# 概念

Vite（法语意为 "快速的"，发音 /vit/，发音同 "veet"）是一种新型前端构建工具，能够显著提升前端开发体验。它主要由两部分组成：

* 一个开发服务器，它基于 原生 ES 模块 提供了 丰富的内建功能，如速度快到惊人的 模块热更新（HMR）。
* 一套构建指令，它使用 Rollup 打包你的代码，并且它是预配置的，可输出用于生产环境的高度优化过的静态资源。

Vite 意在提供开箱即用的配置，同时它的 插件 API 和 JavaScript API 带来了高度的可扩展性，并有完整的类型支持。

## index.html 与项目根目录

在一个 Vite 项目中，index.html 在项目最外层而不是在 public 文件夹内。这是有意而为之的：在开发期间 Vite 是一个服务器，而 index.html 是该 Vite 项目的入口文件。

Vite 将 index.html 视为源码和模块图的一部分。Vite 解析 <script type="module" src="..."> ，这个标签指向你的 JavaScript 源码。甚至内联引入 JavaScript 的 <script type="module"> 和引用 CSS 的 <link href> 也能利用 Vite 特有的功能被解析。另外，index.html 中的 URL 将被自动转换，因此不再需要 %PUBLIC\_URL% 占位符了。

与静态 HTTP 服务器类似，Vite 也有 “根目录” 的概念，即服务文件的位置，在接下来的文档中你将看到它会以 <root> 代称。源码中的绝对 URL 路径将以项目的 “根” 作为基础来解析，因此你可以像在普通的静态文件服务器上一样编写代码（并且功能更强大！）。Vite 还能够处理依赖关系，解析处于根目录外的文件位置，这使得它即使在基于 monorepo 的方案中也十分有用。

Vite 也支持多个 .html 作入口点的多页面应用模式。

## 命令行界面

在安装了 Vite 的项目中，可以在 npm scripts 中使用 vite 可执行文件，或者直接使用 npx vite 运行它。下面是通过脚手架创建的 Vite 项目中默认的 npm scripts：

{

"scripts": {

"dev": "vite", // 启动开发服务器，别名：`vite dev`，`vite serve`

"build": "vite build", // 为生产环境构建产物

"preview": "vite preview" // 本地预览生产构建产物

}

}

可以指定额外的命令行选项，如 --port 或 --https。运行 npx vite --help 获得完整的命令行选项列表。

# 功能

对非常基础的使用来说，使用 Vite 开发和使用一个静态文件服务器并没有太大区别。然而，Vite 还通过原生 ESM 导入提供了许多主要用于打包场景的增强功能。

## NPM 依赖解析和预构建

原生 ES 导入不支持下面这样的裸模块导入：

import { someMethod } from 'my-dep'

上面的代码会在浏览器中抛出一个错误。Vite 将会检测到所有被加载的源文件中的此类裸模块导入，并执行以下操作:

* 预构建 它们可以提高页面加载速度，并将 CommonJS / UMD 转换为 ESM 格式。预构建这一步由 esbuild 执行，这使得 Vite 的冷启动时间比任何基于 JavaScript 的打包器都要快得多。
* 重写导入为合法的 URL，例如 /node\_modules/.vite/my-dep.js?v=f3sf2ebd 以便浏览器能够正确导入它们。

## 模块热重载

Vite 提供了一套原生 ESM 的 HMR API。 具有 HMR 功能的框架可以利用该 API 提供即时、准确的更新，而无需重新加载页面或清除应用程序状态。Vite 内置了 HMR 到 Vue 单文件组件（SFC） 和 React Fast Refresh 中。也通过 @prefresh/vite 对 Preact 实现了官方集成。

## TypeScript

Vite 天然支持引入 .ts 文件。Vite 仅执行 .ts 文件的转译工作，并 不 执行任何类型检查。并假设类型检查已经被你的 IDE 或构建过程接管了（你可以在构建脚本中运行 tsc --noEmit 或者安装 vue-tsc 然后运行 vue-tsc --noEmit 来对你的 \*.vue 文件做类型检查）。

Vite 使用 esbuild 将 TypeScript 转译到 JavaScript，约是 tsc 速度的 20~30 倍，同时 HMR 更新反映到浏览器的时间小于 50ms。

