

# 录Contents

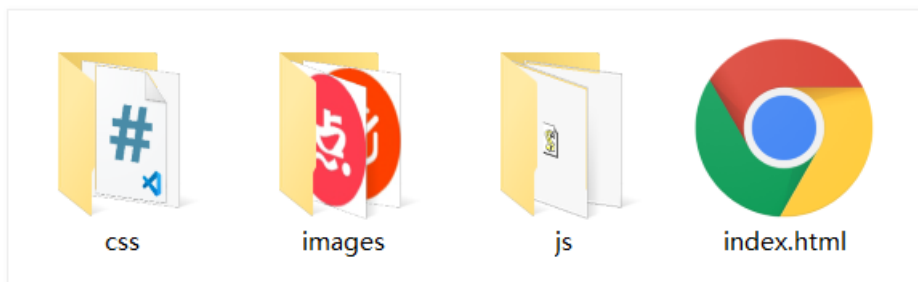
- ◆ 单页面应用程序
- ◆ vite 的基本使用
- ◆ 组件化开发思想
- ◆ vue 组件的构成
- ◆ 组件的基本使用
- ◆ 封装组件的案例

# ■ 单页面应用程序

## 1. 什么是单页面应用程序

**单页面应用程序**（英文名：Single Page Application）简称 SPA，顾名思义，指的是一个 Web 网站中只有唯一的一个 HTML 页面，所有的功能与交互都在这唯一的一个页面内完成。

例如资料中的这个 Demo 项目：



# 单页面应用程序

## 2. 单页面应用程序的特点

单页面应用程序将所有的功能局限于一个 web 页面中，**仅在该 web 页面初始化时加载相应的资源**（HTML、JavaScript 和 CSS）。

一旦页面加载完成了，SPA **不会**因为用户的操作而**进行页面的重新加载或跳转**。而是利用 JavaScript 动态地变换 HTML 的内容，从而实现页面与用户的交互。

# 单页面应用程序

## 3. 单页面应用程序的优点

SPA 单页面应用程序最显著的 3 个优点如下：

### ① 良好的交互体验

- 单页应用的内容的改变不需要重新加载整个页面
- 获取数据也是通过 Ajax 异步获取
- 没有页面之间的跳转，不会出现“白屏现象”

### ② 良好的前后端工作分离模式

- 后端专注于提供 API 接口，更易实现 API 接口的复用
- 前端专注于页面的渲染，更利于前端工程化的发展

### ③ 减轻服务器的压力

- 服务器只提供数据，不负责页面的合成与逻辑的处理，吞吐能力会提高几倍

## 4. 单页面应用程序的缺点

任何一种技术都有自己的局限性，对于 SPA 单页面应用程序来说，主要的缺点有如下两个：

### ① 首屏加载慢

- 路由懒加载
- 代码压缩
- CDN 加速
- 网络传输压缩

### ② 不利于 SEO


- SSR 服务器端渲染

## 5. 如何快速创建 vue 的 SPA 项目

vue 官方提供了**两种**快速创建工程化的 SPA 项目的方式：

- ① 基于 **vite** 创建 SPA 项目
- ② 基于 **vue-cli** 创建 SPA 项目

|               | vite              | vue-cli      |
|---------------|-------------------|--------------|
| 支持的 vue 版本    | 仅支持 <b>vue3.x</b> | 支持 3.x 和 2.x |
| 是否基于 webpack  | 否                 | 是            |
| 运行速度          | <b>快</b>          | 较慢           |
| 功能完整度         | <b>小而巧（逐渐完善）</b>  | 大而全          |
| 是否建议在企业级开发中使用 | 目前不建议             | 建议在企业级开发中使用  |



