

# vue工具库搭建

## 技术栈

vue3 + vite + 文档产出框架（内置 vitePress、vuePress）的 monorepo 项目

内容：公共的组件、方法、hooks

## 使用手册

1. 字符串全局替换 vmono-seed -> 你的项目名(将作为所有子包的名称前缀)
2. 删除 README.pdf、更新 README.md 内容

## 整体框架搭建

### 项目初始化

1. 创建项目目录 vmono-seed
2. 运行 pnpm init，编辑部分字段

代码块

```
1  {
2    "name": "vmono-seed",
3    "version": "0.0.0",
4    "type": "module",
5    "author": "astfn",
6    "description": "A project that includes common components from the H5
project (dependent on vant) and some utility functions",
7    "scripts": {},
8    "keywords": [],
9    "license": "ISC",
10 }
```

3. 创建 monorepo 工作区配置文件：`pnpm-workspace.yaml` 其中配置的包目录，后续可以在整个项目中共享，实时引入最新代码。

```
1 packages:
2   - 'internal/*'
3   - 'packages/*'
4   - 'vuepress-docs'
```

- internal 用于放一些公共的内部配置
  - eslint-config
  - ts-config
- packages 就是维护的工具包
  - 目前统一放在 cpn-kit 目录中，后续可以将工具函数单独抽出去。
- vuepress-docs
  - 用于文档产出，使用 vuepress 构建

#### 4. 创建对应的包目录

代码块

```
1 vmono-seed/
2   └─ internal/
3     └─ eslint-config/      # 通用的 eslint 配置
4     └─ ts-config/          # 通用的 ts 规则配置
5   |
6   └─ packages/
7     └─ cpn-kit/             # Vue 工具库 (组件 + Hook + 方法)
8   |
9   └─ vuepress-docs/         # VuePress 文档站点
10  |
11  └─ pnpm-workspace.yaml     # pnpm Monorepo 配置
12  └─ package.json
13  └─ README.md
```

#### 5. 进入 cpn-kit 工具包，初始化 vite 项目

代码块

```
1 cd packages/cpn-kit
2 pnpm init
```

修改部分字段

代码块

```
1  {
2    "name": "@vmono-seed/cpn-kit",
3    "version": "0.0.0",
4    "type": "module",
5    "description": "A project that includes common components from the H5
project (dependent on vant) and some utility functions",
6    "main": "index.ts",
7    "scripts": {},
8    "keywords": [],
9    "author": "astfn",
10   "license": "ISC",
11 }
12
```

## 安装依赖

代码块

```
1  pnpm add -D typescript vite @vitejs/plugin-vue vue vue-tsc
```

创建 `vite.config.ts` （顺便创建下入口文件 `index.ts` 做预留）

代码块

```
1  import { defineConfig } from 'vite';
2  import vue from '@vitejs/plugin-vue';
3
4  export default defineConfig({
5    plugins: [vue()],
6    build: {
7      lib: {
8        entry: './src/index.ts',
9        name: 'VueUtils',
10       fileName: (format) => `cpn-kit.${format}.js`,
11     },
12     rollupOptions: {
13       external: ['vue'],
14       output: {
15         globals: {
16           vue: 'Vue',
17         },
18       },
19     },
20   },
21 });
```

## 6. Placeholder

### 配置 tsconfig

#### 配置在哪

可以配置在全局，也可以配置在各个子包中。如果都配置了，则以当前包的为准。

由于在项目根层级中，目前不需要编写额外的 ts 代码，所以目前只在子包 (cpn-kit) 中配置即可

#### 配置复用

如果后续新增其它工具包 (例如把常用的工具函数、hooks单独抽成一个包)，那这些 tsconfig 都是通用的，保持风格一致。

因此有必要单独抽离一下，然后在子包中引入这些配置。



在抽离某些配置之前，要先看看这些配置是否支持插拔式引入。

tsconfig.json 文件是支持 extends 配置项的，可以直接引入外部包，继承其配置。因此我们的想法才可以进行实践。

#### 配置抽离

在 internal/ts-config 中进行工具包的初始化

代码块

```
1  pnpm init
```

并修改 package.json 部分字段

代码块

```
1  {
2    "name": "@vmono-seed/ts-config",
3    "version": "0.0.0",
4    "author": "astfn",
5    "private": true,
6    "files": [
7      "tsconfig.json",
8      "tsconfig.app.json",
9      "tsconfig.node.json"
10   ]
11 }
```

