



下载APP



35 | Vite原理：写一个迷你的Vite

2022-01-10 大圣

《玩转Vue 3全家桶》

课程介绍 >



讲述：大圣

时长 08:44 大小 8.00M



你好，我是大圣。

上一讲学完了 Vue 的编译原理后，我们就把 Vue 的整体流程梳理完毕了，但是我们在使用 Vue 的时候，还会用到很多 Vue 生态的库。所以从今天开始，我会带你了解几个 Vue 生态中重要成员的原理和源码，今天我先带你剖析一下我们项目中用的工程化工具 Vite 的原理。

现在工程化的痛点



现在前端开发项目的时候，工程化工具已经成为了标准配置，webpack 是现在使用率最高的工程化框架，它可以很好地帮助我们完成从代码调试到打包的全过程，但是随着项目规模的爆炸式增长，**webpack 也带来了一些痛点问题。**


最早 webpack 可以帮助我们在 JavaScript 文件中使用 require 导入其他 JavaScript、CSS、image 等文件，并且提供了 dev-server 启动测试服务器，极大地提高了我们开发项目的效率。

webpack 的核心原理就是通过分析 JavaScript 中的 require 语句，分析出当前 JavaScript 文件所有的依赖文件，然后递归分析之后，就得到了整个项目的一个依赖图。对图中不同格式的文件执行不同的 loader，比如会把 CSS 文件解析成加载 CSS 标签的 JavaScript 代码，最后基于这个依赖图获取所有的文件。

进行打包处理之后，放在内存中提供给浏览器使用，然后 dev-server 会启动一个测试服务器打开页面，并且在代码文件修改之后可以通过 WebSocket 通知前端自动更新页面，**也就是我们熟悉的热更新功能。**

由于 webpack 在项目调试之前，要把所有文件的依赖关系收集完，打包处理后才能启动测试，很多大项目我们执行调试命令后需要等 1 分钟以上才能开始调试。这对于开发者来说，这段时间除了摸鱼什么都干不了，而且热更新也需要等几秒钟才能生效，极大地影响了我们开发的效率。所以针对 webpack 这种打包 bundle 的思路，社区就诞生了 bundless 的框架，Vite 就是其中的佼佼者。

前端的项目之所以需要 webpack 打包，是因为**浏览器里的 JavaScript 没有很好的方式去引入其他文件**。webpack 提供的打包功能可以帮助我们更好地组织开发代码，但是现在大部分浏览器都支持了 ES6 的 module 功能，我们在浏览器内使用 type="module" 标记一个 script 后，在 src/main.js 中就可以直接使用 import 语法去引入一个新的 JavaScript 文件。这样我们其实可以不依赖 webpack 的打包功能，利用浏览器的 module 功能就可以重新组织我们的代码。

 复制代码

```
1 <script type="module" src="/src/main.js"></script>
```

Vite 原理

了解了 script 的使用方式之后，我们来实现一个**迷你的 Vite** 来讲解其大致的原理。

首先，浏览器的 module 功能有一些限制需要额外处理。浏览器识别出 JavaScript 中的 import 语句后，会发起一个新的网络请求去获取新的文件，所以只支持 /、./ 和 ../ 开头的路径。

而在下面的 Vue 项目启动代码中，首先浏览器并不知道 Vue 是从哪来，我们第一个要做的，就是分析文件中的 import 语句。如果路径不是一个相对路径或者绝对路径，那就说明这个模块是来自 node_modules，我们需要去 node_modules 查找这个文件的入口文件后返回浏览器。然后 ./App.vue 是相对路径，可以找到文件，但是浏览器不支持 .vue 文件的解析，并且 index.css 也不是一个合法的 JavaScript 文件。

我们需要解决以上三个问题，才能让 Vue 项目很好地在浏览器里跑起来。

[复制代码](#)

```
1 import { createApp } from 'vue'
2 import App from './App.vue'
3 import './index.css'
4
5 const app = createApp(App)
6 app.mount('#app')
7
```


怎么做呢？首先我们需要使用 Koa 搭建一个 server，用来拦截浏览器发出的所有网络请求，才能实现上述功能。在下面代码中，我们使用 Koa 启动了一个服务器，并且访问首页内容读取 index.html 的内容。

