# 样式方案：在 Vite 中接入现代化的 CSS 工程化方案

发布于 2022-05-09

上一小节，我们使用 Vite 初始化了一个 Web 项目，迈出了使用 Vite 的第一步。但在实际工作中，仅用 Vite 官方的脚手架项目是不够的，往往还需要考虑诸多的工程化因素，借助 Vite 本身的配置以及业界的各种生态，才能搭建一个名副其实的脚手架工程。

那在接下来的几个小节内容中，我们将以 实战 的方式逐个击破项目工程化的要素。你可以跟着我一起进行编码，从0搭建一个完整的 Vite 项目架构。不仅如此，在实战的过程中，你也会对 Vite 本身的功能有全面了解，能够熟练地将它应用到实际项目。

样式方案是前端工程化离不开的一个话题，也是本节要具体探讨的内容。在最原始的开发阶段大家都是手写原生的 CSS，但原生 CSS 存在着诸多问题。本小节，我们通过引入现代的各种 CSS 样式方案，一起动手实践，让你学会如何在 Vite 中落地这些样式方案。

## 样式方案的意义

对初学者来说，谈到开发前端的样式，首先想到的便是直接写原生 CSS。但时间一长，难免会发现原生 CSS 开发的各种问题。那么，如果我们不用任何 CSS 工程方案，又会出现哪些问题呢？

**开发体验**欠佳。比如原生 CSS 不支持选择器的嵌套:

```
// 选择器只能平铺，不能嵌套
.container .header .nav .title .text {
  color: blue;
}

.container .header .nav .box {
  color: blue;
  border: 1px solid grey;
}
```

**样式污染**问题。如果出现同样的类名，很容易造成不同的样式互相覆盖和污染。

```
// a.css
.container {
```

```
  color: red;
}

// b.css
// 很有可能覆盖 a.css 的样式!
.container {
  color: blue;
}
```

**浏览器兼容**问题。为了兼容不同的浏览器，我们需要对一些属性(如 `transition` )加上不同的浏览器前缀，比如 `-webkit-` 、 `-moz-` 、 `-ms-` 、 `-o-` ，意味着开发者要针对同一个样式属性写很多的冗余代码。

打包后的**代码体积**问题。如果不用任何的 CSS 工程化方案，所有的 CSS 代码都将打包到产物中，即使有部分样式并没有在代码中使用，导致产物体积过大。

针对如上原生 CSS 的痛点，社区中诞生了不少解决方案，常见的有 5 类。

`CSS 预处理器` ：主流的包括 `Sass/Scss` 、 `Less` 和 `Stylus` 。这些方案各自定义了一套语法，让 CSS 也能使用嵌套规则，甚至能像编程语言一样定义变量、写条件判断和循环语句，大大增强了样式语言的灵活性，解决原生 CSS 的**开发体验问题**。

`CSS Modules` ：能将 CSS 类名处理成哈希值，这样就可以避免同名的情况下**样式污染**的问题。

CSS 后处理器 `PostCSS` ，用来解析和处理 CSS 代码，可以实现的功能非常丰富，比如将 `px` 转换为 `rem` 、根据目标浏览器情况自动加上类似于 `--moz--` 、 `-o-` 的属性前缀等等。

`CSS in JS` 方案，主流的包括 `emotion` 、 `styled-components` 等等，顾名思义，这类方案可以实现直接在 JS 中写样式代码，基本包含 `CSS 预处理器` 和 `CSS Modules` 的各项优点，非常灵活，解决了开发体验和全局样式污染的问题。

CSS 原子化框架，如 `Tailwind CSS` 、 `Windi CSS` ，通过类名来指定样式，大大简化了样式写法，提高了样式开发的效率，主要解决了原生 CSS **开发体验**的问题。

不过，各种方案没有孰优孰劣，各自解决的方案有重叠的部分，但也有一定的差异，大家可以根据自己项目的痛点来引入。接下来，我们进入实战阶段，在 Vite 中应用上述常见的 CSS 方案。

## CSS 预处理器

Vite 本身对 CSS 各种预处理器语言( `Sass/Scss` 、 `Less` 和 `Stylus` )做了内置支持。也就是说，即使你不经过任何的配置也可以直接使用各种 CSS 预处理器。我们以 `Sass/Scss` 为例，来具体感受一下 Vite 的 `零配置` 给我们带来的便利。

