前端新工具-Vite

GitHub: https://github.com/vitejs/vite/

Vite 是一个由原生 ESM 驱动的 Web **开发构建工具。**在开发环境下基于浏览器原生 **ES Module** 开发，在生产环境下基于 **Rollup** 打包。

表象上看，Vite 可以取代基于 Webpack 的 vue-cli 或者 cra 的集成式开发工具，它的优势主要体现在提升开发者在开发过程中的体验。

特点

- 快速的冷启动(Lightning-fast cold server start)
- 即时的模块热更新(Instant hot module replacement (HMR))
- 真正的按需编译(True on-demand compilation)

由来

- Webpack Dev Server冷启动时间比较长
- Webpack HMR热更新的反应速度比较慢

一、快速上手

根据vite文档搭建初始项目

对于Vue项目，Vite 官方提供了一个比较简单的脚手架：create-vite-app，可以使用这个脚手架快速创建一个使用 Vite 构建的 Vue.js 应用

```
$ npm init vite-app <project-name>
$ cd <project-name>
$ npm install
$ npm run dev
```

针对React项目，可以使用 npm init vite-app --template react 来构建。

ES Module

script module 是 ES 模块在浏览器端的实现，目前主流的浏览器都已经支持https://caniuse.com/es6-module

其最大的特点是在浏览器端使用 export、import 的方式导入和导出模块，在 script 标签里设置 type="module"

```
// index.html

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
```

```
    <title>Document</title>
  </head>
  <body>
    <div id="app"></div>

    <script type="module">
      import { createApp } from "./main.js";
      createApp();
    </script>
  </body>
</html>
```

```
// main.js

export function createApp() {
  console.log("create app!");
}
```

浏览器会识别添加 type="module"的 script 元素，把这段内联 script 或者外链 script 认为是 ECMAScript 模块，对其内部的 import 引用发起 http 请求获取模块内容。

我们再回顾一下 vite 宣称的几个特性，并和webpack做个对比:

- webpack 之类的打包工具为了在浏览器里加载各模块，会借助胶水代码用来组装各模块，比如 webpack 使用 map 存放模块 id 和路径，使用 webpack_require 方法获取模块导出，vite 利用浏览器原生支持模块化导入这一特性，省略了对模块的组装，也就不需要生成 bundle，所以 冷启动是非常快的
- 打包工具会将各模块提前打包进 bundle 里，但打包的过程是静态的——不管某个模块的代码是否执行到，这个模块都要打包到 bundle 里，这样的坏处就是随着项目越来越大打包后的 bundle 也越来越大。而 ESM 天生就是按需加载的，只有 import 的时候才会去按需加载

