

## 08 | 如何从零搭建自研的Vue组件库？

2022-12-09 杨文坚 来自北京

天下无鱼  
<https://shikey.com/>

《Vue 3 企业级项目实战课》

课程介绍 >



讲述：杨文坚

时长 18:21 大小 16.76M



你好，我是杨文坚。

回顾我们之前的几节课，讲的都是 Vue.js 3.x 的入门级操作。从这节课开始，我们将以 Vue.js 3.x 组件库的开发为线索，展开 Vue.js 3.x 企业级项目的进阶学习。

作为一个前端开发者，你肯定对前端组件库并不陌生。相信你在用 Vue.js 或者 React.js 开发实际项目时，或多或少都使用过相关开源组件库，例如 Vue.js 的 ElementUI 组件库和 React 的 Ant Design 组件库。

前端组件库的出现是为了方便我们实现更多的样式和交互效果。毕竟 JavaScript + HTML + CSS 的原生技术能力比较有限，如果基于原生技术能力来实现网页的样式和交互效果，要付出很大的工作量。而组件库能让前端开发者省去这部分工作量，直接进入页面的功能开发。

Vue.js 3.x 有很多现成的开源组件库，例如 Element Plus 和 Ant Design Vue 等都可以直接拿来用。为什么我们还要自己学 Vue.js 3.x 的组件库开发呢？



## 自研开发 Vue.js 3.x 组件库能带来什么？

**一方面是定制化的需要。**企业产品必定是根据客户或者业务的特色量身定制的，也就是说它有一定的自定义性质，而开源的组件库不一定能满足所有定制化的前端功能。

这个时候，作为前端工程师，你就必须掌握自研组件库的开发能力，为可能出现的定制化组件的要求做好准备。

**另一方面，也能锻炼和提高自己的技术能力，提升工作效率。**

自研组件库的过程中，我们需要考虑很多的技术问题，例如“如何让组件库能开箱即用？”“如何让组件库能定制化主题？”“如何保证组件库里所有组件的复用性和通用性”等等。在实现组件库过程中，你还需要考虑和解决很多业务外的技术问题，这是一个非常好的锻炼技术能力的机会。

而且，组件库开发到最后要沉淀一套 UI 框架，让以后项目直接复用，这就大大减少了以后重复性开发的工作量。例如现在组件库开发了一个通用对话框组件，以后其它项目就不需重新开发，直接复用就行。

说了这么多自研组件库的好处，那我们要怎么学习 Vue.js 3.x 自研组件库的开发呢？有哪些开发要点呢？

## Vue.js 3.x 组件库开发要点

Vue.js 3.x 组件库的核心作用是可以给其它项目复用。既然是复用，那么标准做法就是将组件库封装成 npm 模块进行使用。

所以这个时候，我们就需要考虑将组件库里的 Vue.js 代码编译成 JavaScript 文件，保证能支持引入组件库 npm 模块的同时，还要能自动识别 TypeScript 类型、支持 ES Module 和 CommonJS 模块格式、组件按需加载等操作。

总体来讲，这里可以划分为三个技术要点：

- monorepo 管理组件代码；
- Vue.js 3.x 源文件的多种模块格式编译；
- 基于 Less 开发 CSS 样式文件和独立编译。



## 先看第一点，monorepo 管理组件库代码。

我们在企业中开发组件库的时候，并不会像开源组件库那样，一概而论地将组件库划分为基础功能组件（例如 Button 组件、Dialog 组件），而是要考虑到多业务场景类型，划分成基础功能组件类型、某某业务组件类型等。

这个时候，我们就需要把组件库划分成多个 npm 模块进行输出管理，同时需要尽量用一个代码仓库进行维护，因为不同类型的组件可能存在互相依赖或者引用的关系，要保证能在一个代码仓库中快速调试多个 npm 模块的代码效果。一个仓库管理多个 npm 模块（多个子项目），就需要用到 monorepo 的项目管理形式。

## 第二个要点，了解 Vue.js 3.x 源文件的多种模块格式编译。

刚刚提到，组件库是以 npm 模块的形式提供给其他开发者使用的。开发者在使用组件库的时候，可以有多种 JavaScript 模块的使用格式，常见的是 ES Module 和 CommonJS 模块格式，所以我们要将源码编译成 ES Module 和 CommonJS 格式。

但开发者在使用组件库的时候，不可能一次性把 npm 模块中所有组件都打包进业务项目，这样做会导致业务项目代码编译结果的体积非常大。为了避免冗余代码结果，我们必须支持组件库能够**按需加载**使用。这时候就需要每个组件独立编译，输出单独的文件，也支持使用者在自己的业务项目中单独使用某个 npm 里的某个组件。

