第1章

项目概览与环境准备

本书所介绍的电商项目，其前端将使用Vue.js框架进行开发，前端项目包含用户端和后台管理端；后端则将采用Node.js平台下的Express框架进行搭建。与Vue.js的设计理念类似，Express框架也以小巧灵活为主要特点，对于快速开发中小型项目非常高效。

本章首先介绍Vue.js和Express这两个开发框架，并搭建开发项目所需要的基础环境。通过本章内容，读者将能够了解一个成熟的商业项目从0到1的全过程，并且可以将自己的理论学习经验付诸实践，完成让自己满意的编程作品。

我们一起开始这段有趣的编程之旅吧。

本章学习目标：

- 传统电商项目的功能构成。
- 了解 Vue.js 框架。
- 了解 Express 框架。
- 了解 TypeScript 语言。
- Vue.js 框架和 Express 框架的脚手架的应用。
- 安装 Node.js 环境以及 Visual Studio Code 编程工具。
- 前端和后端项目的 HelloWorld 工程解析。

1.1 项目概览

在互联网技术如此发达的今天，我们生活的方方面面几乎都有着互联网的影子。通过互联网，我们可以进行资讯阅读，也可以听音乐、看电影、进行社交通信以及游戏娱乐等。互联网使得我们的生活和工作变得更加便利和高效。在电子商务领域，互联网改变了传统的购物方式，让买家更容易找到心仪的产品，并且足不出户即可收到自己所购买的物品。同时，这也让卖家的开店

成本降低，卖家能更高效地推广和销售自己的产品。相信大部分人都有过电商购物的经验，在享受电商购物带来的便利的同时，不知你是否思考过这些问题：

- 这样一个电商系统软件的架构是怎么样的？
- 在进行商品购买时，下单逻辑和购物车逻辑是如何串联在一起的？
- 商家怎么管理自己的商铺？如何进行商品的上架、下架和库存管理？
- 订单的状态是如何变更和维护的？
- 历史订单的数据保存在哪里？
- 商品的用户评价是怎么产生的？
- ……

如果你对这些问题很有兴趣，那么恭喜你，通过本项目的学习，这些疑问都将得到解答。

1.1.1　电商项目的功能构成

电商项目的核心是购物，因此所有功能都是围绕这个主题来展开的。平时我们在使用电商网站进行网络购物时，实际上使用的是整个电商项目中的客户端部分。当然，随着移动设备的普及，客户端已经不仅仅局限于网页上，iOS App、Android App也都是客户端的一部分。本书所涉及的客户端主要指网页客户端。

1. 电商客户端

对于电商客户端，将要实现的功能按照模块进行划分，可以大致得到如下几个模块：

- 用户登录和注册模块。
- 网站首页商品分类与列表模块。
- 商品搜索模块。
- 商品详情页。
- 购物车模块。
- 订单模块。
- 商品评价模块。

其中，用户登录和注册模块提供基础的用户体系支持，要在电商网站中购买商品，首先需要注册一个会员账户，像购物车、商品评价、订单等模块的功能都是和会员账户绑定的。

网站首页商品分类与列表模块主要提供商品展示能力，会员用户可以在这个功能模块中进行商品的选购。

商品搜索模块也是电商系统很重要的一部分，有些用户有很强的购物方向，通过搜索可以快速匹配到用户感兴趣的商品。

商品详情页将展示某个商品的详细信息，比如商家设置的商品介绍、商品的价格等，并且在此页面可以将商品加入购物车中。

购物车模块用来管理用户所添加到购物车的商品，方便用户进行下单。

订单模块用来管理用户所有的订单，并且用户可以实时地掌握订单的状态。

商品评价模块用来管理用户的评价。用户购买了商品后，可以在该模块对商品进行评价，其他用户可以在商品详情页看到与此商品相关的评价。

2. 后台管理系统

有了供用户使用的客户端，配套的还需要有电商商家管理员的客户端，即电商平台的后台管理系统。当一个电商项目在上线运行时，不可能让管理员通过执行代码来进行商品的维护、订单的管理等操作。因此，我们需要一个图形化的后台管理系统来为商家提供服务。对应电商客户端的功能模块，电商后台管理系统需要包含如下几个功能模块：

- 管理员的登录和注册模块。
- 商品类别的管理模块。
- 商品管理模块。
- 用户评价管理模块。
- 订单的管理模块。
- 财务管理及数据统计模块。

后台管理系统需要管理员来进行具体的管理操作，因此需要有管理员登录和注册系统。管理员在后台管理系统中可以对商品类别和商品进行添加或删除操作，对商品的评价进行审核，等等。相比用户端，后台管理端的主要服务对象是商家，因此少不了要有一些数据报表的服务，以帮助商家更方便地了解电商业务的运营情况。

3. 服务后端与支付功能

一个完整的电商系统，除了有用户端和后台管理端两个前端外，还需要有一个服务后端来提供数据存储、数据处理等逻辑，例如用户数据的存储、商品数据的存储等，并且将这些操作封装为接口提供给前端使用。这也是本项目需要完成的。

本项目中所涉及的支付部分并不真正接入，只做模拟。在实际项目中，支付通常可以接第三方的支付平台，例如银联支付、微信支付、支付宝支付等。这些支付平台的接入也很简单，每种支付方式都有完整的官方文档和客服人员提供支持，本书不涉及这部分内容。

总之，我们最终要完成的是一个功能简约但完整的电商系统，在客户端，用户可以完整地登录注册、浏览商品、下单等；在后台管理端，管理员可以自主添加商品、管理评价等。

1.1.2　前端框架 Vue.js 及其周边工具

自2010年开始，相继出现了多个影响深远的前端开发框架，如Angular、React.js和Vue.js等，这些框架的应用，开启了互联网网站开发的SPA（Single Page Application）时代，即单页面应用程序时代，这也是当今互联网Web应用开发的主流方向。

在众多前端开发框架中，Vue.js尤其在中国获得了广泛的支持。它不仅被企业广泛采用，还经常出现在学校的教学课程中，拥有庞大的用户群体。这种普及程度使Vue.js成为国内最流行的前端框架之一。正因如此，本书选择Vue.js作为主要的前端项目开发框架，希望借助其广泛的社区支持和丰富的应用实例，帮助读者更有效地学习和掌握前端开发技术。

Vue.js被定义为渐进式的JavaScript框架，所谓渐进式，是指其被设计为可以自底向上逐层应

用。我们可以只使用Vue.js框架中提供的某层的功能，也可以与其他第三方库整合使用。当然，Vue.js本身也提供了完整的工具链，使用其全套功能进行项目的构建也非常简单。

在使用Vue.js之前，需要掌握基础的HTML、CSS和JavaScript（或者TypeScript）的相关基础技能。Vue.js的渐进式性质使其使用变得非常灵活，在使用时，我们可以使用其完整的框架，也可以只使用其部分功能。一个完整的前端项目通常会包含UI界面、网络能力、路由管理能力、状态管理能力等。在本项目中，我们将使用Vue.js生态圈中的提供这些功能的插件。下面对这些插件进行简要介绍。

1. Element Plus

Element Plus是一款基于Vue.js 3的面向设计师和开发者的UI组件库，提供了大量功能强大、使用简洁的组件，让开发者搭建UI页面变得非常高效。

2. axios

axios本身是一个使用promise封装的HTTP网络库，可以在浏览器或Node.js环境中运行。它通过Promise类型的API来处理HTTP请求和响应，包括请求发送、请求和响应的拦截处理、数据解析等功能。vue-axios是基于Vue封装的axios网络工具库，在Vue前端项目中可以使用它来与后端服务进行交互。