[复制代码](#)

```
1 const fs = require('fs')
2 const path = require('path')
3 const Koa = require('koa')
4 const app = new Koa()
5
6 app.use(async ctx=>{
7   const {request:{url,query}} = ctx
8   if(url==='/'){
9     ctx.type="text/html"
10    let content = fs.readFileSync('./index.html','utf-8')
11
12    ctx.body = content
13  }
14 })
15 app.listen(24678, ()=>{
```

```
16 console.log('快来快来数一数，端口24678')
17 })
```


下面就是首页 index.html 的内容，一个 div 作为 Vue 启动的容器，并且通过 script 引入 src.main.js。我们访问首页之后，就会看到浏览器内显示的 geektime 文本，并且发起了一个 main.js 的 HTTP 请求，**然后我们来解决页面中的报错问题。**

 复制代码

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <link rel="icon" href="/favicon.ico" />
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Vite App</title>
8 </head>
9 <body>
10  <h1>geek time</h1>
11  <div id="app"></div>
12  <script type="module" src="/src/main.js"></script>
13 </body>
14 </html>
15
```

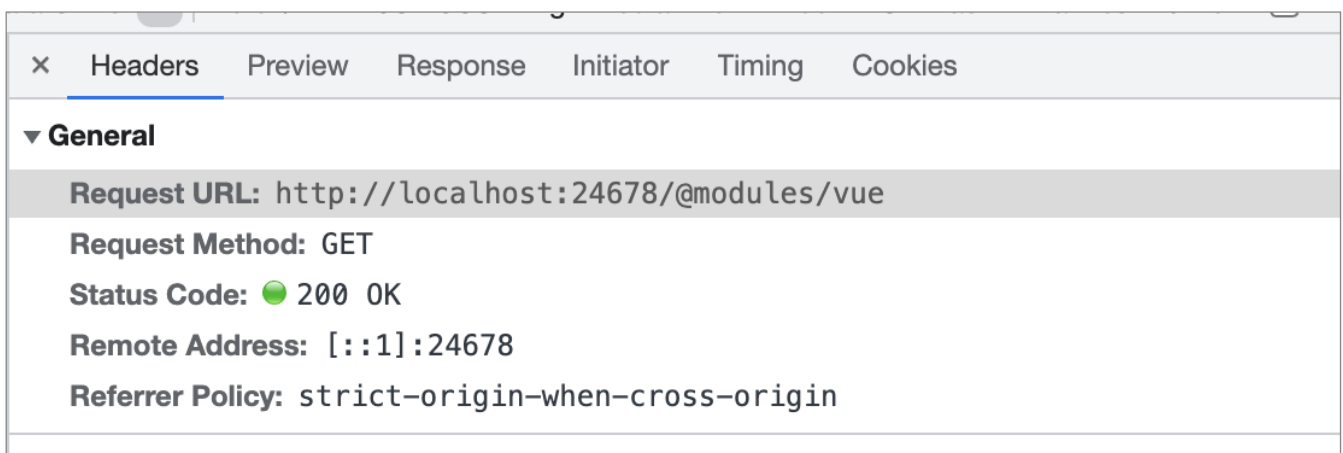
首先 import {createApp} from Vue 这一步由于浏览器无法识别 Vue 的路径，就会直接抛出错误，所以我们要在 Koa 中把 Vue 的路径重写。为了方便演示，我们可以直接使用 replace 语句，把 Vue 改成 /@modules/vue，使用 @module 开头的地址来告诉 Koa 这是一个需要去 node_modules 查询的模块。

在下面的代码中，我们判断如果请求地址是 js 结尾，就去读取对应的文件内容，使用 rewriteImport 函数处理后再返回文件内容。在 rewriteImport 中我们实现了路径的替换，把 Vue 变成了 @modules/vue，现在浏览器就会发起一个 <http://localhost:24678/@modules/vue> 的请求，下一步我们要在 Koa 中拦截这个请求，并且返回 Vue 的代码内容。

 复制代码

```
1 const fs = require('fs')
2 const path = require('path')
3 const Koa = require('koa')
4 const app = new Koa()
```