上节课我们提到，保证企业项目中代码质量，有一点就是用 TypeScript 进行开发，但编译后的 npm 模块是 JavaScript 代码，怎么能让使用者识别到组件库的代码类型呢？这时就需要在输出 JavaScript 的 npm 模块里，加上编译输出组件库的 JavaScript 的 TypeScript 类型描述文件（\*.d.ts 文件）。

## 第三点，基于 Less 开发 CSS 样式文件和独立编译。

前面说过，企业级的业务项目有很多定制化的需求，特别是前端页面的主题定制化能力，甚至是同一个企业内不同业务项目的主题定制都不一样。



考虑到要支持主题配置，我们采用 **Less** 来进行“编程方式”开发 **CSS**。因为原生 **CSS** 是静态文件，不能像 **JavaScript** 那样有丰富的动态脚本能力可以执行动态的内容，例如函数复用、循环逻辑等特性，导致开发复用度高的 **CSS** 样式需要写很多重复性的代码。所以，这时候要让 **CSS** 能“编程化”，可以使用 **Less** 这个 **CSS** 的“预处理语言”。

**Less** 是 **CSS** 的“预处理语言”，意思是可以让 **CSS** 像写 **JavaScript** 那样支持变量、循环、继承和自定义方法等多种特性，极大提高 **CSS** 的开发效率和样式复用率，最终再通过工具编译成 **CSS** 代码。

这里，考虑到组件也是按需加载，所以不同组件的 **CSS** 样式内容也要独立拆分。

到这儿，**Vue.js 3.x** 组件库自研的开发要点就分析得差不多了，接下来根据三个要点我们一一讲解相关的技术实现，先来看如何搭建 **monorepo** 项目。

## 如何搭建 monorepo 项目？

从前面的分析中，我们多少可以了解到，**monorepo** 项目就是一个仓库管理多个子项目的概念。在前端领域中，用 **monorepo** 管理同仓库多项目的方案有很多种，例如传统的 **Lerna** 技术方案、最近一两年比较流行的 **pnpm** 管理方案。

既然有这么多方案，那要怎么选择呢？我们可以参考 **Vue.js** 官方在 **GitHub** 上的代码仓库 <https://github.com/vuejs/core> 的 **monorepo** 方案选择，就是用 **pnpm** 来管理 **monorepo** 项目，主要利用 **pnpm** 天然支持 **monorepo** 的管理能力，同时 **pnpm** 安装 **node\_modules** 也能更省体积空间。

现在我们开始进入基于 **pnpm** 的 **monorepo** 方案的实现操作：

- 初始化代码目录；
- 基于 **pnpm** 配置 **monorepo** 项目；
- 安装所有子项目依赖。

先看第一步，初始化项目的代码目录，项目的代码基础目录如下所示：



```
1  .
2  |— package.json
3  |— packages/           # 多子项目的目录
4  |   |— business/      # 业务组件库 - 子项目目录
5  |   |   |— package.json # 业务组件库 - 子项目package.json声明
6  |   |   |— src/*       # 业务组件库 - 子项目源码目录
7  |   |— components/    # 基础组件库 - 子项目目录
8  |   |   |— package.json # 基础组件库 - 子项目package.json声明
9  |   |   |— src/*       # 基础组件库 - 子项目源码目录
10 |— pnpm-workspace.yaml
11 |— scripts/*
12 |— tsconfig.json
```

我来讲解一下上述目录中各自的作用：

- 根目录的 `package.json` 主要是用来声明公共的操作脚本和公共的开发编译所需的 `npm` 模块；
- `packages/*` 目录用来管理多个子项目，每个子项目都有各自的 `package.json` 项目声明文件；
- `pnpm-workspace.yaml` 是 `pnpm` 管理项目的配置文件；
- `scripts/*` 目录是用来存放项目通用编译脚本的；
- `tsconfig.json` 是用来声明 `TypeScript` 的项目配置的。

目录初始化还有一个最重要的点，就是对主项目和子项目目录下的 `package.json` 依赖的初始化，我们现在对两个子项目的 `package.json` 文件进行初始化。

