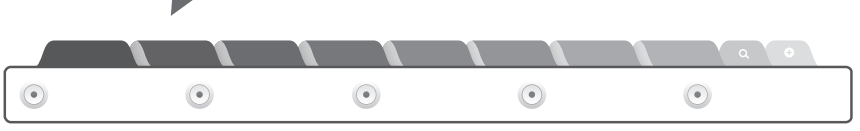
Web开发典藏大系

Vue.js 3+ TypeScript 从入门到项目实践

李一鸣◎编著



清華大學出版社
北京

内 容 简 介

本书以实战为主线，结合众多代码示例和一个完整的项目案例，全面、系统地介绍 Vue 3 和 TypeScript 的相关技术及其在实际项目开发中的应用。本书在讲解中穿插介绍了一些开发技巧，可以帮助读者提高代码质量和项目开发的效率。

本书共 13 章，分为 3 篇。第 1 篇基础知识，包括初识 Vue、TypeScript 基础知识、Vue 的基本指令、CSS 样式绑定、数据响应式基础、组件化开发。第 2 篇进阶提升，包括 HTTP 网络请求、使用 Vue Router 构建单页应用、Vuex 状态管理与应用调优、项目构建利器 Webpack、搭建后台模拟环境。第 3 篇项目实战，包括商城后台管理系统项目设计与框架搭建以及功能模块的实现。

本书内容丰富，实用性强，适合有一定 Web 开发和 JavaScript 编程基础的前端工程师阅读，也适合熟悉 Vue 2 而想进一步系统学习 Vue 3 的 Web 前端开发从业人员阅读，还适合大中专院校和社会培训机构作为 Web 开发类课程的教材。

图书在版编目（CIP）数据

Vue.js 3+TypeScript 从入门到项目实践 / 李一鸣编著. -- 北京 : 清华大学出版社, 2024. 9. -- (Web 开发典藏大系). -- ISBN 978-7-302-67259-3

Ⅰ. TP393.092.2；TP312.8

中国国家版本馆 CIP 数据核字第 2024YA2826 号

责任编辑：王中英
封面设计：欧振旭
责任校对：胡伟民
责任印制：刘海龙

出版发行：清华大学出版社
网　　址：https://www.tup.com.cn，https://www.wqxuetang.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-83470000　　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：河北鹏润印刷有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张：**18　　**字　　数：**453 千字
版　　次：2024 年 9 月第 1 版　　**印　　次：**2024 年 9 月第 1 次印刷
定　　价：79.80 元

产品编号：107413-01

前言

在当今这个 Web 技术发展令人眼花缭乱的时代，Vue.js（后文简称为 Vue）以其不断的创新而引领潮流。从 2013 年 12 月 Vue 的诞生，到 2016 年 10 月 Vue 2 的正式发布，再到 2020 年 9 月 Vue 3 的横空面世，这一开源框架历经多年的打磨与迭代，终于取得了空前的成功，国内有大量的公司都将其作为 Web 前端开发的首选框架。

Vue 3 是一套渐进式框架，它同 Vue 2 一样具有优雅的设计、出色的性能和友好的文档；它专注于视图层，采用自底向上的增量开发设计，可以构建优秀的用户界面，而且非常容易上手；它在兼容 Vue 2 的基础上进行了革新，引入了组合式 API（Composition API），这一革命性的特性使得代码编写更具聚焦性，不仅有助于提升代码的可重用性和可维护性，而且使得开发更为流畅，效率更高；它还以崭新的方式摒弃了 Vue 2 所依赖的 Object.defineProperty 方法，转而以 Proxy 实现响应式编程，从而能够更加灵活地追踪和触发数据变化。另外，伴随着 TypeScript 的逐渐流行，Vue 3 还进一步提升了对其支持与集成，这为项目开发提供了更加便捷和友好的环境。

上述特性使得 Vue 3 深受前端开发者的青睐，无数软件公司摈弃了传统的开发框架而转用这一革命性的新框架，无数开发者纷纷开始了解和学习 Vue 3。可以说，Vue 3 已经是 Web 前端开发人员必须掌握的一项技术。

目前，图书市场上已经可以找到多本 Vue 3 开发图书，但是还鲜见一本基于 Vue 3+ TypeScript 的 Web 项目开发图书。基于此，笔者编写了本书，全面介绍 Vue 3 和 TypeScript 的相关技术及其在实际项目开发中的应用，从而帮助读者全面、系统地学习 Web 前端开发知识。本书以实战为主旨，首先从基础知识讲起，然后进阶提升，系统讲述一些核心技术，最后进行项目实战，详细介绍一个商城后台管理系统的实现，从而帮助读者融会贯通前面所学的知识，并提高实际项目开发水平。

本书特色

1．内容新颖，技术前瞻

本书重点介绍 Vue 3 的全新特性，帮助读者快捷、顺畅地从 Vue 2 过渡到 Vue 3，从而为自己的项目提供强大的技术支撑。

2．内容全面，涵盖多项关键技术

本书不但全面介绍 Vue 3 前端开发的基础知识，而且介绍组件开发、路由管理、状态维护、数据响应式、性能分析与优化等多项 Vue 3 的核心技术。通过阅读本书，读者可以系统地了解 Vue 3 的各项技术及其应用场景与技术要点。

3．Vue 3 结合 TypeScript 进行开发

本书深入介绍 Vue 3 与 TypeScript 技术的结合使用，为读者展示编写类型安全代码的技巧，以及单元测试与性能优化等关键技术，其在实际项目开发中具有很大的价值。

4．示例丰富，注重实践

本书多数章节给出了丰富的代码示例，其难度由易到难，讲解由浅入深，循序渐进，代码注释丰富，非常适合读者上手练习，从而更好地理解相关的知识点。

5．详解经典项目实战案例

本书通过一个紧贴实际业务场景的经典项目实战案例——商城后台管理系统，引导读者理解实际项目开发，并在实际业务场景中应用所学的知识，以及提高代码质量和开发效率。本书不但详解该项目案例的实现思路，而且给出了完整的源代码并对其进行详细的注释，帮助读者深入理解项目开发的细节。

本书内容

第 1 篇　基础知识

本篇涵盖第 1～6 章，包括初识 Vue、TypeScript 基础知识、Vue 的基本指令、CSS 样式绑定、数据响应式基础、组件化开发。通过学习本篇内容，读者可以掌握 Vue 框架的背景、历史、目录结构和 Vue CLI 的使用方法，以及 TypeScript 的类型系统、接口和泛型等，并系统掌握构建出色的 Web 应用所需要具备的各种知识和技能。

第 2 篇　进阶提升

本篇涵盖第 7～11 章，包括 HTTP 网络请求、使用 Vue Router 构建单页应用、Vuex 状态管理与应用调优、项目构建利器——Webpack、搭建后台模拟环境。通过学习本篇内容，读者可以进一步拓展自己的技术视野，从而系统掌握构建更复杂、高效和实用的前端应用所需要的核心技术。

第 3 篇　项目实战

本篇涵盖第 12、13 章，包括商城后台管理系统——项目设计与框架搭建、商城后台管理系统——功能模块的实现。通过学习本篇内容，读者可以更加深入地理解前面章节所学技术在实际项目中的应用，并详细了解一个完整的 Web 项目的开发流程。

配套资源获取方式

本书涉及的源代码等配套资源有两种获取方式：一是关注微信公众号“方大卓越”，回复数字“30”自动获取下载链接；二是在清华大学出版社网站（www.tup.com.cn）上搜索到本书，然后在本书页面上找到“资源下载”栏目，单击“网络资源”或“课件下载”

按钮进行下载。

读者对象

- Vue.js 3 初学者；
- Vue.js 3 进阶者；
- 从 Vue.js 2 转向 Vue.js 3 的开发者；
- Web 前端开发工程师；
- Web 服务器端开发工程师；
- 对前端开发感兴趣的后端开发人员；
- 软件开发项目经理；
- 高等院校的学生；
- 相关培训机构的学员。

致谢

由衷地感谢参与本书出版的所有工作人员，是你们让我与本书结缘，才得以分享自己的知识与经验，并让本书高质量出版。还要感谢那些在本书写作过程中给予帮助的人，你们的支持和交流使得本书内容更加完善和有价值。另外，还要特别感谢我的家人，是你们的坚定支持和鼓励才让我能够坚持下去，最终完成本书的创作。最后，衷心地感谢本书的所有读者，正是因为有了你们，本书才能够体现其最大的价值和意义。

售后支持

由于水平所限，加之写作时间较为仓促，书中可能还存在一些疏漏和不足之处，敬请广大读者批评与指正。阅读本书的过程中如果有疑问，请发电子邮件联系笔者。邮箱地址：bookservice2008@163.com。

李一鸣
2024 年 7 月

目录

第 1 篇　基础知识

第 3 篇　项目实战

第 1 篇
基础知识

第 1 章　初识 Vue.js

本章介绍 Vue.js（下文简称为 Vue，涉及版本号时简称为 Vue 2 或 Vue 3）框架的基本发展历程，并提供从零开始搭建开发环境到运行自己第一个程序的详细指导，以帮助读者初步了解 Vue 框架的核心特性和使用方法。

在对 Vue 有一定了解后，可深入探究前端领域中其他框架的现状和发展趋势。通过比较各框架的特点和适用环境，协助读者在不同项目中做出恰当的选择。

本章涉及的主要内容点如下：

- Vue 简介；
- 从零开始搭建 Vue 开发环境；
- 第一个 Vue 程序；
- 丰富的界面体验：探索 Vue UI 库。

1.1　Vue 简介

2021 年，GitHub 发布了一份年度报告，其中，JavaScript 的热度常年位居第一，TypeScript 的热度也在近几年一路飙升。虽然近两年就业市场相对严峻，但前端的生存环境还算差强人意，其招聘数量依然高于其他开发领域。随着前端技术的不断演进，涌现出了众多新技术和框架，其中，Vue 成为学习前端不可忽视的一个框架。本书出版的目的就是帮助更多想学习 Vue 的前端开发者，通过更多的练习掌握这个框架。

目前还有一些公司使用 Vue 2 进行开发，大多是因为有一些老项目需要进行维护。对于框架本身而言，Vue 2 和 Vue 3 中的很多概念都是相通的，而且 Vue 3 在很大程度上兼容 Vue 2 的语法。Vue 3 已经面世几年了，因此不管是初学者还是工作多年的前端开发人员，推荐直接学习 Vue 3。总而言之，千万不要在学习方向上过度纠结，这会浪费自己的精力与时间。新技术不断涌现，但本质并没有太大的变化，对于开发人员来说，更重要的是思考问题的角度和快速解决问题的能力。不要将自己局限在单一编程语言或领域，也不要把自己定位在一个方向或岗位上。一旦精通一门开发语言，学习另一门开发语言将变得更加容易。

接下来，一同来了解 Vue 的历史和现状吧！

1.1.1　Vue 的诞生与发展

2013 年，一位叫尤雨溪开发者受到 AngularJS 的启发，提取了自己喜欢的部分开发了一款轻量级框架 Seed。由于 NPM 中已有同名框架，同年 12 月，该框架正式更名为 Vue。

2014 年 2 月，Vue 正式对外发布，版本号为 0.6.0。2016 年 10 月正式发布了 Vue 2 版本，2020 年 9 月正式发布了 Vue 3 版本。

Vue 在国内的流行，不仅在于尤雨溪个人，也在于框架自身的优秀和可靠。2016 年一项针对 JavaScript 框架的调查表明，Vue 的开发者满意度为 89%。在 GitHub 上，该项目平均每天能收获 95 颗星，为 GitHub 有史以来星标数第 3 多的项目。2018 年，在 JavaScript 框架/函式库中，Vue 所获得的星标数已超过 React，并高于 Angular 和 jQuery 等项目。