```
5 function rewriteImport(content){
6   return content.replace(/ from ['"]([^"]+)[']/g, function(s0,s1){
7     // . ../ 开头的, 都是相对路径
8     if(s1[0] !== '.' && s1[1] !== '/'){
9       return ` from '@modules/${s1}'`
10    }else{
11      return s0
12    }
13  })
14 }
15 }
16
17 app.use(async ctx=>{
18   const {request:{url,query}} = ctx
19   if(url === '/'){
20     ctx.type = "text/html"
21     let content = fs.readFileSync('./index.html', 'utf-8')
22
23     ctx.body = content
24   }else if(url.endsWith('.js')){
25     // js文件
26     const p = path.resolve(__dirname, url.slice(1))
27     ctx.type = 'application/javascript'
28     const content = fs.readFileSync(p, 'utf-8')
29     ctx.body = rewriteImport(content)
30   }
31 })
32 app.listen(24678, ()=>{
33   console.log('快来说一本书, 端口24678')
34 })
35
```



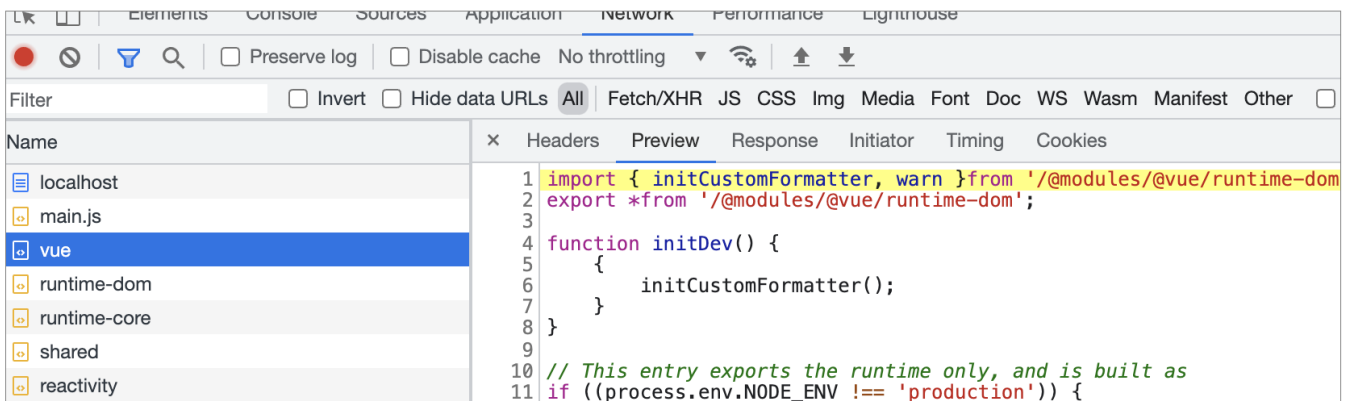
然后我们在 Koa 中判断请求地址, 如果是 @module 的地址, 就把后面的 Vue 解析出来, 去 node_modules 中查询。然后拼接出目标路径 ./node_modules/vue/package.json 去读取 Vue 项目中 package.json 的 module 字

段，这个字段的地址就是 ES6 规范的入口文件。在我们读取到文件后，再使用 `rewriteImport` 处理后返回即可。

这里还要使用 `rewriteImport` 的原因是，Vue 文件内部也会使用 `import` 的语法去加载其他模块。然后我们就可以看到浏览器网络请求列表中多了好几个 Vue 的请求。

[复制代码](#)