我们先初始化“基础组件库”子项目的 `packages/components/package.json` 文件，如下面代码所示：

复制代码

```
1 {
2   "name": "@my/components",
3   "version": "0.0.1",
4   "main": "dist/cjs/index.cjs",
5   "module": "dist/esm/index.mjs",
```



```
6  "types": "dist/esm/index.d.ts",
7  "devDependencies": {
8    "vue": "^3.2.39"
9  },
10 "peerDependencies": {
11   "vue": "^3.2.39"
12 }
13 }
14
```



上述代码中，我们给“基础组件库”子项目加上了“@my”这个私有前缀名称。这个是可以自定义的，你可以根据自己所在企业 npm 源站可使用的前缀名称进行定义，方便统一命名子项目，后续的其他子项目也可以加上同样的“@xxx”的前缀进行统一命名。

注意，这里使用“@xxx”这类私有命名前缀不是必要的操作，但是为了方便管理子项目 npm 模块名称，一般都要加上这类命名前缀。


接下来，我们来初始化“业务组件库”子项目的 packages/business/package.json 文件，如下面代码所示：

 复制代码


```
1  {
2    "name": "@my/business",
3    "version": "0.0.1",
4    "main": "dist/cjs/index.cjs",
5    "module": "dist/esm/index.mjs",
6    "types": "dist/esm/index.d.ts",
7    "dependencies": {
8      "@my/components": "^0.0.1"
9    },
10   "devDependencies": {
11     "vue": "^3.2.39"
12   },
13   "peerDependencies": {
14     "vue": "^3.2.39"
15   }
16 }
```

在“业务组件库”子项目这个 npm 模块依赖里，我们使用了“基础组件库”子项目模块“@my/components”。其实 npm 站点上并不需要存在这个模块，后续通过 pnpm 进行 monorepo 的管理，实现项目子依赖模块 @my/components 直接指向和引用 packages/components 的代码。

第一步完成，我们进行第二步，基于 **pnpm** 配置 **monorepo** 项目。

在 **pnpm-workspace.yaml** 这个文件里，进行 **monorepo** 的项目配置，具体代码如下所示：

```
1 packages:
2   - packages/*
```

 复制代码

声明 **packages/\*** 目录是用来管理所有子项目的。

接着是第三步，安装所有子项目依赖。这里我们要先保证本地电脑有全局的 **pnpm** 命令，可以通过以下脚本进行安装：

```
1 npm i -g pnpm
```