3. Vue Router

Vue Router是Vue.js框架配套的前端路由插件，为Vue.js项目提供了可配置的、动态的路由管理功能。

4. Pinia

Pinia是Vue.js状态管理工具插件，相较于Vuex，Pinia提供了更简单的API，并且提供了组合式API风格的实践用法。目前它也是Vue.js官方最为推荐的状态管理工具。

上面提到的这些插件在后续章节都会使用，到时再做详细介绍。即使之前没有使用过这些框架也没有关系，我们会以项目驱动的方式进行介绍，相信读者在使用的过程中可以快速理解与掌握相关技能。

1.1.3　熟悉 Node.js 与 Express

提到JavaScript技术栈，不得不提Node.js运行环境。在Node.js出现之前，JavaScript常常被认为是一种玩具语言，当时JavaScript编写的程序主要是为了配合HTML来增强网页的交互性，使其在浏览器中提供更佳的用户体验。因此JavaScript的应用场景十分有限。2009年5月，Node.js发布，它本身是一个基于Chrome V8引擎的JavaScript运行环境，采用了事件驱动、非阻塞式的I/O模型。Node.js的出现，V8引擎功不可没，V8引擎使用的最新编译技术使得JavaScript脚本语言编写的代码的运行速度得到了极大的提升。我们知道，一个语言的应用场景是和其性能表现相关的，性能的提升为JavaScript语言带来了更多用武之地，有了Node.js，JavaScript语言编写的程序可以脱离浏览器而单独运行，可以用于开发Web服务、桌面应用、移动端应用以及终端命令行工具等。

对于本书将要完成的电商项目，我们就将使用Node.js平台来搭建后端服务，为电商网站的用户端和后台管理端提供数据服务支持。

Express是Node.js平台上的一款极简的Web开发框架。通过Vue.js+Express的技术栈组合，可以让我们使用一致的编程语言和设计思路来进行项目的构建，从而更好地理解一个完整项目的前后端开发流程。另外，这也减少了编程语言和编程思想的学习障碍，使得我们能更轻松地完成整个项目。

下面，我们再来详细看一下Express这个框架。Express可用来构建Web应用程序，简单来说，它可以帮助我们以最小的规模和极大的灵活度来构建功能完整的Web应用。这是十分重要的，对于初学者来说，保持最小规模可以减少很多不必要的干扰，更聚焦于逻辑设计本身。Express框架虽然小巧，但其性能并不差，用来构建一般的商业应用也完全没有问题。

使用Express搭建一个Web应用非常简单，只需要创建一个Express应用实例，然后设置要监听的端口号即可。一个Express应用实例可以设置多个路由函数，当用户向浏览器中输入地址访问某个网页时，或者直接使用网络API来向某个接口请求数据时，就会在Express应用的路由系统中进行匹配，当匹配到合适的路由后，就会执行对应的路由方法处理逻辑，并且可以返回数据到客户端。Express应用返回给客户端的可以是普通的文本数据，也可以是HTML数据、JSON数据或其他格式的数据。我们可以根据客户端和服务端的约定来选择合适的数据格式。后面小节中我们会具体演示Express应用的创建。图1-1描述了一个Express应用的运行流程。

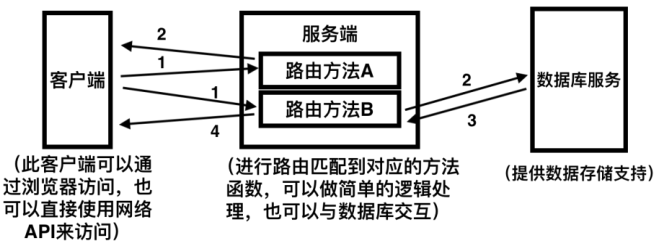

图 1-1　Express 应用运行示例

1.1.4　从 JavaScript 到 TypeScript

TypeScript是JavaScript的一种超集。所谓超集，是指TypeScript本身就包含JavaScript的所有功能，所有JavaScript的语法在TypeScript中依然适用。TypeScript是对JavaScript功能的一种增强。

在互联网时代初期，互联网应用大多非常简单，更多的是提供信息供用户阅读，可进行的用户交互并不多。此时的应用直接使用JavaScript语言来开发是非常简单和方便的，JavaScript提供的功能也绰绰有余。随着互联网技术的发展，互联网应用的规模越来越庞大，应用涉及的页面逐渐增多，用户交互逐渐复杂，并且JavaScript开发的软件的应用场景也越来越广泛。这时JavaScript本身的灵活性反倒为开发者带来困扰，过度的灵活导致程序中的错误不易排查、模块化能力弱、重构困难等问题。为了解决JavaScript的这些问题，发明了TypeScript，它更适用于大型项目的开发。

关于TypeScript的用法，我们会在后面的项目开发中穿插介绍。这里我们简单对比一下TypeScript与JavaScript的主要区别。

（1）TypeScript提供了更多面向对象编程的特性。

JavaScript是面向对象的语言，它的面向对象是基于原型实现的，本身并没有“类”和“接

口”这类概念。总体来说，JavaScript的面向对象功能较弱，项目越大，其劣势就越明显。TypeScript中增加了类、模块、接口等功能，增强了JavaScript的面向对象能力。

（2）TypeScript为JavaScript提供了静态类型功能。

JavaScript中的变量没有明确的类型，TypeScript则要求变量有明确的类型。静态类型对于大型项目来说非常重要，很多编码错误在编译时即可通过静态检查发现。同时，TypeScript还提供了泛型、枚举、类型推论等高级功能。

（3）函数相关功能的增强。

TypeScript为函数提供了默认参数值，引入了装饰器、迭代器和生成器的语法特性。这些特性增强了编程语言的可用性，用更少的代码可以实现更复杂的功能。

对于TypeScript，读者可能还有一点疑惑，大部分浏览器的引擎都只支持JavaScript的语法，那么如何保证TypeScript编写的项目可以在所有主流浏览器上运行呢？这就需要通过编译器进行编译。编译器的作用是将TypeScript代码编译成通用的JavaScript代码，保证在各种环境下的兼容性。当然，在编译的过程中，编译器也会对语法规则进行检查，从而在一定程度上保证了代码的质量。

最后，对于为什么要使用TypeScript而不是JavaScript，这其实是分场景而言的，对于大型项目来说，无论在开发效率、可维护性还是代码质量上，TypeScript都具有明显的优势，是前端开发语言的未来与方向。

1.2　脚手架工具的应用

脚手架本身是建筑领域中的一个概念，可以理解为是为了保证施工过程的顺利进行而搭建的工作平台。在编程领域，项目的开发也需要一些软件开发平台的协助，对应的也会有一些脚手架工具供开发者使用。

Vue.js本身是一个渐进式的前端Web开发框架，它不仅允许我们只在项目的部分页面中使用Vue.js进行开发，也允许我们仅使用Vue.js中的部分功能来进行项目开发。但是，如果目标是完成一个风格统一的、可扩展性强的、现代化的Web单页面应用，那么使用Vue.js提供的一整套完整的功能进行开发是非常适合的，并且通过工具链的配合，我们可以创建集开发、编译、调试、发布为一体的开发流程，还可以在开发过程中使用TypeScript、Sass等更高级的编程语言。