新建 tsconfig.json 文件，将 tsconfig.app.json、tsconfig.node.json 再抽成单独的文件配置

代码块

```
1  {
2    "files": [],
3    "references": [{ "path": "./tsconfig.app.json" }, { "path":
4      "./tsconfig.node.json" }]
5  }
```

新建 tsconfig.app.json 文件

代码块

```
1  {
2    "compilerOptions": {
3      "target": "ES2020",
4      "noImplicitAny": false,
5      "useDefineForClassFields": true,
6      "module": "ESNext",
7      "lib": ["ES2020", "DOM", "DOM.Iterable"],
8      "skipLibCheck": true,
9      "baseUrl": ".",
10     "moduleResolution": "bundler",
11     "allowImportingTsExtensions": true,
12     "isolatedModules": true,
13     "moduleDetection": "force",
14     "noEmit": true,
15     "jsx": "preserve",
16     "sourceMap": true,
17     "strict": true,
18     "noUnusedLocals": true,
19     "noUnusedParameters": true,
20     "noFallthroughCasesInSwitch": true
21   },
22   "include": ["src/**/*.ts", "src/**/*.tsx", "src/**/*.vue", "*.d.ts"]
23 }
24
```

新建 tsconfig.node.json 文件

代码块

```
1  {
```

```

2    "compilerOptions": {
3      "target": "ES2022",
4      "lib": ["ES2023"],
5      "module": "ESNext",
6      "skipLibCheck": true,
7      "moduleResolution": "bundler",
8      "allowImportingTsExtensions": true,
9      "isolatedModules": true,
10     "moduleDetection": "force",
11     "noEmit": true,
12     "noUnusedLocals": true,
13     "noUnusedParameters": true,
14     "noFallthroughCasesInSwitch": true
15   },
16   "include": ["vite.config.ts"]
17 }
18

```

## 使用配置

进入 cpn-kit 包，先把抽离的 tsconfig 依赖添加到 package.json 中，并执行 `pnpm i` 进行下载

代码块

```

1  {
2    "name": "@vmono-seed/cpn-kit",
3    .....,
4    "devDependencies": {
5      "@vmono-seed/ts-config": "workspace:*",
6      .....
7    }
8  }
9

```

新建 tsconfig.json 文件，同理也将 tsconfig.app.json、tsconfig.node.json 再抽成单独的文件配置

- 复用 @vmono-seed/ts-config 中 tsconfig.json 配置

代码块

```

1  {
2    "extends": "@vmono-seed/ts-config/tsconfig.json",
3    "files": [],
4    "references": [{ "path": "./tsconfig.app.json" }, { "path":
5      "./tsconfig.node.json" }]
6  }

```

新建 `tsconfig.app.json` 文件，复用 `@vmono-seed/ts-config` 中 `tsconfig.app.json` 配置的同时，再针对该包，新增一些配置

代码块

```
1  {
2    "extends": "@vmono-seed/ts-config/tsconfig.app.json",
3    "compilerOptions": {
4      "baseUrl": ".",
5      "paths": {
6        "@/*": ["src/*"]
7      }
8    },
9    "include": ["src/**/*.ts", "src/**/*.tsx", "src/**/*.vue", "*.d.ts"]
10 }
```

新建 `tsconfig.node.json` 文件，复用 `@vmono-seed/ts-config` 中 `tsconfig.node.json` 配置的同时，再针对该包，新增一些配置

代码块

```
1  {
2    "extends": "@vmono-seed/ts-config/tsconfig.node.json",
3    "include": ["vite.config.ts"]
4  }
```

到这里就把 `cpn-kit` 包的 `tsconfig` 配置完了，如果后续新增其它子包，可遵循相同的配置过程。

## 配置 eslint

### 配置在哪

可以配置在全局，也可以配置在各个子包中。如果都配置了，则以当前包的为准。

由于我们是在一个大的 `monorepo` 项目中，`eslint` 风格对于所有包来说应该是要一致的，不像 `tsconfig` 那样需要针对不同包需要个性化配置。