 复制代码

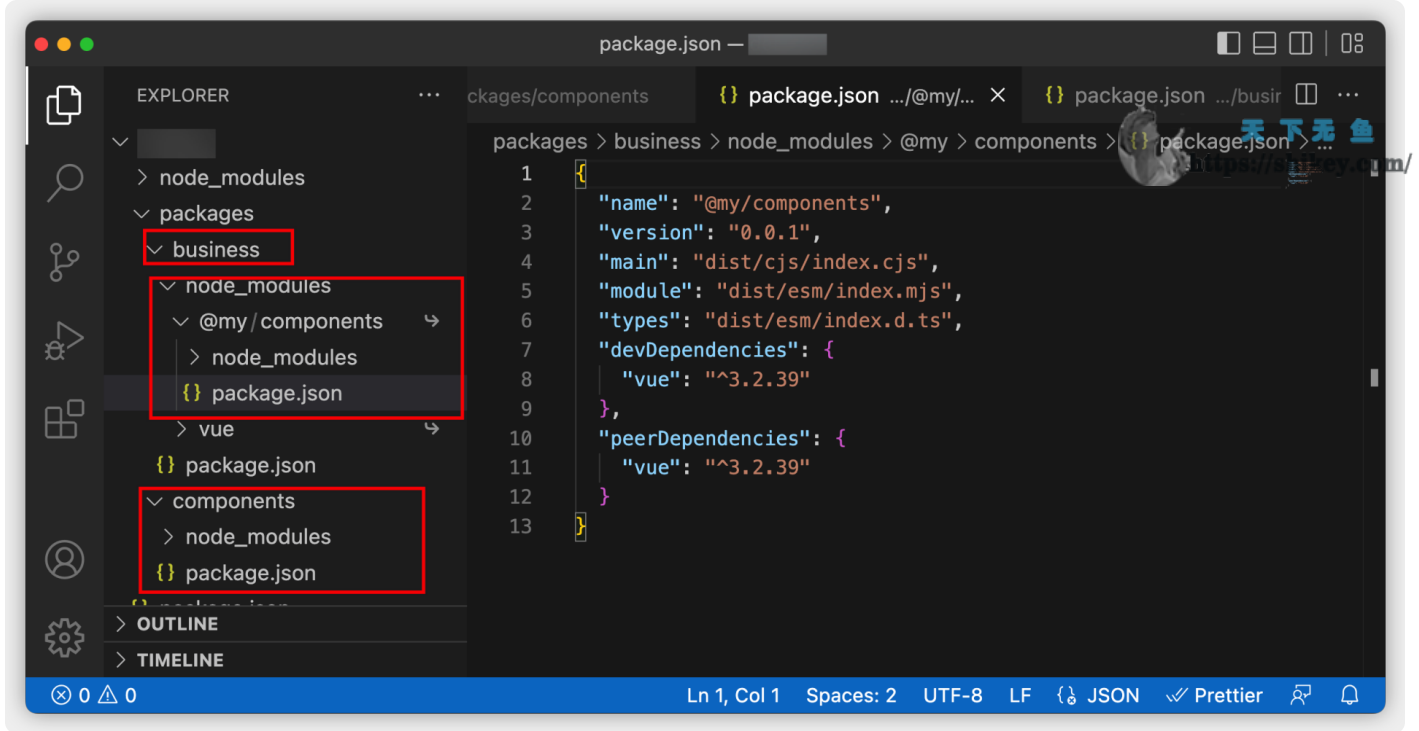
最后再在项目的根目录下执行：

```
1 pnpm i
```

 复制代码

就可以实现 **monorepo** 项目的依赖安装和管理了。

**pnpm** 安装依赖过后，如下图所示：



业务组件库子项目（@my/business）里依赖了基础组件库的子项目（@my/components），通过 pnpm 管理的 monorepo 项目方式，将依赖的 @my/components 子项目通过“软链接”形式指向了真正的 components/\* 目录。

monorepo 里有“软链接”实现子项目的 npm 模块依赖关系，我们就可以放心拆分不同类型组件库，以及管理不同类型组件库的嵌套依赖关系了。

## 怎么对组件库做编译设置？

实现了多种类型组件库项目聚合的 monorepo 项目目录，我们就要开始准备各个子项目的初始化源码和源码的编译脚本。

我们可以这样设计子项目里组件库的文件目录格式：

复制代码

```
1 .
2 |— README.md
3 |— env.d.ts
4 |— package.json
5 |— packages
6 |   |— business/
7 |     |— package.json
8 |     |— src/
9 |       |— comp-000/
10 |         |— xxxx.vue
11 |         |— index.ts
```

# 业务组件库 - 子项目源码目录  
# 业务组件 - 单独组件目录  
# 业务组件 - 组件索引文件







```
1 import fs from 'node:fs';
2 import { rollup } from 'rollup'
3 import vue from '@vitejs/plugin-vue'
4 import vueJsx from '@vitejs/plugin-vue-jsx'
5 import VueMacros from 'unplugin-vue-macros/rollup'
6 import { nodeResolve } from '@rollup/plugin-node-resolve'
7 import commonjs from '@rollup/plugin-commonjs'
8 import esbuild from 'rollup-plugin-esbuild'
9 import glob from 'fast-glob'
10 import type { OutputOptions } from 'rollup';
11 import { resolvePackagePath } from './util';
12
13 const getExternal = async (pkgDirName: string) => {
14   const pkgPath = resolvePackagePath(pkgDirName, 'package.json');
15   const manifest = require(pkgPath) as any;
16   const { dependencies = {}, peerDependencies = {}, devDependencies = {} } = manifest;
17   const deps: string[] = [...new Set([
18     ...Object.keys(dependencies),
19     ...Object.keys(peerDependencies),
20     ...Object.keys(devDependencies)
21 ])]);
22   return (id: string) => {
23     if (id.endsWith('.less')) {
24       return true;
25     }
26     return deps.some(
27       (pkg) => id === pkg || id.startsWith(`${pkg}/`)
28     )
29   }
30 }
31
32 const build = async (pkgDirName: string) => {
33
34   const pkgDistPath = resolvePackagePath(pkgDirName, 'dist');
35   if (fs.existsSync(pkgDistPath) && fs.statSync(pkgDistPath).isDirectory()) {
36     fs.rmSync(pkgDistPath, { recursive: true })
37   }
38
39   const input = await glob([
40     '**/*.{js,jsx,ts,tsx,vue}',
41     '!node_modules'
42   ], {
43     cwd: resolvePackagePath(pkgDirName, 'src'),
44     absolute: true,
45     onlyFiles: true,
46   })
47
48   const bundle = await rollup({
49     input,
50     plugins: [
51       VueMacros({
```

```
52     setupComponent: false,  
53     setupSFC: false,  
54     plugins: {  
55       vue: vue({  
56         isProduction: true,  
57       }),  
58       vueJsx: vueJsx(),  
59     },  
60   }),  
61   nodeResolve({  
62     extensions: ['.mjs', '.js', '.json', '.ts'],  
63   }),  
64   commonjs(),  
65   esbuild({  
66     sourceMap: true,  
67     target: 'es2015',  
68     loaders: {  
69       '.vue': 'ts',  
70     },  
71   }),  
72 ],  
73   external: await getExternal(pkgDirName),  
74   treeshake: false,  
75 })  
76  
77 const options: OutputOptions[] = [  
78   // CommonJS 模块格式的编译  
79   {  
80     format: 'cjs',  
81     dir: resolvePackagePath(pkgDirName, 'dist', 'cjs'),  
82     exports: 'named',  
83     preserveModules: true,  
84     preserveModulesRoot: resolvePackagePath(pkgDirName, 'src'),  
85     sourcemap: true,  
86     entryFileNames: '[name].cjs',  
87   },  
88   // ES Module 模块格式的编译  
89   {  
90     format: 'esm',  
91     dir: resolvePackagePath(pkgDirName, 'dist', 'esm'),  
92     exports: undefined,  
93     preserveModules: true,  
94     preserveModulesRoot: resolvePackagePath(pkgDirName, 'src'),  
95     sourcemap: true,  
96     entryFileNames: '[name].mjs',  
97   }  
98 ]  
99 return Promise.all(options.map((option) => bundle.write(option)))  
100 }  
101  
102 console.log('[TS] 开始编译所有子模块...')  
103 await build('components');
```

```
104 await build('business');
105 console.log('[TS] 编译所有子模块成功!')
106
```