### TypeScript 编译器选项

tsconfig.json中compilerOptions下的一些配置项需要特别注意。

isolatedModules应该设置为 true。这是因为 esbuild 只执行没有类型信息的转译，它并不支持某些特性，如 const enum 和隐式类型导入。你必须在 tsconfig.json 中的 compilerOptions 下设置 "isolatedModules": true。如此做，TS 会警告你不要使用隔离（isolated）转译的功能。

useDefineForClassFields。从 Vite v2.5.0 开始，如果 TypeScript 的 target 是 ESNext，此选项默认值则为 true。这与 tsc v4.3.2 及以后版本的行为 一致。这也是标准的 ECMAScript 的运行时行为。但对于那些习惯其他编程语言或旧版本 TypeScript 的开发者来说，这可能是违反直觉的。 你可以参阅 TypeScript 3.7 发布日志 中了解更多关于如何兼容的信息。如果你正在使用一个严重依赖 class fields 的库，请注意该库对此选项的预期设置。

大多数库都希望 "useDefineForClassFields": true，如 MobX，Vue Class Components 8.x 等。但是有几个库还没有兼容这个新的默认值，其中包括 lit-element。如果遇到这种情况，请将 useDefineForClassFields 设置为 false。

影响构建结果的其他编译器选项

* extends
* importsNotUsedAsValues
* jsxFactory
* jsxFragmentFactory

### 客户端类型

Vite 默认的类型定义是写给它的 Node.js API 的。要将其补充到一个 Vite 应用的客户端代码环境中，请添加一个 d.ts 声明文件：

/// <reference types="vite/client" />

同时，你也可以将 vite/client 添加到 tsconfig 中的 compilerOptions.types 下：

{

"compilerOptions": {

"types": ["vite/client"]

}

}

这将会提供以下类型定义补充：

* 资源导入 (例如：导入一个 .svg 文件)
* import.meta.env 上 Vite 注入的环境变量的类型定义
* import.meta.hot 上的 HMR API 类型定义

## Vue

Vite 为 Vue 提供第一优先级支持：

* Vue 3 单文件组件支持：@vitejs/plugin-vue
* Vue 3 JSX 支持：@vitejs/plugin-vue-jsx
* Vue 2 支持：underfin/vite-plugin-vue2

## JSX

.jsx 和 .tsx 文件同样开箱即用。JSX 的转译同样是通过 esbuild，默认为 React 16 风格。Vue 用户应使用官方提供的 @vitejs/plugin-vue-jsx 插件，它提供了 Vue 3 特性的支持，包括 HMR，全局组件解析，指令和插槽。

如果不是在 React 或 Vue 中使用 JSX，自定义的 jsxFactory 和 jsxFragment 可以使用 esbuild 选项 进行配置。例如对 Preact：

// vite.config.js

import { defineConfig } from 'vite'

export default defineConfig({

esbuild: {

jsxFactory: 'h',

jsxFragment: 'Fragment'

}

})

可以使用 jsxInject（这是一个仅在 Vite 中使用的选项）为 JSX 注入 helper，以避免手动导入：

// vite.config.js

import { defineConfig } from 'vite'

export default defineConfig({

esbuild: {

jsxInject: `import React from 'react'`

}

})

## CSS

导入 .css 文件将会把内容插入到 <style> 标签中，同时也带有 HMR 支持。也能够以字符串的形式检索处理后的、作为其模块默认导出的 CSS。

### @import 内联和变基

Vite 通过 postcss-import 预配置支持了 CSS @import 内联，Vite 的路径别名也遵从 CSS @import。换句话说，所有 CSS url() 引用，即使导入的文件在不同的目录中，也总是自动变基，以确保正确性。Sass 和 Less 文件也支持 @import 别名和 URL 变基。

### PostCSS

如果项目包含有效的 PostCSS 配置 (任何受 postcss-load-config 支持的格式，例如 postcss.config.js)，它将会自动应用于所有已导入的 CSS。

### CSS Modules

任何以 .module.css 为后缀名的 CSS 文件都被认为是一个 CSS modules 文件。导入这样的文件会返回一个相应的模块对象：

/\* example.module.css \*/

.red {

color: red;

}

import classes from './example.module.css'

document.getElementById('foo').className = classes.red

CSS modules 行为可以通过 css.modules 选项 进行配置。如果 css.modules.localsConvention 设置开启了 camelCase 格式变量名转换（例如 localsConvention: 'camelCaseOnly'），你还可以使用按名导入。

// .apply-color -> applyColor

import { applyColor } from './example.module.css'

document.getElementById('foo').className = applyColor

### CSS 预处理器

由于 Vite 的目标仅为现代浏览器，因此建议使用原生 CSS 变量和实现 CSSWG 草案的 PostCSS 插件（例如 postcss-nesting）来编写简单的、符合未来标准的 CSS。话虽如此，但 Vite 也同时提供了对 .scss, .sass, .less, .styl 和 .stylus 文件的内置支持。没有必要为它们安装特定的 Vite 插件，但必须安装相应的预处理器依赖：