因此配置在项目根目录中，再加上 `husky` 其实就已经可以实现整个项目在代码提交时遵循同一套 `lint` 规则进行代码校验。

但在进行实际开发过程中，你会发现，尽管在根项目中配置了 `eslint`，但 `VSCode` 的 `ESLint` 插件默认只会查找当前打开文件所在目录下的 `eslint` 配置文件，如果找不到，就不会激活 `ESLint` 校验。

而你在子包中没有配置 `eslint.config.js`，所以 VSCode 不知道要使用根项目的 ESLint 配置，因此在进行子包开发时，如果违反了 eslint 规则，但是编辑器不会爆红，只会在提交时，通过控制台看到错误。

因此为了达到最佳的开发体验，我们还是要在各个子包中配置 `eslint.config.js`。

## 配置复用



同理，在抽离配置之前，要先看看这些配置是否支持插拔式引入。

当前最新版本 [eslint9.x](#) 统一改为了使用 js 配置，配置风格也变成了扁平化的形式。既然是 js 配置文件，意味着我们可以将配置抽离为方法，供外部使用，并通过函数传参实现更灵活的配置。

上面我们已经说过了，为了达到最佳开发体验，项目根目录和各个子包都要配置 eslint，而现在 eslint 配置支持可插拔形式，就大大降低了维护成本，并且还能在复用全局配置的同时，对子包的 lint 规则进行定制化，虽然目前没有定制需求。

## 配置抽离

在 `internal/eslint-config` 中进行工具包的初始化

代码块

```
1  pnpm init
```

并修改 `package.json` 部分字段

代码块

```
1  {
2    "name": "@vmono-seed/eslint-config",
3    "version": "0.0.0",
4    "author": "astfn",
5    "type": "module",
6    "private": true,
7    "main": "index.js",
8    "files": [
9      "index.js"
10   ],
11 }
```

安装依赖



```
代码块
npm add @eslint/js eslint-config-prettier eslint-plugin-vue globals
typescript-eslint vue-eslint-parser
```

## 新建 index.js 文件

- 将 ts 相关的 lint 规则抽离到 ./tsLintConfig.js 中
- 将 vue 相关的 lint 规则抽离到 ./vueLintConfig.js 中

代码块

```
1  import globals from 'globals';
2  import eslint from '@eslint/js';
3  import eslintConfigPrettier from 'eslint-config-prettier';
4  import { genTsLintConfig } from './tsLintConfig.js';
5  import { genVueLintConfigArr } from './vueLintConfig.js';
6
7  export * from './tsLintConfig.js';
8  export * from './vueLintConfig.js';
9
10 export default [
11   {
12     ignores: ['.history/**', '.husky/**', '.vscode/**', 'coverage/**',
13     'lib/**', 'public/**', 'node_modules/**'],
14   },
15   { languageOptions: { globals: { ...globals.browser, ...globals.node } } },
16   eslint.configs.recommended,
17   ...genTsLintConfig(),
18   ...genVueLintConfigArr(),
19   eslintConfigPrettier,
20 ];
```

## 新建 tsLintConfig.js

代码块

```
1  import tseslint from 'typescript-eslint';
2
3  export const genTsNormalRules = ({ customRules }) => {
4    return {
5      'no-sparse-arrays': 'off',
6      '@typescript-eslint/no-explicit-any': 'off',
7      '@typescript-eslint/no-unused-expressions': 'off',
8      '@typescript-eslint/no-unsafe-function-type': 'off',
9      '@typescript-eslint/consistent-type-imports': 'off',
10     'no-console': ['error', { allow: ['warn', 'error'] }],
11     '@typescript-eslint/no-unused-vars': [
```

```

12     'error',
13     {
14         argsIgnorePattern: '^_',
15         varsIgnorePattern: '^_',
16     },
17 ],
18 ...(customRules ?? {}),
19 };
20 };
21
22 export const genTsLintConfig = ({ customRules } = {}) => {
23     return [
24         ...tseslint.configs.recommended,
25         {
26             files: ['**/*.ts'],
27             plugins: {
28                 '@typescript-eslint': tseslint.plugin,
29             },
30             languageOptions: {
31                 parser: tseslint.parser,
32             },
33             rules: genTsNormalRules({ customRules }),
34         },
35     ];
36 };