天下无鱼

<https://shikey.com/>

以上的编译脚本，就是基于 Rollup 来遍历所有 Vue.js 3.x 源码文件和 TypeScript 文件，再进行一一对应的编译。

我们最后将编译出来的 CommonJS 模块格式文件命名成 \*.cjs 后缀的 JavaScript 文件，将 ES Module 模块格式文件命名成 \*.mjs 后缀的 JavaScript 文件，为后续的 package.json 输出格式做准备。

最后生成的结果是以组件形式一一对应在每个子项目的 dist 目录下，如下所示：

 复制代码

```
1  .
2  |
3  |--- packages
4  |   |
5  |   |--- components/
6  |   |   |
7  |   |   |--- dist/
8  |   |   |   |
9  |   |   |   |--- esm/
10 |   |   |   |   |
11 |   |   |   |   |--- comp-000/*
12 |   |   |   |   |--- comp-001/*
13 |   |   |   |   |--- comp-002/*
14 |   |   |   |--- cjs/
15 |   |   |   |   |
16 |   |   |   |   |--- comp-000/*
17 |   |   |   |   |--- comp-001/*
18 |   |   |   |   |--- comp-002/*
19 |   |   |--- src/
20 |   |   |   |
21 |   |   |   |--- comp-000/*
22 |   |   |   |--- comp-001/*
23 |   |   |   |--- comp-002/*
24 |   |--- ...
25 |--- ...
```

接下来我们进入第二步，编译出所有 JavaScript 文件的 TypeScript 类型描述文件。这就需要在项目的 scripts/\* 目录下编写以下编译脚本。