```
1 else if(url.startsWith('/@modules/')){
2   // 这是一个node_module里的东西
3   const prefix = path.resolve(__dirname, 'node_modules', url.replace('/@module
4   const module = require(prefix + '/package.json').module
5   const p = path.resolve(prefix, module)
6   const ret = fs.readFileSync(p, 'utf-8')
7   ctx.type = 'application/javascript'
8   ctx.body = rewriteImport(ret)
9 }
```



这样我们就实现了 `node_modules` 模块的解析，然后我们来处理浏览器无法识别 `.vue` 文件的错误。

`.vue` 文件是 Vue 中特有的文件格式，我们上一节课提过 Vue 内部通过 `@vue/compiler-sfc` 来解析单文件组件，把组件分成 `template`、`style`、`script` 三个部分，我们要做的就是 Node 环境下，把 `template` 的内容解析成 `render` 函数，并且和 `script` 的内容组成组件对象，再返回即可。

其中，`compiler-dom` 解析 `template` 的流程我们学习过，今天我们来看下如何使用。

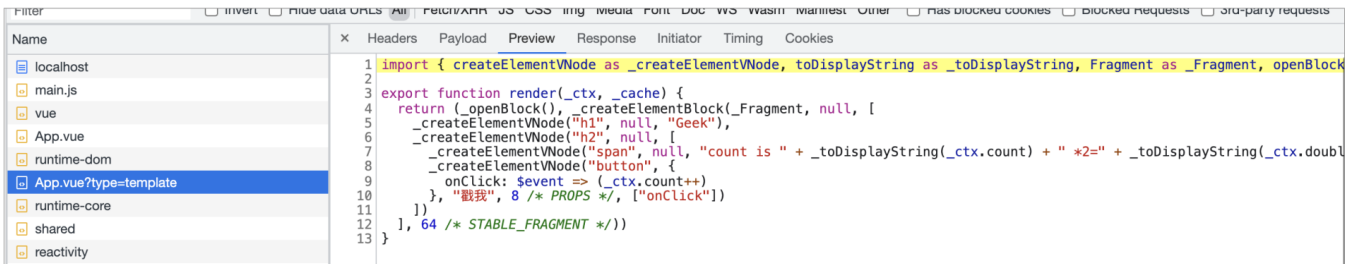
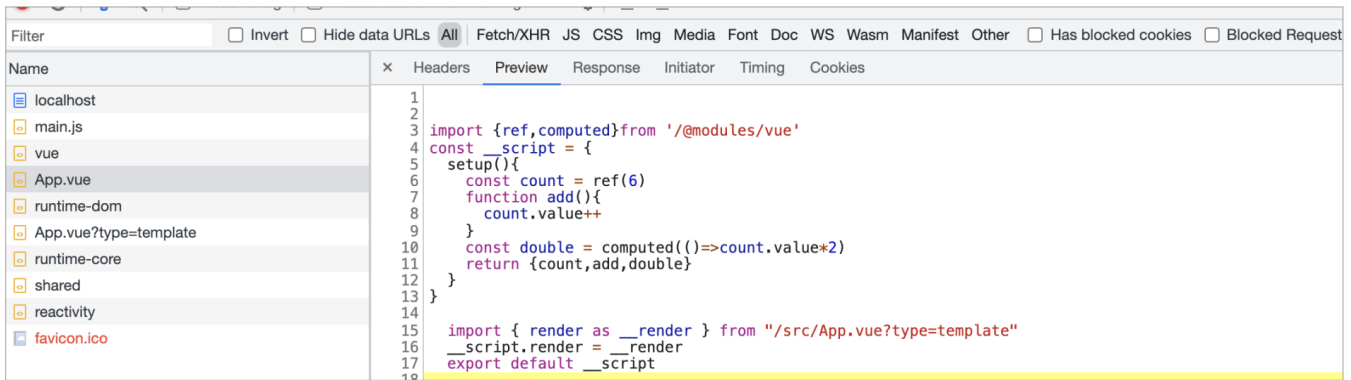
在下面的代码中，我们判断 `.vue` 的文件请求后，通过 `compilerSFC.parse` 方法解析 Vue 组件，通过返回的 `descriptor.script` 获取 JavaScript 代码，并且发起一个

type=template 的方法去获取 render 函数。在 query.type 是 template 的时候，调用 compilerDom.compile 解析 template 内容，直接返回 render 函数。

[复制代码](#)