同样地，对于后端框架Express来说，它也有相关的脚手架工具。Express的应用生成器工具可以帮助我们快速搭建工程，也有相应的开源工具对TypeScript进行支持，可以减少开发者很多配置相关的工作。

1.2.1　安装 Node.js 环境

无论是Vue还是Express，都是JavaScript相关技术栈下、Node.js平台上的框架。因此，我们需要先安装Node.js环境。Node.js的安装非常简单，可以通过如下地址打开Node.js的官网：

https://nodejs.org

官网首页如图1-2所示。

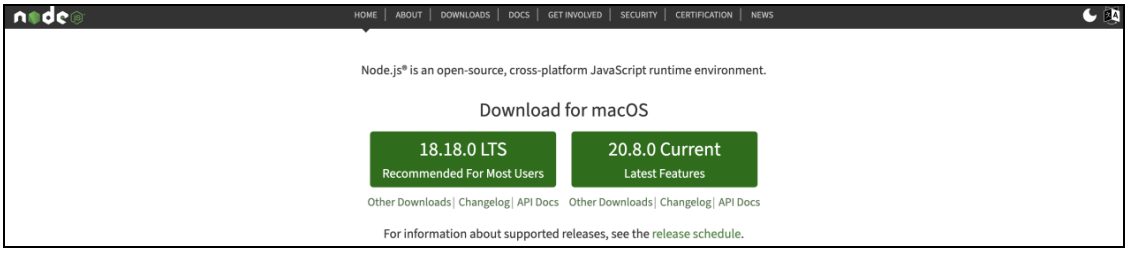

图 1-2　Node.js 官网首页

可以看到，官网的首页非常简单，它会根据使用者的操作系统自动推荐要下载的软件。目前Node.js的标准稳定版本是18.18.0，最新的版本是20.8.0。其中稳定版本是大多数用户目前所使用的版本，我们选择这个版本进行下载即可。

无论是Mac OS、Windows还是Linux操作系统，Node.js的安装方法都类似，先下载软件包，然后像安装普通软件一样进行安装即可。安装完成后，可以尝试在终端输入如下指令来检查所安装的Node.js的版本：

```
node -v
```

如果终端正常输出了版本号信息，则表明Node.js已经安装成功。

温馨提示

有时候不同的项目所要求的 Node.js 的版本并不相同，因此能够方便地对当前所使用的 Node.js 环境的版本进行切换是十分必要的。nvm 是一款 Node.js 版本管理工具，支持进行 Node.js 版本的下载、查看、切换等常用操作。如果读者有兴趣，可以尝试使用 nvm 工具来安装 Node.js。

1.2.2　使用 Vue.js 脚手架工具 Vite

理论上来说，我们可以不使用任何额外的开发脚手架工具，而是直接引入Vue.js的核心代码库来使用Vue.js。但是，使用脚手架工具可以大大减少复杂项目的配置和编译处理流程。Vue.js官网推荐的脚手架工具有Vite和Vue CLI两种。其中Vue CLI曾经是构建大型Vue.js项目不可或缺的工具，它提供带有热重载模块的开发服务器、插件管理系统，甚至用户管理界面等非常多的高级功能。相比Vue CLI，Vite是一款更新的Vue.js脚手架工具，其设计理念也与Vue.js框架本身更加切合。Vite本身的目标是作为一款轻量级的、速度极快的构建工具。Vite工具的作者同时也是Vue.js框架的作者。

在本书中，我们将优先采用Vite作为Vue.js项目开发的首选工具。当然如果有读者更喜欢使用Vue CLI，也没有任何问题。

Vite本身并不像Vue CLI那样功能大而全，它只专注于提供基本构建功能和开发服务器。因此，Vite更加小巧迅捷，其开发服务器的响应速度会比Vue CLI的开发服务器的响应速度快10倍左

右。这对开发者来说太重要了，开发服务器的响应速度会直接影响到开发者的编程体验和开发效率。对于非常大型的项目来说，可能会有成千上万个JavaScript模块，这时构建效率的速度差异就会非常明显。

创建Vite工程非常简单，直接使用npm工具即可（安装完成Node.js后，npm也会自动安装）。执行如下指令：

```
npm create vite@latest
```

之后一步一步地选择一些配置项：首先输入工程名和包名，例如我们可以取名为“1_HelloWorld”；然后选择要使用的框架，Vite不止支持构建Vue.js项目，也支持构建基于React.js等框架的项目，这里我们选择Vue.js即可，使用的语言选择TypeScript。

项目创建完成后，生成的工程目录结构如图1-3所示。

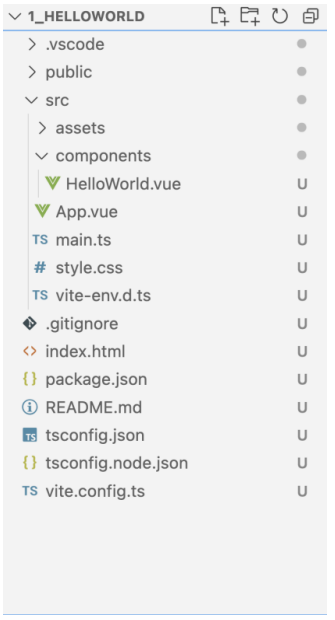

图 1-3　Vite 创建的 Vue 工程目录结构

我们主要关注package.json文件，此文件是Node.js项目的核心文件：

```
{
  "name": "1-helloworld",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vue-tsc && vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "vue": "^3.3.4"
  },
  "devDependencies": {
    "@vitejs/plugin-vue": "^4.2.3",
    "typescript": "^5.0.2",
    "vite": "^4.4.5",
    "vue-tsc": "^1.8.5"
  }
}
```

其中除了配置一些基础信息之外，scripts中配置了Vite工程开发、编译和预览所使用的指令；dependencies中配置了项目生产环境所依赖的库；devDependencies配置了项目开发环境所依赖的库。可以看到，当前项目依赖的Vue版本为3.3.4。

通过观察使用Vite创建出的工程目录，可以发现其中主要包含3个文件夹和7个独立文件。我

们先来看这7个独立文件（其中，以“.”开头的文件都是隐藏文件）：

（1）.gitignore：用来配置Git版本管理工具需要忽略的文件或文件夹，在创建工程时，默认会忽略一些依赖、编译产物、log日志等文件，不需要修改。

（2）index.html：是整个项目的入口文件，其中定义了承载Vue.js应用的HTML元素。

（3）package.json：该文件相对比较重要，其中存储的是一个JSON对象结构的数据，用来配置当前的项目名称、版本号、脚本命令以及模块依赖等。package.json可以用来配置依赖库版本的匹配规则。当我们向项目中添加额外的依赖时，就会被记录到这个文件中。

（4）README.md：该文件是一个MarkDown格式的文件，其中记录了项目的编译和调试方式。我们也可以将项目的介绍编写在这个文件中。

（5）tsconfig.json和tsconfig.node.json：是TypeScript的编译配置文件，可以在其中配置一些编译规则和依赖库。这两个文件的配置分别影响Vue项目的TypeScript环境和Vite本身的TypeScript环境。