# 录Contents

- ◆ 单页面应用程序
- ◆ vite 的基本使用
- ◆ 组件化开发思想
- ◆ vue 组件的构成
- ◆ 组件的基本使用
- ◆ 封装组件的案例

# vite 的基本使用

## 1. 创建 vite 的项目

按照顺序执行如下的命令，即可基于 vite 创建 vue 3.x 的工程化项目：

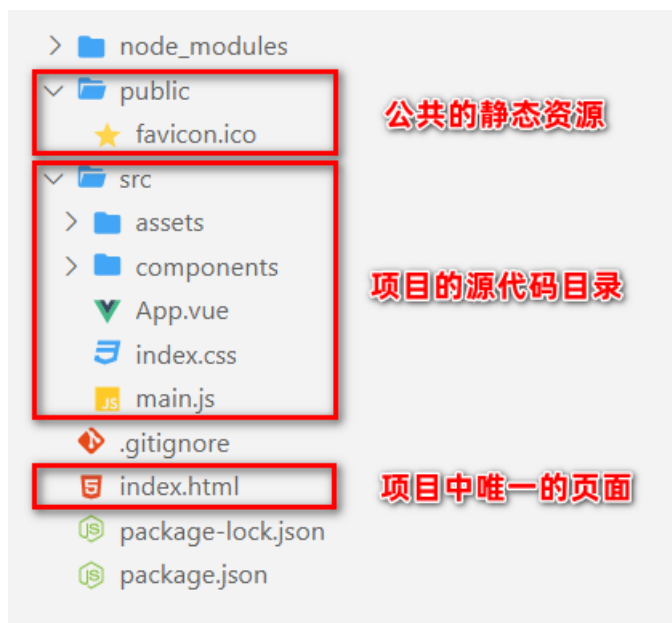
```
1 npm init vite-app 项目名称
2
3 cd 项目名称
4 npm install
5 npm run dev
```



# vite 的基本使用

## 2. 梳理项目的结构

使用 vite 创建的项目结构如下：



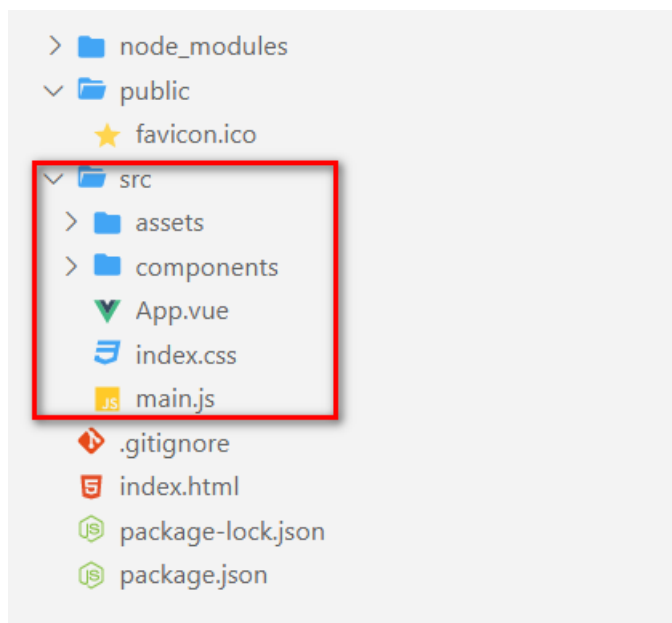
其中：

- `node_modules` 目录用来存放第三方依赖包
- `public` 是公共的静态资源目录
- `src` 是项目的源代码目录（程序员写的所有代码都要放在此目录下）
- `.gitignore` 是 Git 的忽略文件
- `index.html` 是 SPA 单页面应用程序中唯一的 HTML 页面
- `package.json` 是项目的包管理配置文件

# vite 的基本使用

## 2. 梳理项目的结构

在 `src` 这个项目源代码目录之下，包含了如下的文件和文件夹：



其中：

- `assets` 目录用来存放项目中所有的静态资源文件（css、fonts等）
- `components` 目录用来存放项目中所有的自定义组件
- `App.vue` 是项目的根组件
- `index.css` 是项目的全局样式表文件
- `main.js` 是整个项目的打包入口文件

# vite 的基本使用

## 3. vite 项目的运行流程

在工程化的项目中，vue 要做的事情很单纯：通过 `main.js` 把 `App.vue` 渲染到 `index.html` 的指定区域中。

其中：

- ① `App.vue` 用来编写待渲染的模板结构
- ② `index.html` 中需要预留一个 `el` 区域
- ③ `main.js` 把 `App.vue` 渲染到了 `index.html` 所预留的区域中

# vite 的基本使用

## 3.1 在 App.vue 中编写模板结构

清空 App.vue 的默认内容，并书写如下的模板结构：

```
1 <template>
2   <h1>这是 App 根组件</h1>
3 </template>
```

# vite 的基本使用

## 3.2 在 index.html 中预留 el 区域

打开 index.html 页面，确认预留了 el 区域：


```
1 <body>
2   <!-- id 为 app 的 div 元素，就是将来 vue 要控制的区域 -->
3   <div id="app"></div>
4   <script type="module" src="/src/main.js"></script>
5 </body>
```

# vite 的基本使用

## 3.3 在 main.js 中进行渲染

按照 **vue 3.x** 的**标准用法**，把 **App.vue** 中的模板内容渲染到 **index.html** 页面的 **el** 区域中：

```
1 // 1. 从 vue 中按需导入 createApp 函数，
2 //    createApp 函数的作用：创建 vue 的“单页面应用程序实例”
3 import { createApp } from 'vue'
4 // 2. 导入待渲染的 App 组件
5 import App from './App.vue'
6
7 // 3. 调用 createApp() 函数，返回值是“单页面应用程序的实例”，用常量 spa_app 进行接收，
8 //    同时把 App 组件作为参数传给 createApp 函数，表示要把 App 渲染到 index.html 页面上
9 const spa_app = createApp(App)
10 // 4. 调用 spa_app 实例的 mount 方法，用来指定 vue 实际要控制的区域
11 spa_app.mount('#app')
```