通过图 1.1 和图 1.2 可以看出，Vue 在国内是最热门的框架，并且在全球范围内也拥有相当大的市场。

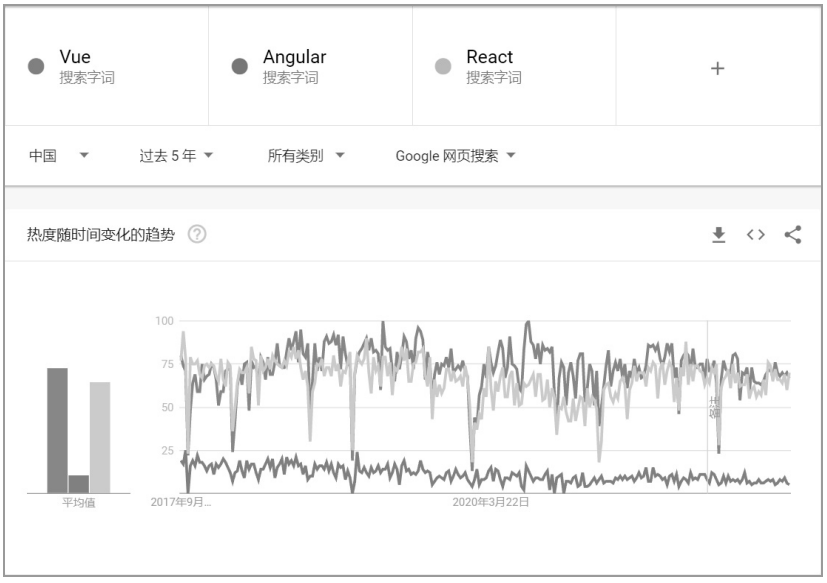


图 1.1　三大框架在国内的热度趋势

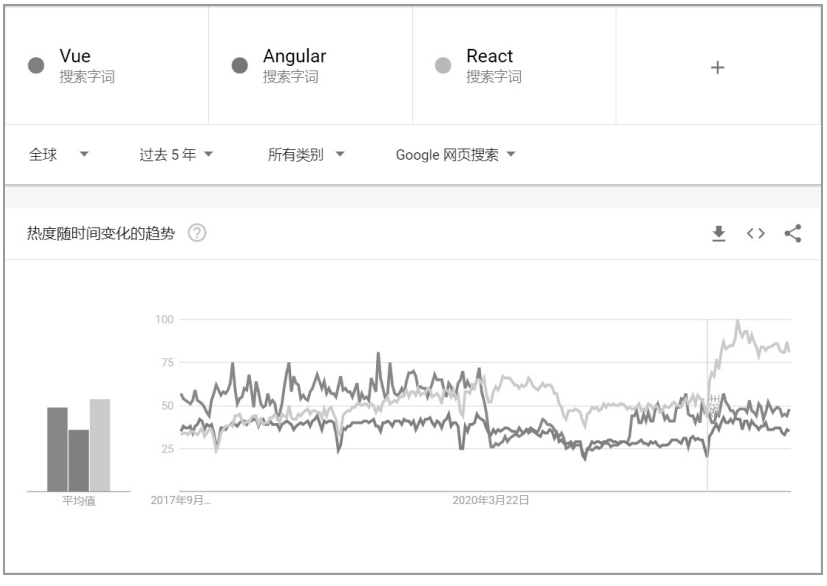


图 1.2　三大框架全球热度趋势

1.1.2　Vue 版本的区别

从 2013 年 Vue 的诞生到本书完成写作时的 Vue 3.4 版本，Vue 的发展十分迅速。其中，每个大版本都有明显的改动。以下是 Vue 各版本的一些主要变化。

1．Vue 1.x

Vue 1.x 是第一个发布的 Vue 版本。它的核心思想是“双向数据绑定”，这意味着当数据发生变化时，视图会自动更新。Vue 1.x 的 API 相对简单，并且性能较好，但也存在一些缺点，如性能受限于实现方式、缺乏虚拟 DOM 等。

2．Vue 2.x

Vue 2.x 是 Vue 的第二个版本，它在 Vue 1.x 的基础上进行了重构和优化。Vue 2.x 的核心优化是引入了虚拟 DOM，这样就能更高效地进行视图更新。此外，Vue 2.x 还加入了许多新特性和语法糖，如计算属性、指令修饰符和渲染函数等，使其更加灵活和易用。

3．Vue 3.x

Vue 3.x 是 Vue 的最新版本，它在 Vue 2.x 的基础上进行了进一步的改进和优化。Vue 3.x 主要优化了性能和开发体验。首先，Vue 3.x 使用了新的响应式系统，它的性能比 Vue 2.x 更好，并且更易于调试。其次，Vue 3.x 采用了模块化的架构，可以更好地支持 Tree Shaking 功能和按需加载。此外，Vue 3.x 还提供了 Composition API，这是一种新的 API 风格，可以更好地组织和复用代码。

Vue 在版本升级过程中一直在改进其性能、易用性和可维护性。Vue 1.x 相对简单，Vue 2.x 加入了许多新特性和语法糖，Vue 3.x 则在新的响应式系统、模块化架构和 Composition API 上进行了改进和优化。

1.1.3　前端框架的选择

以前，jQuery 一统天下，现在，Vue、React、Angular 三分天下。鉴于这三大框架的盛行，有必要了解它们之间的区别，见表 1.1。

表 1.1　三大框架对比

区　　别	Vue	React	Angular
组织方式	模块化	模块化	MVC
路由	动态路由	动态路由	静态路由
模板能力	自由	自由	强大
数据绑定	双向绑定	单向绑定	双向绑定
自由度	较大	大	较小

从表 1.1 中可以看出，Vue 具有快速上手、灵活简单的特点，适用于中小型应用，有

许多公司采用。React 具备极高的灵活性，但学习曲线较陡，广泛被多家公司使用。Angular 拥有强大的功能，提供丰富的特性，然而使用的公司相对较少且自由度有所限制。

对于目前前端的发展情况来说，数量众多的框架让开发者心生畏惧，觉得每天都在出新的东西，永远学不完，心里十分焦虑。程序员应该是框架的主人，为了提高效率、解决某些问题才会选择某个框架，千万不能成为框架的奴隶。新框架那么多，出一个学一个，什么都会一点，将什么都不精。只要拥有扎实的基础知识，对所学框架进行深入的理解，就像学编程语言一样，精通了一门后再学其他的是很快的。

如果读者打算学会一个框架然后去找一份工作，那么对于国内的前端开发者来说选择 Vue 是毫无疑问的。国外对于 React 的开发岗位需求更多一些，因此欧美国家的开发者可以选择 React 进行学习。

1.2　从零开始搭建 Vue 开发环境

本节笔者将从搭建环境开始讲起，并在下一节介绍如何运行第一个 Vue 程序。

☎**提示：**本节的环境搭建将以用户比例最多的 Windows 操作系统进行演示。由于软件在 Windows、macOS、Linux 环境下的安装方法基本一致，所以不再一一讲述。

1.2.1　安装 Node.js 和 NPM

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行时环境，而 NPM（Node Package Manager，Node 包管理器）是 Node.js 默认的以 JavaScript 编写的软件包管理系统。

Vue 的官方文档中明确指出，使用 Vue 的前提条件是安装 15.0 或更高版本的 Node.js。安装方法十分简单，打开 Node.js 官网的网址 https://nodejs.org/zh-cn/download/，选择对应自己操作系统版本的 Node.js 安装包，单击下载即可，如图 1.3 所示。

图 1.3　Node.js 与 NPM 下载

安装 Node.js 时 NPM 会随之自动安装到系统中。在命令行中输入 npm -v 和 node -v 命

令可以检测是否成功安装，如图 1.4 所示。

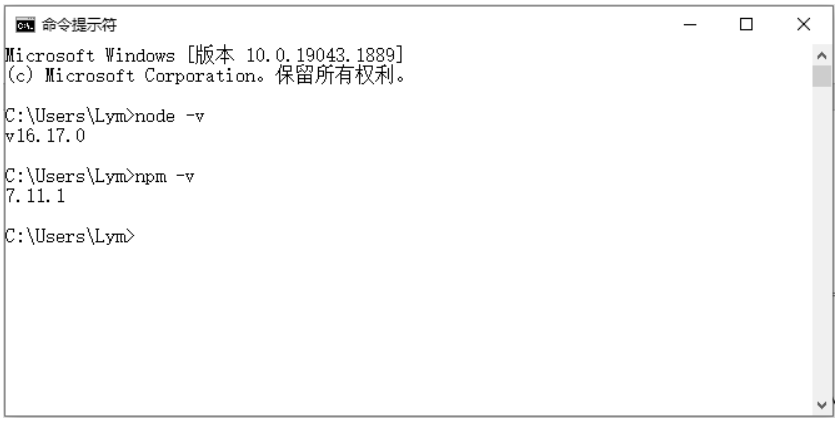


图 1.4　查看当前操作系统安装的 Node.js 和 NPM 版本

如果读者在使用 NPM 安装第三方库时失败了，可以选择使用 CNPM。在安装或升级完 Node.js 后，运行以下命令可以安装淘宝提供的 NPM 软件包库的镜像 CNPM，命令运行结果如图 1.5 所示。

```
npm install -g cnpm --registry=https://registry.npm.taobao.org
```

图 1.5　使用 NPM 安装 CNPM

安装命令成功执行后，以后使用 NPM 命令的时候就都可以用 CNPM 来代替了。下面的内容中依然使用 NPM 命令，安装了 CNPM 的读者可以自行替换。CNPM 支持 NPM 中除 publish 之外的所有命令。

☎**提示：**如果使用 npm install 等命令安装失败，Windows 用户可以尝试使用管理员身份运行命令行，macOS 和 Linux 用户可以尝试在命令前加上 sudo。

至此，Node.js 和 NPM 的安装就完成了。最后，读者可以检查一下本地的版本号，尽量保证所使用的版本大于或等于笔者的版本。

1.2.2　安装 Git

Git 是一个开源的分布式版本控制系统，由 Linux 之父 Linus Torvalds 开发。目前，几乎所有的开源项目都发布在使用 Git 的 GitHub 网站上，包括 Vue 这个开源项目也上传到了该平台上，其 GitHub 的网址为 https://github.com/vuejs/core。

使用 Vue CLI 之前，需要在操作系统中安装好 Git。这样当使用 Vue CLI 创建项目时，将会自动调用 Git 的命令，从 GitHub 把对应版本号的 Vue 模板与支持文件下载到本地。

对于 Git 的安装，首选当然是从官方网站上获取安装文件，打开 https://www.git-scm.com/download/，选择对应的操作系统，单击下载即可，如图 1.6 所示。

图 1.6　Git 的官网下载网站

安装完毕后输入以下指令，验证 Git 是否成功安装并检查安装的版本，如图 1.7 所示。

```
git --version
```

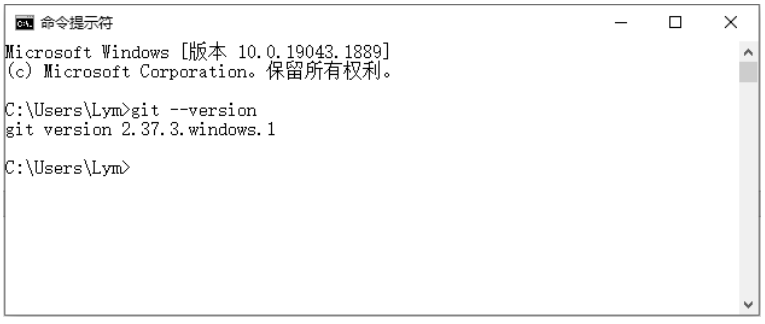


图 1.7　验证 Git 是否成功安装以及安装的版本

1.2.3　安装 Vue CLI

接下来需要使用刚才安装的 NPM 进行 Vue CLI 的安装。Vue CLI 是 Vue 的命令行界面工具，具有创建项目、添加文件及启动服务等功能。使用以下命令安装 Vue CLI。

```
npm install -g @vue/cli
```