二、对比vue-cli

项目结构区别

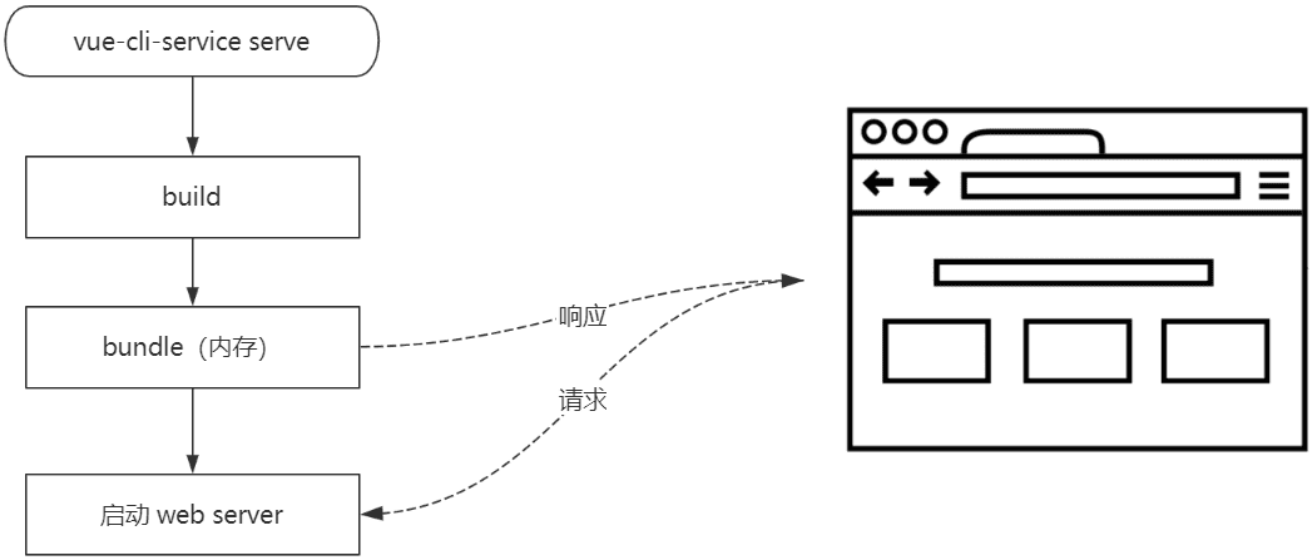
开发依赖非常简单，只有 vite 和 @vue/compiler-sfc

与vue2不同，vue3的index.html不在public目录中，是在根目录下。vite是运行在浏览器上的，index页面中直接引入了main.js。favicon.ico图标的引入也是直接根目录引入，不是vue的 `<link rel="icon" href="<%= BASE_URL %>favicon.ico">`方式。 值得注意的是/favicon.ico路径前面的/不能省略，build后的结构是favicon.ico和index.html在同层级下

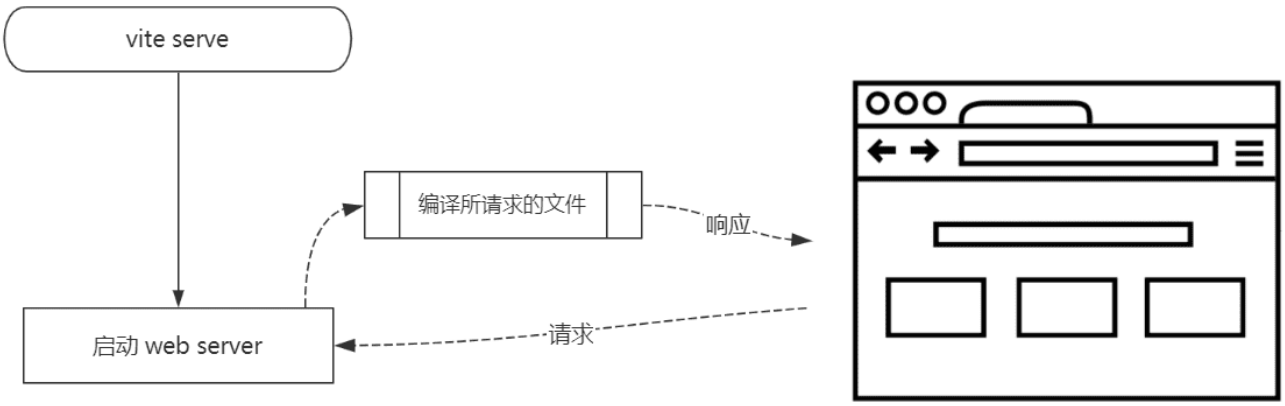
vite在开发服务器启动时间上提升很多

vue-cli 的dev-server：

- 本地运行前需要加载所有模块文件并导出bundle才能展示页面(包括对每个文件导入/导出关系的解析;将各个模块排序、重写、关联)



- 而 Vite 则完全不同，当我们执行 vite serve 时，内部直接启动了 Web Server，并不会先编译所有的代码文件。



像 Webpack 这类工具的做法是将所有模块提前编译、打包进 bundle 里，换句话说，不管模块是否会被执行，都要被编译和打包到 bundle 里。随着项目越来越大打包后的 bundle 也越来越大，打包的速度自然也就越来越慢。

