI 中运行，完成与 NPM 的交互，常见的命令如表 2.3 所示。

表 2.3　NPM的常用命令

命　　令	说　　明
npm help(npm -h)	查看NPM帮助信息
npm version（npm -v）	查看NPM版本信息
npm init	引导创建package.json文件
npm install（npm -i）	安装模块，可以通过npm install –help命令查看参数
npm ls	查看已安装的模块
npm config	管理NPM配置文件。子命令有set/get/delete/list/edit,可以通过帮助命令npm config --help进行查看
npm cache	管理模块缓存，子命令有add/clean/verify,可以通过帮助命令npm cache –help进行查看
npm adduser	添加用户
npm publish	发布模块
npm start	启动模块
npm test	测试模块
npm update（npm -up）	更新模块
npm root	线上NPM根目录
npm access	设置模块的访问级别，子命令可以通过帮助命令查看：npm access –help

注意：命令不需要死记，忘记时可以通过帮助命令进行查看。npm help 命令可以查看所有的命令，“npm 命令 help”可以查看具体命令的使用。

2．案例：使用NPM安装第三方包

【本节示例参考：\源代码\C2\thirdModule】

学习完 NPM 命令后，下面来看如何使用 2.2.3 节中提到的第三方模块 Moment。

（1）在 C2 目录下新建 thirdModule 目录，在终端中通过命令 cd thirdModule 切换到该目录，运行 npm init 命令，按提示输入相应信息（也可以一直按 Enter 键保持默认值），等待初始化完成，如图 2.11 所示。