# .scss and .sass

npm install -D sass

# .less

npm install -D less

# .styl and .stylus

npm install -D stylus

如果是用的是单文件组件，可以通过 <style lang="sass">（或其他预处理器）自动开启。Vite 为 Sass 和 Less 改进了 @import 解析，以保证 Vite 别名也能被使用。另外，url() 中的相对路径引用的，与根文件不同目录中的 Sass/Less 文件会自动变基以保证正确性。由于 Stylus API 限制，@import 别名和 URL 变基不支持 Stylus。

你还可以通过在文件扩展名前加上 .module 来结合使用 CSS modules 和预处理器，例如 style.module.scss。

## 静态资源处理

导入一个静态资源会返回解析后的 URL：

import imgUrl from './img.png'

document.getElementById('hero-img').src = imgUrl

添加一些特殊的查询参数可以更改资源被引入的方式：

// 显式加载资源为一个 URL

import assetAsURL from './asset.js?url'

// 以字符串形式加载资源

import assetAsString from './shader.glsl?raw'

// 加载为 Web Worker

import Worker from './worker.js?worker'

// 在构建时 Web Worker 内联为 base64 字符串

import InlineWorker from './worker.js?worker&inline'

## JSON

JSON 可以被直接导入 —— 同样支持具名导入：

// 导入整个对象

import json from './example.json'

// 对一个根字段使用具名导入 —— 有效帮助 treeshaking！

import { field } from './example.json'

## Glob 导入

Vite 支持使用特殊的 import.meta.glob 函数从文件系统导入多个模块：

const modules = import.meta.glob('./dir/\*.js')

以上将会被转译为下面的样子：

// vite 生成的代码

const modules = {

'./dir/foo.js': () => import('./dir/foo.js'),

'./dir/bar.js': () => import('./dir/bar.js')

}

可以遍历 modules 对象的 key 值来访问相应的模块：

for (const path in modules) {

modules[path]().then((mod) => {

console.log(path, mod)

})

}

匹配到的文件默认是懒加载的，通过动态导入实现，并会在构建时分离为独立的 chunk。如果你倾向于直接引入所有的模块（例如依赖于这些模块中的副作用首先被应用），你可以使用 import.meta.globEager 代替：

const modules = import.meta.globEager('./dir/\*.js')

以上会被转译为下面的样子：

// vite 生成的代码

import \* as \_\_glob\_\_0\_0 from './dir/foo.js'

import \* as \_\_glob\_\_0\_1 from './dir/bar.js'

const modules = {

'./dir/foo.js': \_\_glob\_\_0\_0,

'./dir/bar.js': \_\_glob\_\_0\_1

}

注意：

* 这只是一个 Vite 独有的功能而不是一个 Web 或 ES 标准
* 该 Glob 模式会被当成导入标识符：必须是相对路径（以 ./ 开头）或绝对路径（以 / 开头，相对于项目根目录解析）。
* Glob 匹配是使用 fast-glob 来实现的 —— 阅读它的文档来查阅 支持的 Glob 模式。
* 你还需注意，glob 的导入不接受变量，你应直接传递字符串模式。

## WebAssembly

预编译的 .wasm 文件可以直接被导入 —— 默认导出一个函数，返回值为所导出 wasm 实例对象的 Promise：

import init from './example.wasm'

init().then((exports) => {

exports.test()

})

init 函数还可以将传递给 WebAssembly.instantiate 的导入对象作为其第二个参数：

init({

imports: {

someFunc: () => {

/\* ... \*/

}

}

}).then(() => {

/\* ... \*/

})

在生产构建当中，体积小于 assetInlineLimit 的 .wasm 文件将会被内联为 base64 字符串。否则，它们将作为资源复制到 dist 目录中，并按需获取。

## Web Worker

一个 web worker 脚本可以直接通过添加一个 ?worker 或 ?sharedworker 查询参数来导入。默认导出一个自定义的 worker 构造器：

import MyWorker from './worker?worker'

const worker = new MyWorker()

Worker 脚本也可以使用 import 语句来替代 importScripts() —— 注意，在开发过程中，这依赖于浏览器原生支持，目前只在 Chrome 中适用，而在生产版本中，它已经被编译掉了。