```

## 新建 vueLintConfig.js

### 代码块

```

1  import vueParser from 'vue-eslint-parser';
2  import { genTsNormalRules } from './tsLintConfig.js';
3  import pluginVue from 'eslint-plugin-vue';
4  import tseslint from 'typescript-eslint';
5
6  export const genVueNormalRules = ({ customRules }) => {
7      return {
8          'vue/no-v-html': 'off',
9          'vue/attributes-order': 'off',
10         'vue/require-emit-validator': 'warn',
11         'vue/multi-word-component-names': 'off',
12         'vue/no-setup-props-destructure': 'off',
13         'vue/v-on-event-hyphenation': 'off',
14         'vue/no-mutating-props': 'off',
15         'vue/html-self-closing': [
16             'error',

```

```

17     {
18       html: {
19         void: 'always',
20         normal: 'never',
21         component: 'always',
22       },
23       svg: 'always',
24       math: 'always',
25     },
26   ],
27   ...(customRules ?? {}),
28 };
29 };
30
31 export const genVueLintConfigArr = ({ customRules } = {}) => {
32   return [
33     ...pluginVue.configs['flat/recommended'],
34     {
35       files: ['**/*.vue'],
36       plugins: {
37         '@typescript-eslint': tseslint.plugin,
38       },
39       languageOptions: {
40         parser: vueParser, // 使用vue解析器, 这个可以识别vue文件
41         parserOptions: {
42           parser: tseslint.parser, // 在vue文件上使用ts解析器
43           sourceType: 'module',
44         },
45       },
46       rules: {
47         ...genTsNormalRules({}),
48         ...genVueNormalRules({ customRules }),
49       },
50     },
51   ];
52 };

```

## 使用配置

分别在全局、子包中执行以下步骤

1. 在 package.json 的 devDependencies 中添加 "@vmono-seed/eslint-config": "workspace:\*" , 并执行 `pnpm i` 下载
2. 新建 eslint.config.js 文件, 引入公共配置, 并复用

代码块

```
1  import eslintConfig from '@vmono-seed/eslint-config';
2
3  export default eslintConfig;
```

## 配置 husky & prettier

这两个配置都是全局性质的，只需要在根目录进行统一配置即可

在根目录安装 husky 和 prettier

代码块

```
1  pnpm add -D -w husky prettier
```

## prettier



如果不同子包需要定制 prettier，也在子包中配置 prettier 即可，会以子包中的为准。

配置 .prettierrc 文件

代码块

```
1  {
2    "semi": true,
3    "singleQuote": true,
4    "trailingComma": "es5",
5    "printWidth": 120,
6    "tabWidth": 2,
7    "useTabs": false,
8    "bracketSpacing": true,
9    "jsxBracketSameLine": true,
10   "arrowParens": "always",
11   "endOfLine": "lf"
12 }
```

配置 .prettierignore

代码块

```
1  .history
```

```
2  .husky
3  .vscode
4  coverage
5  dist
6  node_modules
7  public
```

## husky

配置 .lintstagedrc 文件

代码块

```
1  {
2    "*.{js?(x),ts?(x),vue,json}": ["npm run format", "npm run lint"],
3    "*.{html,css,less,scss}": ["npm run format"]
4  }
```

更新根目录 package.json 的脚本指令

代码块

```
1  {
2    "name": "vmono-seed",
3    .....,
4    "scripts": {
5      "prepare": "husky install",
6      "format": "prettier --config .prettierrc . --write",
7      "lint": "eslint --config eslint.config.js"
8    }
9  }
10
```

运行 `pnpm i`，会自动生成 `.husky` 目录，在其中创建 `pre-commit` (做提交前的代码校验逻辑)、`commit-msg` (做提交时 message 的校验逻辑)

pre-commit 文件

代码块

```
1  echo "🔧 lint-staged 正在执行 🔧"
2  npx lint-staged --quiet
3  echo "🎉 lint-staged 检测完毕,通过校验 🎉"
```