# 录Contents

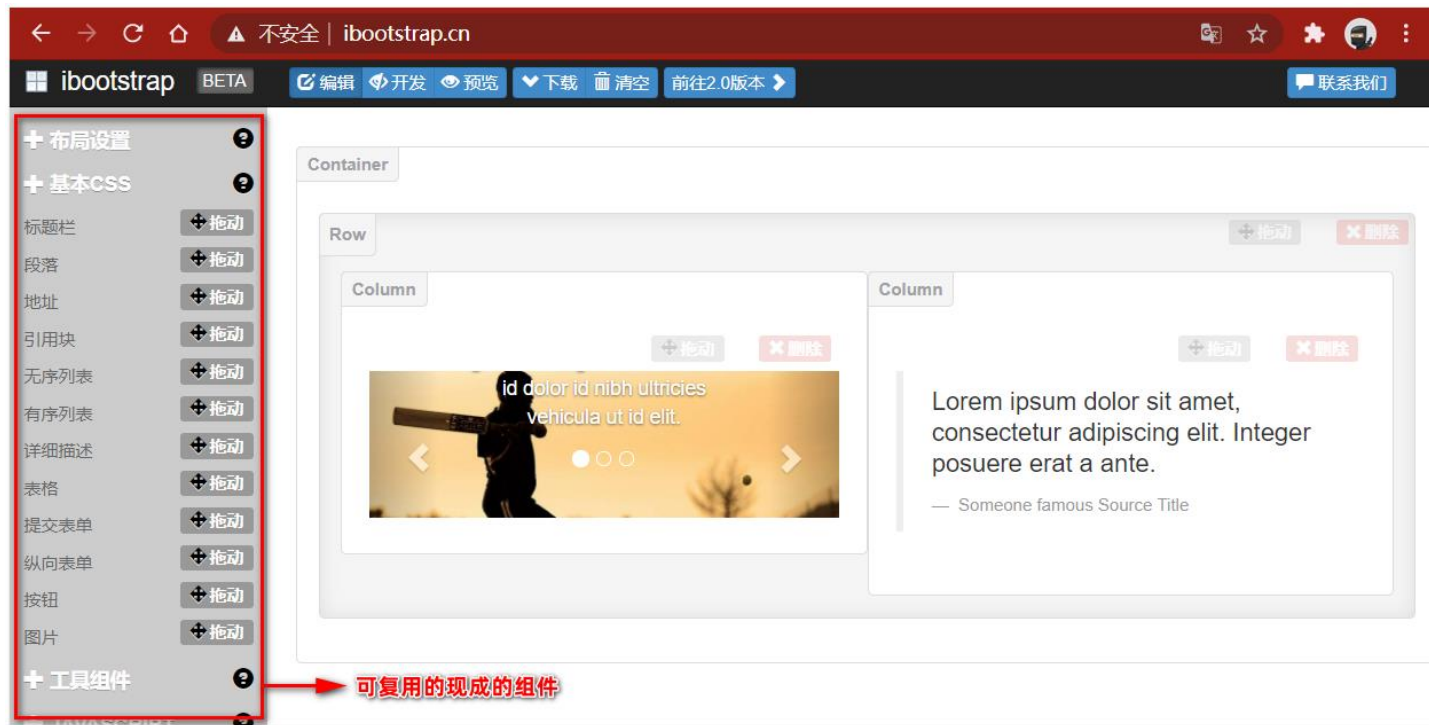
- ◆ 单页面应用程序
- ◆ vite 的基本使用
- ◆ 组件化开发思想
- ◆ vue 组件的构成
- ◆ 组件的基本使用
- ◆ 封装组件的案例

