快速上手: 如何用 Vite 从零搭建前端项目?

发布于 2022-05-09

通过前面的学习，我们已经知道了前端构建工具的意义，也明确了 Vite 相比于传统构建工具 Webpack 的优势。相信对于为什么要学习和使用 Vite 这个问题，你已经有了自己的答案。

回到实际的应用场景当中，我们应该如何使用 Vite 来搭建前端工程项目呢？这一节，我将和你一起近距离接触 Vite，学完本节你不仅能学会前端开发环境的搭建，更重要的是，你能上手使用 Vite 来初始化一个脚手架项目，并理解这个项目究竟是如何运行起来的。

环境搭建

首先需要的是代码编辑器和浏览器，我推荐安装 `VSCode` 和 `Chrome` 浏览器。

其次是安装 Node.js，如果你的系统中还没有安装 `Node.js` ，可以进入 Nodejs 官网下载相应的安装包进行手动安装；如果已经安装了 `Node.js` ，你可以使用这个命令检查一下 Node.js 版本:

```
node -v
```

推荐 `12.0.0` 及以上版本，如果低于这个版本，推荐使用 `nvm` 工具切换 Nodejs 版本。

安装完 Nodejs 之后，包管理器 `npm` 也会被自动安装，你可以执行下面的命令来验证:

```
npm -v
```

当然，在现代的前端项目中，我非常不推荐使用 npm 作为项目的包管理器，甚至也不再推荐 `yarn` (`npm` 的替代方案)，因为两者都存在比较严重的性能和安全问题，而这些问题

在 pnpm 中得到了很好的解决，更多细节可以参考我的这篇博客: 关于现代包管理器的深度思考——为什么现在我更推荐 pnpm 而不是 npm/yarn? 。

因此，包管理器方面我推荐使用 pnpm，安装方式非常简单，输入如下命令即可:

```
npm i -g pnpm
```

由于默认的镜像源在国外，包下载速度和稳定性都不太好，因此我建议你换成国内的镜像源，这样 `pnpm install` 命令的体验会好很多，命令如下:

```
pnpm config set registry https://registry.npmmirror.com/
```

项目初始化

在搭建了基本的开发环境之后，我们进入到 项目初始化 阶段。你可以在终端命令行中输入如下的命令:

```
pnpm create vite
```

在执行完这个命令后，pnpm 首先会自动下载 `create-vite` 这个第三方包，然后执行这个包中的项目初始化逻辑。因此，你很快就可以看到这样的交互界面:

```
➜  ~ pnpm create vite
Packages: +6
++++++
Packages are hard linked from the content-addressable store to the virtual store.
  Content-addressable store is at: /Users/yangxingyuan/.pnpm-store/v3
  Virtual store is at:             node_modules/.pnpm

/private/var/folders/hb/wtdqvf3n7fn_2l76fv_fwr340000gn/T/dlx-46431/5:
+ create-vite 2.7.2

Progress: resolved 6, reused 6, downloaded 0, added 6, done
? Project name: › vite-project
```


后续的交互流程梳理如下:

- • 输入项目名称;
- • 选择前端框架;
- • 选择开发语言。

首先是输入项目名称，这里你可以输入 `vite-project` ，然后按下回车，进入 选择前端框架 的部分:

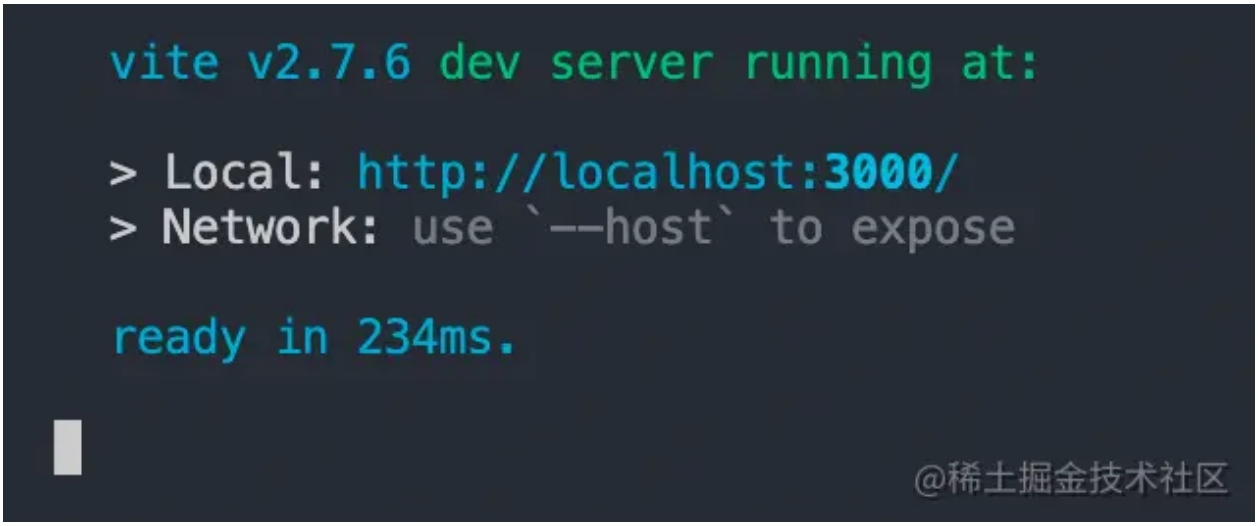
```
✓ Project name: vite-project
? Select a framework: › - Use arrow-keys. Return to submit.
    vanilla // 无前端框架
    vue     // 基于 Vue
  > react   // 基于 React
    preact  // 基于 Preact（一款精简版的类 React 框架）
    lit     // 基于 lit（一款 Web Components 框架）
    svelte  // 基于 Svelte
```

Vite 内置了以上不同前端框架的脚手架模板，这里我们以其中的 `react` 框架为例来讲解，选择 `react` 并按回车，紧接着选择 `react-ts` 完成命令交互。

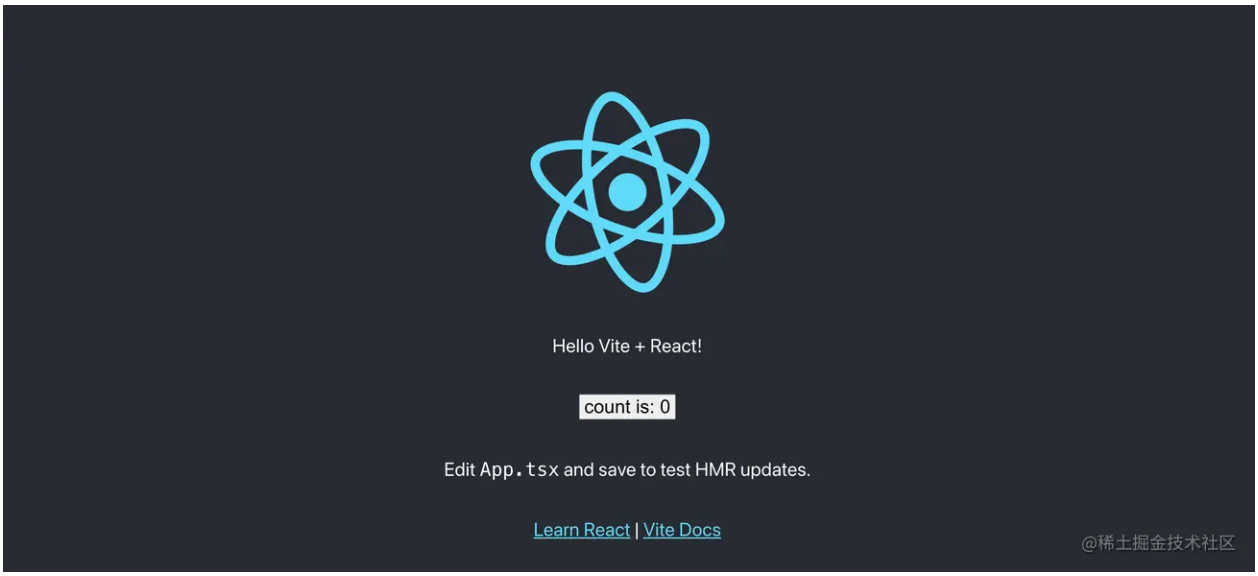
好，现在脚手架的模板已经生成完毕。你可以执行如下命令在本地启动项目:

```
// 进入项目目录
cd vite-project
// 安装依赖
pnpm install
// 启动项目
pnpm run dev
```