由于 Vite 底层会调用 CSS 预处理器的官方库进行编译，而 Vite 为了实现按需加载，并没有内置这些工具库，而是让用户根据需要安装。因此，我们首先安装 Sass 的官方库，

安装命令如下:

```
pnpm i sass -D
```

然后，在上一节初始化后的项目中新建 `src/components/Header` 目录，并且分别新建 `index.tsx` 和 `index.scss` 文件，代码如下:

```
// index.tsx
import './index.scss';
export function Header() {
  return <p className="header">This is Header</p>
};

// index.scss
.header {
  color: red;
}
```

这样就完成了一个最简单的 demo 组件。接着我们在 `App.tsx` 应用这个组件:

```
import { Header } from "./components/Header";

function App() {
  return (
    <div>
      <Header />
    </div>
  );
}

export default App;
```

现在你可以执行 `pnpm run dev` ，然后到浏览器上查看效果:

内容比较简单，如果页面出现红色的文字部分，就说明 `scss` 文件中的样式已经成功生效。好，现在我们封装一个全局的主题色，新建 `src/variable.scss` 文件，内容如下:

```
// variable.scss
$theme-color: red;
```

然后，我们在原来 Header 组件的样式中应用这个变量:

```scss
@import "../../variable";

.header {
  color: $theme-color;
}
```

回到浏览器访问页面，可以看到样式依然生效。你可能会注意到，每次要使用 `$theme-color` 属性的时候我们都需要手动引入 `variable.scss` 文件，那有没有自动引入的方案呢？这就需要在 Vite 中进行一些自定义配置了，在配置文件中增加如下的内容:

```ts
// vite.config.ts
import { normalizePath } from 'vite';
// 如果类型报错，需要安装 @types/node: pnpm i @types/node -D
import path from 'path';

// 全局 scss 文件的路径
// 用 normalizePath 解决 window 下的路径问题
const variablePath = normalizePath(path.resolve('./src/variable.scss'));


export default defineConfig({
  // css 相关的配置
  css: {
    preprocessorOptions: {
      scss: {
        // additionalData 的内容会在每个 scss 文件的开头自动注入
        additionalData: `@import "${variablePath}";`
      }
    }
  }
})
```

现在你可以直接在文件中使用全局文件的变量，相当于之前手动引入的方式显然方便了许多:

```scss
.header {
  color: $theme-color;
}
```

同样的，你可以对 `less` 和 `stylus` 进行一些能力的配置，如果有需要你可以去下面的官方文档中查阅更多的配置项:

- Sass
- Less
- Stylus

- Styles

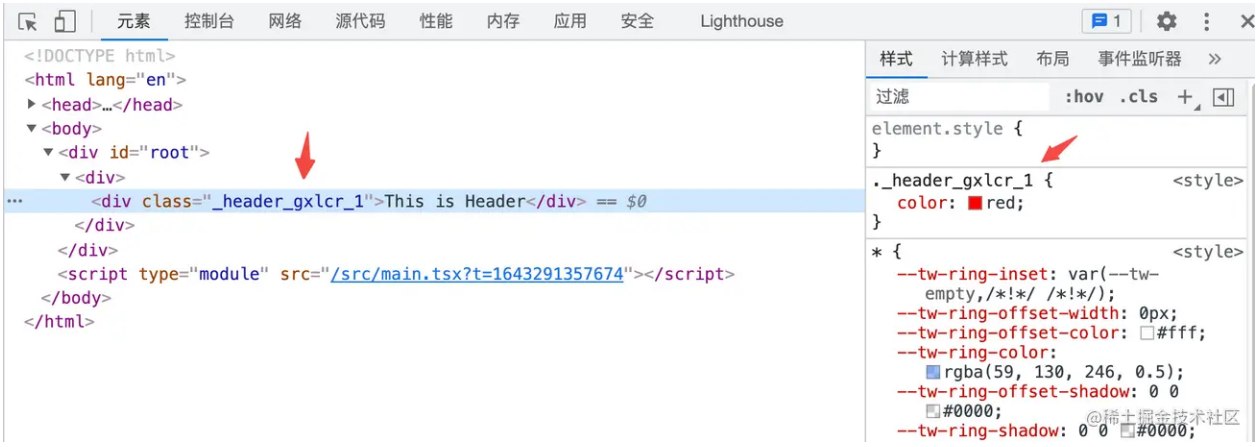
## CSS Modules

CSS Modules 在 Vite 也是一个开箱即用的能力，Vite 会对后缀带有 `.module` 的样式文件自动应用 CSS Modules。接下来我们通过一个简单的例子来使用这个功能。