Vite 利用现代浏览器原生支持 ESM 特性，省略了对模块的打包(只解析，不打包)。

对于需要编译的文件，Vite 采用的是另外一种模式：即时编译。**只有具体去请求某个文件时才会编译这个文件**，这种「即时编译」的好处主要体现在：按需编译。

三、Vite配置

具体可参考：https://github.com/vitejs/vite/blob/master/src/node/config.ts

Optimize

Vite 提供了一个optimize命令，它的作用就是单独的去「优化依赖」。

所谓的「优化依赖」，指的就是**自动去把代码中依赖的第三方模块提前编译出来。**

例如，我们在代码中通过 import 载入了 vue 这个模块，那通过这个命令就会自动将这个模块打包成一个单独的 ESM bundle, 放到 node_modules/.vite_opt_cache 目录中。

这样后续请求这个文件时就不需要再即时去加载了。

```
// src/node/server/index.ts

  server.listen = (async (port: number, ...args: any[]) => {
    if (optimizeDeps.auto !== false) {
      await require('../optimizer').optimizeDeps(config)
    }
    return listen(port, ...args)
  }) as any
```

```
// src/node/optimizer/index.ts

export interface DepOptimizationOptions {
  // 需要被处理的依赖
  include?: string[]
  // 不需要被处理的依赖
  exclude?: string[]
  // 在 link 中指定的依赖不会被 optimize 处理，因为需要防止被缓存。而依赖的依赖会被优化。在 monorepo 这种架构中使用。(monorepo架构 可参考 lerna)
  link?: string[]
  // 使用 node 原生模块，但是不直接在浏览器中使用
  allowNodeBuiltins?: string[]
  // 是否在启动时自动执行 vite optimize
  auto?: boolean
}
```

四、Vite做了什么

命令解析

主要内容是借助 minimist —— 一个轻量级的命令解析工具解析 npm scripts，解析的函数是 resolveOptions，精简后的代码片段如下。

```
// src/node/cli.ts

const argv = require('minimist')(process.argv.slice(2))

async function resolveOptions() {
    // command 可以是 dev/build/optimize
    if (argv._[0]) {
        argv.command = argv._[0];
    }
    return argv;
}
```

拿到 options 后，会根据 options.command 的值判断是执行在开发环境需要的 runServe 命令或生产环境需要的 runBuild 命令。

server

```
// src/node/server/index.ts

export function createServer(config: ServerConfig): Server {
  const app = new Koa<State, Context>()
  const server = resolveServer(config, app.callback())
  const watcher = chokidar.watch(root, {
    ignored: ['**/node_modules/**', '**/.git/**'],
    ignoreInitial: true,
    ...chokidarWatchOptions
  }) as HMRWatcher
  const resolver = createResolver(root, resolvers, alias, assetsInclude)

  const context: ServerPluginContext = {...}
}
```

vite 里是借用了 koa 来启动了一个服务，使用 chokidar 创建了一个监听文件改动的 watcher，同时实现了一个插件机制，将 koa-app 和 watcher 以及其他必要工具组合成一个 context 对象注入到每个 plugin 中。

context 组成如下：

plugin 依次从 context 里获取上面这些组成部分，有的 plugin 在 koa 实例添加了几个 middleware，有的借助 watcher 实现对文件的改动监听，这种插件机制带来的好处是整个应用结构清晰，同时每个插件处理不同的事情，职责更分明。(处理文件的模式可以参考下洋葱模型)

都有哪些plugin

- 用户注入的 plugins —— 自定义
- pluginhmrPlugin —— 处理 hmr
- htmlRewritePlugin —— 重写 html 内的 script 内容
- moduleRewritePlugin —— 重写模块中的 import 导入
- moduleResolvePlugin ——获取模块内容
- vuePlugin —— 处理 vue 单文件组件
- esbuildPlugin —— 使用 esbuild 处理资源
- assetPathPlugin —— 处理静态资源
- serveStaticPlugin —— 托管静态资源
- cssPlugin —— 处理 css/less/sass 等引用