```
1  const compilerSfc = require('@vue/compiler-sfc') // .vue
2  const compilerDom = require('@vue/compiler-dom') // 模板
3
4
5
6
7
8  if(url.indexOf('.vue')>-1){
9      // vue单文件组件
10     const p = path.resolve(__dirname, url.split('?')[0].slice(1))
11     const {descriptor} = compilerSfc.parse(fs.readFileSync(p,'utf-8'))
12
13     if(!query.type){
14         ctx.type = 'application/javascript'
15         // 借用vue自导的compile框架 解析单文件组件，其实相当于vue-loader做的事情
16         ctx.body = `
17     ${rewriteImport(descriptor.script.content.replace('export default ','const _
18     import { render as __render } from "${url}?type=template"
19     __script.render = __render
20     export default __script
21     `
22     }else if(query.type==='template'){
23         // 模板内容
24         const template = descriptor.template
25         // 要在server端吧compiler做了
26         const render = compilerDom.compile(template.content, {mode:"module"}).co
27         ctx.type = 'application/javascript'
28
29         ctx.body = rewriteImport(render)
30     }
31 }
```

上面的代码实现之后，我们就可以在浏览器中看到 App.vue 组件解析的结果。App.vue 会额外发起一个 App.vue?type=template 的请求，最终完成了整个 App 组件的解析。



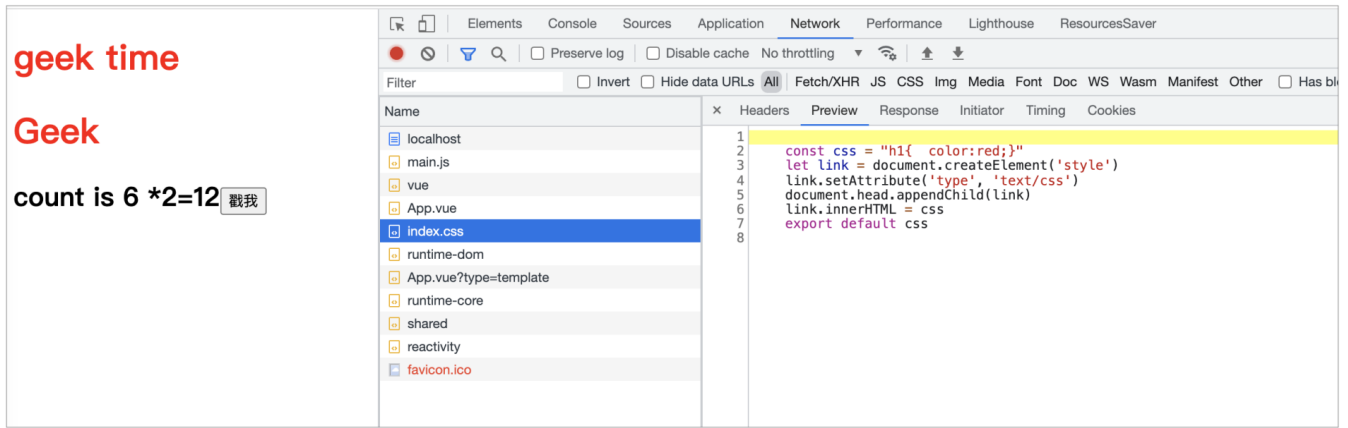
接下来我们再来实现对 **CSS 文件的支持**。下面的代码中，如果 url 是 CSS 结尾，我们就返回一段 JavaScript 代码。这段 JavaScript 代码会在浏览器里创建一个 style 标签，标签内部放入我们读取的 CSS 文件代码。这种对 CSS 文件的处理方式，让 CSS 以 JavaScript 的形式返回，这样我们就实现了在 Node 中对 Vue 组件的渲染。

[复制代码](#)

```

1 if(url.endsWith('.css')){
2   const p = path.resolve(__dirname,url.slice(1))
3   const file = fs.readFileSync(p,'utf-8')
4   const content = `
5   const css = "${file.replace(/\n/g, '\n')}"
6   let link = document.createElement('style')
7   link.setAttribute('type', 'text/css')
8   document.head.appendChild(link)
9   link.innerHTML = css
10  export default css
11  `
12  ctx.type = 'application/javascript'
13  ctx.body = content
14 }

```

Vite 的热更新

最后我们再来看一下热更新如何实现。热更新的目的就是在我们修改代码之后，**浏览器能够自动渲染更新的内容**，所以我们要在客户端注入一个额外的 JavaScript 文件，这个文件用来和后端实现 WebSocket 通信。然后后端启动 WebSocket 服务，通过 chalk 库监听文件夹的变化后，再通过 WebSocket 去通知浏览器即可。

下面的代码中，我们通过 `chokidar.watch` 实现了文件夹变更的监听，并且通过 `handleHMRUpdate` 通知客户端文件更新的类型。

[复制代码](#)