在安装完成后，使用以下命令验证 Vue CLI 的安装与版本与命令帮助，如图 1.8 所示。

```
ng --version
```

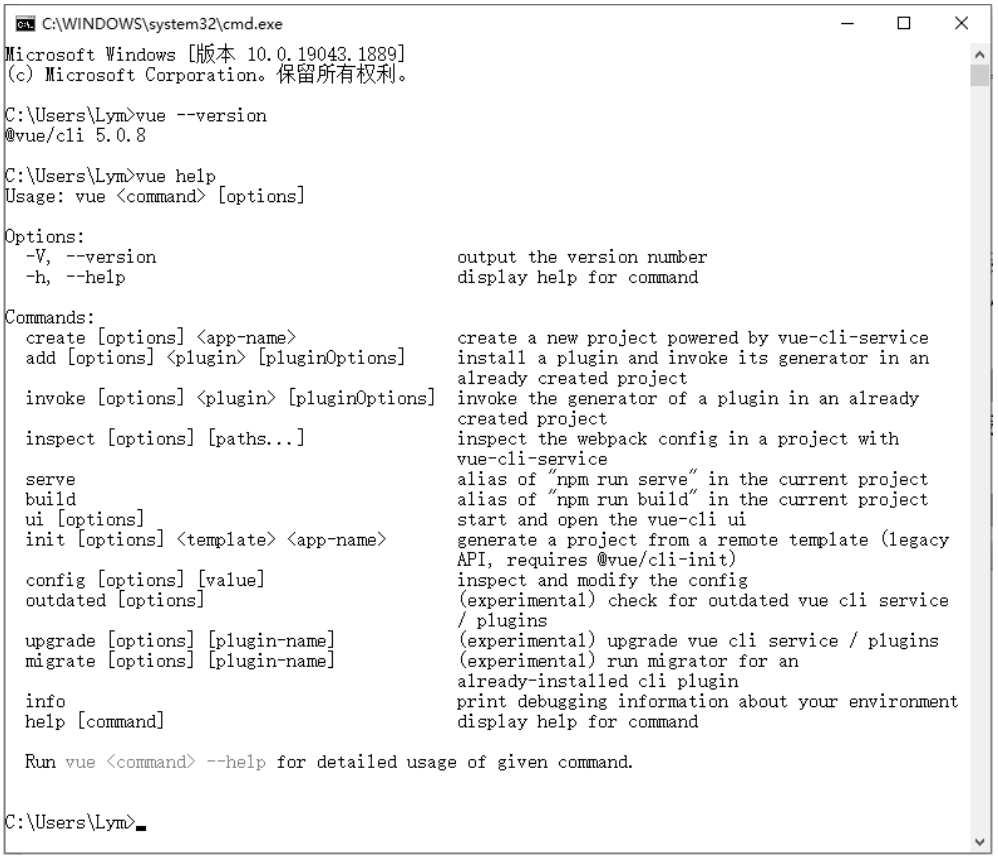


图 1.8　验证 Vue CLI 安装的版本与命令帮助

1.2.4　安装 Visual Studio Code

Visual Studio Code 是由微软开发的一个轻量且强大的代码编辑器，不仅免费开源，而且还提供相当丰富的插件。Vue 所使用的编程语言 TypeScript 也是由微软开发的，因此 Visual Studio Code 编辑器对其的支持性也比较强，这里推荐读者使用该编辑器进行开发。如果读者已经习惯了 WebStorm、Sublime 和 Nodepad 之类的软件，则可以跳过这一节直接进入下一节的内容。

关于 Visual Studio Code 编辑器，可以直接到其官方网站 https://code.visualstudio.com/ 下载安装文件进行安装，如图 1.9 所示。

Visual Studio Code 安装完毕后，可以启动它，使用它打开任意目录，也可以直接将文件夹拖入或者通过选择“文件”|“打开文件夹”的方式打开项目，如图 1.10 所示。在欢迎使用窗口中，可以清晰地看到它的方便之处，可以直接打开文件夹，也可以自定义对各种编程语言的支持，还可以根据用户的习惯设置快捷键，颜色主题也可以进行自定义。

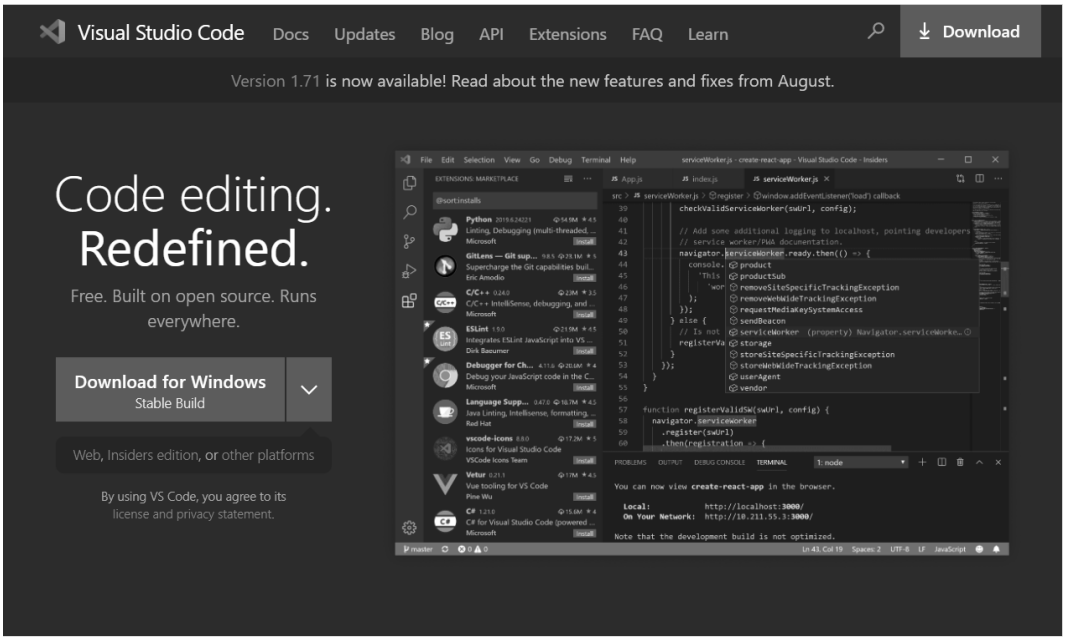


图 1.9　Visual Studio Code 官方网站下载页

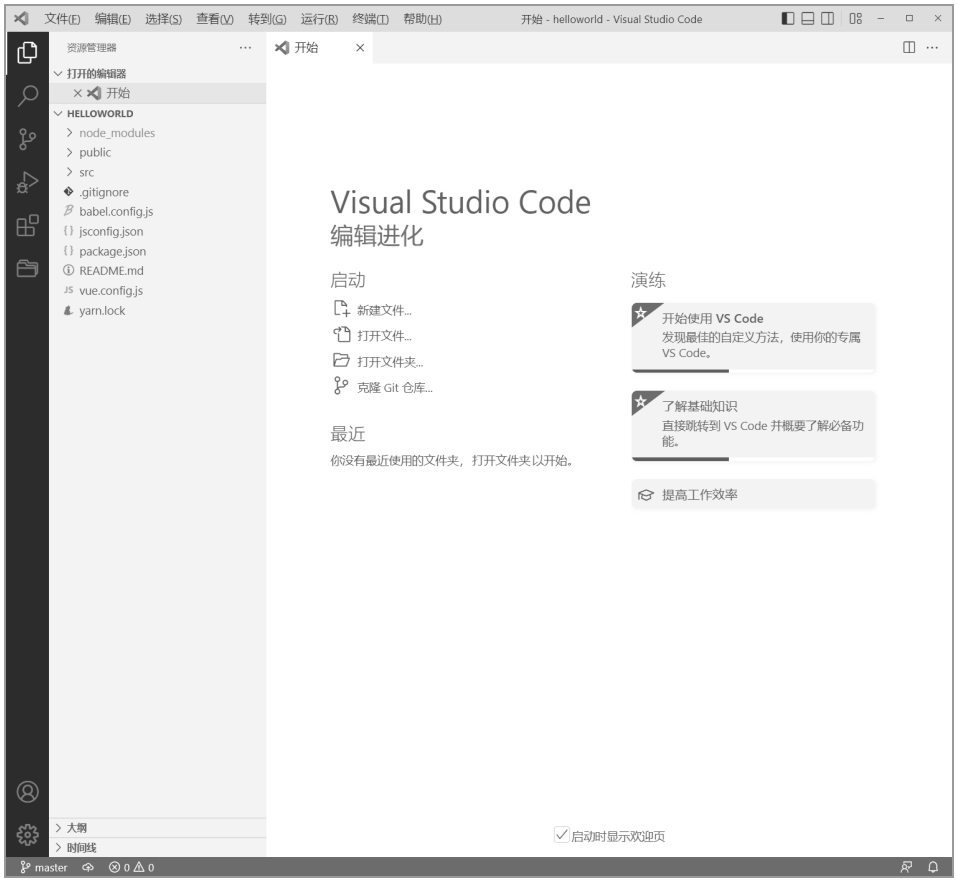


图 1.10　Visual Studio Code 的窗口布局

在键盘上分别按 Ctrl+K 和 Ctrl+T 组合键（苹果计算机上是 Command+K 和 Command+T 组合键）可以进行主题颜色设置，也可以选择安装扩展的主题颜色和扩展的图标等，如图 1.11 所示。这里为了图书印刷效果，选择了方便读者阅读的浅色主题。

图 1.11　Visual Studio Code 的颜色主题

1.2.5　安装 Vue 辅助编码插件

Visual Studio Code 的扩展功能非常强大，可以直接进行搜索和安装。 为了方便 Vue 的开发，应该选择当前最新并且评分比较高的扩展插件进行安装。安装完成后，单击“重新加载”按钮，然后刷新页面即可使用。这些扩展插件主要包括相关的代码提示等功能，可以给开发者带来一定的便利，但并不是必需的。如图 1.12 是笔者推荐的 Vue 3 的常用插件，读者根据名称搜索并安装即可。

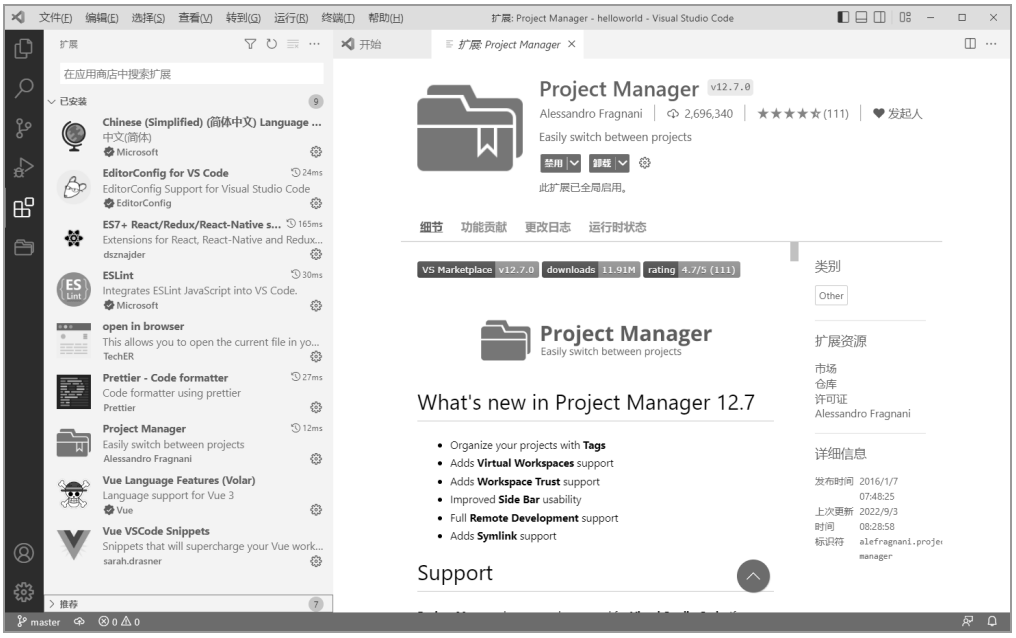


图 1.12　安装 Visual Studio Code 的 Vue 扩展

以上大部分插件安装完毕后，不需要进行操作，自动就具有代码提示等功能。也有一部分插件是手动操作，如 open in browser 插件需要使用 Alt+B 组合键启动浏览器打开 HTML 文件。