## commit-msg 文件

代码块

```
1 echo "commit msg 验证中🕒"  
2 npx --no-install commitlint --edit "$1"  
3 echo "commit msg 验证通过🌞"
```

## 安装 commit-msg 所需依赖

- @commitlint/config-conventional 是比较通用的校验规则包

代码块

```
1 pnpm add -D -w @commitlint/cli @commitlint/config-conventional
```

在根目录创建 .commitlintrc 配置文件，继承通用的规则校验包

代码块

```
1 {  
2   "extends": [  
3     "@commitlint/config-conventional"  
4   ]  
5 }
```

## 其它基础配置

### .npmrc

代码块

```
1 registry=https://registry.npmmirror.com
```

### .gitignore

代码块

```
1 # Logs  
2 logs  
3 *.log
```

```
4  npm-debug.log*
5  yarn-debug.log*
6  yarn-error.log*
7  pnpm-debug.log*
8  lerna-debug.log*
9
10 node_modules
11 dist
12 dist-ssr
13 *.local
14
15 # Editor directories and files
16 .vscode/*
17 !.vscode/extensions.json
18 .idea
19 .DS_Store
20 *.suo
21 *.ntvs*
22 *.njsproj
23 *.sln
24 *.sw?
```

## .gitattributes

代码块

```
1  # 告诉 Git: 这些是文本文件, 并统一使用 LF 换行符
2  *.txt text eol=lf
3  *.md text eol=lf
4  *.js text eol=lf
5  *.ts text eol=lf
6  *.tsx text eol=lf
7  *.vue text eol=lf
8  *.json text eol=lf
9  *.json5 text eol=lf
10 *.mjs text eol=lf
11 *.cjs text eol=lf
12 *.css text eol=lf
13 *.scss text eol=lf
14 *.html text eol=lf
15 *.yaml text eol=lf
16 *.yml text eol=lf
17 *.toml text eol=lf
18 *.lock text eol=lf
19 *.log text eol=lf
20 *.env text eol=lf
```

```
21 *.prettierignore text eol=lf
22 *.gitignore text eol=lf
23 *.editorconfig text eol=lf
```

## Vue 组件包

模板中以 cpn-kit 为例子

与之前直接开发 vite web 应用不同，我们现在要构建的是库，所以 vit.config.ts 中的打包配置要遵循库模式

## 必要的基础配置

vite.config

代码块

```
1  import { defineConfig } from 'vite';
2  import vue from '@vitejs/plugin-vue';
3  import path from 'path';
4  import unpluginComponents from 'unplugin-vue-components/vite';
5  import { VantResolver } from '@vant/auto-import-resolver';
6
7  export default defineConfig({
8    plugins: [
9      vue(),
10     // 全局自动引入 vant 组件
11     unpluginComponents({ resolvers: [VantResolver()] }),
12   ],
13   build: {
14     lib: {
15       //打包时的入口文件
16       entry: path.resolve(__dirname, './src/index.ts'),
17       //应用名
18       name: 'cpn-kit',
19       //构建产物文件名, js 产物默认有两种 es、umd (format 的值)
20       fileName: (format) => `cpn-kit.${format}.js`,
21     },
22     rollupOptions: {
23       external: ['vue'],
24       output: {
25         globals: {
26           vue: 'Vue',
27         },
28       },
29     },
30   },
31 });
```



```
30    },
31  });
```

## package.json

代码块

```
1  {
2    // 外部默认识别的文件入口
3    "main": "dist/cpn-kit.umd.js",
4    // esm 模块规范默认识别的文件入口
5    "module": "dist/cpn-kit.es.js",
6    // 类型声明文件的识别入口
7    "types": "dist/index.d.ts",
8    // 在发布 npm 包时, 包含的文件/目录有哪些
9    "files": [
10     "dist"
11   ],
12 }
```

## Css 支持

<https://cn.vitejs.dev/guide/build.html#css-support>

在 package.json 中配置 exports

代码块

```
1    "exports": {
2      ".": {
3        "import": "./dist/cpn-kit.es.js",
4        "require": "./dist/cpn-kit.umd.cjs"
5      },
6      "./style.css": "./dist/cpn-kit.css"
7    },
```