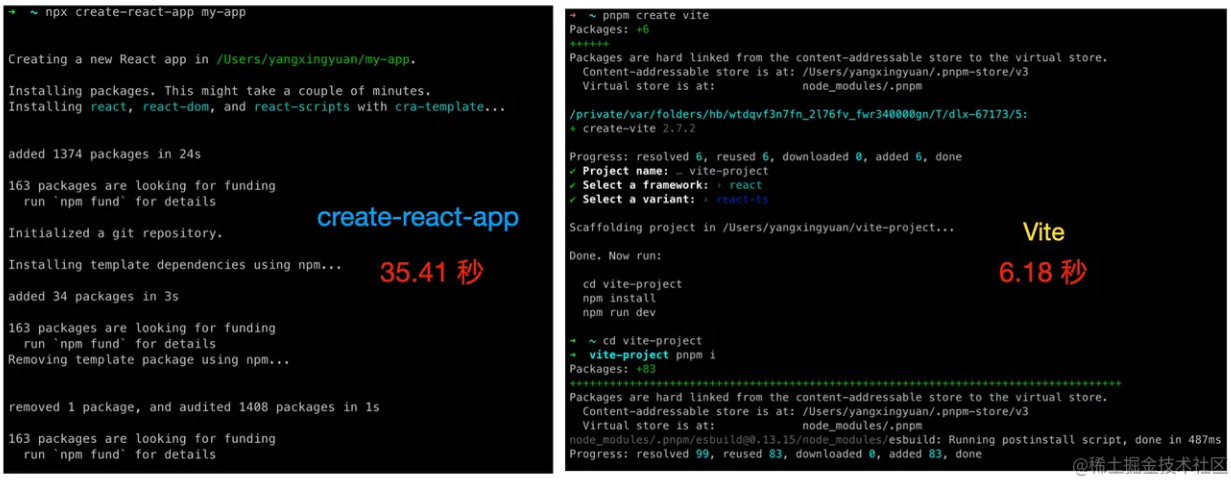
执行 `pnpm run dev` 之后你可以看到如下界面，表示项目已经成功启动啦。



紧接着，我们立马去浏览器中打开 `http://localhost:3000` 页面，你可以看到：



至此，我们成功搭建起了一个 React 前端项目。怎么样？利用 Vite 来初始化一个前端项目是不是非常简单？经过初步尝试，Vite 给人的第一感觉就是简洁、轻量、快速。我曾经拿 react 官方基于 Webpack 的脚手架 `create-react-app` ，也就是大家常说的 `cra` 来测试过，从项目初始化到依赖安装所花的时间与 Vite 对比如下：



Vite 已经比 cra 快了接近 6 倍，并且一开始就甩了 cra 一大截，显而易见地提升了初始化速度和开发体验。

项目入口加载

言归正传，我们继续学习 Vite 初始化后的项目。项目的目录结构如下：

```
.
├── index.html
├── package.json
```

```
├── pnpm-lock.yaml
├── src
│   ├── App.css
│   ├── App.tsx
│   ├── favicon.svg
│   ├── index.css
│   ├── logo.svg
│   ├── main.tsx
│   └── vite-env.d.ts
├── tsconfig.json
└── vite.config.ts
```

值得注意的是，在项目根目录中有一个 `index.html` 文件，这个文件十分关键，因为 Vite 默认会把项目根目录下的 `index.html` 作为入口文件。也就是说，当你访问 `http://localhost:3000` 的时候，Vite 的 Dev Server 会自动返回这个 HTML 文件的内容。我们来看看这个 HTML 究竟写了什么:

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/src/favicon.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite App</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.tsx"></script>
  </body>
</html>
```

可以看到这个 HTML 文件的内容非常简洁，在 `body` 标签中除了 id 为 root 的根节点之外，还包含了一个声明了 `type="module"` 的 `script` 标签:

```html
<script type="module" src="/src/main.tsx"></script>
```

由于现代浏览器原生支持了 ES 模块规范，因此原生的 ES 语法也可以直接放到浏览器中执行，只需要在 script 标签中声明 `type="module"` 即可。比如上面的 script 标签就声明了 type="module"，同时 src 指向了 `/src/main.tsx` 文件，此时相当于请求了 `http://localhost:3000/src/main.tsx` 这个资源，Vite 的 Dev Server 此时会接受到这个请求，然后读取对应的文件内容，进行一定的中间处理，最后将处理的结果返回给浏览器。

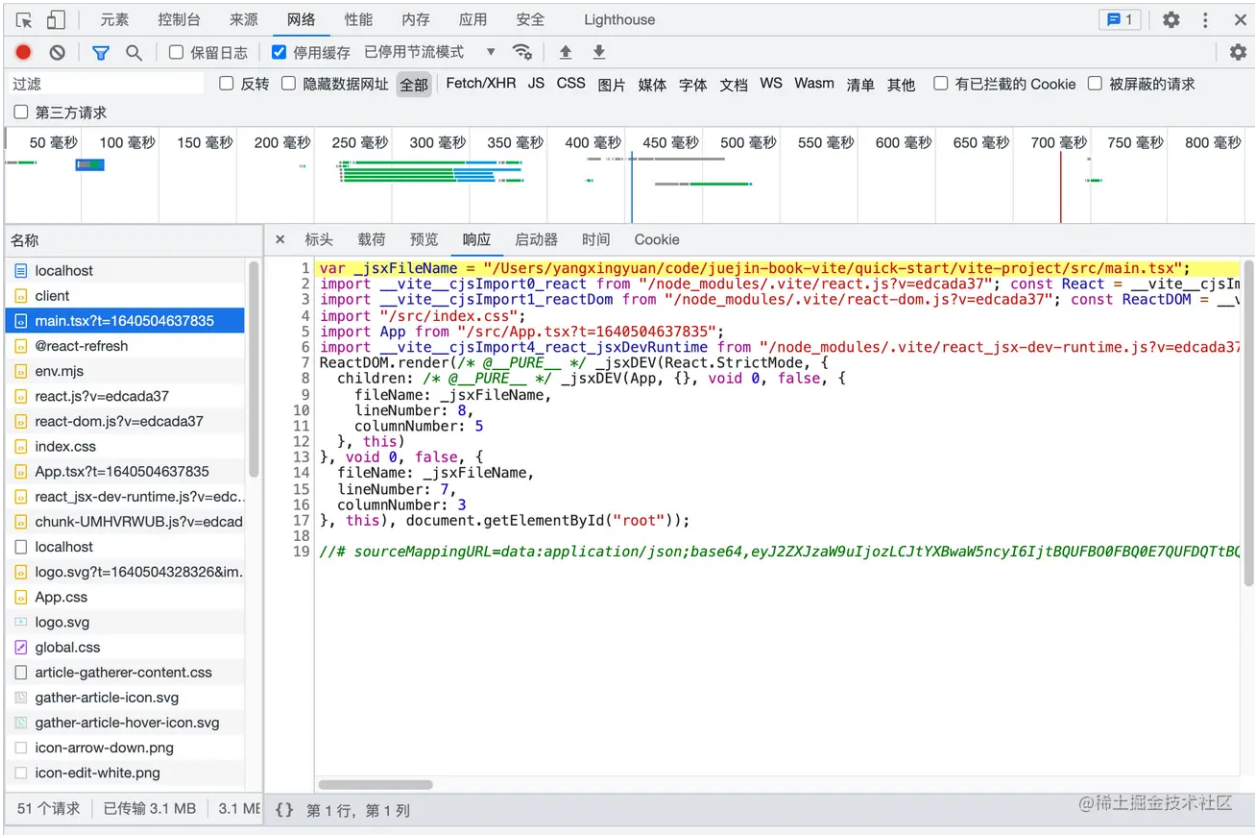
我们可以来看看 `main.tsx` 的内容:

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
)
```

到这里可能你会诧异: 浏览器并不识别 tsx 语法，也无法直接 import css 文件，上面这段代码究竟是如何被浏览器正常执行的呢?