1.3　第一个 Vue 程序

开发环境搭建好之后，本节将创建第一个 Vue 程序。在接下来的内容中，我们将一步步探索 Vue 3 的强大功能和灵活性，从零开始构建一个完整的知识体系。无论初学者还是有经验的开发者，相信通过本节的学习都会有新的收获。

1.3.1　Hello Vue 实例解析

创建 Vue 项目十分方便，打开终端，使用 cd 命令进入想要创建项目的目录下，执行以下命令即可创建项目。

```
vue create hello-vue
```

笔者本地的 Vue CLI 版本是 5.0.8，输入该指令后，会提示生成 Vue 3 还是 Vue 2 项目，这里选择生成 Vue 3 项目。

接下来输入以下代码即可启动 Vue 开发服务器。

```
cd hello-vue
npm run serve
```

至此，第一个 Vue 程序运行成功，效果如图 1.13 所示。

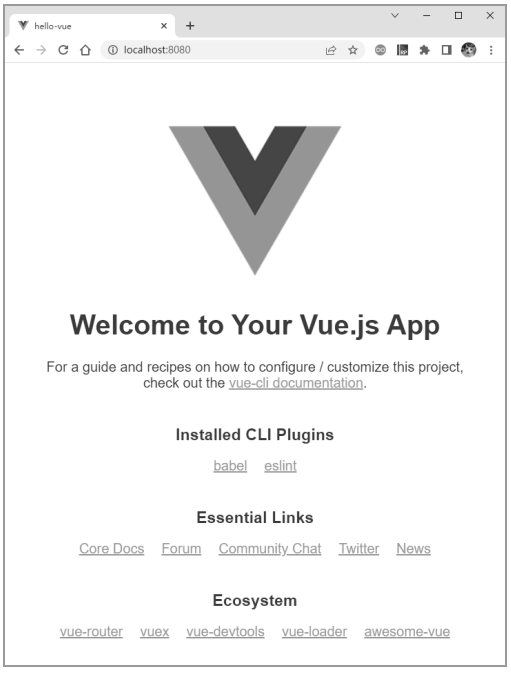


图 1.13　示例项目运行效果

1.3.2　Vue 的目录结构

Vue 安装完成后的项目目录如图 1.14 所示。

从图 1.14 中可以看出，在 hello-vue 文件夹目录下生成了很多文件与子目录，接下来按照文件的排列顺序进行讲解。

- .git/：创建 Git 仓库需要的目录。
- node_modules/：执行 npm install 后生成的文件夹，package.json 中的第三方模块将会全部安装在其中。
- pubilc/：公共资源目录，该文件夹的资源不会被 Webpack 处理，需要用绝对路径来引用。
- src/：主要代码的存放目录，后面的工作大多在这个目录下完成。
- .gitignore：Git 配置文件，会在提交时忽略不需要的文件。
- babel.config.js：babel 的配置文件，作用于整个项目。
- jsconfig.json：JavaScript 配置文件，可以用来配置默认的根路径等。
- package.json：NPM 配置文件，列出了项目使用的第三方模块。
- README.md：项目说明文档，使用 markdown 进行编写。
- vue.config.js：项目配置文件，Webpack 等大多数配置都在这里进行，参考地址为 https://cli.vuejs.org/zh/config/。
- yarn.lock：YARN 的一个配置文件，文件由 YARN 自动生成和编辑，不需要进行操作。

src 文件夹下存放的是主要的代码，接下来分析其中的文件。src 文件夹下的目录如图 1.15 所示。

图 1.14　项目文件夹目录

图 1.15　src 文件夹目录

- assets/：该目录下一般存放图片等资源文件。
- components：用于存放组件。
- App.vue：根组件，之后创建的页面都在这个节点之下。
- main.js：程序入口，createApp 方法在本文件里执行。

从图 1.15 中可以看出，文件并不多，因为这个指令生成的是最基本的项目，并没有 router 和 TypeScript 等配置。

1.3.3　使用 Vue CLI 创建项目

Vue CLI 的全名是 Vue 命令行界面，它是一个帮助开发者快速构建 Vue 集成相关工具链的工具。它可以确保各种构建工具能够基于智能的默认配置平稳衔接，这样开发者可以专注在撰写应用上，而不必纠结配置的问题。

首先来熟悉一下命令行中的各种指令。输入以下命令可以查看 Vue CLI 帮助信息，如图 1.16 所示。

```
vue help
```

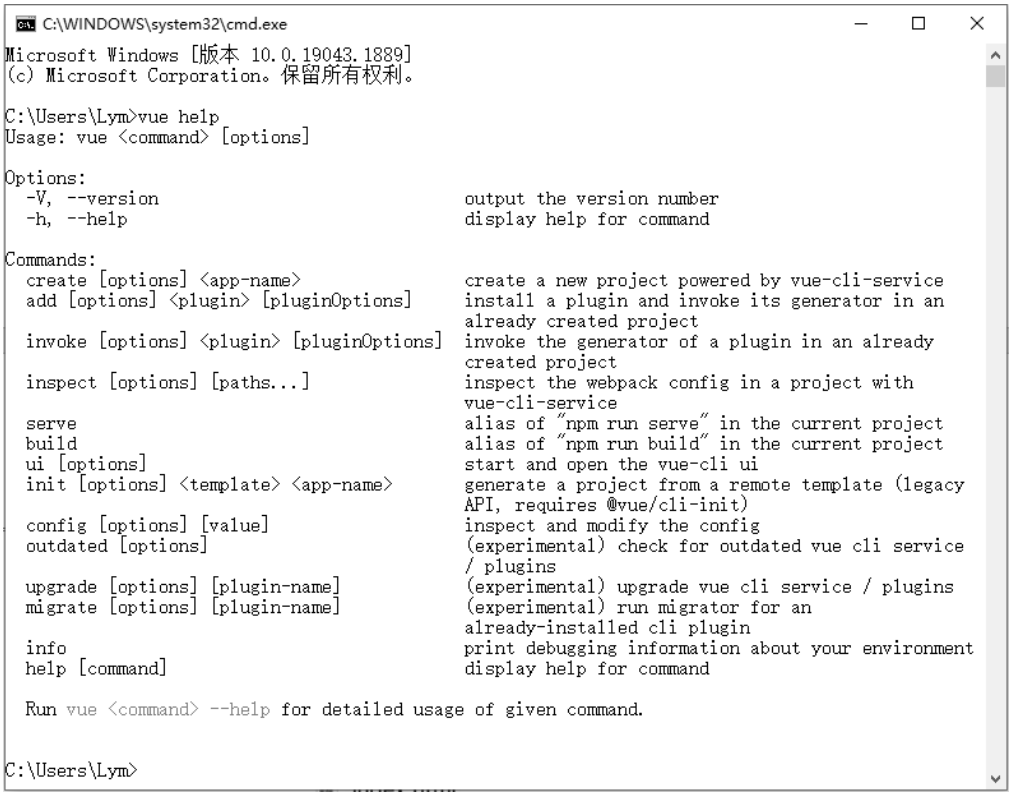


图 1.16　使用 vue help 命令查看 Vue CLI 帮助信息

- vue create：新建项目，前面创建的 hello-vue 就是用这个命令创造的。
- vue add：安装插件，相当于 npm install 和 vue invoke 命令都执行了。
- vue invoke：vue add 的子功能，用户更改插件的配置。
- vue inspect：通过 vue-cli-service 导出 Webpack 的配置到项目目录中。
- vue serve：相当于 npm run serve 命令。
- vue build：相当于 npm run build 命令。
- vue ui：可视化的图形界面，用于创建、更新和管理 Vue 项目。

- vue init：初始化项目，与 vue create 类似，需要详细选择不同的配置项。
- vue config：检查和修改配置。
- vue outdated：检查服务或者插件是否过期。
- vue upgrade：升级 cli-service 和 plugins。
- vue migrate：迁移已安装的插件。
- vue info：打印当前环境的调试信息。
- vue help：输出帮助信息。

vue.config.js 是一个可选的配置文件，如果项目的（和 package.json 同级的）根目录中存在这个文件，那么它会被@vue/cli-service 自动加载。当然，也可以使用 package.json 中的 vue 字段，但是注意这种写法需要严格遵照 JSON 的格式。接下来介绍 vue.config.js 的常用配置项，如表 1.2 所示。

表 1.2　vue.config.js的常用配置项

配　置　项	默　认　值	说　　明
pubilcPath	'/'	部署应用包时的基本URL。从Vue CLI 3.3起baseURL已弃用，改用pubilcPath
outputDir	'dist'	运行npm run build/vue-cli-service build时生成的生产环境构建文件的目录
assetsDir	''	放置生成的静态资源（js、css、img、fonts）的目录
indexPath	'index.html'	指定生成的index.html的输出路径
filenameHashing	true	开关文件名哈希，可以更好地控制缓存
pages	undefined	在multi-page模式下构建应用。平时做的都是单页面应用
lintOnSave	'default'	是否在开发环境下通过eslint-loader在每次保存时自动校对代码
runtimeCompiler	false	是否使用包含运行时编译器的Vue构建版本
transpileDependencies	false	是否对所有的依赖都进行转译
productionSourceMap	true	是否需要生产环境的source map，设置为false可以加速构建，但是无法准确定位错误信息
crossorigin	undefined	设置HTML中的crossorigin属性
integrity	false	是否在生成的HTML中启用SRI，如果构建的文件部署在CDN上，那么启用该选项可以提供额外的安全性
configureWebpack	无	值是一个对象，会合并到Webpack的最终配置中
chainWebpack	无	其是一个函数，可以对内部的Webpack配置进行更细粒度的修改
css.requireModuleExtension	true	是否将所有的*.(css|scss|sass|less|styl(us)?)文件视为CSSModules模块
css.extract	生产环境true 开发环境false	是否将组件中的CSS提取至一个独立的CSS文件中
css.sourceMap	false	是否为CSS开启source map。设置为true之后可能会影响项目打包的速度
css.loaderOptions	{}	向CSS相关的loader传递选项
devServer.proxy	无	将API请求代理到API服务器

续表

配　置　项	默　认　值	说　　明
pwa	无	向PWA插件传递选项
pluginOptions	无	传递任何第三方插件选项

vue.config.js 的配置是开发过程中很重要的一步，初学者一下看到这么多配置项可能会觉得乱，在后面的项目中会经常用到它们，多尝试、多使用自然就会记住了。Vue CLI 在未来可能会有改动，可以参考其最新的官方文档，网址为 https://cli.vuejs.org/zh/config/。