首先，将 Header 组件中的 `index.scss` 更名为 `index.module.scss` ，然后稍微改动一下 `index.tsx` 的内容，如下:

```
// index.tsx
import styles from './index.module.scss';
export function Header() {
  return <p className={styles.header}>This is Header</p>
};
```

现在打开浏览器，可以看见 p 标签的类名已经被处理成了哈希值的形式:

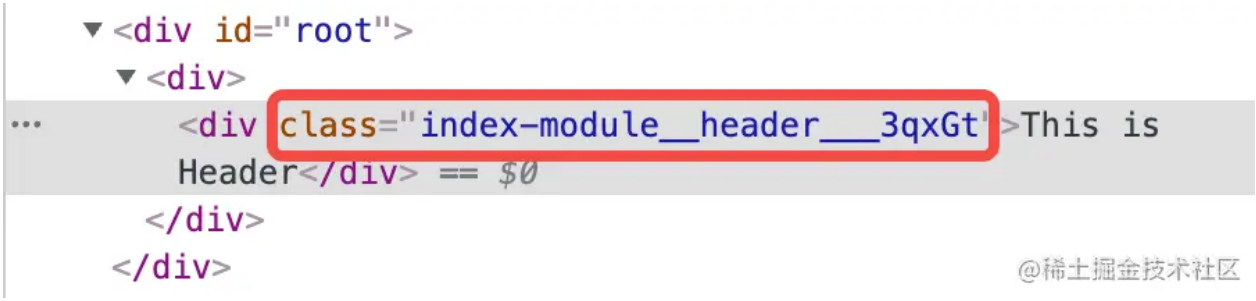



说明现在 CSS Modules 已经正式生效了！同样的，你也可以在配置文件中的 `css.modules` 选项来配置 CSS Modules 的功能，比如下面这个例子:

```
// vite.config.ts
export default {
  css: {
    modules: {
      // 一般我们可以通过 generateScopedName 属性来对生成的类名进行自定义
      // 其中，name 表示当前文件名，local 表示类名
      generateScopedName: "[name]__[local]___[hash:base64:5]"
    },
    preprocessorOptions: {
      // 省略预处理器配置
    }

  }
}
```

再次访问页面，我们可以发现刚才的类名已经变成了我们自定义的形式:




这是一个 CSS Modules 中很常见的配置，对开发时的调试非常有用。其它的一些配置项不太常用，大家可以去这个地址进行查阅。

## PostCSS

一般你可以通过 `postcss.config.js` 来配置 postcss，不过在 Vite 配置文件中已经提供了 PostCSS 的配置入口，我们可以直接在 Vite 配置文件中进行操作。

首先，我们来安装一个常用的 PostCSS 插件—— `autoprefixer`：

```
pnpm i autoprefixer -D
```

这个插件主要用来自动为不同的目标浏览器添加样式前缀，解决的是浏览器兼容性的问题。接下来让我们在 Vite 中接入这个插件:

```
// vite.config.ts 增加如下的配置
import autoprefixer from 'autoprefixer';

export default {
  css: {
    // 进行 PostCSS 配置
    postcss: {
      plugins: [
        autoprefixer({
          // 指定目标浏览器
          overrideBrowserslist: ['Chrome > 40', 'ff > 31', 'ie 11']
        })
      ]
    }
  }
}
```

配置完成后，我们回到 Header 组件的样式文件中添加一个新的 CSS 属性:

```
.header {
  <!-- 前面的样式省略 -->
```

```
  text-decoration: dashed;
}
```

你可以执行 `pnpm run build` 命令进行打包，可以看到产物中自动补上了浏览器前缀，如:

```
._header_kcvt0_1 {
  <!-- 前面的样式省略 -->
  -webkit-text-decoration: dashed;
  -moz-text-decoration: dashed;
  text-decoration: dashed;
}
```