默认情况下，worker 脚本将在生产构建中编译成单独的 chunk。如果你想将 worker 内联为 base64 字符串，请添加 inline 查询参数：

import MyWorker from './worker?worker&inline'

## 构建优化

### CSS 代码分割

Vite 会自动地将一个异步 chunk 模块中使用到的 CSS 代码抽取出来并为其生成一个单独的文件。这个 CSS 文件将在该异步 chunk 加载完成时自动通过一个 <link> 标签载入，该异步 chunk 会保证只在 CSS 加载完毕后再执行，避免发生 FOUC 。

如果你更倾向于将所有的 CSS 抽取到一个文件中，你可以通过设置 build.cssCodeSplit 为 false 来禁用 CSS 代码分割。

### 预加载指令生成

Vite 会为入口 chunk 和它们在打包出的 HTML 中的直接引入自动生成 <link rel="modulepreload"> 指令。

### 异步 Chunk 加载优化

在实际项目中，Rollup 通常会生成 “共用” chunk —— 被两个或以上的其他 chunk 共享的 chunk。与动态导入相结合，会很容易出现下面这种场景：

[异步chunk加载优化](异步chunk加载优化.png)

在无优化的情境下，当异步 chunk A 被导入时，浏览器将必须请求和解析 A，然后它才能弄清楚它也需要共用 chunk C。这会导致额外的网络往返：

Entry ---> A ---> C

Vite 将使用一个预加载步骤自动重写代码，来分割动态导入调用，以实现当 A 被请求时，C 也将 同时 被请求：

Entry ---> (A + C)

C 也可能有更深的导入，在未优化的场景中，这会导致更多的网络往返。Vite 的优化会跟踪所有的直接导入，无论导入的深度如何，都能够完全消除不必要的往返。

# 使用插件

Vite 可以使用插件进行扩展，这得益于 Rollup 优秀的插件接口设计和一部分 Vite 独有的额外选项。这意味着 Vite 用户可以利用 Rollup 插件的强大生态系统，同时根据需要也能够扩展开发服务器和 SSR 功能。

## 添加一个插件

若要使用一个插件，需要将它添加到项目的 devDependencies 并在 vite.config.js 配置文件中的 plugins 数组中引入它。例如，要想为传统浏览器提供支持，可以按下面这样使用官方插件 @vitejs/plugin-legacy：

$ npm i -D @vitejs/plugin-legacy

// vite.config.js

import legacy from '@vitejs/plugin-legacy'

import { defineConfig } from 'vite'

export default defineConfig({

plugins: [

legacy({

targets: ['defaults', 'not IE 11']

})

]

})

plugins 也可以接受包含多个插件作为单个元素的预设。这对于使用多个插件实现的复杂特性（如框架集成）很有用。该数组将在内部被扁平化。Falsy 虚值的插件将被忽略，可以用来轻松地启用或停用插件。

## 查找插件

查看 Plugins 章节 获取官方插件信息。社区插件列表请参见 awesome-vite。而对于兼容的 Rollup 插件，请查看 Vite Rollup 插件 获取一个带使用说明的兼容 Rollup 官方插件列表，若列表中没有找到，则请参阅 Rollup 插件兼容性章节。

## 强制插件排序

为了与某些 Rollup 插件兼容，可能需要强制执行插件的顺序，或者只在构建时使用。这应该是 Vite 插件的实现细节。可以使用 enforce 修饰符来强制插件的位置:

* pre：在 Vite 核心插件之前调用该插件
* 默认：在 Vite 核心插件之后调用该插件
* post：在 Vite 构建插件之后调用该插件

## 按需应用

默认情况下插件在开发 (serve) 和生产 (build) 模式中都会调用。如果插件在服务或构建期间按需使用，请使用 apply 属性指明它们仅在 'build' 或 'serve' 模式时调用：

// vite.config.js

import typescript2 from 'rollup-plugin-typescript2'

import { defineConfig } from 'vite'

export default defineConfig({

plugins: [

{

...typescript2(),

apply: 'build'

}

]

})

# 依赖预构建

当你首次启动 vite 时，可能会注意到打印出了以下信息：

Optimizable dependencies detected: （侦测到可优化的依赖：）

react, react-dom

Pre-bundling them to speed up dev server page load...（将预构建它们以提升开发服务器页面加载速度）

(this will be run only when your dependencies have changed)（这将只会在你的依赖发生变化时执行）

## 原因

这就是 Vite 执行的所谓的“依赖预构建”。这个过程有两个目的:

1. CommonJS 和 UMD 兼容性: 开发阶段中，Vite 的开发服务器将所有代码视为原生 ES 模块。因此，Vite 必须先将作为 CommonJS 或 UMD 发布的依赖项转换为 ESM。

当转换 CommonJS 依赖时，Vite 会执行智能导入分析，这样即使导出是动态分配的（如 React），按名导入也会符合预期效果：

// 符合预期

import React, { useState } from 'react'

1. 性能： Vite 将有许多内部模块的 ESM 依赖关系转换为单个模块，以提高后续页面加载性能。

一些包将它们的 ES 模块构建作为许多单独的文件相互导入。例如，lodash-es 有超过 600 个内置模块！当我们执行 import { debounce } from 'lodash-es' 时，浏览器同时发出 600 多个 HTTP 请求！尽管服务器在处理这些请求时没有问题，但大量的请求会在浏览器端造成网络拥塞，导致页面的加载速度相当慢。

通过预构建 lodash-es 成为一个模块，我们就只需要一个 HTTP 请求了！

## 自动依赖搜寻

如果没有找到相应的缓存，Vite 将抓取你的源码，并自动寻找引入的依赖项（即 "bare import"，表示期望从 node\_modules 解析），并将这些依赖项作为预构建包的入口点。预构建通过 esbuild 执行，所以它通常非常快。

在服务器已经启动之后，如果遇到一个新的依赖关系导入，而这个依赖关系还没有在缓存中，Vite 将重新运行依赖构建进程并重新加载页面。

## Monorepo 和链接依赖

在一个 monorepo 启动中，该仓库中的某个依赖可能会成为另一个包的依赖。Vite 会自动侦测没有从 node\_modules 解析的依赖项，并将链接的依赖视为源码。它不会尝试打包被链接的依赖，而是会分析被链接依赖的依赖列表。

由于依赖关系的处理方式不同，链接的依赖关系在最终构建时可能无法正常工作。 使用 npm package 代替所有本地依赖，以避免最终的 bundle 问题。

## 自定义行为

默认的依赖项发现为启发式可能并不总是可取的。在你想要显式地从列表中包含/排除依赖项的情况下, 请使用 optimizeDeps 配置项。

当你遇到不能直接在源码中发现的 import 时，optimizeDeps.include 或 optimizeDeps.exclude 就是典型的用例。例如，import 可能是插件转换的结果。这意味着 Vite 无法在初始扫描时发现 import —— 它只能在浏览器请求文件时转换后才能发现。这将导致服务器在启动后立即重新打包。

include 和 exclude 都可以用来处理这个问题。如果依赖项很大（包含很多内部模块）或者是 CommonJS，那么你应该包含它；如果依赖项很小，并且已经是有效的 ESM，则可以排除它，让浏览器直接加载它。

## 缓存

### 文件系统缓存

Vite 会将预构建的依赖缓存到 node\_modules/.vite。它根据几个源来决定是否需要重新运行预构建步骤:

* package.json 中的 dependencies 列表
* 包管理器的 lockfile，例如 package-lock.json, yarn.lock，或者 pnpm-lock.yaml
* 可能在 vite.config.js 相关字段中配置过的

只有在上述其中一项发生更改时，才需要重新运行预构建。

如果出于某些原因，你想要强制 Vite 重新构建依赖，你可以用 --force 命令行选项启动开发服务器，或者手动删除 node\_modules/.vite 目录。

### 浏览器缓存

解析后的依赖请求会以 HTTP 头 max-age=31536000,immutable 强缓存，以提高在开发时的页面重载性能。一旦被缓存，这些请求将永远不会再到达开发服务器。如果安装了不同的版本（这反映在包管理器的 lockfile 中），则附加的版本 query 会自动使它们失效。如果你想通过本地编辑来调试依赖项，你可以:

* 通过浏览器调试工具的 Network 选项卡暂时禁用缓存；
* 重启 Vite dev server，并添加 --force 命令以重新构建依赖；
* 重新载入页面。

# 静态资源处理

## 将资源引入为 URL

服务时引入一个静态资源会返回解析后的公共路径：

import imgUrl from './img.png'

document.getElementById('hero-img').src = imgUrl

例如，imgUrl 在开发时会是 /img.png，在生产构建后会是 /assets/img.2d8efhg.png。行为类似于 Webpack 的 file-loader。区别在于导入既可以使用绝对公共路径（基于开发期间的项目根路径），也可以使用相对路径。