1.3.4　使用 Vite 创建项目

目前，Vue CLI 的官方网站处于维护模式，建议读者逐步切换到使用 Vite 来创建项目，如图 1.17 所示。

图 1.17　Vue CLI 官网

Vite 是一个快速的 Web 开发构建工具，由 Vue 核心开发团队维护，它的主要目标是提供一种快速的开发体验。Vite 提供了快速的开发服务器和即时热更新，支持 Vue、React 和 Svelte 等前端框架，它使用了原生 ES 模块来加载代码，可以显著提升构建速度。执行以下命令安装 Vite：

```
// 全局安装 Vite 的命令行
npm install vite -g

// 完成后输入以下代码查看版本号，验证是否安装成功
vite -v
```

安装完成后，使用 create-vue 创建 Vite 项目。直接使用 NPM 命令即可创造 Vite 项目，代码如下：

```
npm create vue@3
```

命令成功运行窗口如图 1.18 所示。

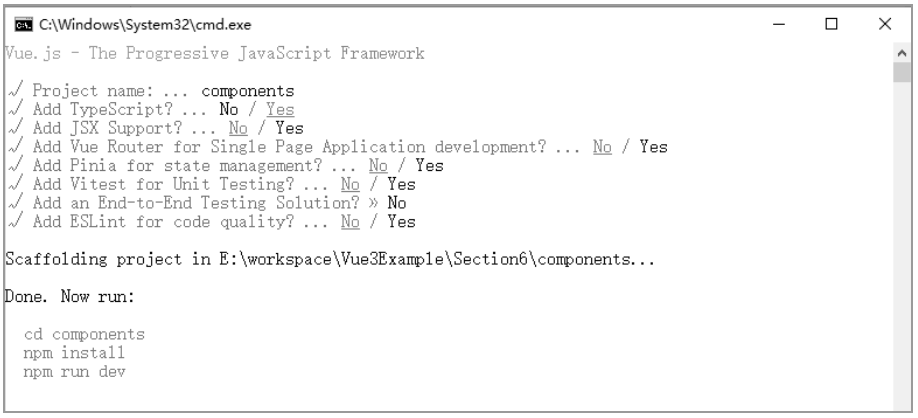


图 1.18　create-vue 命令运行窗口

根据提示，输入项目名、是否需要 TypeScript、JSX 等配置后，即可完成项目的创建，方式与 Vue CLI 十分相似。Vite 的优势就在于其可以极大地提高构建速度。在学习阶段使用 Vue CLI 完全没有问题，但是未来的项目开发可能会逐渐转为 Vite，因此读者有必要对 Vite 有一些了解。

1.3.5　使用 CDN 创建项目

新建一个文件 cdn.html，用于展示不使用构建工具，使用 CDN 来加载 Vue。用 VSCode 打开 cdn.html 并输入 html:5，会自动生成一段 HTML 5 的空白模板代码，如图 1.19 所示。

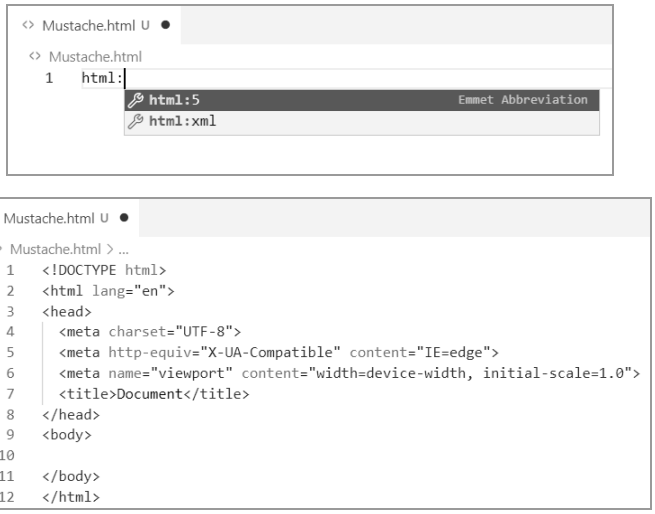


图 1.19　生成 HTML 5 空白模板代码

接下来在 head 标签中引入 CDN，并在 body 标签中输入以下代码。

```
...
<!-- 引入 Vue 3 的脚本 -->
<script src="https://unpkg.com/vue@3"></script>
...
<!-- 在页面主体部分 -->
```

```
<body>
  <!-- 创建一个 ID 为"app"的 div 元素，其将在 Vue 应用中使用 -->
  <div id="app">{{ message }}</div>
  <script>
    // 从 Vue 中解构出 createApp 函数
    const { createApp } = Vue
    // 创建一个 Vue 应用
    createApp({
      // data 函数，用于定义应用中的数据
      data() {
        return {
          message: 'Hello Vue!',
        }
      },
    }).mount('#app')
  </script>
</body>
...
```

接下来右击鼠标，在弹出的快捷菜单中选择 Open in Default Browser 命令运行代码，如图 1.20 所示（如果没有这个选项，请参考 1.2.5 节的内容）。可以看到，在浏览器中已经出现了 Hello Vue，如图 1.21 所示。这种方式比较简单易懂，因此在后面的章节中主要通过 CDN 来进行代码的编写。

图 1.20　Open in Default Browser 命令

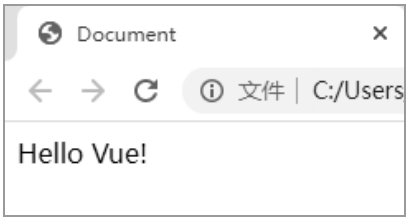


图 1.21　浏览器的显示效果

1.3.6　如何高效地学习 Vue

在学习 Vue 的过程中，首先，需要了解 Vue 的基本概念和语法，包括 Vue 的核心概念、组件化思想及指令等基础内容。这些内容可以通过阅读 Vue 3 的官方文档和相关教程来学习。

其次，完整掌握 Vue 的生命周期、组件和指令等进阶内容。这些内容的学习可以通过阅读 Vue 3 的官方文档和相关教程来完成。同时还可以通过阅读其他人的 Vue 3 项目代码来学习更多的编程技巧和实践经验。建议使用 Vue 3 搭建一些小型的项目，这样可以更加熟悉 Vue 3 的应用。在实践操作中肯定会遇到一些问题，这些问题可以通过查阅官方文档和其他人的代码来解决。

最后，深入阅读 Vue 的源码，从底层了解其实现原理，这样可以更加深入地理解 Vue，并且可以提高代码调试和排错能力。另外，建议了解一些周边的生态工具，如 Vuex 和 Vue Router 等，这些工具可以帮助开发者更好地应用 Vue。

总之，Vue 的学习属于上手容易，精通困难，进行简单的学习之后就可以开发出一些应用，但是想熟练掌握还是需要费一些功夫的。跟着本书或者网上的 Demo 多动手练习，写的代码多了自然就熟练了。此外，要多花费精力形成自己的知识结构体系，知道 Vue 的各个模块是如何配合工作的，这样学习起来才能事半功倍。

1.4　丰富的界面体验：探索 Vue UI 库

Vue 提供了一个方便的框架用于编写业务逻辑。同样，了解 Vue 中流行的 UI 框架在日常开发中也是非常有必要的。一个优秀的框架不仅提供了吸引用户的页面风格，还可以显著提高开发效率，使开发人员能够更专注于业务开发，而不必在 UI 和 CSS 方面花费过多精力。本节将介绍一些常见的 UI 框架。

1.4.1　构建精美的界面：Element-Plus 库简介

Element-Plus 是“饿了么”的前端团队开源出品的一套为开发者、设计师和产品经理准备的组件库，提供了配套设计资源，帮助网站开发工作快速成型。可以说 Element-Plus 是 Vue 开发者一定要掌握的框架。如果读者开发过 Vue 2，那么一定听说过 Element-UI，这个框架可以说是 Vue 开发者使用率最高的 UI 框架了，而这个 Element-Plus 就是 Element-UI 的 Vue 3 版本。

Element-Plus 的官方网址为 https://element-plus.org/zh-CN/，如图 1.22 所示。

图 1.22　Element-Plus 的官方网站

1.4.2　借助 Ant Design Vue 进行快速开发

Ant Design 作为一门优秀的设计语言，经历过多年的迭代和发展，它的 UI 设计已经有了一套别具一格的风格，截止到目前，在 GitHub 上 Ant Design Vue 已经有 8 万余颗星，可以说是 React 开发者手中的神兵利器。作为 Ant Design 的 Vue 实现，Ant Design Vue 不仅继承了 Ant Design 的设计思想和极致体验，而且结合了 Vue 框架的优点和特性。Vue 3 上的 Ant Design 包更小、更轻并且支持 SSR。Ant Design 拥有成熟的复杂组件，如数据表、统计框、pop 确认、模态和弹出窗口等。

Ant Design 的官方网址为 https://www.antdv.com/docs/vue/introduce-cn，如图 1.23 所示。

图 1.23　Ant Design Vue 的官方网站

1.4.3　打造轻巧的应用：认识 Vant 3 组件库

Vant 系列是有赞出品的开源移动 UI 组件库，基于 Vue 3 重构发布了 Vant 3。Vant 系列在移动端的地位相当于桌面端的 Element，是非常流行的一款 UI 组件库，经过一段时间的迭代，Vant 目前的版本已经非常稳定了。Vant 是一个轻量级的框架，性能极佳，其一共有 70 多个高质量组件，能覆盖移动端的主流场景，组件平均体积小于 1KB。不仅如此，Vant 还同时支持 Vue 2、Vue 3 和微信小程序。

Vant 3 的官方网址为 https://vant-ui.github.io/vant/#/zh-CN，如图 1.24 所示。

1.4.4　跨平台开发利器：uni-app 框架简介

uni-app 不能说是一个 UI 框架，但是在业内还是拥有很多受众的，因此有必要了解一下它。uni-app 是一个使用 Vue 开发所有前端应用的框架，一套代码可以发布到 iOS、

Android、Web 和小程序等平台上，类似于 ionic。它的页面文件遵循 Vue 的单文件组件规范，组件标签规范类似于微信小程序。总之，这种一套代码发布多平台的框架，还是比较受欢迎的。

uni-app 的官方网址为 https://zh.uniapp.dcloud.io/，如图 1.25 所示。

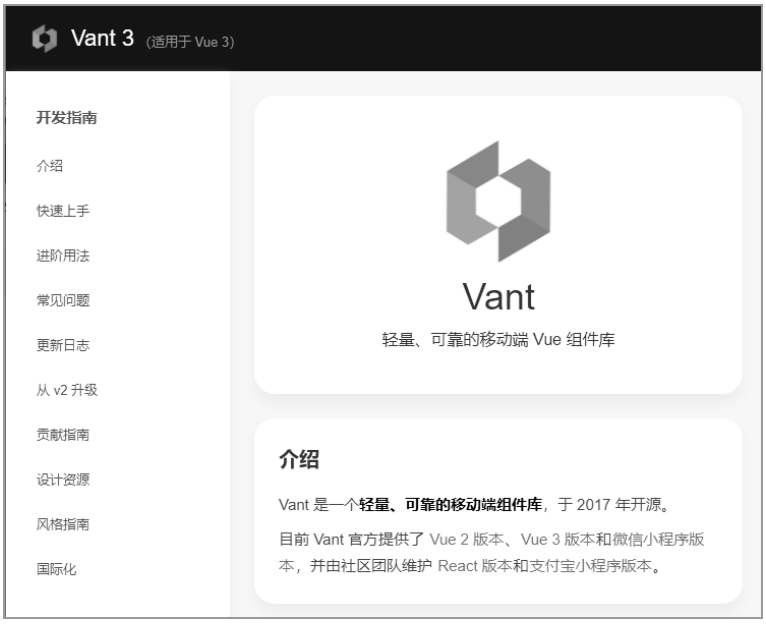


图 1.24　Vant 3 的官方网站

图 1.25　uni-app 的官方网站

1.5　小　　结

本章主要介绍了 Vue 框架的背景和历史，以及如何编写第一个 Vue 程序。除此之外，本章还列举了 Vue 项目的目录结构和 Vue CLI 的使用方法，方便读者快速查阅。在接下来的章节中，读者将逐步学习 Vue 框架的各个部分。