我们来看 plugin 的实现方式，开发一个用来拦截 json 文件 plugin 可以这么实现：

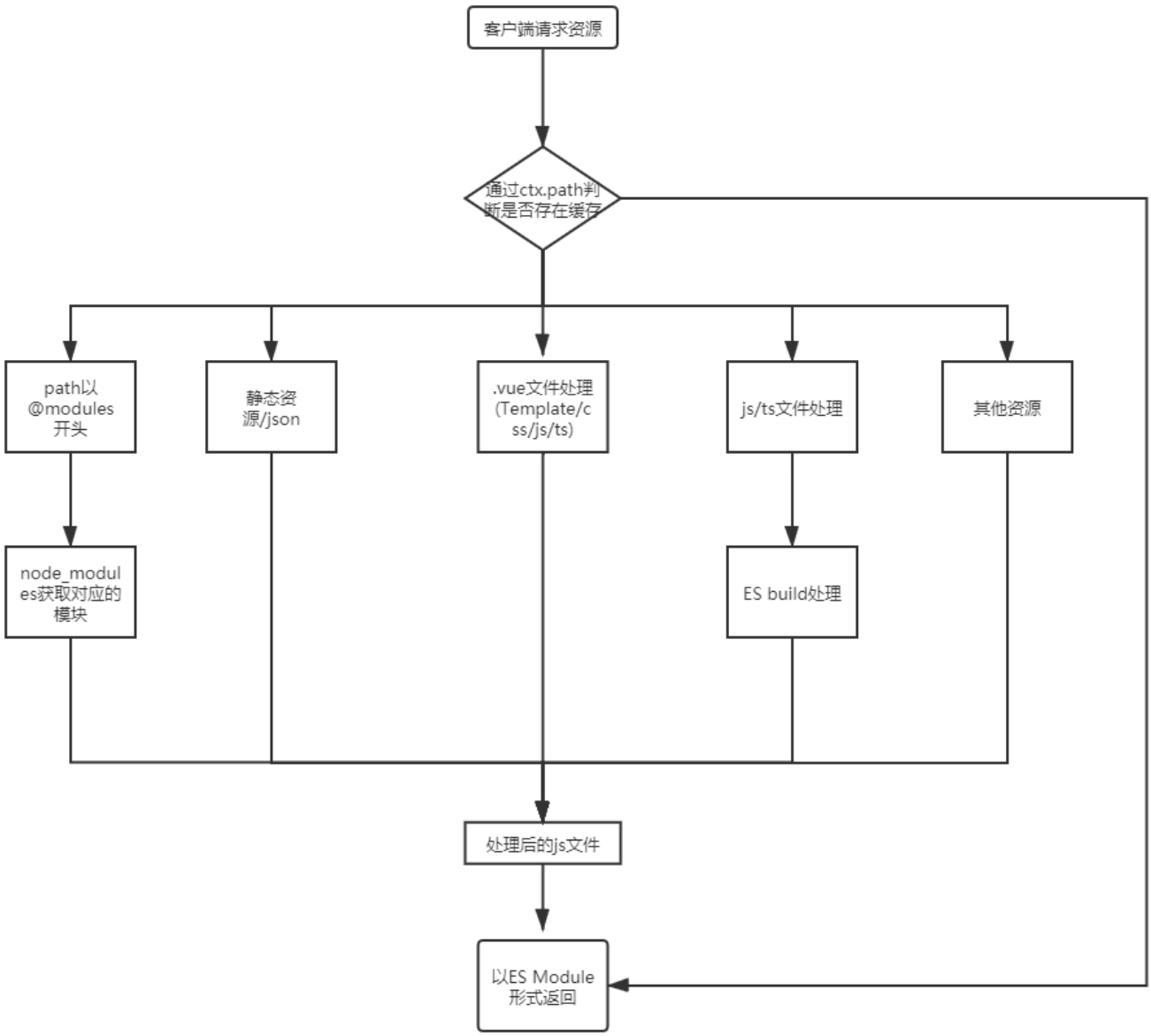
```
interface ServerPluginContext {
  root: string
  app: Koa
  server: Server
  watcher: HMRWatcher
  resolver: InternalResolver
  config: ServerConfig
}

type ServerPlugin =  (ctx:ServerPluginContext) => void;

const JsonInterceptPlugin:ServerPlugin = ({app})=>{
    app.use(async (ctx, next) => {
      await next()
      if (ctx.path.endsWith('.json') && ctx.body) {
        ctx.type = 'js'
        ctx.body = `export default json`
      }
  })
}
```