# ■ 组件化开发思想

## 1. 什么是组件化开发

组件化开发指的是：根据封装的思想，把页面上可重用的部分封装为组件，从而方便项目的开发和维护。

例如：<http://www.ibootstrap.cn/> 所展示的效果，就契合了组件化开发的思想。用户可以通过拖拽组件的方式，快速生成一个页面的布局结构。





# 组件化开发思想

## 2. 组件化开发的好处


前端组件化开发的好处主要体现在以下两方面：

- 提高了前端代码的复用性和灵活性
- 提升了开发效率和后期的可维护性

# 组件化开发思想

## 3. vue 中的组件化开发

vue 是一个完全支持组件化开发的框架。vue 中规定组件的后缀名是 `.vue`。之前接触到的 `App.vue` 文件本质上就是一个 vue 的组件。



# 录Contents

- ◆ 单页面应用程序
- ◆ vite 的基本使用
- ◆ 组件化开发思想
- ◆ vue 组件的构成
- ◆ 组件的基本使用
- ◆ 封装组件的案例

# vue 组件的构成

## 1. vue 组件组成结构

每个 .vue 组件都由 3 部分构成，分别是：

- `template` -> 组件的模板结构
- `script` -> 组件的 JavaScript 行为
- `style` -> 组件的样式

其中，每个组件中必须包含 `template` 模板结构，而 `script` 行为和 `style` 样式是可选的组成部分。

# vue 组件的构成

## 2. 组件的 template 节点

vue 规定：每个组件对应的模板结构，需要定义到 `<template>` 节点中。

```
1 <template>
2   <!-- 当前组件的 DOM 结构，需要定义到 template 标签的内部 -->
3 </template>
```

注意：`<template>` 是 vue 提供的容器标签，只起到包裹性质的作用，它不会被渲染为真正的 DOM 元素。

# vue 组件的构成

## 2.1 在 template 中使用指令

在组件的 `<template>` 节点中，支持使用前面所学的**指令语法**，来辅助开发者渲染当前组件的 DOM 结构。  
代码示例如下：

```
1 <template>
2   <h1>这是 App 根组件</h1>
3   <!-- 使用 {{ }} 插值表达式 -->
4   <p>生成一个随机数字: {{ (Math.random() * 10).toFixed(2) }}</p>
5   <!-- 使用 v-bind 属性绑定 -->
6   <p :title="new Date().toLocaleTimeString()">我在黑马程序员学习 vue.js</p>
7   <!-- 属性 v-on 事件绑定 -->
8   <button @click="showInfo">按钮</button>
9 </template>
```

# vue 组件的构成

## 2.2 在 template 中定义根节点

在 **vue 2.x** 的版本中，<template> 节点内的 DOM 结构**仅支持单个根节点**：

```
1 <template>
2   <!-- vue 2.x 中，template 节点内的所有元素，最外层“必须有”唯一的根节点进行包裹，否则报错 -->
3   <div>
4     <h1>这是 App 根组件</h1>
5     <h2>这是副标题</h2>
6   </div>
7 </template>
```

但是，在 **vue 3.x** 的版本中，<template> 中支持定义**多个根节点**：

```
1 <template>
2   <!-- 这是包含多个根节点的 template 结构，因为 h1 标签和 h2 标签外层没有包裹性质的根元素 -->
3   <h1>这是 App 根组件</h1>
4   <h2>这是副标题</h2>
5 </template>
```

# vue 组件的构成

## 3. 组件的 script 节点

vue 规定：组件内的 `<script>` 节点是可选的，开发者可以在 `<script>` 节点中封装组件的 JavaScript 业务逻辑。

`<script>` 节点的基本结构如下：

```
1 <script>
2 // 今后，组件相关的 data 数据、methods 方法等，
3 // 都需要定义到 export default 所导出的对象中。
4 export default {}
5 </script>
```



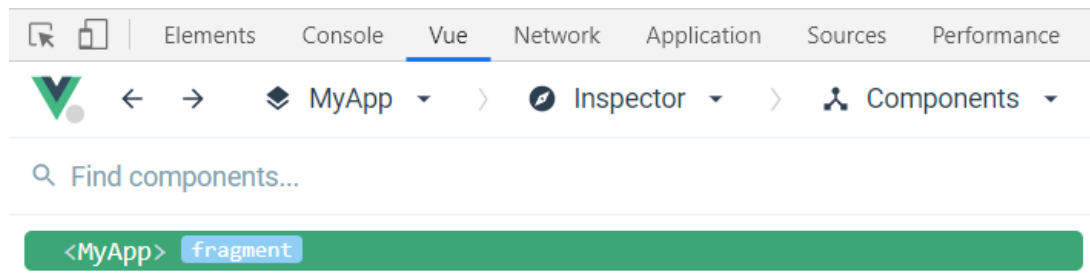
# vue 组件的构成

## 3.1 script 中的 **name** 节点

可以通过 name 节点为当前组件定义一个名称：

```
1 <script>
2 export default {
3   // name 属性指向的是当前组件的名称（建议：每个单词的首字母大写）
4   name: 'MyApp',
5 }
6 </script>
```