由于有 CSS 代码的 AST (抽象语法树)解析能力，PostCSS 可以做的事情非常多，甚至能实现 CSS 预处理器语法和 CSS Modules，社区当中也有不少的 PostCSS 插件，除了刚刚提到的 `autoprefixer` 插件，常见的插件还包括:

- postcss-pxtorem： 用来将 px 转换为 rem 单位，在适配移动端的场景下很常用。
- postcss-preset-env: 通过它，你可以编写最新的 CSS 语法，不用担心兼容性问题。
- cssnano: 主要用来压缩 CSS 代码，跟常规的代码压缩工具不一样，它能做得更加智能，比如提取一些公共样式进行复用、缩短一些常见的属性值等等。

关于 PostCSS 插件，这里还给大家推荐一个站点：www.postcss.parts/ ，你可以去里面探索更多的内容。

## CSS In JS

社区中有两款主流的 `CSS In JS` 方案: `styled-components` 和 `emotion` 。

对于 CSS In JS 方案，在构建侧我们需要考虑 `选择器命名问题` 、 `DCE` (Dead Code Elimination 即无用代码删除)、 `代码压缩` 、 `生成 SourceMap` 、 `服务端渲染(SSR)` 等问题，而 `styled-components` 和 `emotion` 已经提供了对应的 babel 插件来解决这些问题，我们在 Vite 中要做的就是集成这些 babel 插件。

具体来说，上述的两种主流 CSS in JS 方案在 Vite 中集成方式如下:

```
// vite.config.ts
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

// https://vitejs.dev/config/
```

```
export default defineConfig({

  plugins: [
    react({
      babel: {
        // 加入 babel 插件
        // 以下插件包都需要提前安装
        // 当然，通过这个配置你也可以添加其它的 Babel 插件
        plugins: [
          // 适配 styled-component
          "babel-plugin-styled-components"
          // 适配 emotion
          "@emotion/babel-plugin"
        ]
      },
      // 注意：对于 emotion，需要单独加上这个配置
      // 通过 `@emotion/react` 包编译 emotion 中的特殊 jsx 语法
      jsxImportSource: "@emotion/react"
    })
  ]
})
```

## CSS 原子化框架

在目前的社区当中，CSS 原子化框架主要包括 `Tailwind CSS` 和 `Windi CSS` 。Windi CSS 作为前者的替换方案，实现了按需生成 CSS 类名的功能，开发环境下的 CSS 产物体积大大减少，速度上比 `Tailwind CSS v2` 快 20~100 倍！当然，Tailwind CSS 在 v3 版本也引入 JIT(即时编译) 的功能，解决了开发环境下 CSS 产物体积庞大的问题。接下来我们将这两个方案分别接入到 Vite 中，在实际的项目中你只需要使用其中一种就可以了。我个人比较喜欢 Windi CSS 本身的 `attributify` 、 `shortcuts` 等独有的特性，因此首先从 windicss 开始说起。

## 1. Windi CSS 接入

首先安装 `windicss` 及对应的 Vite 插件:

```
pnpm i windicss vite-plugin-windicss -D
```

随后我们在配置文件中来使用它:

```
// vite.config.ts
import windi from "vite-plugin-windicss";

export default {
  plugins: [
```

```
    // 省略其它插件
    windi()
  ]
}
```

接着要注意在 `src/main.tsx` 中引入一个必需的 import 语句:

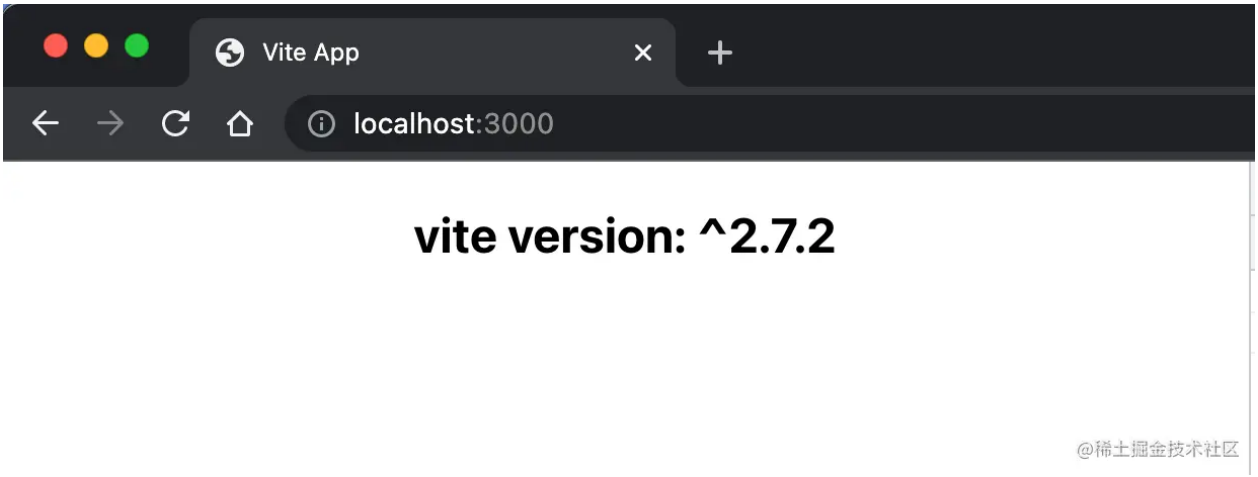
```
// main.tsx
// 用来注入 Windi CSS 所需的样式，一定要加上！
import "virtual:windi.css";
```

这样我们就完成了 Windi CSS 在 Vite 中的接入，接下来我们在 Header 组件中来测试，组件代码修改如下:

```
// src/components/Header/index.tsx
import { devDependencies } from "../../../package.json";

export function Header() {
  return (
    <div className="p-20px text-center">
      <h1 className="font-bold text-2xl mb-2">
        vite version: {devDependencies.vite}
      </h1>
    </div>
  );
}
```

启动项目可以看到如下的效果，说明样式已经正常生效:




除了本身的原子化 CSS 能力，Windi CSS 还有一些非常好用的高级功能，在此我给大家推荐自己常用的两个能力: **attributify** 和 **shortcuts**。

要开启这两个功能，我们需要在项目根目录新建 `windi.config.ts` ，配置如下:

```
import { defineConfig } from "vite-plugin-windicss";

export default defineConfig({
  // 开启 attributify
  attributify: true,
});
```

首先我们来看看 attributify ，翻译过来就是 属性化 ，也就是说我们可以用 props 的方式去定义样式属性，如下所示:

```
<button
  bg="blue-400 hover:blue-500 dark:blue-500 dark:hover:blue-600"
  text="sm white"
  font="mono light"
  p="y-2 x-4"
  border="2 rounded blue-200"
>
  Button
</button>
```

这样的开发方式不仅省去了繁琐的 className 内容，还加强了语义化，让代码更易维护，大大提升了开发体验。

不过使用 attributify 的时候需要注意类型问题，你需要添加 types/shim.d.ts 来增加类型声明，以防类型报错:

```
import { AttributifyAttributes } from 'windicss/types/jsx';

declare module 'react' {
  type HTMLAttributes<T> = AttributifyAttributes;
}
```

shortcuts 用来封装一系列的原子化能力，尤其是一些常见的类名集合，我们在 windi.config.ts 来配置它:

```
//windi.config.ts
import { defineConfig } from "vite-plugin-windicss";

export default defineConfig({
  attributify: true,
  shortcuts: {
    "flex-c": "flex justify-center items-center",
  }
});
```

比如这里封装了 `flex-c` 的类名，接下来我们可以在业务代码直接使用这个类名:

```
<div className="flex-c"></div>
<!-- 等同于下面这段 -->
<div className="flex justify-center items-center"></div>
```

如果你也有过 Windi CSS 的开发经历，欢迎把你用到的高级功能分享到评论区，让大家一起来见识见识。

## 2. Tailwind CSS

接下来我们来接入 Tailwind CSS 方案，为了避免和之前的 Windi CSS 混淆，这里我建议你新起一个 Vite 项目。

> 小册中对应的 GitHub 代码地址。

首先安装 `tailwindcss` 及其必要的依赖:

```
pnpm install -D tailwindcss postcss autoprefixer
```