* url() 在 CSS 中的引用也以同样的方式处理。
* 如果 Vite 使用了 Vue 插件，Vue SFC 模板中的资源引用都将自动转换为导入。
* 常见的图像、媒体和字体文件类型被自动检测为资源。你可以使用 assetsInclude 选项 扩展内部列表。
* 引用的资源作为构建资源图的一部分包括在内，将生成散列文件名，并可以由插件进行处理以进行优化。
* 较小的资源体积小于 assetsInlineLimit 选项值 则会被内联为 base64 data URL。

### 显式 URL 引入

未被包含在内部列表或 assetsInclude 中的资源，可以使用 ?url 后缀显式导入为一个 URL。这十分有用，例如，要导入 Houdini Paint Worklets 时：

import workletURL from 'extra-scalloped-border/worklet.js?url'

CSS.paintWorklet.addModule(workletURL)

### 将资源引入为字符串

资源可以使用 ?raw 后缀声明作为字符串引入。

import shaderString from './shader.glsl?raw'

### 导入脚本作为 Worker

脚本可以通过 ?worker 或 ?sharedworker 后缀导入为 web worker。

// 在生产构建中将会分离出 chunk

import Worker from './shader.js?worker'

const worker = new Worker()

// sharedworker

import SharedWorker from './shader.js?sharedworker'

const sharedWorker = new SharedWorker()

// 内联为 base64 字符串

import InlineWorker from './shader.js?worker&inline'

### public 目录

如果你有下列这些资源：

* 不会被源码引用（例如 robots.txt）
* 必须保持原有文件名（没有经过 hash）
* ...或者你压根不想引入该资源，只是想得到其 URL。

那么你可以将该资源放在指定的 public 目录中，它应位于你的项目根目录。该目录中的资源在开发时能直接通过 / 根路径访问到，并且打包时会被完整复制到目标目录的根目录下。目录默认是 <root>/public，但可以通过 publicDir 选项 来配置。

请注意：

* 引入 public 中的资源永远应该使用根绝对路径 —— 举个例子，public/icon.png 应该在源码中被引用为 /icon.png。
* public 中的资源不应该被 JavaScript 文件引用。

## new URL(url, import.meta.url)

import.meta.url 是一个 ESM 的原生功能，会暴露当前模块的 URL。将它与原生的 URL 构造器 组合使用，在一个 JavaScript 模块中，通过相对路径我们就能得到一个被完整解析的静态资源 URL：

const imgUrl = new URL('./img.png', import.meta.url).href

document.getElementById('hero-img').src = imgUrl

这在现代浏览器中能够原生使用 - 实际上，Vite 并不需要在开发阶段处理这些代码！这个模式同样还可以通过字符串模板支持动态 URL：

function getImageUrl(name) {

return new URL(`./dir/${name}.png`, import.meta.url).href

}

# 构建生产版本

当需要将应用部署到生产环境时，只需运行 vite build 命令。默认情况下，它使用 <root>/index.html 作为其构建入口点，并生成能够静态部署的应用程序包。

## 浏览器的兼容性

用于生产环境的构建包会假设目标浏览器支持现代 JavaScript 语法。默认情况下，Vite 的目标浏览器是指能够 支持原生 ESM script 标签 和 支持原生 ESM 动态导入 的。可以通过 build.target 配置项 指定构建目标，最低支持 es2015。

请注意，默认情况下 Vite 只处理语法转译，且 默认不包含任何 polyfill。你可以前往 Polyfill.io 查看，这是一个基于用户浏览器 User-Agent 字符串自动生成 polyfill 包的服务。

传统浏览器可以通过插件 @vitejs/plugin-legacy 来支持，它将自动生成传统版本的 chunk 及与其相对应 ES 语言特性方面的 polyfill。兼容版的 chunk 只会在不支持原生 ESM 的浏览器中进行按需加载。

## 公共基础路径

如果你需要在嵌套的公共路径下部署项目，只需指定 base 配置项，然后所有资源的路径都将据此配置重写。这个选项也可以通过命令行参数指定，例如 vite build --base=/my/public/path/。

由 JS 引入的资源 URL，CSS 中的 url() 引用以及 .html 文件中引用的资源在构建过程中都会自动调整，以适配此选项。

当然，情况也有例外，当访问过程中需要使用动态连接的 url 时，可以使用全局注入的 import.meta.env.BASE\_URL 变量，它的值为公共基础路径。注意，这个变量在构建时会被静态替换，因此，它必须按 import.meta.env.BASE\_URL 的原样出现（例如 import.meta.env['BASE\_URL'] 是无效的）