外部使用该库时, 需要引入样式文件, 加载该库的样式

代码块

```
1  import '@vmono-seed/cpn-kit/style.css';
```

# 构建产物

## 输出类型文件

<https://github.com/qmhc/unplugin-dts>

注意下文档中提到的 `@microsoft/api-extractor` 这个包，用于解决构建时的问题

### 安装依赖

代码块

```
1 pnpm add -D unplugin-dts@beta @microsoft/api-extractor
```

## 配置 vite.config.ts

这里注意下 unplugin-dts/vite 用的是 CommonJs 规范，使用 ESM 规范导入时，虽然不影响打包 (vite 已处理)，但是 ts 类型会提示没有默认导出项。

1. 改成 `* as dts` 虽然 ts 没有报错，但是打包会失败，此时 vite 会真的按照 ESM 规范导入该包内容
2. 因此最后采取用 ts 跳过校验的注释

代码块

```
1 // @ts-expect-error 默认导出类型有问题，忽略 ts 校验
2 import dts from 'unplugin-dts/vite';
3
4 export default defineConfig({
5   .....,
6   plugins: [
7     .....,
8     dts({ tsconfigPath: './tsconfig.app.json' }),
9   ],
10 });
11
```

## 开发体验相关

### 组件自动引入

#### 安装依赖

代码块

```
1  pnpm add -D unplugin-vue-components @vant/auto-import-resolver
```

## 配置 vite.config.ts

代码块

```
1
2  import unpluginComponents from 'unplugin-vue-components/vite';
3  import { VantResolver } from '@vant/auto-import-resolver';
4
5  export default defineConfig({
6    .....,
7    plugins: [
8      .....,
9      // 全局自动引入 vant 组件
10     unpluginComponents({ resolvers: [VantResolver()] }),
11   ],
12 });
13
```

## Vue 工具包

在 vue 组件包（模板中以 cpn-kit 为例子）目录同级，创建 tools 目录，用于构建 vue 工具包，包含公共函数、hook 等偏向纯逻辑类的工具。

同理也要注意，与之前直接开发 vite web 应用不同，我们现在要构建的是库，所以 vit.config.ts 中的打包配置要遵循库模式

主要构件流程和 Vue 组件包的差不多，相对更简单，因为不需要注入 vant，里面都是 ts 方法。也不需要处理 css。

## Vant-cli (H5 cpn & doc)

<https://github.com/youzan/vant/blob/main/packages/vant-cli/README.zh-CN.md>

### 为何使用？

是 vant 官方维护的 cli 项目，能够轻松构建 vue 组件库。

- 内置文档工程，和手机端 demo 演示
- 支持组件库、文档的独立打包。
  - 自动根据目录结构生成构建产物

如果你要基于 vant 封装一套自己的组件库，那这肯定是不二之选。

## 基础配置修改

### 1. 删除 git 相关配置

- a. .gitignore 配置文件
- b. husky 相关配置
  - nano-staged (package.json)

### 2. 更新包管理器选项（因为我们的项目是 pnpm 创建的 monorepo，为了相关指令能正常使用，要把包管理器切换为 pnpm）

vant.config.mjs

代码块

```
1  export default {
2      .....
3      build: {
4          packageManager: 'pnpm',
5          .....
6      }
7  };
8
```

### 3. 更新描述、作者 (package.json)

代码块

```
1  {
2      .....
3      "description": "A project that includes common components from the H5
4      project (dependent on vant) and some utility functions",
5      "author": "astfn"
6  }
```