（6）vite.config.js：是使用Vite创建项目时自动生成的文件，用来对项目的部署进行配置，目前无须关心。

现在，可以尝试运行此Vite项目，在工程目录下执行如下指令来安装依赖：

```
npm install
```

之后直接执行如下指令在开发模式下运行项目：

```
vite
```

模板工程的运行效果如图1-4所示。

图 1-4　Vite 工程示例

1.2.3　使用 Express 项目生成工具

Node.js本身是JavaScript的运行环境，基于Node.js平台，我们可以通过它来开发各种各样的互联网应用。Express框架的特点是非常轻量，可以快速搭建项目。对于学习Web应用开发来说，使用它非常方便。

首先需要安装一个脚手架工具来帮助创建Express项目。在终端执行如下指令来全局安装一个Node.js工具包：

```
npm install -g yo generator-express-no-stress-typescript
```

generator-express-no-stress-typescript是一款开源的适用于Express框架的脚手架工具，它会帮助我们生成一个集开发服务器、交互式文档、结构化日志等功能于一体的Express项目，并且generator-express-no-stress-typescript 是 generator-express-no-stress 开源项目的扩展，加入了

TypeScript的支持。

下面我们尝试使用此脚手架来创建一个Express下的HelloWorld项目。在终端输入如下指令创建工程：

```
    yo express-no-stress-typescript
1_HelloWorld_backend
```

其中，1_HelloWorld_backend是设置的工程名称。之后在终端会提示一些选项供开发者设置，比如需要填写项目的描述、版本以及所使用的OpenAPI版本等。创建出的工程的目录结构如图1-5所示。

图 1-5　Express 模板工程示例

关于工程中每个目录和独立文件的作用，目前无须过多关注，只需要了解后端接口的代码都编写在server目录下即可，在工程目录下执行如下指令即可运行项目：

```
    npm run dev
```

运行成功后，将自动开启开发服务器，监听本机的3000端口。可以在浏览器中输入地址http://localhost:3000/来查看，效果如图1-6所示。

图 1-6　Express 示例工程

使用generator-express-no-stress-typescript脚手架非常棒的一点在于它会自动帮助我们生成API接口文档页面，这对前后端分离的开发方式来说非常重要，以结构化的方式生成文档可以减少很多沟通合作的成本，并且方便以面向接口编程的方式来开发大型项目。文档页面如图1-7所示。

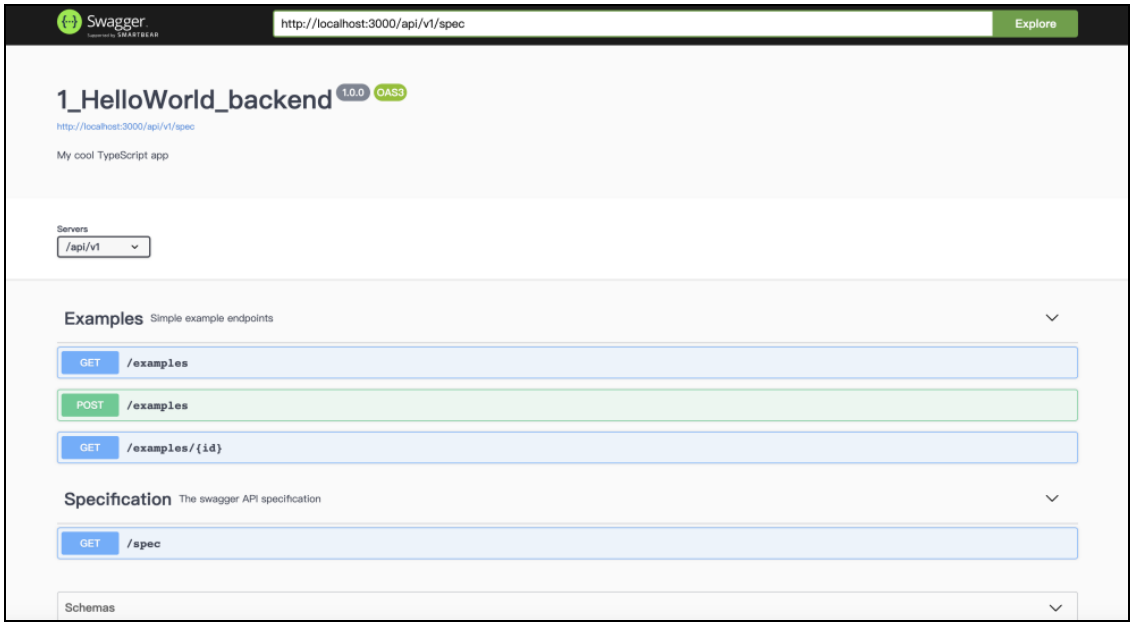

图 1-7　自动生成的 API 文档

1.2.4　使用 Visual Studio Code 编程工具

有句话说得好，叫“工欲善其事，必先利其器”。编程本身也是一种技术工作，要想高效地完成编程工作，离不开各种编程工具的支持。虽然编程的本质是代码的编写，原则上我们使用任何文本编辑器都可以进行项目开发，但是一款优秀的编程工具不仅可以极大地提高开发者的开发效率，而且可以让开发者在编程的过程中享受极致的编程体验。

我们将Visual Studio Code（也称VSCode）编程工具作为此项目的主要开发工具。VSCode是Microsoft开源的一款代码编辑器，其本身非常轻量，运行速度快；更重要的是VSCode以插件的方式来扩展功能，针对不同的编程场景可以安装不同的扩展插件。插件可以为VSCode提供代码提示、关键词高亮、索引跳转、预编译等一系列高级功能。开发者可以根据自己的需求进行灵活配置。

首先，在VSCode官方网站下载VSCode软件，官方网站地址如下：

https://code.visualstudio.com/

VSCode官网页面如图1-8所示。

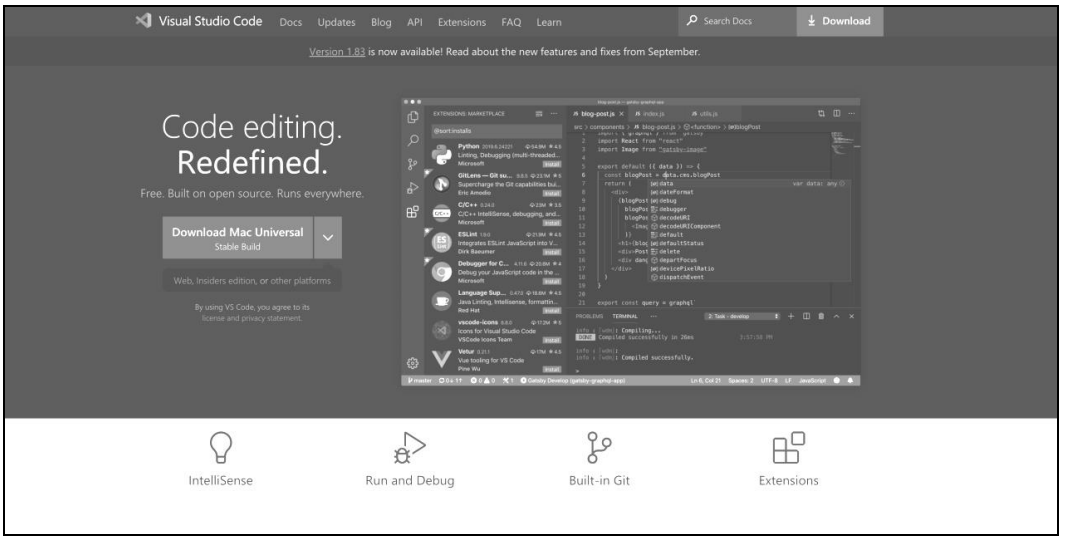

图 1-8　VSCode 官网页面

在VSCode官网上，除了有对编辑器的基本介绍外，还有一个非常显著的下载按钮，通过此入口即可直接下载该软件的稳定版本。

下载完成后，可以使用VSCode来打开前面创建的Vue.js项目工程1_HelloWorld。直接将此工程文件夹拖入VSCode编辑器即可。VSCode编辑器的整体页面布局如图1-9所示。