请求拦截

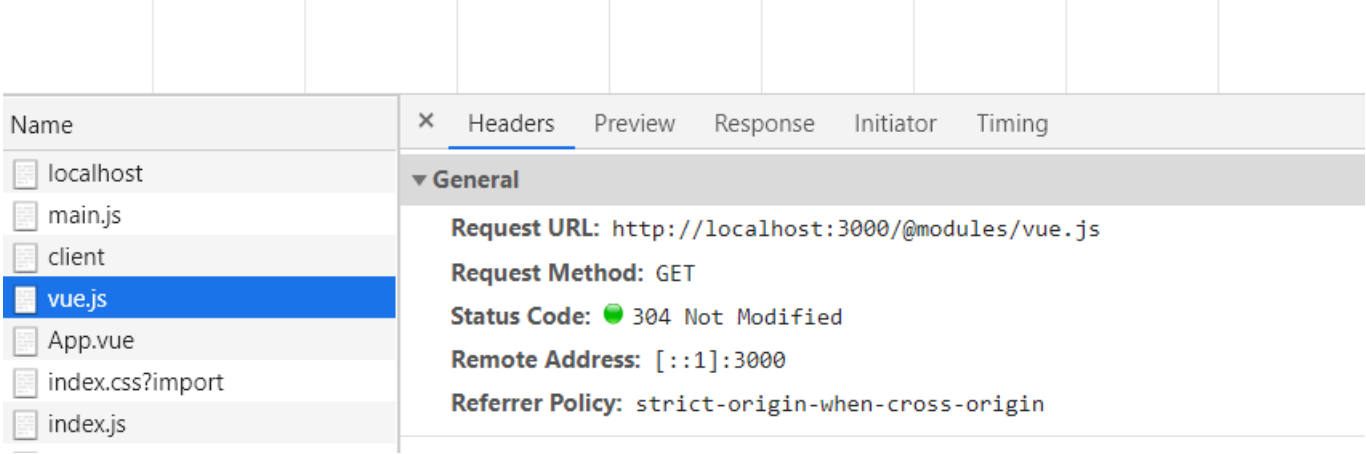
直接运行上面第一个html文件，发现是会报错的。在浏览器里使用 ES module 是使用 http 请求拿到模块，所以 vite 的一个任务就是**启动一个 web server 去代理这些模块。**

平时我们写代码，如果不是引用相对路径的模块，而是引用 node_modules 的模块，都是直接 import xxx from 'xxx'，由 Webpack 等工具来帮我们找这个模块的具体路径进行打包。但是浏览器不知道你项目里有 node_modules，它只能通过相对路径或者绝对路径去寻找模块。

这就引出了 vite 的一个实现核心 - **拦截浏览器对模块的请求并返回处理后的结果**



通过工程下的 main.js 和开发环境下的实际加载的 main.js 对比，我们发现浏览器访问的时候，路径前缀已经被处理成了 /、/@module/:id。



koa 中间件里对 import 都做了一层处理，过程如下：

- 在 koa 中间件里获取请求 body
- 通过 es-module-lexer 解析资源 ast 拿到 import 的内容

- 判断 import 的资源是否是绝对路径，绝对视为 npm 模块
- 返回处理后的资源路径："vue" => "/@modules/vue"

```
// src/node/server/serverPluginModuleRewrite.ts

export const resolveImport = (
  root: string,
  importer: string,
  id: string,
  resolver: InternalResolver,
  timestamp?: string
): string => {
  id = resolver.alias(id) || id

  if (bareImportRE.test(id)) {
    // directly resolve bare module names to its entry path so that relative
    // imports from it (including source map urls) can work correctly
    id = `/@modules/${resolveBareModuleRequest(root, id, importer, resolver)}`
  } else {
    // 1. relative to absolute
    //    ./foo -> /some/path/foo
    let { pathname, query } = resolver.resolveRelativeRequest(importer, id)

    // 2. resolve dir index and extensions.
    pathname = resolver.normalizePublicPath(pathname)

    // 3. mark non-src imports
    if (!query && path.extname(pathname) && !jsSrcRE.test(pathname)) {
      query += `?import`
    }

    id = pathname + query
  }
  // ...
}
```