这就归功了 Vite Dev Server 所做的"中间处理"了，也就是说，在读取到 `main.tsx` 文件的内容之后，Vite 会对文件的内容进行编译，大家可以从 Chrome 的网络调试面板看到编译后的结果:



当然，大家不用纠结每句代码的含义，因为这涉及 Vite 内部的编译流程，我们会在后面的章节深入分析。这里你只需要知道，Vite 会将项目的源代码编译成浏览器可以识别的代码，与此同时，一个 import 语句即代表了一个 HTTP 请求，如下面两个 import 语句:

```
import "/src/index.css";
import App from "/src/App.tsx";
```

需要注意的是，在 Vite 项目中，一个 `import 语句即代表一个 HTTP 请求` 。上述两个语句则分别代表了两个不同的请求，Vite Dev Server 会读取本地文件，返回浏览器可以解析的代码。当浏览器解析到新的 import 语句，又会发出新的请求，以此类推，直到所有的资源都加载完成。

现在，你应该知道了 Vite 所倡导的 `no-bundle` 理念的真正含义: **利用浏览器原生 ES 模块的支持，实现开发阶段的 Dev Server，进行模块的按需加载**，而不是**先整体打包再进行加载**。相比 Webpack 这种必须打包再加载的传统构建模式，Vite 在开发阶段省略了繁琐且耗时的打包过程，这也是它为什么快的一个重要原因。

初识配置文件

在使用 Vite 的过程，我们需要对 Vite 做一些配置，以满足日常开发的需要。你可以通过两种方式来对 Vite 进行配置，一是通过命令行参数，如 `vite --port=8888` ，二是通过配置文件，一般情况下，大多数的配置都通过配置文件的方式来声明。

Vite 当中支持多种配置文件类型，包括 `.js` 、 `.ts` 、 `.mjs` 三种后缀的文件，实际项目中一般使用 `vite.config.ts` 作为配置文件，以脚手架项目中的配置为例，具体的配置代码如下:

```
// vite.config.ts
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()]
})
```

可以看到配置文件中默认在 `plugins` 数组中配置了官方的 react 插件，来提供 React 项目编译和热更新的功能。

接下来，我们可以基于这个文件完成更加丰富的配置。之前我就遇到过这样一个需求: 页面的入口文件 `index.html` 并不在项目根目录下，而需要放到 `src` 目录下，如何在访问 `localhost:3000` 的时候让 Vite 自动返回 src 目录下的 `index.html` 呢？我们可以通过 `root` 参数配置项目根目录的位置:

```
// vite.config.ts
import { defineConfig } from 'vite'
// 引入 path 包注意两点:
// 1. 为避免类型报错，你需要通过 `pnpm i @types/node -D` 安装类型
// 2. tsconfig.node.json 中设置 `allowSyntheticDefaultImports: true`，以允许下面的 default 导入
import path from 'path'
import react from '@vitejs/plugin-react'

export default defineConfig({
  // 手动指定项目根目录位置
  root: path.join(__dirname, 'src')
  plugins: [react()]
})
```

当手动指定 `root` 参数之后，Vite 会自动从这个路径下寻找 `index.html` 文件，也就是说当我直接访问 `localhost:3000` 的时候，Vite 从 `src` 目录下读取入口文件，这样就成功实现了刚才的需求。

当然，这只是让你体验了一个简单的配置案例，在 Vite 中还有非常多的配置，由于篇幅所限，本文就不再逐个进行演示了，对于一些经常使用或者比较难理解的配置，后面的文章中会给大家一一介绍。

生产环境构建

有人说 `Vite` 因为其不打包的特性而不能上生产环境，其实这种观点是相当有误的。在开发阶段 Vite 通过 Dev Server 实现了不打包的特性，而在生产环境中，Vite 依然会基于 Rollup 进行打包，并采取一系列的打包优化手段。从脚手架项目的 `package.json` 中就可见一斑：

```
"scripts": {
  // 开发阶段启动 Vite Dev Server
  "dev": "vite",
  // 生产环境打包
  "build": "tsc && vite build",
  // 生产环境打包完预览产物
  "preview": "vite preview"
},
```