图1-9中标注出了常用的4大工作区；最左边为功能导航区，用来切换功能模块，包括编码模块、搜索模块、分支管理模块、运行模块和插件模块等；文件导航区会列出当前工程目录下所有的文件夹和文件，用来进行文件的切换；代码编辑区是核心的编码区域；日志与调试区也是开发中很常用的功能区，项目在编译运行时的输出信息都会在这个区域进行展示，也包括对应的调试功能。

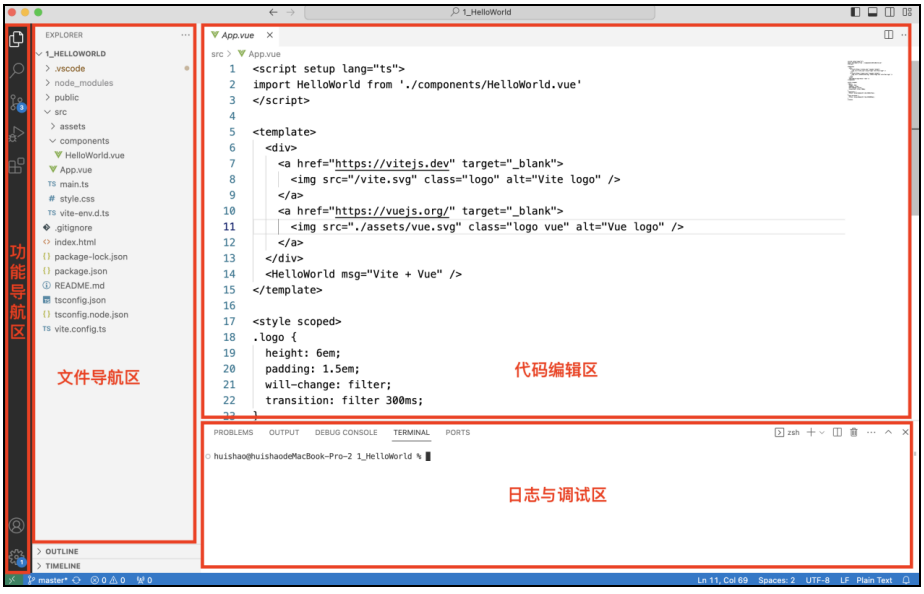

图 1-9　VSCode 页面布局

VSCode工具的强大之处在于其丰富的插件系统。可以看到，在未安装任何插件的情况下，代码编辑区的Vue.js组件代码非常难读，并且对于引入的其他组件也无法直接进行连接跳转。

我们可以先来安装一个名为Vue Language Features的插件，此插件可以为Vue.js组件代码增加不同颜色，并且为组件和CSS样式引用增加点击功能。在VSCode的插件管理模块中搜索此插件，如图1-10所示。

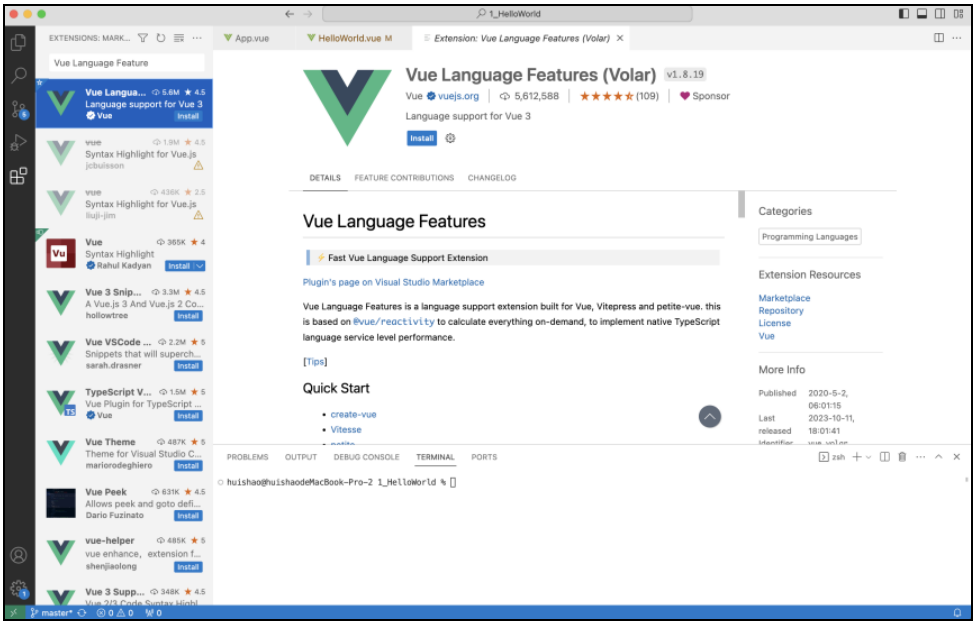

图 1-10　安装 Vue Language Features 插件

安装完成后，在VSCode中打开一个Vue.js组件文件，效果如图1-11所示，可以看到代码已经根

据语法类别的不同进行了颜色区分，并且可以方便地从引用部分跳转到定义部分。

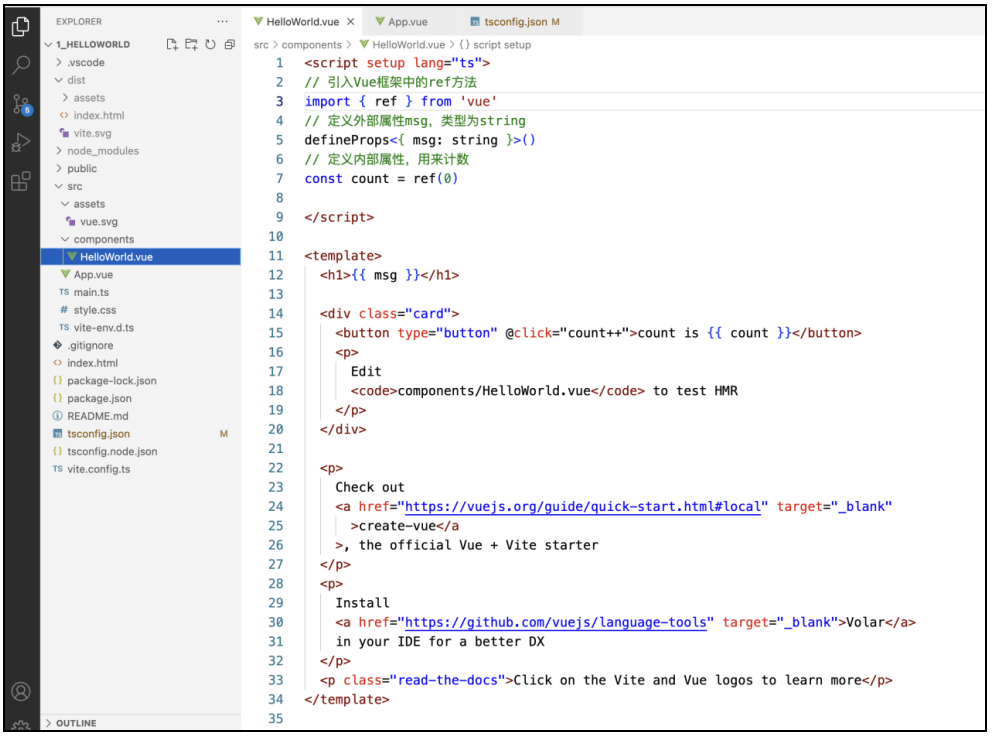

图 1-11　Vue Language Features 插件效果

温馨提示

在 Windows 系统上，按住 Win 键单击引用进行跳转，在 Mac OS 上按住 Command 键单击引用进行跳转。

1.3　HelloWorld 工程解析

本节将对前面所创建的Vue.js工程和Express工程做简单介绍。这两个模板工程的结构虽然简单，但麻雀虽小，五脏俱全。通过对HelloWorld工程的分析，我们可以更好地理解成熟项目的基本架构，并了解应该从何处入手开发应用。