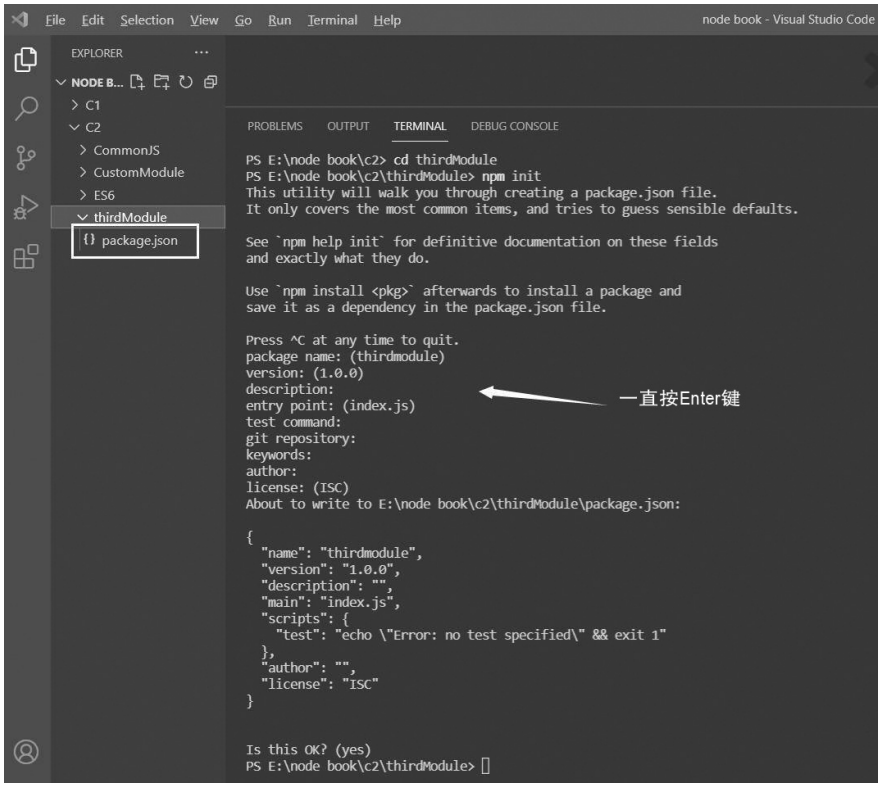

图 2.11　初始化目录

初始化完成后，可以看到目录下新增了一个 package.json 文件，此文件的内容稍后再进行分析。

（2）运行 npm install moment --save 命令，安装 Moment 模块，安装完成后如图 2.12 所示。

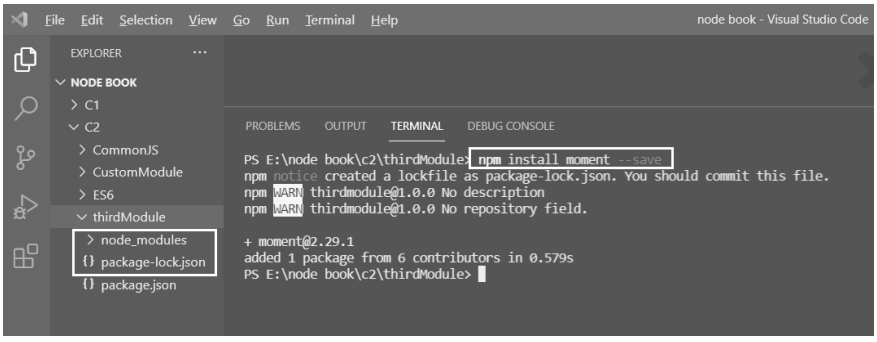

图 2.12　安装 Moment 插件模块

安装完成后，可以看到项目目录下新加了一个 package-lock.json 文件及 node_modules 目录。

（3）使用 Moment 模块新建 useMoment.js 文件，代码如下。

代码 2.16　自定义模块：useMoment.js

```
//使用第三方模块 Moment
var moment = require("moment");     //引入模块
var today = moment();               //通过 Moment 提供的方法获取当前时间
//通过 Moment 提供的格式化方法将日期格式化输出
console.log(today.format('YYYY-MM-DD HH:mm:ss'));
```

在代码中通过 require 函数导入刚才安装的 Moment 模块，采用 Moment 提供的方法获取当前日期，并通过 format 函数进行格式化输出，结果如图 2.13 所示。

图 2.13　Moment 插件的使用

这样就可以非常简便地使用第三方模块的功能了，避免重复造轮子，从而极大提高了工作效率。在示例中通过命令 npm init 生成了 package.json 文件；通过 npm install 命令生成了 package-lock.json 文件和 node_modules 目录，接下来进行分析。

3．package.json文件及包结构分析

package.json 文件是通过 npm init 命令创建（当然也可以手动创建）的，NPM 会在创建过程中进行引导，开发者根据提示填写相应信息，一步步按 Enter 键即可完成文件的创建。该文件中包含项目的元数据信息（名称、版本、作者、许可证等），也包含模块之间的依赖关系。上例中最终的 package.json 文件内容如下。

代码 2.17　package.json文件

```
{
  "name": "thirdmodule",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "moment": "^2.29.1"
  }
}
```

注意：文件中的 dependencies 节点在创建时是不存在的，只有安装了 Moment 模块后才会自动将该依赖记录到 package.json 文件中。

package.json 文件中各个字段的含义如表 2.4 所示。

表 2.4　package.json文件中的常见字段及其含义

字 段 名 称	含　义
Name	模块名称
Version	模块版本号
Description	模块描述
Scripts	可用于运行的脚本命令
Author	作者
License	许可证
Dependencies	正常运行时所需的模块

在安装 Moment 的过程中还生成了 package-lock.json 文件，细心的读者可能已经发现此文件的内容与 package.json 相似。既然如此，为何还要生成此文件呢？

实际上，package-lock.json 文件是用来做模块版本兼容处理的，它记录了当前状态下安装的所有模块信息，防止模块包不一致，确保在下载时间、开发者、下载源、机器都不同的情况下也能得到完全一样的模块包。

在安装 Moment 的过程中还会生成 node_modules 目录，并在其下生成 moment 目录。node_modules 目录为包目录，以后安装的所有包都会自动保存到此目录下。moment 目录则保存了 moment 包的内容，通过该目录可以大概了解 Node.js 的 NPM 包的基本组成部分。由于不同的包采用的模块规范可能不同，所以包结构可能会有一些差异。

完全符合 CommonJS 规范的模块应该包含以下几个文件：

- package.json：模块的描述文件。
- bin：存放可执行的二进制文件。
- lib：存放 JavaScript 代码。
- doc：存放文档。
- test：存放单元测试用例。