在使用 vue-devtools 进行项目调试的时候，自定义的组件名称**可以清晰的区分每个组件**：



# vue 组件的构成

## 3.2 script 中的 data 节点

vue 组件渲染期间需要用到的数据，可以定义在 data 节点中：

```
1 <script>
2 export default {
3   // 组件的名称
4   name: 'MyApp',
5   // 组件的数据（data 方法中 return 出去的对象，就是当前组件渲染期间需要用到的数据对象）
6   data() {
7     return {
8       username: '哇哈哈',
9     }
10  },
11 }
12 </script>
```

# vue 组件的构成

## 组件中的 data 必须是函数

vue 规定：组件中的 data **必须是一个函数**，**不能直接指向一个数据对象**。因此在组件中定义 data 数据节点时，下面的方式是**错误的**：

```
1 data: { // 组件中，不能直接让 data 指向一个数据对象（会报错）
2   count: 0
3 }
```

# vue 组件的构成

## 3.3 script 中的 **methods** 节点

组件中的**事件处理函数**，必须定义到 **methods** 节点中，示例代码如下：

```
1 export default {  
2   name: 'MyApp', // 组件的名称  
3   data() {       // 组件的数据  
4     return {  
5       count: 0,  
6     }  
7   },  
8   methods: {     // 处理函数  
9     addCount() {  
10      this.count++  
11    },  
12  },  
13 }
```

# vue 组件的构成

## 4. 组件的 style 节点

vue 规定：组件内的 `<style>` 节点是可选的，开发者可以在 `<style>` 节点中编写样式美化当前组件的 UI 结构。

`<script>` 节点的基本结构如下：

```
1 <style lang="css">
2 h1 {
3   font-weight: normal;
4 }
5 </style>
```

其中 `<style>` 标签上的 `lang="css"` 属性是可选的，它表示所使用的样式语言。默认只支持普通的 css 语法，可选值还有 less、scss 等。


# vue 组件的构成

## 4.1 让 style 中支持 less 语法

如果希望使用 less 语法编写组件的 style 样式，可以按照如下两个步骤进行配置：

- ① 运行 `npm install less -D` 命令安装依赖包，从而提供 less 语法的编译支持
- ② 在 `<style>` 标签上添加 `lang="less"` 属性，即可使用 less 语法编写组件的样式

```
1 <style lang="less">
2 h1 {
3   font-weight: normal;
4   i {
5     color: red;
6     font-style: normal;
7   }
8 }
9 </style>
```