本书注重实战，因此不会在框架的介绍上花费过多篇幅。后续章节将通过大量的实例和代码演示，帮助读者快速掌握 Vue 框架的应用。正如 Linus Torvalds 所言：“Talk Is Cheap, Show Me The Code.”，笔者也认为实践比概念更重要。

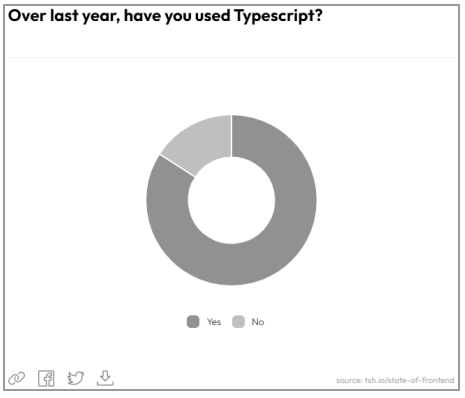
第 2 章　TypeScript 基础知识

2022 年 5 月 31 日，The Software House 发布了一份 2022 前端行业报告“State of Frontend 2022”。其中有一项关于 TypeScript 的调查显示，2022 年使用 TypeScript 的前端开发者占比上升到了 84.1%，如图 2.1 所示。可以说这些年随着前端技术的发展，TypeScript 所处的地位越来越重要。为什么它会如此火热呢？因为 TypeScript 依托于微软的官方支持，而且它完全兼容 JavaScript。

TypeScript 是 JavaScript 的一种超集，兼容 JavaScript 的所有语法，并提供了额外的静态类型系统、类和接口等特性。这些额外的特性使得 TypeScript 的代码变得更加易读和易维护，并可以在编译时发现一些编码错误。

与 JavaScript 不同，TypeScript 要求声明变量类型，并使用类型检查进行静态类型检查。这可以在编码过程中帮助开发人员发现问题，减少错误，并提高代码质量。TypeScript 还提供了一些高级语法，如类、接口和泛型等，使得编写大型应用程序更加容易。此外，TypeScript 还支持 ES 6 和 ES 7 等最新的 JavaScript 语法，并且可以编译为兼容性更好的 JavaScript 代码。

总体而言，TypeScript 是一种强类型的编程语言，提供了一些额外的静态类型系统、高级语法和工具，使得开发大型应用程序更加容易和高效。希望读者通过本章的学习可以加深对 TypeScript 的理解。



Company type	Typescript	
	Not using TypeScript	Using TypeScript
Not provided	32.67%	67.33%
Government organization	19.35%	80.65%
Non-tech-first company	19.21%	80.79%
Software development company / developer agency	12.93%	87.07%
Tech-first / digital-first company	12.16%	87.84%

图 2.1　The Software House 2022 前端报告 TypeScript 部分截图

本章涉及的主要内容点如下：

- TypeScript 简介；
- 基础数据类型；
- 函数；
- 类；

- 泛型；
- 交叉类型与联合类型。

2.1　TypeScript 简介

TypeScript 作为 JavaScript 的超集，添加了可选的静态类型和基于类的面向对象编程。TypeScript 是由微软公司的 Anders Hejlsberg 主导开发的，并于 2013 年 6 月 19 日正式发布。TypeScript 是一门开源的编程语言，众多的开发者都在为完善这个项目而努力，公用项目地址为 https://github.com/Microsoft/TypeScript。

对于有 JavaScript 基础的读者来说，虽然入门 TypeScript 很容易，只需要简单掌握其基础的类型系统就可以轻松上手，但是当应用越来越复杂时，容易把 TypeScrip 写成 AnyScript（即大量地把变量设置为 any 类型）。因此，想要完全掌握 TypeScript 的特性，还是需要进行系统化的学习。

2.1.1　动态语言与静态语言

在学习 TypeScript 之前，先了解一下动态类型语言和静态类型语言的基本概念。

1．动态类型语言

动态类型语言在运行期才进行类型检查。其主要优点在于可以少写很多类型声明代码，更自由并且易于阅读。JavaScript 就是一门动态类型语言。

2．静态类型语言

静态类型语言在编译期就会进行数据类型检查。其主要优点在于类型的强制声明，使得 IDE 有很强的代码感知能力，能在早期发现一些问题，方便调试。TypeScript 就是一门静态类型语言。

总之，动态类型语言和静态类型语言各有优势，它们的主要区别就是适合的使用场景不同，没有必要在哪个更好上进行争论。

2.1.2　搭建开发环境

本节搭建一套开发环境，以方便运行 TypeScript 练习代码。接下来笔者会介绍两种方案，读者自行选择一种即可。

1．通过在线网站运行

如果计算机可以连接互联网，推荐使用 TypeScript 官方网站的演练场功能直接运行，网站地址是 https://www.typescriptlang.org/zh/play，如图 2.2 所示。

图 2.2　在线直接运行 TypeScript 代码

从图 2.2 中可以看到，这个网站的功能简单、明了，开箱即用，还支持切换 TypeScript 版本，不想折腾的话直接打开就可以进行下一小节的练习了。

2．使用ts-node在本地运行

如果读者的动手能力比较强，那么可以自己搭建一套本地环境。输入以下代码使用 NPM 进行安装。

```
npm install -g typescript
npm install -g ts-node
```

第一个命令安装的是 TypeScript，第二个命令安装的是 TypeScript 的 Node.js 运行环境。安装完成后可以输入以下代码检查版本号，如图 2.3 所示。

```
tsc --version
ts-node --version
```

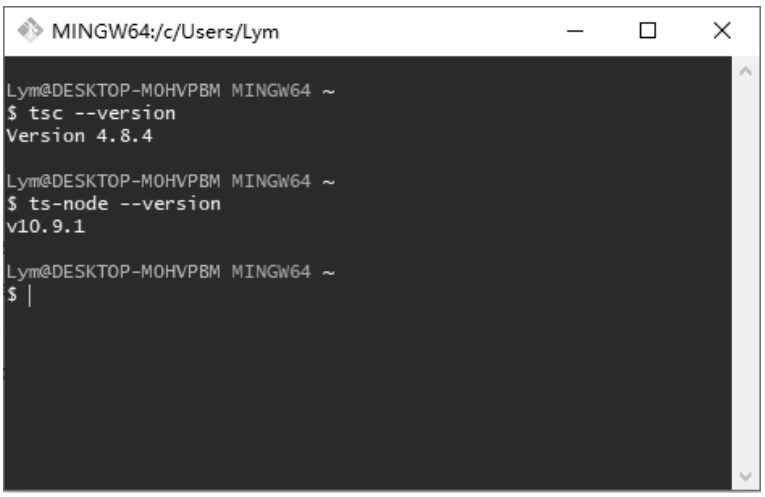


图 2.3　查看当前操作系统安装的 TypeScript 和 ts-node 版本

运行代码的方式也很简单，新建一个 test.ts 文件并在其中输入一行测试代码。

```
console.log('Hello TypeScript');
```

之后使用命令行进入 test.ts 文件目录并使用 ts-node 运行该文件。

```
ts-node test.ts
```

运行成功后的效果如图 2.4 所示。

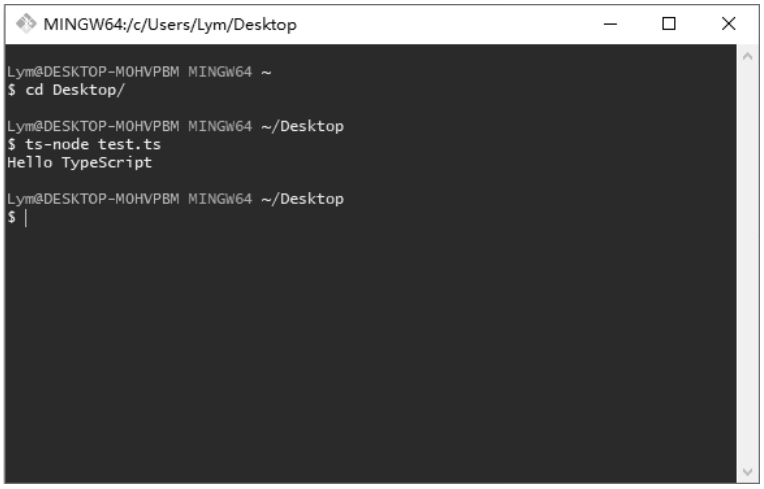


图 2.4　使用 ts-node 运行本地的 TypeScript 文件

如果在运行过程中报出如图 2.5 所示的错误，那么可能是因为新版 TypeScript 缺少运行依赖包，可以输入以下代码安装@types/node 予以解决。

```
npm i -g tslib @types/node
```

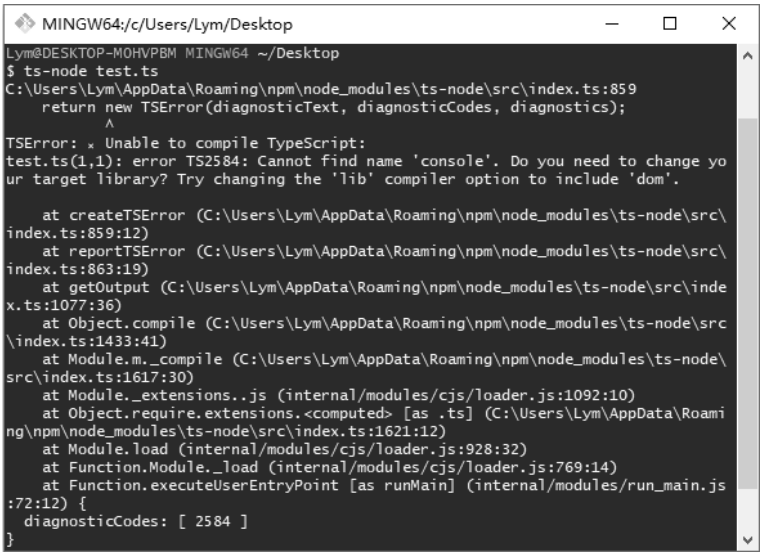


图 2.5　缺少运行依赖包报错

运行以上命令后，会在当前目录下生成 node_module 文件夹和 package-lock.json 文件，如图 2.6 所示。建议把要运行的代码单独放一个文件夹内，避免文件过多产生混乱。

如果运行困难，又不希望下载这么多的依赖文件，可以安装旧版的 TypeScript，则不需要安装@types/node 也可以运行。执行以下代码，可以退回到指定的版本。

```
npm i -g ts-node@8.5.4
```

笔者已经测试过，8.5.4 版本可以直接运行.ts 文件。虽然缺少一些新特性，但是练习使用也足够了。

图 2.6　执行命令后所生成的新文件

☎**提示**：ts-node 工具主要依赖的是 Node.js，让代码可以脱离浏览器运行，它的作用是将 TypeScript 代码编译为 JavaScript 代码。

2.2　基础数据类型

无论学习哪一种编程语言，首要任务都是掌握数据类型。在 TypeScript 中，有多种数据类型需要了解。本节将着重介绍 TypeScript 中常用的数据类型，并为每种类型提供相关的示例代码。为确保内容的连贯性和易懂性，本节的实例代码将着重讲解各种常用数据类型的使用方法。如果读者在运行代码时遇到报错情况，可下载配书代码，将其复制并粘贴到相应的运行环境下进行调试。