1.3.1　Vue.js 工程解析

回顾我们使用Vite脚手架创建的Vue.js模板工程，其中关于工程配置的相关文件可暂不关注。在工程目录下执行npm run build指令来对项目进行编译。编译完成后，在工程的根目录下会生成一个名为dist的文件夹，其中存放的是编译后的生产环境的项目代码。编译后生成的JavaScript代码和CSS代码是经过压缩的，阅读起来非常困难，但是HTML文件并未进行压缩，dist文件夹下的

index.html文件代码如下：

【代码片段1-1　源码见目录1/1_HelloWorld/dist/index.html】

```
<!doctype html>
<html lang="en">
  <head>
    <!-- 设置文件编码格式 -->
    <meta charset="UTF-8" />
    <!-- svg资源文件引入 -->
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <!-- meta配置 -->
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <!-- 网页标题 -->
    <title>Vite + Vue + TS</title>
    <!-- 引入JS脚本 -->
    <script type="module" crossorigin
src="/assets/index-08e9f3d0.js"></script>
    <!-- 引入CSS样式 -->
    <link rel="stylesheet" href="/assets/index-b31fd3ba.css">
  </head>
  <body>
    <!-- 挂载Vue应用的标签 -->
    <div id="app"></div>
  </body>
</html>
```

生产环境下的index.html非常简单，与项目工程中的index.html相比，不同之处在于将编译和压缩处理后的JavaScript文件和CSS文件进行了引入。对于整个Vue.js项目来说，工程中的main.ts才是应用程序的真正入口，此文件代码如下：

【代码片段1-2　源码见目录1/1_HelloWorld/src/main.ts】

```
// 引入Vue框架中的createApp方法
import { createApp } from 'vue'
// 引入全局的CSS样式
import './style.css'
// 引入自定义的App组件
import App from './App.vue'
// 使用App作为根组件来创建Vue应用实例，并挂载到指定HTML元素上
createApp(App).mount('#app')
```

其中App组件被指定为应用实例的根组件。App.vue文件是一个纯粹的Vue.js组件文件，被编译后才能使用。我们之后在开发项目时，会用到非常多的自定义组件，它们都是类似的单文件组件。

Vue.js组件文件大致分为3个部分：逻辑脚本部分、模板部分和样式表部分。

- 逻辑脚本部分用来编写核心的数据和交互逻辑，如子组件的使用、变量的定义和计算、方法的定义和调用等。
- 模板部分用来定义页面的结构，其中主体部分是HTML代码，可以通过Vue.js的一些语法规则来实现不同的渲染逻辑。

- 样式表部分用来定义所需要使用的 CSS 样式，可以定义为局部的，避免污染其他组件。

App.vue模板代码如下：

【代码片段1-3　源码见目录1/1_HelloWorld/src/App.vue】

```
<!-- 逻辑脚本部分 -->
<script setup lang="ts">
// 引入HelloWorld组件
import HelloWorld from './components/HelloWorld.vue'
</script>

<!-- 模板部分 -->
<template>
  <div>
    <a href="https://vitejs.dev" target="_blank">
      <img src="/vite.svg" class="logo" alt="Vite logo" />
    </a>
    <a href="https://vuejs.org/" target="_blank">
      <img src="./assets/vue.svg" class="logo vue" alt="Vue logo" />
    </a>
  </div>
  <!-- 这里使用了HelloWorld组件 -->
  <HelloWorld msg="Vite + Vue" />
</template>

<!-- 样式表部分 -->
<style scoped>
.logo {
  height: 6em;
  padding: 1.5em;
  will-change: filter;
  transition: filter 300ms;
}
.logo:hover {
  filter: drop-shadow(0 0 2em #646cffaa);
}
.logo.vue:hover {
  filter: drop-shadow(0 0 2em #42b883aa);
}
</style>
```

示例代码中，script标签将lang属性设置为ts，表示要使用的脚本语言为TypeScript，并且使用setup关键字进行了修饰。setup关键字的作用是告诉Vue.js编译器此脚本模块采用了组合式API的语法，在其中直接使用组合式API的方式编写Vue.js组件即可。style标签使用了scoped修饰，scoped的作用是将当前style标签内定义的样式约束为组件内部样式，即只在当前组件内生效。

再来观察HelloWorld组件，这个组件被嵌入了App组件内部，我们也可以称它为子组件。调用HelloWorld组件的代码如下：

```
<HelloWorld msg="Vite + Vue" />
```

可以看到，自定义组件的使用和普通的HTML标签并没有太大区别，msg属性是HelloWorld组件定义的一个外部属性，外部属性的作用是进行组件间的数据传递。HelloWorld组件的Script部分代码如下：

【代码片段1-4　源码见目录1/1_HelloWorld/src/Components/HelloWorld.vue】

```
<script setup lang="ts">
// 引入Vue框架中的ref方法
import { ref } from 'vue'
// 定义外部属性msg，类型为string
defineProps<{ msg: string }>()
// 定义内部属性，用来计数
const count = ref(0)
</script>
```

具体的Vue.js语法这里不做过多解释，此处定义的count变量其实将会编译成一个具有响应性的组件内部属性，对count值的修改会直接影响到页面对应的展示。这也是HelloWorld项目模板的基本功能，项目运行后会在页面展示一个按钮，按钮上显示单击次数，单击按钮后，对应的单击次数也增加。

在实际的项目开发中，其实也是编写各种Vue.js组件，单个组件将逻辑聚焦于其本身的功能，之后将组件进行组合，进行数据交互和跳转管理即可。


1.3.2　Express 工程解析

相比前端的Vue.js工程而言，后端的Express工程要更加复杂一些。这是因为Express框架的功能扩展大多是以中间件的方式进行构建的，而后端工程相比前端有更复杂的日志、接口文档定义、数据解析等逻辑，所使用的中间件也比较多，整体看上去更复杂。

我们还是先从程序的入口文件开始分析。Express工程运行时，入口文件是index.ts，此文件比较简单，内容如下：

【代码片段1-5　源码见目录1/1_HelloWorld_backend/server/index.ts】

```
import './common/env'; // 这里的import的功能是直接注入环境变量
// 引入Server和routes模块
import Server from './common/server';
import routes from './routes';
// 获取配置的端口号
const port = parseInt(process.env.PORT ?? '3000');
// 开启服务
export default new Server().router(routes).listen(port);
```