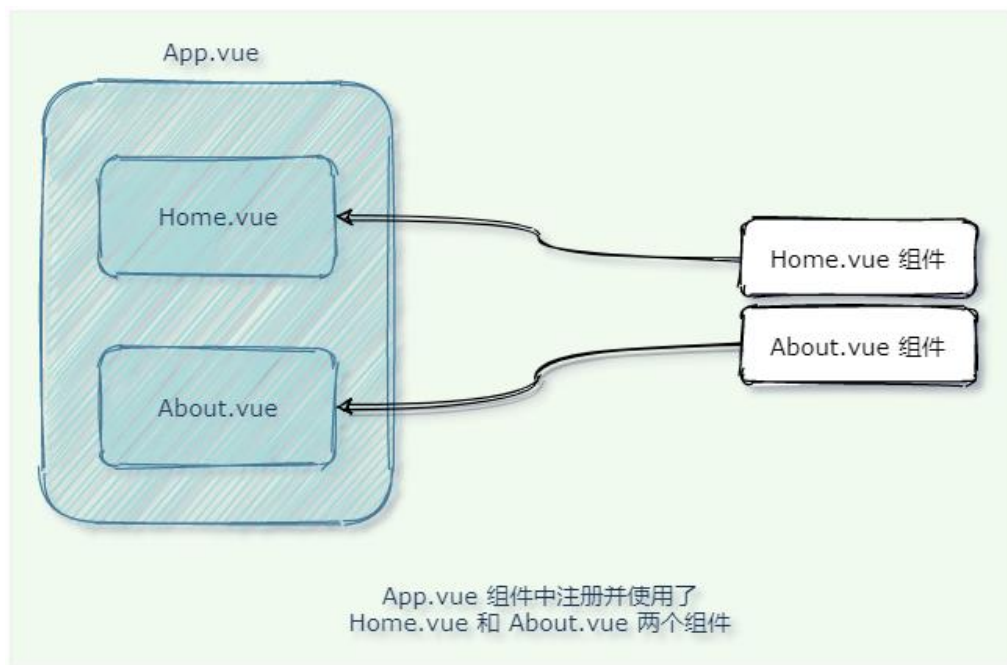
# 录Contents

- ◆ 单页面应用程序
- ◆ vite 的基本使用
- ◆ 组件化开发思想
- ◆ vue 组件的构成
- ◆ 组件的基本使用
- ◆ 封装组件的案例

# ■ 组件的基本使用

## 1. 组件的注册

组件之间可以进行相互的引用，例如：



vue 中组件的引用原则：先注册后使用。



# ■ 组件的基本使用

## 1.1 注册组件的两种方式

vue 中注册组件的方式分为“全局注册”和“局部注册”两种，其中：

- 被全局注册的组件，可以在全局任何一个组件内使用
- 被局部注册的组件，只能在当前注册的范围范围内使用



# ■ 组件的基本使用

## 1.2 全局注册组件

```
1 import { createApp } from 'vue'
2 import App from './App.vue'
3 // 1. 导入 Swiper 和 Test 两个组件
4 import Swiper from './components/MySwiper.vue'
5 import Test from './components/MyTest.vue'
6
7 const app = createApp(App)
8
9 // 2. 调用 app 实例的 component() 方法，在全局注册 my-swiper 和 my-test 两个组件
10 app.component('my-swiper', Swiper)
11 app.component('my-test', Test)
12
13 app.mount('#app')
```

# ■ 组件的基本使用

## 1.3 使用全局注册组件

使用 `app.component()` 方法注册的全局组件，**直接以标签的形式进行使用**即可，例如：

```
1 // 在 main.js 中，注册了 my-swiper 和 my-test 两个全局组件
2 spa_app.component('my-swiper', Swiper)
3 spa_app.component('my-test', Test)
4
5 <!-- 在其他组件中，直接以标签的形式，使用刚才注册的全局组件即可 -->
6 <template>
7   <h1>这是 App 根组件</h1>
8   <hr/>
9   <my-swiper></my-swiper>
10  <my-test></my-test>
11 </template>
```

# 组件的基本使用

## 1.4 局部注册组件

```
1 <template>
2   <h1>这是 App 根组件</h1>
3   <my-swiper></my-swiper>
4   <my-search></my-search>
5 </template>
6
7 <script>
8   import Search from './components/MySearch.vue'
9   export default {
10     components: { // 通过 components 节点，为当前的组件注册私有子组件
11       'my-search': Search,
12     },
13   }
14 </script>
```

## 组件的基本使用

### 1.5 全局注册和局部注册的区别

- 被全局注册的组件，可以在全局任何一个组件内使用
- 被局部注册的组件，只能在当前注册的范围内使用

应用场景：

如果某些组件在开发期间的使用频率很高，推荐进行全局注册；

如果某些组件只在特定的情况下会被用到，推荐进行局部注册。

# ■ 组件的基本使用

## 1.6 组件注册时名称的大小写

在进行组件的注册时，定义组件注册名称的方式有两种：

- ① 使用 **kebab-case** 命名法（俗称**短横线命名法**，例如 my-swiper 和 my-search）
- ② 使用 **PascalCase** 命名法（俗称**帕斯卡命名法**或**大驼峰命名法**，例如 MySwiper 和 MySearch）

短横线命名法的特点：

- 必须严格按照短横线名称进行使用

帕斯卡命名法的特点：

- 既可以严格按照帕斯卡名称进行使用，又可以**转化为短横线名称**进行使用

注意：在实际开发中，**推荐使用帕斯卡命名法**为组件注册名称，因为它的**适用性更强**。

# ■ 组件的基本使用

## 1.7 通过 name 属性注册组件

在注册组件期间，除了可以**直接提供组件的注册名称**之外，还可以把组件的 **name 属性**作为注册后**组件的名称**，示例代码如下：

```

Swiper 组件
1 <template>
2   <h3>轮播图组件</h3>
3 </template>
4
5 <script>
6 export default {
7   name: 'MySwiper' // name 属性为当前组件的名字
8 }
9 </script>
10
全局注册 Swiper 组件
11 import Swiper from './components/MySwiper.vue'
12 app.component(Swiper.name, Swiper) // 相当于 app.component('MySwiper', Swiper)
```

# ■ 组件的基本使用

## 2. 组件之间的样式冲突问题

默认情况下，**写在 .vue 组件中的样式会全局生效**，因此很容易造成**多个组件之间的样式冲突问题**。导致组件之间样式冲突的根本原因是：

- ① 单页面应用程序中，所有组件的 DOM 结构，都是基于**唯一的 index.html 页面**进行呈现的
- ② 每个组件中的样式，都会**影响整个 index.html 页面**中的 DOM 元素



# ■ 组件的基本使用

## 2.1 思考：如何解决组件样式冲突的问题

为每个组件分配唯一的自定义属性，在编写组件样式时，通过属性选择器来控制样式的作用域，示例代码如下：

```
1 <template>
2   <div class="container" data-v-001>
3     <h3 data-v-001>轮播图组件</h3>
4   </div>
5 </template>
6
7 <style>
8   /* 通过中括号“属性选择器”，来防止组件之间的样式冲突问题，
9      因为每个组件分配的自定义属性是“唯一的” */
10  .container[data-v-001] {
11    border: 1px solid red;
12  }
13 </style>
```