然后新建两个配置文件 `tailwind.config.js` 和 `postcss.config.js`：

```
// tailwind.config.js
module.exports = {
  content: [
    "./index.html",
    "./src/**/*.{vue,js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}

// postcss.config.js
// 从中你可以看到，Tailwind CSS 的编译能力是通过 PostCSS 插件实现的
// 而 Vite 本身内置了 PostCSS，因此可以通过 PostCSS 配置接入 Tailwind CSS
// 注意：Vite 配置文件中如果有 PostCSS 配置的情况下会覆盖掉 post.config.js 的内容！
module.exports = {
  plugins: {
    tailwindcss: {},
    autoprefixer: {},
  },
```

```
}
```

接着在项目的入口 CSS 中引入必要的样板代码:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

现在，你就可以在项目中安心地使用 Tailwind 样式了，如下所示:

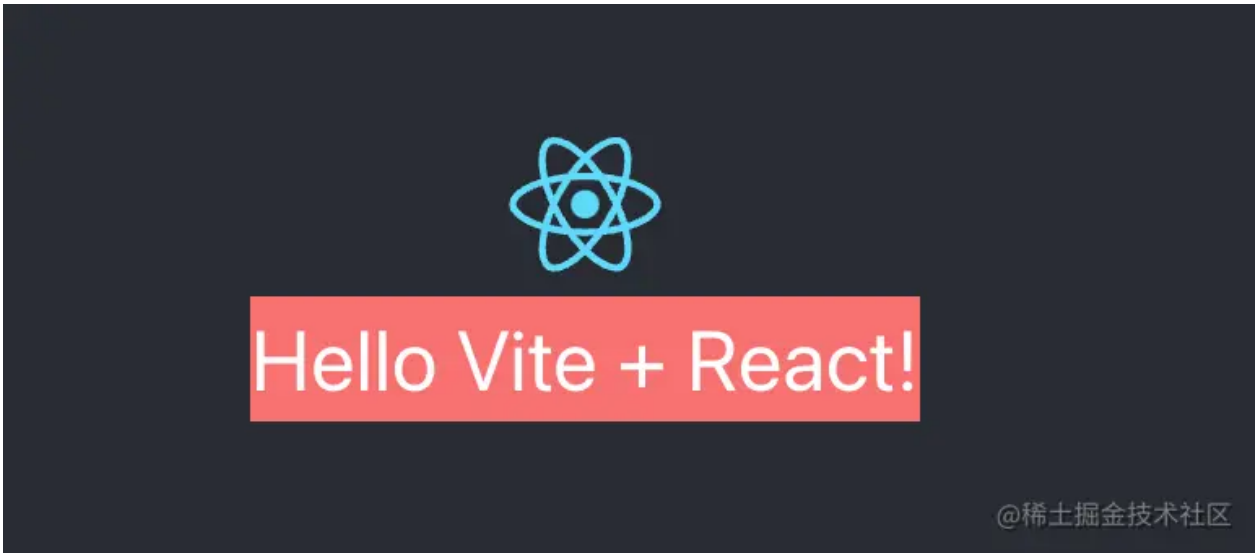
```
// App.tsx

import logo from "./logo.svg";
import "./App.css";

function App() {
  return (
    <div>
      <header className="App-header">
        <img src={logo} className="w-20" alt="logo" />
        <p className="bg-red-400">Hello Vite + React!</p>
      </header>
    </div>
  );
}

export default App;
```

当你启动项目之后可以看到 Tailwind CSS 的样式已经正常生效:




## 小结

OK，本小节的内容到这里就结束了。这一节我们完成了脚手架项目样式部分的搭建，你需要重点掌握前端工程中各种样式方案在 Vite 的接入方法。这些样式方案包括，包括 `CSS 预处理器`、`CSS Modules`、`PostCSS`、`CSS In JS` 和 `CSS 原子化框架(Windi CSS)`。与此同时，你应该明白了各种样式方案的含义以及背后所解决的问题。接下来，我们将会进入项目规范搭建的部分，让我们下一节再见！

上一篇：快速上手: 如何用 Vite 从零搭建前端项目?

下一篇：代码规范: 如何利用 Lint 工具链来保证代码风格和质量?