上面代码中import './common/env'的作用并非引入某个工具模块，而是进行环境变量的注入，执行此行代码时，会直接执行./common/env文件中的TypeScript代码，此文件内容如下：

【源码见目录1/1_HelloWorld_backend/server/common/env.ts】

```
import dotenv from 'dotenv';  // 引入dotenv模块
dotenv.config();  // 进行环境配置
```

其中，dotenv是一个第三方的环境配置工具，调用config方法后，会直接将本地配置的环境变量注入当前应用执行的Node.js环境中去。我们可以在工程的根目录下找到一个名为.env的隐藏文件，所需要的环境变量即可配置在此文件中。.env文件内容如下：

```
APP_ID=1_HelloWorld_backend
PORT=3000
LOG_LEVEL=debug
REQUEST_LIMIT=100kb
SESSION_SECRET=mySecret
OPENAPI_SPEC=/api/v1/spec
```

可以看到，此模板工程的环境变量配置了应用的id、端口号、日志等级、请求体字节数限制等。关于环境变量，我们可以将其简单理解为全局可使用的静态变量。

再回到index.ts文件，引入的Server和routes是两个自定义的模块，现在可以简单地将Server理解为一个自定义的服务器对象类，routers为路由注册方法。

process.env是Node.js中的环境对象，前面注册的环境变量数据都会被存储到此对象中。调用Server()构造方法将实例化出一个服务器对象，之后调用router方法来进行路由的注册，调用listen方法来开启服务，监听指定的端口。Server服务器类内部略微复杂，我们稍后会做详细介绍。当listen方法执行完后，Express服务器就启动完成了。

common目录下的server.ts文件是实现服务器逻辑的核心文件。先看其导入部分，代码如下：

【源码见目录1/1_HelloWorld_backend/server/common/server.ts】

```
// 引入Node.js自带的工具模块
import path from 'path';
import http from 'http';
import os from 'os';
// 引入Express框架中的模块
import express, { Application } from 'express';
// 引入中间件模块
import bodyParser from 'body-parser';
import cookieParser from 'cookie-parser';
import errorHandler from '../api/middlewares/error.handler';
import * as OpenApiValidator from 'express-openapi-validator';
// 引入log工具
import l from './logger';
```

其中path、http和os是Node.js自带的模块，提供基础的路径处理、网络和系统调用能力。我们主要关注引入的一些中间件。

- body-parser 是一个请求解析中间件，它可以对请求体中通用的 JSON 格式的数据、URL 编码的表单格式的数据以及纯文本的数据进行解析。
- cookie-parser 是一个用来对 Cookie 进行签名和解析的中间件，在很多 Web 应用中，常使用 Cookie 来存储用户信息。
- express-openapi-validator 是对接口请求数据进行 OpenApi 规则验证的中间件。
- errorHandler 实际上是自定义的一个中间件，其用来处理请求错误时的返回结果。
- logger 只是对 pino 模块的封装，用来进行日志的记录。

简单理解了这些引入的模块的作用后，继续阅读server.ts的代码，可以看到，其中定义了一个全局的对象app：

【源码见目录1/1_HelloWorld_backend/server/common/server.ts】

```
// 创建一个全局应用实例
const app = express();
```

此行代码创建了一个express应用实例，在启动Web服务时会用到。除此之外，server.ts中主要定义了ExpressServer类，此类的实现如下：

【代码片段1-6　源码见目录1/1_HelloWorld_backend/server/common/server.ts】

```
// 定义Express服务器类
export default class ExpressServer {
  private routes: (app: Application) => void;
  constructor() {
    // 定义根路径
    const root = path.normalize(__dirname + '/../..');
    // 挂载bodyParser中间件：控制JSON数据体大小
    app.use(bodyParser.json({ limit: process.env.REQUEST_LIMIT || '100kb' }));
    // 挂载bodyParser中间件：控制表单数据大小和表单的解析器类型
    app.use(
      bodyParser.urlencoded({
        extended: true,
        limit: process.env.REQUEST_LIMIT || '100kb',
      })
    );
    // 挂载bodyParser中间件：控制普通文本数据大小
    app.use(bodyParser.text({ limit: process.env.REQUEST_LIMIT || '100kb' }));
    // 挂载cookieParser中间件
    app.use(cookieParser(process.env.SESSION_SECRET));
    // 挂载静态资源路径中间件
    app.use(express.static(`${root}/public`));
    // 定义API文档描述文件的路径
    const apiSpec = path.join(__dirname, 'api.yml');
    // 是否开启响应验证
    const validateResponses = !!(
      process.env.OPENAPI_ENABLE_RESPONSE_VALIDATION &&
      process.env.OPENAPI_ENABLE_RESPONSE_VALIDATION.toLowerCase() ===
'true'
    );
    // 挂载API文档静态文件路径
    app.use(process.env.OPENAPI_SPEC || '/spec', express.static(apiSpec));
    // 挂载OpenApiValidator中间件
    app.use(
      OpenApiValidator.middleware({
        apiSpec,
        validateResponses,
        ignorePaths: /.*\/spec(\/|$)/,
      })
```

```
      );
    }
    // 定义router方法，用来注册路由，返回当前服务器对象本身
    router(routes: (app: Application) => void): ExpressServer {
      // 通过外部的方法来注册路由
      routes(app);
      // 注册异常处理中间件
      app.use(errorHandler);
      return this;
    }
    // 定义listen方法，开启服务器
    listen(port: number): Application {
      // 定义回调函数
      const welcome = (p: number) => (): void =>
        l.info(
          `up and running in ${
            process.env.NODE_ENV || 'development'
          } @: ${os.hostname()} on port: ${p}}`
        );
      // 创建HTTP服务，监听端口
      http.createServer(app).listen(port, welcome(port));
      // 返回应用实例
      return app;
    }
  }
```

上面代码中有比较详尽的注释，整体来说，当我们在构造ExpressServer实例对象时，会在构造方法中完成对各种中间件的注册，首先注册请求体解析相关的中间件，然后是Cookie签名的中间件，接着是静态文件处理的中间件，最后是OpenAPI校验的中间件。router方法实际上提供了一个外部路由注册的接口，允许使用外部的方法来注册路由中间件，路由注册完成后进行了异常处理中间件的注册。

需要注意，在Express中，中间件的注册顺序是非常重要的。简单理解，中间件的作用是对业务流程的中间环节进行处理。以客户端→服务端→客户端这样的数据传输业务为例，Express允许开发者在接收到客户端的数据后执行一系列的中间逻辑，之后将响应的数据返回给客户端。中间件的处理顺序与注册顺序一致。错误处理类型的中间件比较特殊，当其他中间件的执行有异常抛出时，会执行错误处理中间件的逻辑。示例工程中使用的异常处理函数定义在middlewares目录下的error.handler.ts文件中，代码如下：

【代码片段1-7　源码见目录1/1_HelloWorld_backend/server/api/middlewares/error.handler.ts】

```
// 模块引入
import { Request, Response, NextFunction } from 'express';
// 定义异常处理中间件
export default function errorHandler(
  err: any,
  _req: Request,
  res: Response,
  _next: NextFunction
```

```
  ): void {
    // 直接返回给客户端错误信息
    const errors = err.errors || [{ message: err.message }];
    res.status(err.status || 500).json({ errors });
  }
```