## 自定义构建

构建过程可以通过多种 构建配置选项 来自定义构建。具体来说，你可以通过 build.rollupOptions 直接调整底层的 Rollup 选项：

// vite.config.js

module.exports = defineConfig({

build: {

rollupOptions: {

// https://rollupjs.org/guide/en/#big-list-of-options

}

}

})

例如，你可以使用仅在构建期间应用的插件来指定多个 Rollup 输出。

## 文件变化时重新构建

你可以使用 vite build --watch 来启用 rollup 的监听器。或者，你可以直接通过 build.watch 调整底层的 WatcherOptions 选项：

// vite.config.js

module.exports = defineConfig({

build: {

watch: {

// https://rollupjs.org/guide/en/#watch-options

}

}

})

## 多页面应用模式

假设你有下面这样的项目文件结构：

├── package.json

├── vite.config.js

├── index.html

├── main.js

└── nested

├── index.html

└── nested.js

在开发过程中，简单地导航或链接到 /nested/ - 将会按预期工作，与正常的静态文件服务器表现一致。在构建过程中，你只需指定多个 .html 文件作为入口点即可：

// vite.config.js

const { resolve } = require('path')

const { defineConfig } = require('vite')

module.exports = defineConfig({

build: {

rollupOptions: {

input: {

main: resolve(\_\_dirname, 'index.html'),

nested: resolve(\_\_dirname, 'nested/index.html')

}

}

}

})

## 库模式

当你开发面向浏览器的库时，你可能会将大部分时间花在该库的测试/演示页面上。在 Vite 中你可以使用 index.html 获得如丝般顺滑的开发体验。

当这个库要进行发布构建时，请使用 build.lib 配置项，以确保将那些你不想打包进库的依赖进行外部化处理，例如 vue 或 react：

// vite.config.js

const path = require('path')

const { defineConfig } = require('vite')

module.exports = defineConfig({

build: {

lib: {

entry: path.resolve(\_\_dirname, 'lib/main.js'),

name: 'MyLib',

fileName: (format) => `my-lib.${format}.js`

},

rollupOptions: {

// 确保外部化处理那些你不想打包进库的依赖

external: ['vue'],

output: {

// 在 UMD 构建模式下为这些外部化的依赖提供一个全局变量

globals: {

vue: 'Vue'

}

}

}

}

})

使用如上配置运行 vite build 时，将会使用一套面向库的 Rollup 预设，并且将为该库提供两种构建格式：es 和 umd (可在 build.lib 中配置)：

$ vite build

building for production...

[write] my-lib.es.js 0.08kb, brotli: 0.07kb

[write] my-lib.umd.js 0.30kb, brotli: 0.16kb

# 部署静态站点

建立在以下几个假设基础之上：

* 正在使用的是默认的构建输出路径（dist）。这个路径 可以通过 build.outDir 更改。
* 正在使用 NPM；或者 Yarn 等其他可以运行下面的脚本指令的包管理工具。
* Vite 已作为一个本地开发依赖（dev dependency）安装在你的项目中，并且你已经配置好了如下的 npm scripts：

{

"scripts": {

"build": "vite build",

"preview": "vite preview"

}

}

## 构建应用

运行 npm run build 命令来执行应用的构建。

$ npm run build

当你构建完成应用后，你可以通过运行 npm run preview 命令，在本地测试该应用。

$ npm run build

$ npm run preview

可以通过 --port 参数来配置服务的运行端口。

{

"scripts": {

"preview": "vite preview --port 8080"

}

}

# 环境变量和模式

## 环境变量

Vite 在一个特殊的 import.meta.env 对象上暴露环境变量。这里有一些在所有情况下都可以使用的内建变量：

* import.meta.env.MODE: {string} 应用运行的模式。
* import.meta.env.BASE\_URL: {string} 部署应用时的基本 URL。他由base 配置项决定。
* import.meta.env.PROD: {boolean} 应用是否运行在生产环境。
* import.meta.env.DEV: {boolean} 应用是否运行在开发环境 (永远与 import.meta.env.PROD相反)。

在生产环境中，这些环境变量会在构建时被静态替换，因此，在引用它们时请使用完全静态的字符串。动态的 key 将无法生效。例如，动态 key 取值 import.meta.env[key] 是无效的。

它还将替换出现在 JavaScript 和 Vue 模板中的字符串。这本应是非常少见的，但也可能是不小心为之的。在这种情况下你可能会看到类似 Missing Semicolon 或 Unexpected token 等错误，例如当 "process.env.NODE\_ENV" 被替换为 ""development": "。有一些方法可以避免这个问题：