注意：有的包不一定完全遵从此规范，但至少应该包含 package.json 文件。

2.3.3　使用 YARN 管理模块

YARN 是 Facebook、谷歌等联合推出的新的 JavaScript 包管理工具，主要是为了弥补 NPM 的一些缺陷。与 NPM 相比，YARN 的速度更快、更安全、更可靠。YARN 的官网地址为 https://yarnpkg.com/。

1．YARN的安装及其常用命令

由于 YARN 没包含在 Node.js 中，因此需要单独进行安装。安装 YARN 可以通过 NPM 进

行安装，在终端中运行如下命令即可，其中-g 表示全局安装。

```
npm install yarn -g
```

安装完成后可以通过命令 yarn -v 查看版本号。

```
yarn -v
```

安装成功后如图 2.14 所示。

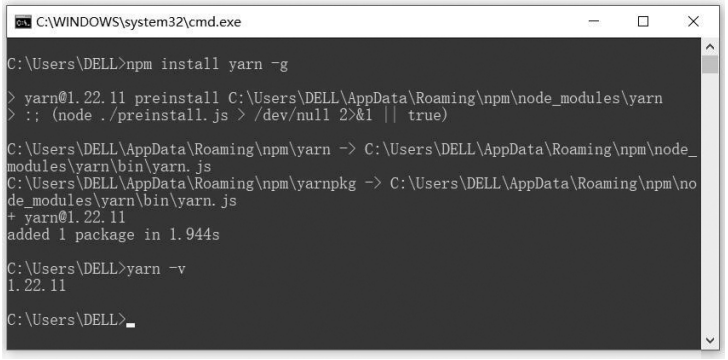

图 2.14　在 Windows 10 中安装 YARN

注意：本书讲的所有操作都是在 Windows 10 中完成的，如果读者的操作系统是 macOS 则需要参考官方文档。

YARN 的常用命令与 NPM 类似，如表 2.5 所示。

表 2.5　NPM与YARN的常用命令对比

NPM	YARN	说　明
npm init	Yarn init	初始化包的开发环境
npm install	Yarn install	安装package文件里定义的所有依赖
npm install xxx –save	Yarn add xxx	安装某个依赖，默认保存到package中
npm uninstall xxx –save	Yarn remove xxx	移除某个依赖项目
npm install xxx –save-dev	Yarn add xxx –dev	安装某个开发环境的依赖项目
npm update xxx –save	Yarn upgrade xxx	更新某个依赖项目
npm install xxx –global	Yarn global add xxx	安装某个全局依赖项目
npm run/test	Yarn run/test	运行命令

2．案例：使用YARN安装第三方包

【本节示例参考：\源代码\C2\Yarn】

了解 YARM 命令后，我们将 2.3.2 节中使用 NPM 安装 Moment，改为通过 YARN 来安装。

（1）在 C2 目录下新建 Yarn 目录，在终端中通过命令 cd Yarn 切换到该目录，运行 yarn init 命令，按提示输入相应信息（也可以一直按 Enter 键，保持默认值），等待初始化完成，如图 2.15 所示。