2.2.1　布尔类型

与其他编程语言一样，在 TypeScript 中，布尔类型的值为 true 和 false，分别代表真和假。

下面的示例定义一个布尔类型变量，赋值为 false 并输出。

```
// 声明一个布尔类型变量
let isDog: boolean = false;
console.log(isDog);

// 输出结果：false
```

注意，代码中的“: boolean”就是为变量定义数据类型，语法是冒号后面跟要设置的数据类型，所有的数据类型都是通过这种方式设置的。

2.2.2　数字类型

TypeScript 中的数字类型都是浮点数，因此整数可以直接与带小数点的数字进行运算。

下面的示例定义两个数字类型变量，分别赋值整数与浮点数进行相加。

```
// 声明两个数字类型变量
let num1: number = 10;
let num2: number = 5.5;
console.log(num1 + num2);

// 输出结果：15.5
```

虽然 TypeScript 分了各种数据类型，但是并没有把数字像其他静态语言一样拆成类似 int、float、double 来区分整数和小数，而是直接使用 number 类型，整数和小数可以直接进行加、减、乘、除。

2.2.3　字符串类型

TypeScript 中的字符串类型使用单引号与双引号来表示，并且支持 ES 6 的反引号“`”来操作。

下面的示例分别用单引号和双引号定义两个字符串类型变量，第三个字符串使用反引号拼接输出。

```
// 声明两个字符串类型变量
let str1: string = 'Hello';
let str2: string = "TypeScript";
// 拼接 str1 与 str2
let str3: string = `${str1} ${str2}!`;
console.log(str3);

// 输出结果：Hello TypeScript!
```

2.2.4　数组类型与元祖类型

在 TypeScript 中定义数组的方式有两种，元祖类型实质上也是数组类型，只是允许数组添加不同类型的值。

下面的示例分别使用不同的方式初始化一个新的数组，并示范如何创建元祖类型数组。

```
// 元素类型后接中括号
let arr1: number[];

arr1 = [1, 2, 3];                                    // 正确赋值
arr1 = ['a', 'b', 'c'];                              // 错误赋值

// 使用数组泛型，Array<元素类型>
let arr2: Array<number> = [1, 2, 3];


// 定义元组类型
let arr3: [string, number];

arr3 = ['张三', 18];                                  // 正确赋值
arr3 = [18, '张三'];                                  // 错误赋值
```

如果给数组或元祖类型传入非定义类型的值，则编译器会直接报错。

2.2.5　枚举类型

为了防止代码出现过多的魔力数字（magic number，通常指缺乏解释或命名的独特数值），因此枚举类型的存在是十分必要的。与其他语言类似，在 TypeScript 中，默认情况下

从 0 开始编号，也可以手动指定成员数值。

下面的示例定义两组枚举并以不同方式指定数值。

```
// 指定第一个
enum Color {Red = 1, Green, Blue}
console.log('Red=' + Color.Red);
console.log('Green=' + Color.Green);
console.log('Blue=' + Color.Blue);

// 输出结果：Red=1  Green=2  Blue=3

// 全部指定
enum Animal {Dog = 2, Cat = 4, Bird = 5}
console.log('Dog=' + Animal.Dog);
console.log('Cat=' + Animal.Cat);
console.log('Bird=' + Animal.Bird);

// 输出结果：Dog=2 Cat=4 Bird=5

// 通过数值获取定义名
console.log('2=' + Animal[2]);
console.log('4=' + Animal[4]);
console.log('5=' + Animal[5]);

// 输出结果：2=Dog 4=Cat 5=Bird
```

枚举的功能还是十分强大的，不仅可以通过名字获取数值，反过来也是可以实现的。掌握枚举类型，可以更好地编写代码。

2.2.6　any 类型

any 类型实际上相当于移除了类型检查，它允许像 JavaScript 一样赋值任意类型。

下面的示例定义一个 any 类型变量，先赋值为数字类型的 18，后赋值为字符串类型的 eighteen，最后进行输出。

```
// 声明一个 any 类型变量
let age: any = 18;
// 跨类型修改值
age = 'eighteen';
console.log(age);

// 输出结果：eighteen
```

2.2.7　void 类型

void 类型的意思就是没有类型。对于变量来说 void 类型只能赋值 undefined 和 null，而当函数返回值为空时通常可以省略，因此用处不大。

void 类型的使用示例如下：

```
// undefined
let unusable: void = undefined;

// 返回值为空
```

```
function someFunction(): void {}
// 通常省略
function someFunction() {}
```

2.2.8　null 与 undefined 类型

null 与 undefined 其实都有自己的类型，只是它们和 void 一样作用并不大。null 和 undefined 是所有类型的子类型，除非指定了--strictNullChecks 标记，才能让它们只能赋值给自己。

null 与 undefined 类型的使用示例如下：

```
// 声明一个变量 A，其类型为 undefined，赋值为 undefined
let A: undefined = undefined;

// 声明一个变量 B，其类型为 null，赋值为 null
let B: null = null;
```

2.2.9　never 类型

never 表示不可能存在的值的类型。never 类型通常用于那些会导致错误或抛出异常而永远不会正常返回的函数的返回值类型。

例如，当函数内部抛出异常或包含无限循环时，函数将永远不会正常返回结果，这时可以将其返回类型标注为 never。

never 类型的使用示例如下：

```
// 函数内部抛出异常，返回类型标注为 never
function throwError(message: string): never {
  throw new Error(message);
}

// 无限循环，返回类型标注为 never
function infiniteLoop(): never {
  while (true) {
    // Do something indefinitely
  }
}
```

2.3　函　　数

在 TypeScript 中，函数的概念十分重要，TypeScript 的函数可以使用静态类型进行声明。这意味着在定义函数时，必须声明函数的参数类型和返回类型。TypeScript 对 JavaScript 中的函数增加了更多很方便的功能，对开发效率的提升十分显著。

2.3.1　函数的使用

与 JavaScript 一样，TypeScript 可以创建带名称的函数和匿名函数。下面展示这两种函

数的使用方法。

void 类型的使用示例如下：

```
function addNumber(a: number, b: number) {
  return a + b;
}
console.log(addNumber(1, 2));

// 输出结果：3

let addString = function (a: string, b: string) {
  return a + b;
};
console.log(addString('Hello', 'TypeScript'));

// 输出结果：HelloTypeScript
```

2.3.2　构造函数

构造函数是一种特殊的函数，主要用于创建对象时初始化对象，常与 new 运算符一起使用。TypeScript 的构造函数用关键字 constructor 来实现，可以通过 this 来访问当前类的属性和方法。

下面的示例创建一个简单的构造函数，然后用该函数初始化一个新对象并输出。

```
class Student {                                    // 定义 Student 类
  name: string;                                    // 定义类的属性 name
  age: number;                                     // 定义类的属性 age
  constructor(name: string, age: number) {         // 定义构造函数
      this.name = name;
      this.age = age;
  }
}

let stu = new Student('张三', 18);
console.log('name=' + stu.name + ' age=' + stu.age);

// 输出结果：name=Wang Cai age=2
```

2.3.3　可选参数

TypeScript 支持给函数设置可选参数，只要在参数后面增加问号标识即可实现。

下面的示例新建一个带有可选参数的函数，并进行可选传值测试。

```
function add(a: number, b?: number) {
  if (b) {
    return a + b;
  } else {
    return a;
  }
}

console.log(add(3, 2));

// 输出结果：5
```

```
console.log(add(3));

// 输出结果：3
```

☎**提示：**使用可选参数时，有一个规则需要注意，就是必选参数不能位于可选参数之后，错误的使用将会导致编译报错。

2.3.4　默认参数

TypeScript 同样支持给函数设置默认参数，只要在参数声明后用等号进行赋值即可。下面的示例新建一个带有默认参数的函数，并进行默认传值测试。

```
function add(a: number, b: number = 5) {
  if (b) {
    return a + b;
  } else {
    return a;
  }
}

console.log(add(3, 2));

// 输出结果：5

console.log(add(3));

// 输出结果：8
```

2.3.5　箭头函数

在 JavaScript 中，this 的作用域是一个常见的问题，注意看下面这个例子。

```
const Person = {
  'name': '张三',
  'printName': function () {
    let fun = function () {
      return '姓名：' + this.name;
    };
    return fun();
  }
};
console.log(Person.printName());

// 输出结果：姓名：undefined
```

遇到这种问题时，通常先声明一个变量 self，在函数外部先绑定正确的 this，在函数内部再通过 self 调用 name 属性，示例如下：

```
const Person = {
  'name': '张三',
  'printName': function () {
    let self = this;
    let fun = function () {
      return '姓名：' + self.name;
```

```
    };
    return fun();
  }
};
console.log(Person.printName());

// 输出结果：姓名：张三
```

箭头函数提供了另一种方便的解决方案，它的内部的 this 是词法作用域，因此不需要再进行多余的操作。

使用箭头函数解决 this 作用域问题示例如下：

```
const Person = {
  'name': '张三',
  'printName': function () {
    let fun = () => {
      return '姓名：' + this.name;
    };
    return fun();
  }
};
console.log(Person.printName());

// 输出结果：姓名：张三
```

使用上面这种方式不仅使代码更加精简、清晰，便于阅读，而且也规避了一些可能会出现的错误。

2.4　类

类（Class）是面向对象编程（Object Oriented Programming）中的一种抽象数据类型，用于描述一组具有相同属性和方法的对象。类是对象的模板，对象是类的实例。不同于 JavaScript 使用函数和基于原型的继承，在 TypeScript 中是基于类的继承并且对象是由类构建出来的。

2.4.1　属性与方法

在 TypeScript 中创建属性的方法与 JavaScript 类似，一般变量使用 let、常量使用 const 进行创建。变量类型在变量名后面加冒号声明，可以省略。TypeScript 中的方法则需要创建在类中，默认为 public，返回值是在方法名后面加冒号声明，如果无返回值则可以省略或写为 void。

属性与方法的使用示例如下：

```
// 属性示例
let name1: string = '张三';
let name2 = '张三';
const name3 = '张三';

// 方法示例
class User {
```

```
  getUserName(): string {
    return '张三';
  }
}
let user = new User();
console.log(user.getUserName());

// 输出结果：张三
```

2.4.2　类的继承

TypeScript 中类的继承是通过 extends 关键字实现的，派生类通常被称为子类，基类通常被称为父类，子类中使用 super 关键字来调用父类的构造函数和方法。

下面的示例简单演示 TypeScript 中类的继承的使用。

```
// 定义一个名为 Person 的类
class Person {
  name: string;
  constructor(name: string) {
      this.name = name;
  }
}

// 定义一个名为 Student 的类，继承自 Person
class Student extends Person {
  constructor(name: string) {
      super(name);
  }
}

// 创建一个名为 stu 的 Student 类的实例，传入'张三'作为名字参数
let stu = new Student('张三');
console.log(stu.name);

// 输出结果：张三
```

在这个例子中 Student 类继承了 Person 类，并在构造函数中调用了父类的构造函数，最终输出父类的属性 name。

2.4.3　类的实现接口

实现（Implement）是 TypeScript 中的重要概念，一般用于实现接口（Interface）。通常一个类只能继承自另一个类，有时候不同类之间可以有一些共有的特性，这时候就可以把特性提取成接口。当一个类实现了某个接口时，它就必须实现该接口中定义的所有属性和方法。这有助于确保类具有所需的结构和行为，并且可以在编译时发现实现错误。

下面的示例简单演示 TypeScript 中的类的实现接口。

```
// 定义一个接口 Student
interface Student {
   sayHello(): void;
}

// 定义一个类 People，实现了接口 Student
```

```
class People implements Student {
    sayHello() {
        console.log('你好');
    }
}

// 创建一个名为 stu 的 People 类的实例
let stu = new People();
// 调用 stu 实例的 sayHello 方法
stu.sayHello();

// 输出结果：你好
```