# ■ 组件的基本使用

## 2.2 style 节点的 scoped 属性

为了提高开发效率和开发体验，vue 为 **style 节点** 提供了 **scoped** 属性，从而防止组件之间的样式冲突问题：

```
1 <template>
2   <div class="container">
3     <h3>轮播图组件</h3>
4   </div>
5 </template>
6
7 <style scoped>
8   /* style 节点的 scoped 属性，用来自动为每个组件分配唯一的“自定义属性”，
9      并自动为当前组件的 DOM 标签和 style 样式应用这个自定义属性，防止组件的样式冲突问题 */
10  .container {
11    border: 1px solid red;
12  }
13 </style>
```

## ■ 组件的基本使用

### 2.3 /deep/ 样式穿透

如果给当前组件的 style 节点添加了 scoped 属性，则当前组件的样式对其子组件是不生效的。如果想让某些样式对子组件生效，可以使用 /deep/ 深度选择器。

```
1 <style lang="less" scoped>
2 .title {
3   color: blue; /* 不加 /deep/ 时，生成的选择器格式为 .title[data-v-052242de] */
4 }
5
6 /deep/ .title {
7   color: blue; /* 加上 /deep/ 时，生成的选择器格式为 [data-v-052242de] .title */
8 }
9 </style>
```

注意：/deep/ 是 vue2.x 中实现样式穿透的方案。在 vue3.x 中推荐使用 :deep() 替代 /deep/。

## 组件的基本使用

### 3. 组件的 props

为了提高组件的复用性，在封装 vue 组件时需要遵守如下的原则：

- 组件的 DOM 结构、Style 样式要尽量复用
- 组件中要展示的数据，尽量由组件的使用者提供

为了方便使用者为组件提供要展示的数据，vue 组件提供了 props 的概念。

## ■ 组件的基本使用

### 3.1 什么是组件的 props

props 是组件的自定义属性，组件的使用者可以通过 props 把数据传递到子组件内部，供子组件内部进行使用。代码示例如下：

```
1 <!-- 通过自定义 props，把文章的标题和作者，传递到 my-article 组件中 -->
2 <my-article title="面朝大海，春暖花开" author="海子"></my-article>
```

props 的作用：父组件通过 props 向子组件传递要展示的数据。

props 的好处：提高了组件的复用性。

## 组件的基本使用

### 3.2 在组件中声明 props

在封装 vue 组件时，可以把动态的数据项声明为 **props** 自定义属性。自定义属性可以在当前组件的模板结构中被直接使用。示例代码如下：

```
1 <!-- my-article 组件的定义如下: -->
2 <template>
3   <h3>标题: {{title}}</h3>
4   <h5>作者: {{author}}</h5>
5 </template>
6
7 <script>
8   export default {
9     props: ['title', 'author'], // 父组件传递给 my-article 组件的数据，必须在 props 节点中声明
10  }
11 </script>
```

## ■ 组件的基本使用

### 3.3 无法使用未声明的 props

如果父组件给子组件传递了未声明的 props 属性，则这些属性会被忽略，无法被子组件使用，示例代码如下：

```
1 <my-article title="致橡树" author="舒婷"></my-article>
2
3 <template>
4   <h3>标题: {{title}}</h3>
5   <h5>作者: {{author}}</h5>
6 </template>
7
8 <script>
9   export default {
10     props: ['title'], // author 属性没有声明，因此子组件中无法访问到 author 的值
11   }
12 </script>
```

## 组件的基本使用

### 3.4 动态绑定 props 的值

可以使用 **v-bind 属性绑定** 的形式，为组件动态绑定 props 的值，示例代码如下：

```
1 <!-- 通过 v-bind 属性绑定，为 title 动态赋予一个变量的值 -->
2 <!-- 通过 v-bind 属性绑定，为 author 动态赋予一个表达式的值 -->
3 <my-article :title="info.title" :author="'post by ' + info.author"></my-article>
```



## 组件的基本使用

### 3.5 props 的大小写命名

组件中如果使用 “**camelCase (驼峰命名法)**” 声明了 props 属性的名称，则有两种方式为其绑定属性的值：

```
1 <template>
2   <p>发布时间: {{pubTime}}</p>
3 </template>
4
5 <script>
6   export default {
7     props: ['pubTime'], // 采用“驼峰命名法”为当前的组件声明了 pubTime 属性
8   }
9 </script>
10
11 <!-- 既可以直接使用“驼峰命名”的形式为组件绑定属性的值 -->
12 <my-article pubTime="1989"></my-article>
13 <!-- 也可以使用其等价的“短横线分隔命名”的形式为组件绑定属性的值 -->
14 <my-article pub-time="1989"></my-article>
```