脚本文件是 scripts/build-dts.ts，具体代码如下：

 复制代码

```
1 import process from 'node:process'
2 import path from 'node:path';
```

```
3 import fs from 'node:fs'
4 import * as vueCompiler from 'vue/compiler-sfc'
5 import glob from 'fast-glob';
6 import { Project } from 'ts-morph'
7 import type { CompilerOptions, SourceFile } from 'ts-morph'
8 import { resolveProjectPath, resolvePackagePath } from './util';
9
10 const tsWebBuildConfigPath = resolveProjectPath('tsconfig.web.build.json');
11
12 // 检查项目的类型是否正确
13 function checkPackageType(project: Project) {
14   const diagnostics = project.getPreEmitDiagnostics();
15   if (diagnostics.length > 0) {
16     console.error(project.formatDiagnosticsWithColorAndContext(diagnostics))
17     const err = new Error('TypeScript类型描述文件构建失败! ')
18     console.error(err)
19     throw err
20   }
21 }
22
23 // 将*.d.ts文件复制到指定格式模块目录里
24 async function copyDts(pkgDirName: string) {
25   const dtsPaths = await glob(['**/*.d.ts'], {
26     cwd: resolveProjectPath('dist', 'types', 'packages', pkgDirName, 'src'),
27     absolute: false,
28     onlyFiles: true,
29   });
30
31   dtsPaths.forEach((dts: string) => {
32     const dtsPath = resolveProjectPath('dist', 'types', 'packages', pkgDirName
33     const cjsPath = resolvePackagePath(pkgDirName, 'dist', 'cjs', dts);
34     const esmPath = resolvePackagePath(pkgDirName, 'dist', 'esm', dts);
35     const content = fs.readFileSync(dtsPath, { encoding: 'utf8' });
36     fs.writeFileSync(cjsPath, content);
37     fs.writeFileSync(esmPath, content);
38   });
39 }
40
41 // 添加源文件到项目里
42 async function addSourceFiles(project: Project, pkgSrcDir: string) {
43   project.addSourceFileAtPath(resolveProjectPath('env.d.ts'))
44
45   const globSourceFile = '**/*.{js?(x),ts?(x),vue}'
46   const filePaths = await glob([globSourceFile], {
47     cwd: pkgSrcDir,
48     absolute: true,
49     onlyFiles: true,
50   });
51
52   const sourceFiles: SourceFile[] = []
53   await Promise.all([
54     ...filePaths.map(async (file) => {
```

```
55     if (file.endsWith('.vue')) {
56         const content = fs.readFileSync(file, { encoding: 'utf8' })
57         const hasTsNoCheck = content.includes('@ts-nocheck')
58
59         const sfc = vueCompiler.parse(content)
60         const { script, scriptSetup } = sfc.descriptor
61         if (script || scriptSetup) {
62             let content =
63                 (hasTsNoCheck ? '/// @ts-nocheck\n' : '') + (script?.content ?? '')
64
65             if (scriptSetup) {
66                 const compiled = vueCompiler.compileScript(sfc.descriptor, {
67                     id: 'temp',
68                 })
69                 content += compiled.content
70             }
71
72             const lang = scriptSetup?.lang || script?.lang || 'js'
73             const sourceFile = project.createSourceFile(
74                 `${path.relative(process.cwd(), file)}.${lang}`,
75                 content
76             )
77             sourceFiles.push(sourceFile)
78         }
79     } else {
80         const sourceFile = project.addSourceFileAtPath(file)
81         sourceFiles.push(sourceFile)
82     }
83 },
84 ])
85
86 return sourceFiles
87 }
88
89 // 生产Typescript类型描述文件
90 async function generateTypesDefinitions(
91     pkgDir: string,
92     pkgSrcDir: string,
93     outDir: string
94 ){
95     const compilerOptions: CompilerOptions = {
96         emitDeclarationOnly: true,
97         outDir,
98     }
99     const project = new Project({
100         compilerOptions,
101         tsConfigFilePath: tsWebBuildConfigPath
102     })
103
104     const sourceFiles = await addSourceFiles(project, pkgSrcDir)
105     checkPackageType(project);
106     await project.emit({
```



```

107     emitOnlyDtsFiles: true,
108   })
109
110   const tasks = sourceFiles.map(async (sourceFile) => {
111     const relativePath = path.relative(pkgDir, sourceFile.getFilePath())
112
113     const emitOutput = sourceFile.getEmitOutput()
114     const emitFiles = emitOutput.getOutputFiles()
115     if (emitFiles.length === 0) {
116       throw new Error(`异常文件: ${relativePath}`)
117     }
118
119     const subTasks = emitFiles.map(async (outputFile) => {
120       const filepath = outputFile.getFilePath()
121       fs.mkdirSync(path.dirname(filepath), {
122         recursive: true,
123       });
124     })
125
126     await Promise.all(subTasks)
127   })
128   await Promise.all(tasks)
129 }
130
131 async function build(pkgDirName) {
132   const outDir = resolveProjectPath('dist', 'types');
133   const pkgDir = resolvePackagePath(pkgDirName);
134   const pkgSrcDir = resolvePackagePath(pkgDirName, 'src');
135   await generateTypesDefinitions(pkgDir, pkgSrcDir, outDir);
136   await copyDts(pkgDirName);
137 }
138
139 console.log('[Dts] 开始编译d.ts文件...')
140 await build('components');
141 await build('business');
142

```

以上代码是基于两个环节进行操作的，第一个环节是基于 vue/compiler-sfc 的 Vue.js 3.x 编译器，将 Vue.js 源码编译成 TypeScript 代码，第二环节是结合原有其它 TypeScript 代码文件，进行 TypeScript 的类型文件生成。