```
1
2 export function watch() {
3   const watcher = chokidar.watch(appRoot, {
4     ignored: ['**/node_modules/**', '**/.git/**'],
5     ignoreInitial: true,
6     ignorePermissionErrors: true,
7     disableGlobbing: true,
8   });
9   watcher;
10
11   return watcher;
12 }
13 export function handleHMRUpdate(opts: { file: string; ws: any }) {
14   const { file, ws } = opts;
15   const shortFile = getShortName(file, appRoot);
16   const timestamp = Date.now();
17
18   console.log(`[file change] ${chalk.dim(shortFile)}`);
19   let updates;
20   if (shortFile.endsWith('.css')) {
21     updates = [
22       {
23         type: 'js-update',
```

```
24     timestamp,
25     path: `/${shortFile}`,
26     acceptedPath: `/${shortFile}`,
27   },
28 ];
29 }
30
31 ws.send({
32   type: 'update',
33   updates,
34 });
35 }
36
```

然后客户端注入一段额外的 JavaScript 代码，判断后端传递的类型是 js-update 还是 css-update 去执行不同的函数即可。

[复制代码](#)

```
1  async function handleMessage(payload: any) {
2    switch (payload.type) {
3      case 'connected':
4        console.log(`[vite] connected.`);
5
6        setInterval(() => socket.send('ping'), 30000);
7        break;
8
9      case 'update':
10       payload.updates.forEach((update: Update) => {
11         if (update.type === 'js-update') {
12           fetchUpdate(update);
13         }
14       });
15       break;
16     }
17   }
18 }
```

总结

以上就是今天的主要内容，我们来总结一下吧！

首先，我们通过了解 webpack 的大致原理，知道了现在 webpack 在开发体验上的痛点。除了用户体验 UX 之外，开发者的体验 DX 也是项目质量的重要因素。

webpack 启动服务器之前需要进行项目的打包，而 Vite 则是可以直接启动服务，通过浏览器运行时的请求拦截，实现首页文件的按需加载，这样开发服务器启动的时间就和整个项目的复杂度解耦。任何时候我们启动 Vite 的调试服务器，基本都可以在一秒以内响应，这极大地提升了开发者的体验，这也是 Vite 的使用率越来越高的原因。

并且我们可以看到，Vite 的主要目的就是提供一个调试服务器。Vite 也可以和 Vue 解耦，实现对任何框架的支持，如果使用 Vite 支持 React，只需要解析 React 中的 JSX 就可以实现。这也是 Vite 项目的现状，我们只需要使用框架对应的 Vite 插件就可以支持任意框架。

Vite 能够做到这么快的原因，还有一部分是因为使用了 esbuild 去解析 JavaScript 文件。esbuild 是一个用 Go 语言实现的 JavaScript 打包器，支持 JavaScript 和 TypeScript 语法，现在前端工程化领域的工具也越来越多地使用 Go 和 Rust 等更高效的语言书写，这也是性能优化的一个方向。

思考题

最后留一个思考题吧。如果一个模块文件是分散的，导致 Vite 首页一下子要加载 1000 个 JavaScript 文件造成卡顿，我们该如何处理这种情况呢？

欢迎在评论区分享你的答案，我们下一讲再见！

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 2  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 34 | 编译原理（下）：编译原理给我们带来了什么？

更多课程推荐

陈天 · Rust 编程第一课

实战驱动，快速上手 Rust

陈天

Tubi TV 研发副总裁



涨价倒计时 🕒

今日订阅 **¥89**，1月12日涨价至**¥199**

精选留言 (2)

💬 写留言



润培

2022-01-10

模块如果是分散的，可以使用“依赖预构建”，通过预构建生成一个模块，这样只会有一个 http 请求。

<https://cn.vitejs.dev/guide/dep-pre-bundling.html>

...

展开 ▾



InfoQ_e521a4ce8a54

2022-01-10

从[Vite 的热更新]开始，代码片段所在的文件目录和所需的依赖就搞不清楚了。。。