## ts 开发(坑点)

github 也有相关未关闭的 issue

1. <https://github.com/youzan/vant/issues/13047>

## 2. <https://github.com/youzan/vant/issues/12912>

这里我主要解决的是 issue 13047，个人参照 vant 库的源码配置进行微调后，解决问题

### tsconfig.json:

1. <https://github.com/youzan/vant/blob/main/packages/vant/tsconfig.json>

2. <https://github.com/youzan/vant/blob/main/tsconfig.json>

组合到项目中相当于

代码块

```
1  {
2    "compilerOptions": {
3      "baseUrl": ".",
4      "jsx": "preserve",
5      "jsxImportSource": "vue",
6      "strict": true,
7      "target": "ES2015",
8      "module": "ESNext",
9      "skipLibCheck": true,
10     "esModuleInterop": true,
11     "moduleResolution": "Node",
12     "lib": ["esnext", "dom"],
13
14     "allowJs": true,
15     "noImplicitThis": true,
16     "types": ["vue/jsx"],
17   },
18   "include": ["src/**/*", "docs/**/*", "test/**/*"],
19   "exclude": ["**/node_modules", "**/*.*/"]
20 }
21
```



在 compilerOptions 中新增配置 "noImplicitAny": false 更符合个人开发习惯

### tsconfig.declaration.json

代码块

```
1  {
2    "extends": "../tsconfig.json",
3    "compilerOptions": {
```

```
4     "declaration": true,  
5     "declarationDir": ".",  
6     "emitDeclarationOnly": true  
7   },  
8   "include": ["es/**/*", "lib/**/*"],  
9   "exclude": [  
10     "**/node_modules",  
11     "**/test/**/*",  
12     "**/demo/**/*",  
13     "**/vue-lazyload/*"  
14   ]  
15 }  
16
```

## css 开发(坑点)

基于 vant 进行二开，就需要依赖 vant 的 css，通常我们会在一个类似入口文件的地方统一引入第三方的依赖

相关 github issue

- <https://github.com/youzan/vant/blob/main/packages/vant-cli/docs/config.zh-CN.md#buildcssbase>
- <https://github.com/youzan/vant/issues/10934>

最后只能在每个组件中引入喽😓

代码块

```
1 <style scoped lang="less">  
2 @import 'vant/lib/index.css';  
3 .....  
4 </style>
```

## 站点相关配置

<https://github.com/youzan/vant/blob/main/packages/vant-cli/docs/config.zh-CN.md#buildsitepublicpath>

## VitePress (待完善)

<https://vitepress.dev/guide/getting-started>

1. vue3 setup 写法 (本身支持)
2. 代码 demo 示例
  - a. <https://github.com/xinlei3166/vitepress-theme-demoblock>
  - b. <https://github.com/flingyp/vitepress-demo-preview>

## vitepress-demo-preview

关于代码 demo 演示，目前使用 vitepress-demo-preview 插件。

因为相较于 vitepress-theme-demoblock 来说，优点如下：

- 不在 md 里面编写 vue 逻辑，虽然 vitePress 支持直接在 md 中编写类 SFC 语法，但是代码提示、格式化等等开发体验肯定不如直接编写 vue 组件
- 通过在 md 中直接引入 vue 组件，即可出现代码示例和复制示例代码的功能。

代码块

```
1 <preview path="./Test.vue" title="Test" description="Test component
  description content"></preview>
```

## vs-code-intellisense-support

<https://vitepress.dev/guide/using-vue#vs-code-intellisense-support>

## 集成工具库样式

我们需要为工具库编写文档，就涉及到使用工具库中的组件，由于工具库(该项目以 cpn-kit 为例)需要引入 css，才能让工具组件的样式正常体现，因此我们需要在 vitePress 中找个地方引入这个 css，并且全局使用工具库组件时，都不必重复导入工具库的 css。

<https://vitepress.dev/guide/extending-default-theme#customizing-css>

在 .vitepress/theme/index.mts 中引入工具库组件的 css

代码块

```
1 import DefaultTheme from 'vitepress/theme';
2 // 引入工具库组件的 css
3 import '@vmono-seed/cpn-kit/style.css';
4
5 export default {
```

```
6    ...DefaultTheme,  
7  };  
8
```

## VuePress

在 vuepress-docs 目录下创建 VuePress 项目，实现工具库的文档产出。

进入 vuepress-docs 目录，初始化项目

代码块

```
1  cd docs  
2  pnpm init
```

<https://vuepress.vuejs.org/zh/guide/getting-started.html#%E6%89%8B%E5%8A%A8%E5%88%9B%E5%BB%BA>