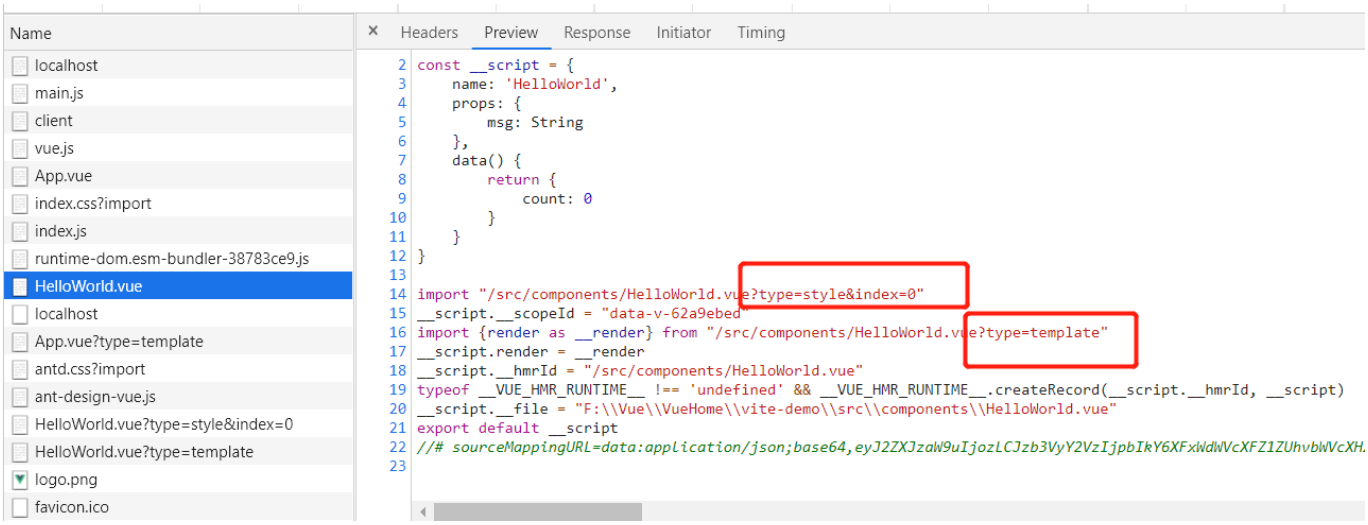
对/@module/的处理

- 在 koa 中间件里获取请求 body
- 判断路径是否以 /@module/ 开头，如果是取出包名
- 去node_module里找到这个库，基于 package.json 返回对应的内容

```
src/node/server/serverPluginModuleResolve.ts

export const moduleRE = /^\/@modules\//

export const moduleResolvePlugin: ServerPlugin = ({ root, app, resolver }) => {
  app.use(async (ctx, next) => {
    const id = decodeURIComponent(ctx.path.replace(moduleRE, ''))
```

```
      const nodeModuleInfo = resolveNodeModule(root, id, resolver)
    }
  }
```

.vue文件处理



- 通过vuePlugin中间件来解析处理(组件的template和style会被处理成单独的的import路径，通过query.type区分)
- 使用@vue/compiler-sfc解析vue文件，重写成新的script以es module形式导出
- 浏览器解析后再次异步请求带有query参数的请求处理template
- 使用@vue/compiler-dom编译，导出render函数处理style
- 借助热更新updateStyle并导出JSON字符串形式的css内容

```
// src/node/server/serverPluginVue.ts

export const vuePlugin: ServerPlugin = ({
  root,
  app,
  resolver,
  watcher,
  config
}) => {
  if (query.type === 'template') {
    const templateBlock = descriptor.template!
    if (templateBlock.src) {
      filePath = await resolveSrcImport(root, templateBlock, ctx, resolver)
    }
    ctx.type = 'js'
  }

  if (query.type === 'style') {}
}
```