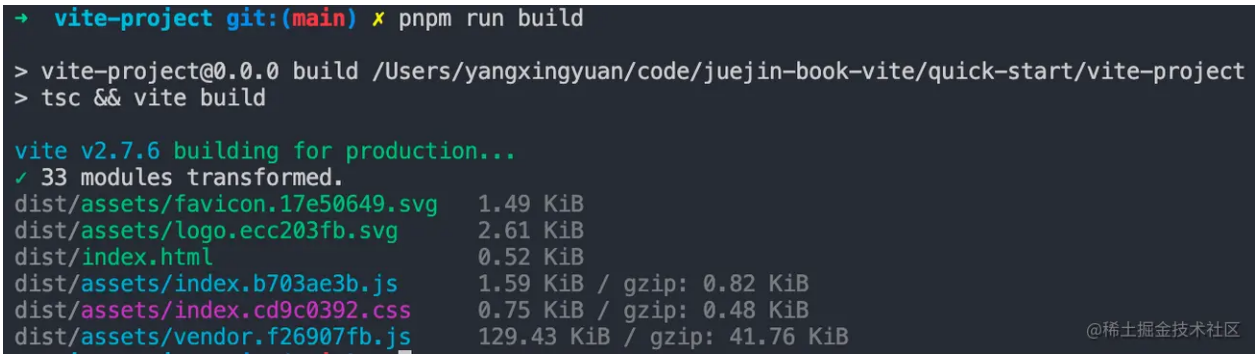
相信你已经注意到其中的 `build` 命令了，没错，这个命令就是 Vite 专门用来进行生产环境打包的。但可能你会有点疑惑，为什么在 `vite build` 命令执行之前要先执行 `tsc` 呢？

`tsc` 作为 TypeScript 的官方编译命令，可以用来编译 TypeScript 代码并进行类型检查，而这里的作用主要是用来做类型检查，我们可以从项目的 `tsconfig.json` 中注意到这样一个配置：

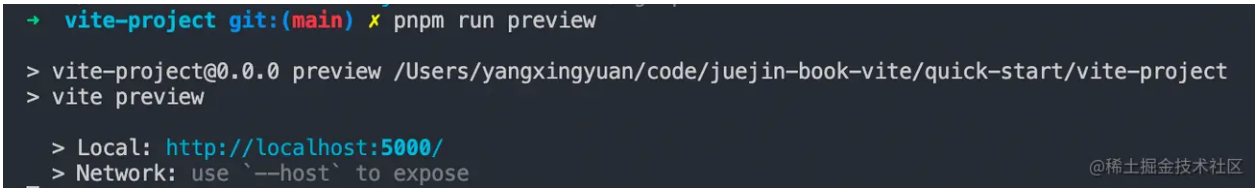
```
{
  "compilerOptions": {
    // 省略其他配置
    // 1. noEmit 表示只做类型检查，而不会输出产物文件
    // 2. 这行配置与 tsc --noEmit 命令等效
    "noEmit": true,
  },
}
```

虽然 Vite 提供了开箱即用的 TypeScript 以及 JSX 的编译能力，但实际上底层并没有实现 TypeScript 的类型校验系统，因此需要借助 `tsc` 来完成类型校验(在 Vue 项目中使用 `vue-tsc` 这个工具来完成)，在打包前提早暴露出类型相关的问题，保证代码的健壮性。

接下来你可以试着执行一下这个打包命令:



此时 Vite 已经生成了最终的打包产物，我们可以通过 `pnpm run preview` 命令预览一下打包产物的执行效果。



在浏览器中打开 `http://localhost:5000` 地址，你将看到和开发阶段一样的页面内容，证明我们成功完成第一个 Vite 项目的生产环境构建。

小结

恭喜你完成了本节的学习！在这一小节中，我们正式地开始近距离接触 Vite，使用它来初始化第一个 Vite 项目。

在最开始，我们一起搭建了基本的前端开发环境，安装常用的编辑器、浏览器、Node.js 环境及包管理器 pnpm，接着我和你使用 Vite 的初始化命令创建一个 React 项目并成功启动，让你真切地体验到 Vite 的快速和轻量。

项目启动之后我也与你分析了项目背后的启动流程，强调了 `一个 import 语句代表一个 HTTP 请求`，而正是 Vite 的 Dev Server 来接收这些请求、进行文件转译以及返回浏览器可以运行的代码，从而让项目正常运行。

不仅如此，我还带你一起初步接触了 Vite 的配置文件，并尝试进行生产环境的打包，为下一节的学习作下了铺垫。在下一小节中，我们将通过 Vite 搭建起一个相对完整的工程化项目框架，你也将会面临更多的开发场景和挑战，逐渐对 Vite 的使用轻车熟路，让我们下一节再见！

上一篇：模块标准：为什么 ESM 是前端模块化的未来?

下一篇： 样式方案：在 Vite 中接入现代化的 CSS 工程化方案