ExpressServer类中除了构造方法外，还定义了两个方法：route和listen。这两个方法比较好理解，route提供了外部注册路由的接口，listen则负责启动服务器，并将当前应用实例返回。前面在介绍index.ts文件的时候，我们说过路由注册的方法是定义在server目录下的routes.ts文件中的，下面我们再来分析这个文件的作用。

server目录下的routes.ts文件代码如下：

【代码片段1-8　源码见目录1/1_HelloWorld_backend/server/routes.ts】

```
import { Application } from 'express'; // 引入模块
import examplesRouter from './api/controllers/examples/router'; // 自定义路由
export default function routes(app: Application): void {
  // 使用use方法进行路由中间件的挂载
  app.use('/api/v1/examples', examplesRouter);
}
```

在Express框架中挂载路由类型的中间件时，use方法的第1个参数为要绑定了路由路径，第2个参数设置处理此路由路径对应的路由对象。上面代码中的examplesRouter才真正定义了处理路由的业务逻辑，此route.ts文件中的代码如下：

【代码片段1-9　源码见目录1/1_HelloWorld_backend/server/api/controllers/examples/route.ts】

```
import express from 'express';              // 引入模块
import controller from './controller';      // 引入处理控制器
// 定义一个路由中间件并进行导出
export default express
  .Router()                                 // 创建路由中间件
  .post('/', controller.create)             // 设置POST请求的处理方法
  .get('/', controller.all)                 // 设置GET请求的处理方法
  .get('/:id', controller.byId);            // 设置GET带参数请求的处理方法
```

上面代码中创建了一个路由中间件，并指定了对客户端发起的POST请求、不带参数的GET请求和带参数的GET请求的处理方法。在进行路径映射时，可以直接使用Express约定的语法进行参数映射，例如上面定义的/:id路由，使用如下URL进行请求时即可命中：

```
http://localhost:3000/api/v1/examples/1
```

我们可以这样理解，server目录下的routes.ts文件的主要作用是在Express应用中挂载路由模块，当应用比较复杂时，可以支持挂载多个路由模块。每个路由模块下可能会有多个API服务，例如示例工程中的examples模块，其下定义了3个API接口。这些接口是在examples目录下的router.ts文件中具体配置的，该文件负责映射当前模块下的API接口路径与对应的处理方法。

再来看一下controllers.ts文件，在Web服务开发中，常常将某个模块的处理类以Controller命名。在Controller类中定义方法来对命中的路由请求进行处理，controllers.ts文件代码如下：

【代码片段1-10　源码见目录1/1_HelloWorld_backend/server/api/controllers/examples/controller.ts】

```
    import ExamplesService from '../../services/examples.service'; // 引入service
模块
    import { Request, Response } from 'express';
    // 控制器类的具体定义
    export class Controller {
      // 对无参数get请求的处理方法
      all(_: Request, res: Response): void {
        // 调用Service来获取数据并返回
        ExamplesService.all().then((r) => res.json(r));
      }
      // 对有id参数的get请求的处理方法
      byId(req: Request, res: Response): void {
        // 调用Service来获取数据并返回
        const id = Number.parseInt(req.params['id']);
        ExamplesService.byId(id).then((r) => {
          if (r) res.json(r);
          else res.status(404).end();
        });
      }
      // 对post请求的处理方法
      create(req: Request, res: Response): void {
        // 调用Service来创建新的数据
        ExamplesService.create(req.body.name).then((r) =>
          res.status(201).location(`/api/v1/examples/${r.id}`).json(r)
        );
      }
    }
    export default new Controller();
```

其中，all方法会通过Service类来获取所有example数据并返回，byId方法查询指定id的数据并返回，create方法会将客户端传递过来的id参数拼接成路径数据存储到内存。ExamplesService是服务类，在整个后端项目架构中，服务类的作用是对数据进行操作，如读写数据、组装数据等。

ExamplesService类的实现比较简单，这里就不再赘述。在模板工程中，ExamplesService类只对内存中的数据进行操作，然后在实际应用中，它更多的是对数据库进行读写操作。

最后，解释一下API文档的生成逻辑，OpenAPI主要是通过YML文件来进行文档的定义。在YML文件中会定义支持的接口路径、请求参数和返回数据的数据结构。项目运行起来后，对前端传递参数的验证也是通过此文档的定义来约束的。另外，前端看到的文档页面其实是一个静态页面，我们在配置Express静态资源中间件的时候配置了工程的public目录，这个目录中的index.html就是前端文档的静态页面入口，文档页面会读取YML文件来进行内容的渲染。后面我们在开发API接口服务时，要做的第一件事情就是定义接口文档。

通过本节的介绍，相信读者对Express工程的结构有了较为清楚的认识，准备在之后的实操章节大展身手吧。

1.4　小结与上机练习

本章是准备章节，首先，介绍了一个完整的电商项目需要包含的功能模块，也介绍了完成这样一个完整项目所需要的技术栈。其次，在编程语言方面，我们首选TypeScript，TypeScript的诸多优势也非常适合大型项目的开发。Vue.js和Express是我们将选用的两个开发框架，电商项目的用户端和后台管理端将使用Vue.js来开发，服务后端将采用Express来开发，我们也对Vue.js和Express这两个框架进行了简单的介绍，并使用一些编辑器和脚手架工具来创建了基础的HelloWorld工程。最后，对前端和后端的HelloWorld工程做了拆解介绍，帮助读者更好地理解项目开发的流程。

现在，我们的准备工作已经完成，相信读者已经迫不及待地想要上手开发项目了。在开始前，请再回顾一下本章所介绍的核心内容，下面的问题或许可以帮助读者回忆。

思考1：Vue.js框架有怎样的特点？

提示：可以从渐进式、小巧、迅捷等方面进行思考。

思考2：使用单文件组件有什么优势，需要编译吗？

提示：单文件组件是以vue为后缀的文件，需要编译成JavaScript代码才能最终使用。单文件组件有很多优势，例如可以更好地解耦逻辑、支持局部的CSS样式等。

思考3：一个Express应用的架构是怎样的？

提示：Express应用通过中间件来构建逻辑管道，中间件的注册是有顺序的，我们可以将中间件分为逻辑处理中间件、路由中间件和异常捕获中间件等。

思考4：开发Vue.js应用，我们通常可以使用什么脚手架？

提示：Vue CLI和Vite是两个流行的Vue脚手架工具，Vue CLI功能全，除了提供流程化的编译链工具外，还提供了插件管理、可视化的Web管理工具等。而Vite最大的特点是快——编译快，动态更新也快，更适合开发大型项目。

练习：请按以下步骤构建Vue.js+Express开发环境。

首先，确保已经安装了Node.js（版本12.0.0以上）。然后，可以使用npm（Node包管理器）来安装vite。

在命令行中运行以下命令：

```
// 新建Vite工程
npm create vite@latest
```

接着，根据终端的提示进行参数配置，这将创建一个包含Vue 3和Vite的新的前端项目。

对于后端，可使用Express框架。首先，需要在项目根目录下创建一个新的文件夹，例如“server”，然后在该文件夹下创建一个新的Node.js项目：

```
cd server
npm init -y
```

接着，安装Express：

```
npm install express
```

现在，可以在server/index.js文件中编写服务器代码，例如：

```
const express = require('express')
const app = express()
const port = 3000
app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Server listening at http://localhost:${port}`)
})
```

最后，可以通过运行node server/index.js来启动服务器。