具体结果如下所示：

 复制代码

```

1  .
2  |— ...
3  |— packages
4  |   |— components/
5  |   |   |— dist/

```



```

22
23 async function build(pkgDirName: string) {
24   const pkgDir = resolvePackagePath(pkgDirName, 'src');
25   const filePaths = await glob(['**/style/index.less'], {
26     cwd: pkgDir,
27   });
28   const indexLessFilePath = resolvePackagePath(pkgDirName, 'src', 'index.less')
29   if (fs.existsSync(indexLessFilePath)) {
30     filePaths.push('index.less')
31   }
32   for (let i = 0; i < filePaths.length; i++) {
33     const file = filePaths[i];
34     const absoluteFilePath = resolvePackagePath(pkgDirName, 'src', file);
35     const cssContent = await compileLess(absoluteFilePath);
36     const cssPath = resolvePackagePath(pkgDirName, 'dist', 'css', file.replace(
37       wirteFile(cssPath, cssContent);
38   }
39 }
40 }
41 console.log('[CSS] 开始编译Less文件...')
42 await build('components');
43 await build('business');
44 console.log('[CSS] 编译Less成功! ')

```



天下无鱼  
<https://shikey.com/>

上述代码，主要是将 Less 文件以组件固定的目录格式一一对应编译到 dist 目录里，具体编译结果如下所示：

复制代码

```

1  .
2  |
3  |--- packages
4  |   |
5  |   |--- components/
6  |   |   |
7  |   |   |--- dist/
8  |   |   |   |
9  |   |   |   |--- css/
10 |   |   |   |   |
11 |   |   |   |   |--- comp-000/
12 |   |   |   |   |   |
13 |   |   |   |   |   |--- style
14 |   |   |   |   |   |   |
15 |   |   |   |   |   |   |--- index.css
16 |   |   |   |   |   |   |
17 |   |   |   |   |   |   |--- **
18 |   |   |   |   |   |
19 |   |   |   |   |--- esm/
20 |   |   |   |   |   |
21 |   |   |   |   |   |--- comp-000/
22 |   |   |   |   |   |   |
23 |   |   |   |   |   |   |--- **
24 |   |   |   |   |   |
25 |   |   |   |   |--- cjs/
26 |   |   |   |   |   |
27 |   |   |   |   |   |--- comp-000/
28 |   |   |   |   |   |   |
29 |   |   |   |   |   |   |--- **
30 |   |   |   |   |   |
31 |   |   |   |--- src/
32 |   |   |   |   |
33 |   |   |   |   |--- comp-000/
34 |   |   |   |   |   |
35 |   |   |   |   |   |--- style/
36 |   |   |   |   |   |   |
37 |   |   |   |   |   |   |--- index.less

```



通过上述三步操作，我们可以发现最终编译结果是存在每个子项目里 `dist/esm/*` 目录、`dist/cjs/*` 目录和 `dist/css/*` 目录里，而且每个组件在这三个目录的位置都是可以一一对应找到的。

当我们把 `Vue.js` 组件库的源码都编译成 `JavaScript` 和 `CSS` 代码后，接下来我们就需要进入到使用组件库的环节了，也就是在其他项目中使用我们本项目里的组件库。

## 在其他项目中使用组件库

在讲解之前，我们回顾刚刚讲过的组件库开发要点，其中一个是可以支持组件的按需加载，也就是其他项目使用组件库的时候，可以按照自己需要使用个别组件，而不会把整个组件库全量打包。

所以在组件库发布到 `npm` 企业内部站点的时候，我们还需要在组件库子项目每个 `package.json` 文件里加上以下配置：

[复制代码](#)

```
1 {  
2   "main": "dist/cjs/index.cjs",  
3   "module": "dist/esm/index.mjs",  
4   "types": "dist/esm/index.d.ts",  
5   "exports": {  
6     ".": {  
7       "require": "./dist/cjs/index.cjs",  
8       "import": "./dist/esm/index.mjs",  
9       "types": "./dist/esm/index.d.ts"  
10    },  
11    "./esm/*": {  
12      "import": "./dist/esm/*/index.mjs",  
13      "types": "./dist/esm/*/index.d.ts"  
14    },  
15    "./cjs/*": {  
16      "require": "./dist/cjs/*/index.cjs",  
17      "types": "./dist/cjs/*/index.d.ts"  
18    },  
19    "./css/*": "./dist/css/*"  
20  }  
21 }
```