本例首先定义 Student 接口并声明了 sayHello 方法。在 People 类中实现了 Student 接口的 sayHello 方法，最后新创造一个对象并调用了 sayHello 方法。

2.4.4　权限修饰符

与众多的编程语言一样，TypeScript 有自己的权限修饰符 public、private、protected 和 readonly，接下来分别举例讲解不同权限修饰符的用法。

1．public

在 TypeScript 中，public 是默认修饰符，表示任何地方都可以访问使用。

public 修饰符的使用示例如下：

```
// 定义一个类 Student
class Student {
  // 声明一个公共（public）实例变量 name，类型为 string
  public name: string;
  constructor(name: string) {
      this.name = name;
  }
}

// 创建一个名为 stu 的 Student 类的实例，传入"张三"作为名字参数
let stu = new Student("张三");
console.log(stu.name);

// 输出结果：张三
```

2．private

使用 private 指定的成员是私有的，只能在当前类中访问。

private 修饰符的使用示例如下：

```
// 定义一个类 Student
class Student {
  // 声明一个私有（private）实例变量 name，类型为 string
  private name: string;
  constructor(name: string) {
      this.name = name;
  }
}
```

```
// 创建一个名为 stu 的 Student 类的实例，传入"张三"作为名字参数
let stu = new Student("张三");
console.log(stu.name);

// 输出结果：错误：'name'是私有属性
```

3．protected

使用 protected 指定的成员是受保护的，只能在当前类和子类中访问。修改前面类继承的例子，增加一个 protected 修饰符，根据输出结果可以看到仍然可以访问。

protected 修饰符的使用示例如下：

```
// 定义一个类 Person
class Person {
  // 声明一个受保护（protected）的实例变量 name，类型为 string
  protected name: string;
  constructor(name: string) {
      this.name = name;
  }
}

// 定义一个类 Student，继承自 Person
class Student extends Person {
  constructor(name: string) {
      super(name);
  }
}

// 创建一个名为 stu 的 Student 类的实例，传入'张三'作为名字参数
let stu = new Student('张三');
console.log(stu.name);

// 输出结果：张三
```

4．readonly

readonly 用于修饰属性或者字段。它表示该属性或字段是只读的，不能在任何地方修改。

private 修饰符的使用示例如下：

```
// 定义一个类 Student
class Student {
  // 声明一个只读（readonly）实例变量 name，类型为 string
  readonly name: string;
  constructor(name: string) {
      this.name = name;
  }
}

// 创建一个名为 stu 的 Student 类的实例，传入"张三"作为名字参数
let stu = new Student("张三");
console.log(stu.name);
// 输出结果：张三
stu.name = "李四";
// 错误：Cannot assign to 'name' because it is a read-only property
```

2.5　泛　　型

在构建自己的项目时，使用的组件不仅要考虑当前的数据类型，而且应该支持未来可能会加入的数据类型，这样在开发大型系统时才会更加灵活，复用性更高。很多初学者认为自己的实力不足，没有机会自己来构建项目。其实，只要努力提高自己的技能，积极工作，总会有机会负责一个项目，这时候就需要从复用性等架构层面来思考问题，而泛型的出现就是用来解决这个问题的。一个组件可以支持多种类型的数据，这样用户就可以以自己的数据类型来使用组件。

2.5.1　泛型示例

首先创造一个简单的示例。假设现在需要一个 getValue 函数，用来返回传入值。如果传入值是 number 类型，那么代码就是这样的：

```
// 定义一个函数 getValue，接收一个 number 类型的参数 arg，返回一个 number 类型的值
function getValue(arg: number): number {
    return arg;
}

let output = getValue(123);
console.log(output)

// 输出结果：123
```

可是这样会产生一个问题，因为声明的类型是 number，所以传入其他类型的参数就会提示错误。为了解决这个问题，把代码修改为如下：

```
// 定义一个函数 getValue，接收一个 any 类型的参数 arg，返回一个 any 类型的值
function getValue(arg: any): any {
    return arg;
}

let output = getValue("Hello");
console.log(output)

// 输出结果："Hello"
```

现在类型问题解决了，但是带来了一个新的问题。使用 any 类型虽然可以让这个函数正常工作，但是无法知道传入类型和返回类型是否相同。这句话可能不太好理解，简单来说，如果传入一个 number 类型且返回值声明的是 any，那么无法确定返回的就是 number 类型。这时候就需要使用类型推论做第二次优化。

在下面的示例中演示如何使用类型推论来解决上述问题。

```
// 定义一个泛型函数 getValue，使用类型参数 T，接收一个参数 arg，返回与传入参数相同的值
function getValue<T>(arg: T): T {
    return arg;
}

let output = getValue("Hello");
console.log(output)
```

```
// 输出结果："Hello"
```

本示例给函数添加了类型变量 T，编译器会根据传入的参数自动确定 T 的类型，然后把这个类型设置给返回值，这样就可以确定参数类型与返回值类型是相同的了。T 这个字母表示 Type 的意思，因此使用频率最高。当然用其他有效名称代替也是可以正常运行的。

有的读者可能会疑惑，这不是跟上个例子的结果一样吗？用 any 应该更方便。在简单的代码结构中看起来确实是这样的，不过项目的复杂度一旦提高，上述的类型推论（Type Inference）功能可以帮助开发者推断变量的类型，规避潜在的问题。用 TypeScript 的目的就是在大型项目中提高类型的预知并降低出现 bug 的概率。如果大量使用 any，那么就失去使用 TypeScript 的意义了。

2.5.2　泛型接口

TypeScript 的泛型接口可以用来定义可以适用于多种类型的接口，该接口可以在不必明确类型的情况下定义复杂的数据结构。

泛型接口示例如下：

```
// 定义一个接口 Student，使用类型参数 T 和 U
interface Student<T, U> {
  name: T,
  age: U
}

// 定义一个泛型函数 getStudentInfo，使用类型参数 T 和 U
function getStudentInfo<T, U> (name: T, age: U): Student<T, U> {
  let studentInfo: Student<T, U> = {
    name,
    age
  };
  return studentInfo;
}

console.log(getStudentInfo("张三", 18));

// 输出结果：{ "name": "张三", "age": 18 }
```

本示例首先定义一个名为 Student 的泛型接口，该接口接收两个类型变量 T 和 U，并定义了两个属性。然后在 getStudentInfo 函数内部创建一个 studentInfo 对象，该对象符合 Student 接口的定义，并通过接收的参数进行初始化。最后输出姓名和年龄信息。

2.5.3　泛型类

TypeScript 的泛型类是带有一个或多个类型变量的类，其可以用来创建适用于多种类型的类。

泛型类示例如下：

```
// 定义一个类 Student，使用类型参数 T
class Student<T> {
```

```
  // 声明一个实例变量 age，类型为 T
  age: T
  constructor(age: T) {
    // 在构造函数中初始化 age 属性
    this.age = age
  }
  getAge(): T {
    // 返回 age 属性的值
    return this.age
  }
}

const myNumberClass1 = new Student<Number>(15);
console.log(myNumberClass1.getAge());
// 输出结果：15

const myNumberClass2 = new Student<string>("15");
console.log(myNumberClass2.getAge());
// 输出结果："15"
```

本示例定义一个 Student 的泛型类，并在构造函数中给 age 属性赋值。getAge 方法用来返回类的 age 属性。分别传入 number 和 string 类型的初始化参数，就可以得到不同类型的返回值了。

2.5.4　泛型约束

在函数内部使用泛型变量的时候，由于事先不知道它是哪种类型，所以不能随意操作它的属性或方法。尝试输入以下代码：

```
// 定义一个泛型函数 getValue，使用类型参数 T
function getValue<T>(value: T): T {
    console.log(value.length);
    return value;
}

// 报错信息：Property 'length' does not exist on type 'T'.
```

很显然，由于编译器不确定 T 类型是否有 length 属性，所以直接报错。因此很有必要对泛型进行约束，如只允许这个函数传入包含 length 属性的变量。

泛型约束示例如下：

```
// 定义一个接口 Lengthwise，具有一个 length 属性
interface Lengthwise {
    length: number;
}

// 定义一个泛型函数 getValue，使用类型参数 T，受限于接口 Lengthwise
function getValue<T extends Lengthwise>(value: T): T {
    console.log(value.length);
    return value;
}

getValue('abcde')
// 输出结果：5
getValue(true)
```

```
// 报错信息：Argument of type 'boolean' is not assignable to parameter of type
'Lengthwise'
```

上面的这段代码比较容易理解，字符串类型有长度，所以输出 5；布尔类型没有长度，所以会报错。

2.6　交叉类型与联合类型

要掌握 TypeScript 的知识点，除了 2.2 节介绍的基础类型外，还有必要了解它的高级类型。高级类型包含交叉类型（Intersection Types）和联合类型（Union Types），本节会分别介绍这两个高级类型的用法及它们的异同点，帮助读者在日后的开发中准确选择应该使用的类型。

2.6.1　交叉类型

交叉类型是指将多个类型合并为一个类型。这样的类型具有所有输入类型的特征，并且它们的实例可以同时具有多个类型的所有属性和方法。

语法：type IntersectionType = TypeA & TypeB，可以使用 & 运算符合并多个类型。

交叉类型示例如下：

```
// 定义一个接口 Person，具有 name 属性和 age 属性
interface Person {
  name: string;
  age: number;
}

// 定义一个接口 Student，具有 studentId 属性和 score 属性
interface Student {
  studentId: string;
  score: number;
}

// 使用交叉类型（&）创建一个新的类型 PersonStudent
// 包含 Person 和 Student 的属性
type PersonStudent = Person & Student;

// 创建一个名为 personStudent 的对象，类型为 PersonStudent
const personStudent: PersonStudent = {
  name: '张三',
  age: 18,
  studentId: '01',
  score: 100
};
```

在本示例中使用了交叉类型创建一个名为 PersonStudent 的新类型，该类型具有 Person 和 Student 两种类型的所有特性。

这种方法非常适合多种类型的合并，它允许合并多个类型以更好地描述一个特定的对象，并且可以确保该对象具有所需的所有属性和方法。

2.6.2　联合类型

联合类型表示一个值可以是几种类型之一。使用联合类型，可以将多种类型的值合并为一个类型。

语法：type Union Types = TypeA | TypeB，可以使用 | 运算符合并多个类型。

联合类型示例如下：

```
// 定义一个名为 test 的变量，类型为 string 或 number
const test: string | number = 'hello';
test = 123;

// 定义一个名为 value 的变量，类型为 string 或 string 数组
const value: string | string[] = 'hello';
value = ['hello', 'world'];
```

在本例中，变量 test 的类型为 string | number，它可以是字符串类型或数字类型。同样，变量 value 的类型为 string | string[]，它可以是字符串类型或字符串数组类型。

联合类型特别适用于不确定的值，如从 API 获取的数据，或者对函数返回值类型的描述。使用联合类型，可以在不损失类型安全性的前提下处理多种情况。

2.7　小　　结

本章首先对 TypeScript 进行了简单介绍，并对现阶段动态语言与静态语言进行了对比，之后搭建了开发环境并对其常见的知识点进行了讲解。最后需要明确一点，TypeScript 的出现并不是要完全取代 JavaScript，二者虽然同源但各有特色，学习 TypeScript 之后并不需要完全放弃 JavaScript。目前，Vue 对于 TypeScript 的支持也在日益完善，在掌握了 TypeScript 的相关基础知识后，使用这门语言进行 Vue 应用开发已经不成问题了，相信熟悉 JavaScript 的读者很快就能掌握本章的内容。

对于基础知识点来说，使用强类型检查反而会增加理解难度，因此在后面的内容中，笔者不会再对每个章节的示例代码都使用 TypeScript。