HMR

热更新的时候，Vite 只需要立即编译当前所修改的文件即可，所以响应速度非常快。

而 Webpack 修改某个文件过后，会自动以这个文件为入口重写 build 一次，所有的涉及到的依赖也都会被加载一遍，所以反应速度会慢很多。

当request.path 路径是 /vite/client 时、请求得到对应的客户端代码，在客户端中创建了一个**websocket服务**并与服务端建立了连接，通过chokidar这个库创建watcher实例，监听文件变化，不同的消息触发一些事件做到浏览器端的即时热模块更换

```
// src/client/client.ts

async function handleMessage(payload: HMRPayload) {
  const { path, changeSrcPath, timestamp } = payload as UpdatePayload
  switch (payload.type) {
    case 'connected':
      console.log(`[vite] connected.`)
      // proxy(nginx, docker) hmr ws maybe caused timeout, so send ping package
let ws keep alive.
      setInterval(() => socket.send('ping'), __HMR_TIMEOUT__)
      break
    case 'vue-reload':...
    case 'vue-rerender':...
    case 'full-reload':
      if (path.endsWith('.html')) {
        // if html file is edited, only reload the page if the browser is
        // currently on that page.
        const pagePath = location.pathname
        if (
          pagePath === path ||
          (pagePath.endsWith('/') && pagePath + 'index.html' === path)
        ) {
          location.reload()
        }
        return
      } else {
        location.reload()
      }
  }
}
```

一些优化

启动服务前的预优化

启动服务前，会进行预优化，把用户项目的npm依赖打包到node_module下的.vite_opt_cache目录缓存下来，这些js文件里不存在import，不用发起多次请求，利用http缓存可以提高读取node_modules里模块的加载速度。

ctx.read方法会从内存中读取已缓存文件，node_modules模块第一次解析后缓存到内存中，不用每次从磁盘中读取。

```
// src/node/server/index.ts

export function createServer(config: ServerConfig): Server {
  // attach server context to koa context
  app.use((ctx, next) => {
    Object.assign(ctx, context)
    ctx.read = cachedRead.bind(null, ctx)
    return next()
  })
}
```

```
// src/node/server/serverPluginModuleResolve.ts

// plugin for resolving /@modules/:id requests.
export const moduleResolvePlugin: ServerPlugin = ({ root, app, resolver }) => {
  app.use(async (ctx, next) => {
    const serve = async (id: string, file: string, type: string) => {
      moduleIdToFileMap.set(id, file)
      moduleFileToIdMap.set(file, ctx.path)
      debug(`(${type}) ${id} -> ${getDebugPath(root, file)}`)
      await ctx.read(file)
      return next()
    }
  }
}
```

```
// src/node/utils/fsUtils.ts

const fsReadCache = new LRUCache<string, CacheEntry>({
  max: 10000
})

export async function cachedRead(
  ctx: Context | null,
  file: string,
  poll = false
): Promise<Buffer> {
  const lastModified = fs.statSync(file).mtimeMs
  const cached = fsReadCache.get(file)
}
```

http2/https

执行 vite --https 会开启http2协议，多个请求都是通过一个 TCP 连接并发完成

build

这部分代码在 node/build/index.ts 中，build 目录的结构虽然与 server 相似，同样导出一个 build 方法，同样也有许多 plugin，不过这些 plugin 与 server 中的用途不一样，因为 build 使用了 rollup ，所以这些 plugin 也是为 rollup 打包的 plugin。

五、实现一个简单的Vite

原理

Vite 的核心功能：Static Server + Compile + HMR

核心思路：

- 将当前项目目录作为静态文件服务器的根目录
- 拦截部分文件请求
    - 处理代码中 import node_modules 中的模块
    - 处理 vue 单文件组件（SFC）的编译
- 通过 WebSocket 实现 HMR