初始化完成后，同样可以看到目录下新增了一个 package.json 文件，文件内容与前面通过

NPM 生成内容稍有不同。

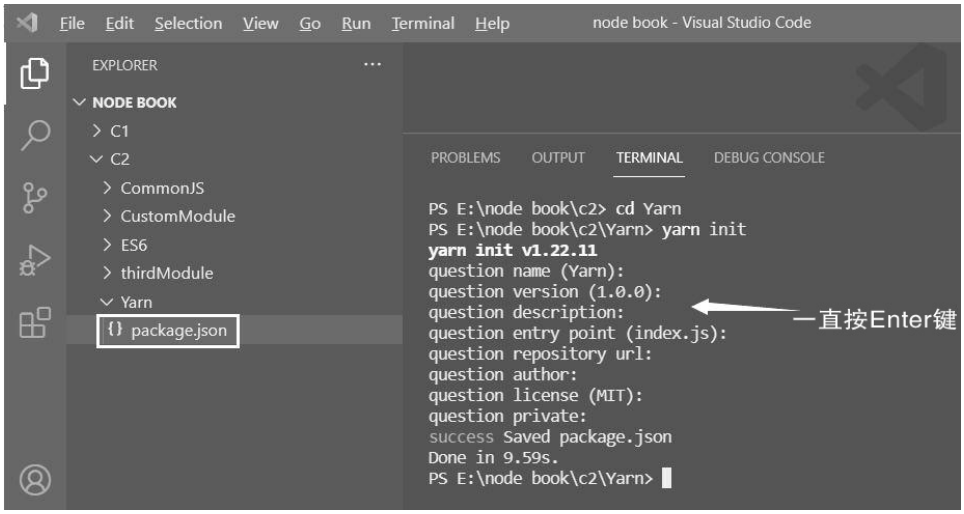

图 2.15　初始化目录

（2）运行 yarn add moment 命令，安装 Moment 模块，安装完成后如图 2.16 所示。

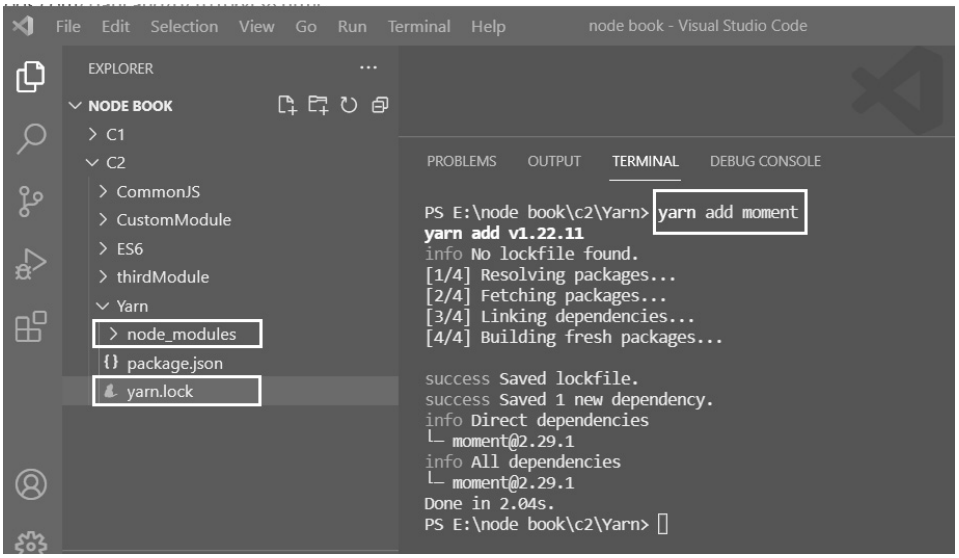

图 2.16　安装 Moment 插件模块

安装完成后，可以看到项目目录下新加了一个 yarn.lock 文件及 node_modules 目录。

（3）此步与 2.3.2 节中完全一致，直接将 useMoment.js 文件复制过来，代码如下。

代码 2.18　自定义模块：useMoment.js

```
//使用 Moment 第三方模块
var moment = require("moment");          //引入模块
var today = moment();                    //通过 Moment 提供的方法获取当前的时间
//通过 Moment 提供的格式化方法将日期格式化输出
console.log(today.format('YYYY-MM-DD HH:mm:ss'));
```

通过 node 命令运行文件，结果如图 2.17 所示。

图 2.17　Moment 插件使用

从上在的例子中可以看出，NPM 与 YARN 在使用上基本一致，仅是命令不同而已。在具体开发过程中，读者可以根据实际情况任选其一即可。

2.4　本 章 小 结

本章先介绍了 JavaScript 模块化的发展历史，详细介绍了 CommonJS 模块规范和 ES 6 的模块规范；接着介绍了 Node.js 的 3 种模块类型，即内置模块、自定义模块和第三方模块，为后续章节介绍 Node.js 核心模块打下基础；最后分别介绍了 NPM 和 YARN 包管理器及它们的常用命令，并且通过安装和使用 Moment 插件演示了它们的用法。

通过本章的学习，读者可以了解 Node.js 模块化编程思想，了解 Node.js 常见的内置模块以及如何使用 NPM 或 YARN 来管理三方模块。

由于 Node.js 采用 JavaScript 进行编程，在具体介绍 Node.js 核心模块之前，第 3 章将会详细介绍 JavaScript 的基础语法以及 ES 6 的规范，如果读者已经掌握了此部分内容，可以直接进入第 4 章的学习。