## 组件的基本使用

### 4. Class 与 Style 绑定

在实际开发中经常会遇到动态操作元素样式的需求。因此，vue 允许开发者通过 `v-bind` 属性绑定指令，为元素动态绑定 `class` 属性的值和行内的 `style` 样式。

## ■ 组件的基本使用

### 4.1 动态绑定 HTML 的 class

可以通过三元表达式，动态的为元素绑定 class 的类名。示例代码如下：

```
1 <h3 class="thin" :class="isItalic ? 'italic' : ''">MyDeep 组件</h3>
2 <button @click="isItalic=!isItalic">Toggle Italic</button>
3
4 data() {
5   return { isItalic: true }
6 }
7
8 .thin { // 字体变细
9   font-weight: 200;
10 }
11 .italic { // 倾斜字体
12   font-style: italic;
13 }
```

## ■ 组件的基本使用

### 4.2 以数组语法绑定 HTML 的 class

如果元素需要动态绑定多个 class 的类名，此时可以使用数组的语法格式：

```
1 <h3 class="thin" :class="[isItalic ? 'italic' : '', isDelete ? 'delete' : '']">
2   MyDeep 组件
3 </h3>
4
5 <button @click="isItalic=!isItalic">Toggle Italic</button>
6 <button @click="isDelete=!isDelete">Toggle Delete</button>
7
8 data() {
9   return {
10     isItalic: true,
11     isDelete: false,
12   }
13 }
```

## ■ 组件的基本使用

### 4.3 以对象语法绑定 HTML 的 class

使用数组语法动态绑定 class 会导致模板结构臃肿的问题。此时可以使用对象语法进行简化：


```
1 <h3 class="thin" :class="classObj">MyDeep 组件</h3>
2 <button @click="classObj.italic = !classObj.italic">Toggle Italic</button>
3 <button @click="classObj.delete = !classObj.delete">Toggle Delete</button>
4
5 data() {
6   return {
7     classObj: { // 对象中, 属性名是 class 类名, 值是布尔值
8       italic: true,
9       delete: false,
10    }
11  }
12 }
```

## ■ 组件的基本使用

### 4.4 以对象语法绑定内联的 style

:style 的对象语法十分直观——看着非常像 CSS，但其实是一个 JavaScript 对象。CSS property 名可以用驼峰式 (camelCase) 或短横线分隔 (kebab-case, 记得用引号括起来) 来命名：

```
1 <div :style="{color: active, fontSize: fsize + 'px', 'background-color': bgcolor}">
2   黑马程序员
3 </div>
4 <button @click="fsize += 1">字号 +1</button>
5 <button @click="fsize -= 1">字号 -1</button>
6
7 data() {
8   return {
9     active: 'red',
10    fsize: 30,
11    bgcolor: 'pink',
12  }
13 }
```



# 录Contents

- ◆ 单页面应用程序
- ◆ vite 的基本使用
- ◆ 组件化开发思想
- ◆ vue 组件的构成
- ◆ 组件的基本使用
- ◆ 封装组件的案例

# ■ 封装组件的案例

## 1. 案例效果

MyHeader 组件

封装要求：

- ① 允许用户自定义 **title** 标题
- ② 允许用户自定义 **bgcolor** 背景色
- ③ 允许用户自定义 **color** 文本颜色
- ④ MyHeader 组件需要在页面顶部进行 **fixed** 固定定位，且 **z-index** 等于 999

使用示例如下：

```
1 <my-header title="黑马程序员" bgcolor="#000" color="#f8f8f8"></my-header>
```



# 封装组件的案例

## 2. 用到的知识点

- 组件的封装与注册
- props
- 样式绑定

## 封装组件的案例

### 3. 整体实现步骤

- 创建 MyHeader 组件
- 渲染 MyHeader 组件的基本结构
- 在 App 组件中注册并使用 MyHeader 组件
- 通过 props 为组件传递数据



# 总结

- ① 能够说出什么是单页面应用程序及组件化开发
  - SPA、只有一个页面、组件是对 UI 结构的复用
- ② 能够说出 .vue 单文件组件的组成部分
  - template、script、style (scoped、lang)
- ③ 能够知道如何注册 vue 的组件
  - 全局注册 (app.component)、局部注册 (components)
- ④ 能够知道如何声明组件的 props 属性
  - props 数组
- ④ 能够知道如何在组件中进行样式绑定
  - 动态绑定 class、动态绑定 style