* 对于 JavaScript 字符串，你可以使用 unicode 零宽度空格 \u200b (一个看不见的分隔符)来分割这个字符串，例如： 'import.meta\u200b.env.MODE'。
* 对于 Vue 模板或其他编译到 JavaScript 字符串的 HTML，你可以使用 <wbr> 标签，例如：import.meta.<wbr>env.MODE。

## .env 文件

Vite 使用 dotenv 从你的 环境目录 中的下列文件加载额外的环境变量：

.env # 所有情况下都会加载

.env.local # 所有情况下都会加载，但会被 git 忽略

.env.[mode] # 只在指定模式下加载

.env.[mode].local # 只在指定模式下加载，但会被 git 忽略

为了防止意外地将一些环境变量泄漏到客户端，只有以 VITE\_ 为前缀的变量才会暴露给经过 vite 处理的代码。例如下面这个文件中：

DB\_PASSWORD=foobar

VITE\_SOME\_KEY=123

默认情况下，Vite 在 vite/client.d.ts 中为 import.meta.env 提供了类型定义。随着在 .env[mode] 文件中自定义了越来越多的环境变量，你可能想要在代码中获取这些以 VITE\_ 为前缀的用户自定义环境变量的 TypeScript 智能提示。

要想做到这一点，你可以在 src 目录下创建一个 env.d.ts 文件，接着按下面这样增加 ImportMetaEnv 的定义：

/// <reference types="vite/client" />

interface ImportMetaEnv {

readonly VITE\_APP\_TITLE: string

// 更多环境变量...

}

interface ImportMeta {

readonly env: ImportMetaEnv

}

## 模式

默认情况下，开发服务器 (dev 命令) 运行在 development (开发) 模式，而 build 以及 serve 命令则运行在 production (生产) 模式。这意味着当执行 vite build 时，它会自动加载 .env.production 中可能存在的环境变量：

# .env.production

VITE\_APP\_TITLE=My App

在你的应用中，你可以使用 import.meta.env.VITE\_APP\_TITLE 渲染标题。然而，重要的是要理解 模式 是一个更广泛的概念，而不仅仅是开发和生产。一个典型的例子是，你可能希望有一个 “staging” (预发布|预上线) 模式，它应该具有类似于生产的行为，但环境变量与生产环境略有不同。

你可以通过传递 --mode 选项标志来覆盖命令使用的默认模式。例如，如果你想为我们假设的 staging 模式构建应用：

vite build --mode staging

为了使应用实现预期行为，我们还需要一个 .env.staging 文件：

# .env.staging

NODE\_ENV=production

VITE\_APP\_TITLE=My App (staging)

# 配置文件

## 配置文件解析

当以命令行方式运行 vite 时，Vite 会自动解析 项目根目录 下名为 vite.config.js 的文件。最基础的配置文件是这样的：

// vite.config.js

export default {

// 配置选项

}

注意：即使项目没有在 package.json 中开启 type: "module"，Vite 也支持在配置文件中使用 ESM 语法。这种情况下，配置文件会在被加载前自动进行预处理。

你可以显式地通过 --config 命令行选项指定一个配置文件（相对于 cwd 路径进行解析）

vite --config my-config.js

## 配置智能提示

因为 Vite 本身附带 Typescript 类型，所以你可以通过 IDE 和 jsdoc 的配合来实现智能提示：

/\*\*

\* @type {import('vite').UserConfig}

\*/

const config = {

// ...

}

export default config

另外可以使用 defineConfig 工具函数，这样不用 jsdoc 注解也可以获取类型提示：

import { defineConfig } from 'vite'

export default defineConfig({

// ...

})

Vite 也直接支持 TS 配置文件。你可以在 vite.config.ts 中使用 defineConfig 工具函数。

## 情景配置

如果配置文件需要基于（dev/serve 或 build）命令或者不同的 模式 来决定选项，则可以选择导出这样一个函数：

export default defineConfig(({ command, mode }) => {

if (command === 'serve') {

return {

// dev 独有配置

}

} else {

// command === 'build'

return {

// build 独有配置

}

}

})

需要注意的是，在 Vite 的 API 中，在开发环境下 command 的值为 serve（在 CLI 中， vite dev 和 vite serve 是 vite 的别名），而在生产环境下为 build（vite build）。

## 异步配置

如果配置需要调用一个异步函数，也可以转而导出一个异步函数：

export default defineConfig(async ({ command, mode }) => {

const data = await asyncFunction()

return {

// 构建模式所需的特有配置

}

})

## 共享配置

[https://vitejs.cn/config/#shared-options](https://vitejs.cn/config/%23shared-options)