以上配置就是可以让使用者在使用组件库的时候，可以全量使用，例如 ES Module 格式使用：



复制代码

```
1 import { Comp001, Comp002 } from '@my/components'
2 import '@my/components/css/index.css'
```

换成 CommonJS 格式使用：

复制代码

```
1 const { Comp001, Comp002 } = require('@my/components');
2 require('@my/components/css/index.css')
```

也可以按需使用组件库，避免出现构建器对代码全量打包，例如 ES Module 格式使用：

复制代码

```
1 import Comp001 from '@my/components/esm/comp-001';
2 import Comp002 from '@my/components/esm/comp-002';
3 import '@my/components/css/comp-001/style/index.css';
4 import '@my/components/css/comp-002/style/index.css'
```

换成 CommonJS 格式使用：

复制代码

```
1 const Comp001 = require('@my/components/cjs/comp-001');
2 const Comp002 = require('@my/components/cjs/comp-002');
3 require('@my/components/css/comp-001/style/index.css');
4 require('@my/components/css/comp-002/style/index.css');
```

至此，我们就完成 Vue.js 3.x 自研组件库的开发入门了。

## 总结

这节课核心是想带你学会 Vue.js 3.x 自研组件库的开发入门，这也是企业级 Vue.js 3.x 项目的进阶部分的第一课。自研组件库的重要性，简单来讲就是开源组件库满足不了企业的定制化需

求，需要自研才能满足企业的特色组件库的需要。

我们总结一下组件库开发的三个要素：



- 用 **monorepo** 管理多种类型组件库，这类项目的代码管理方式，可以一个仓库同时聚合管理多个项目，让项目之间代码依赖使用更方便；
- 源码要编译成多种模块格式（**CommonJS** 和 **ES Module**），主要考虑到前端代码 **npm** 模块的时候，目前主流是 **ES Module** 模块格式，但还是存在很多传统的 **CommonJS** 模块格式的使用兼容。所以在开发自研组件库的时候，尽量要考虑这两种模块格式；
- 基于 **Less** 等预处理 **CSS** 语言来开发组件库的样式，由于 **CSS** 语言能力有限，无法像 **JavaScript** 那样可以使用各种编程逻辑和特性，所以需要借助 **CSS** 预处理语言进行开发 **CSS**。

以上都是大厂内部实现组件库或者开源社区实现组件库的主流技术方案，同时本节课最后也根据主流组件库编译技术方案，通过实际的编译脚本实现，给你演示了如何进行源码编译，以及编译后组件库的结果目录的规范设计和作用。


这节课只是组件库开发入门，后续会逐步进阶增加难度，希望你能掌握本节课要点，为后续进阶学习打下扎实的技术基础。

## 思考题

组件库的按需加载实现方式，还有其它的方案吗？欢迎在留言区参与讨论，期待你的回答，我们下一讲见。

[🔗 完整的代码在这里](#)

分享给需要的人，Ta购买本课程，你将得 **18** 元

 生成海报并分享

 赞 2  提建议



## 精选留言 (4)

写留言



杜子

2022-12-09 来自福建

看懵了，要多看几遍才行



风太大太大

2022-12-09 来自湖北

按需加载实现方式，

1. 文中提及的方案，手动按需加载。

```
import { Comp001, Comp002 } from '@my/components'
import '@my/components/css/index.css'
```

2. 社区还有一种方案，利用babel的能力，（babel-plugin-import）

代码的话就是 `import { Comp001, Comp002 } from '@my/components'`，安装了这个插件，可以不写引入css的文件，但其实本质还是工具帮我们做了引入的事情的事情，将多个文件打包成一个文件。

不过类似全量引入的情况，例如引用elementUi 如果一开始你就直接`Vue.use(ElementUi)`这样就起不到按需加载的作用了就是全量使用了，所以需要注意。

3. 我观察到loadsh关于按需加载的其他方案，例如loadsh.throttle，代码如下：`import throttle from loadsh.throttle`。看了一下代码，这块应该是开发者二次再上传loadsh.throttle到npm里了，所以这块也增加了维护负担，部署loadsh的时候需要二次部署

坦白的说我，我心目中更友好的是方案3，需要我们上传npm包的时候上传写一段脚本，执行上传子包的地部署方案。最后代码如下，然后开发者使用的时候也是按需加载就可以了

```
import Comp001 from '@my/components.Comp001'
```



初煜



2022-12-09 来自陕西

学到很多原来不理解的组件库的知识，感谢老师。



<https://shikey.com/>



**Johnson**

2022-12-09 来自福建

很实用

