

# 前端工程化与 webpack

## 前端工程化

### 1. 小白眼中的前端开发 vs 实际的前端开发

小白眼中的前端开发：

- 会写HTML + CSS + JavaScript 就会前端开发
- 需要美化页面样式，就拽一个bootstrap 过来
- 需要操作DOM 或发起Ajax 请求，再拽一个jQuery过来
- 需要快速实现网页布局效果，就拽一个Layui过来

实际的前端开发：

- 模块化 (js 的模块化、css 的模块化、资源的模块化)
- 组件化 (复用现有的UI 结构、样式、行为)
- 规范化 (目录结构的划分、编码规范化、接口规范化、文档规范化、Git 分支管理)
- 自动化 (自动化构建、自动部署、自动化测试)

### 2. 什么是前端工程化

前端工程化指的是：在企业级的前端项目开发中，把前端开发所需的工具、技术、流程、经验等进行规范化、标准化。

企业中的Vue 项目和React 项目，都是基于工程化的方式进行开发的。

好处：前端开发自成体系，有一套标准的开发方案和流程。

### 3. 前端工程化的解决方案

早期的前端工程化解决方案：

grunt (<https://www.gruntjs.net/>)  
gulp (<https://www.gulpjs.com.cn/>)

目前主流的前端工程化解决方案：

webpack (<https://www.webpackjs.com/>)  
parcel (<https://zh.parceljs.org/>)

## webpack 的基本使用

### 1. 什么是 webpack

概念：webpack 是前端项目工程化的具体解决方案。

主要功能：它提供了友好的前端模块化开发支持，以及代码压缩混淆、处理浏览器端JavaScript 的兼容性、性能优化等强大的功能。

好处：让程序员把工作的重心放到具体功能的实现上，提高了前端开发效率和项目的可维护性。

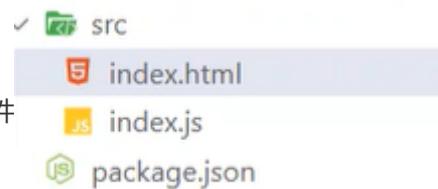
注意：目前Vue，React等前端项目，基本上都是基于webpack进行工程化开发的。

## 2. 创建列表隔行变色项目

① 新建项目空白目录，并运行npm init -y 命令(运行这个目录的前提，必须是纯英文目录)，初始化包管理配置文件package.json

② 新建src 源代码目录

③ 新建src->index.html 首页和src->index.js 脚本文件



④ 初始化首页基本的结构

⑤ 运行npm install jquery -S (这个参数的意思是把jquery添加到dependencies目录下，不过不加这个指令也会安装到这个目录下，我个人感觉没有必要加)命令，安装jQuery

⑥ 通过ES6模块化的方式导入jQuery，实现列表隔行变色效果

## 3. 在项目中安装 webpack

在终端运行如下的命令，安装webpack 相关的两个包：

```
npm install webpack@5.42.1 webpack-cli@4.7.2 -D (这个D的意思是安装到devDependencies)
```

```
4   "description": " ",
5   "main": "index.js",
  ▷ 调试
6   "scripts": {
7     "test": "echo \\\"Error: no test specified\\\" && exit 1"
8   },
9   "keywords": [],
10  "author": "",
11  "license": "ISC",
12  "dependencies": {
13    "jquery": "^3.6.0"
14  },
15  "devDependencies": {
16    "webpack": "^5.42.1",
17    "webpack-cli": "^4.7.2"
18  }
19}
```

至于为什么这样安装，可以看npm的官方

The screenshot shows the webpack NPM package page. In the 'Install' section, there are two examples: one for npm with the command 'npm install --save-dev webpack' and one for yarn with the command 'yarn add webpack --dev'. The npm command is highlighted with a red box.

注意：

- -S是 --save的简写
- -D是 --save-dev的简写

## 4. 在项目中配置 webpack

① 在项目根目录中，创建名为 webpack.config.js 的 webpack 配置文件，并初始化如下的基本配置：

```
1 module.exports = {
2   mode: 'development' // mode 用来指定构建模式。可选值有 development 和 production
3 }
```

② 在 package.json 的 scripts 节点下，新增 dev 脚本如下：

```
1 "scripts": {
2   "dev": "webpack" // script 节点下的脚本，可以通过 npm run 执行。例如 npm run dev
3 }
```

上面后面这个webpack是固定的

③ 在终端中运行npm run dev 命令，启动webpack 进行项目的打包构建

### mode 的可选值

mode 节点的可选值有两个，分别是：

① development

- 开发环境
- 不会对打包生成的文件进行代码压缩和性能优化
- 打包速度快，适合在开发阶段使用

② production

- 生产环境
- 会对打包生成的文件进行代码压缩和性能优化
- 打包速度很慢，仅适合在项目发布阶段使用

```

index.html package.json webpack.config.js index.js main.js
1 // 使用 Node.js 中的导出语法，向外导出一个 webpack 的配置对象
2 module.exports = {
...
3 // 代表 webpack 运行的模式，可选值有两个 development 和 production
4 mode: 'development'          改这个就可以了
5 }
6

```

## webpack.config.js 文件的作用

webpack.config.js 是 webpack 的配置文件。webpack 在真正开始打包构建之前，会先读取这个配置文件，从而基于给定的配置，对项目进行打包。（也就是说，当我们运行npm run dev 的时候，先会读取 webpack.config.js中的配置选项，之后会根据模式运行webpack命令）

注意：由于 webpack 是基于 node.js 开发出来的打包工具，因此在它的配置文件中，支持使用 node.js 相关的语法和模块进行webpack 的个性化配置。

## webpack 中的默认约定

在 webpack 4.x 和 5.x 的版本中，有如下的默认约定：

- ① 默认的打包入口文件为 src -> index.js(src下面的index.js文件)
- ② 默认的输出文件路径为dist -> main.js (dist下面的main.js文件)

注意：可以在webpack.config.js 中修改打包的默认约定

## 自定义打包的入口与出口

在 webpack.config.js 配置文件中，通过entry 节点指定打包的入口。通过output 节点指定打包的出口。示例代码如下：

```

const path = require('path') // 导入 node.js 中专门操作路径的模块
...
module.exports = {
  entry: path.join(__dirname, './src/index.js'), // 打包入口文件的路径
  output: {
    path: path.join(__dirname, './dist'), // 输出文件的存放路径
    filename: 'bundle.js' // 输出文件的名称
  }
}

```

# webpack 中的插件

## webpack 插件的作用

通过安装和配置第三方的插件，可以拓展webpack 的能力，从而让webpack 用起来更方便。最常用的webpack 插件有如下两个：

### ① webpack-dev-server

- 类似于node.js 阶段用到的 nodemon 工具
- 每当修改了源代码，然后ctrl+s， webpack 会自动进行项目的打包和构建,这样我们就可以实时看到我们修改的效果咯

### ② html-webpack-plugin

- webpack 中的HTML 插件（类似于一个模板引擎插件）
- 可以通过此插件自定制 index.html 页面的内容

## webpack-dev-server

webpack-dev-server 可以让webpack 监听项目源代码的变化，从而进行自动打包构建。

### 安装 webpack-dev-server

运行如下的命令，即可在项目中安装此插件：

```
npm install webpack-dev-server@3.11.2 -D
```

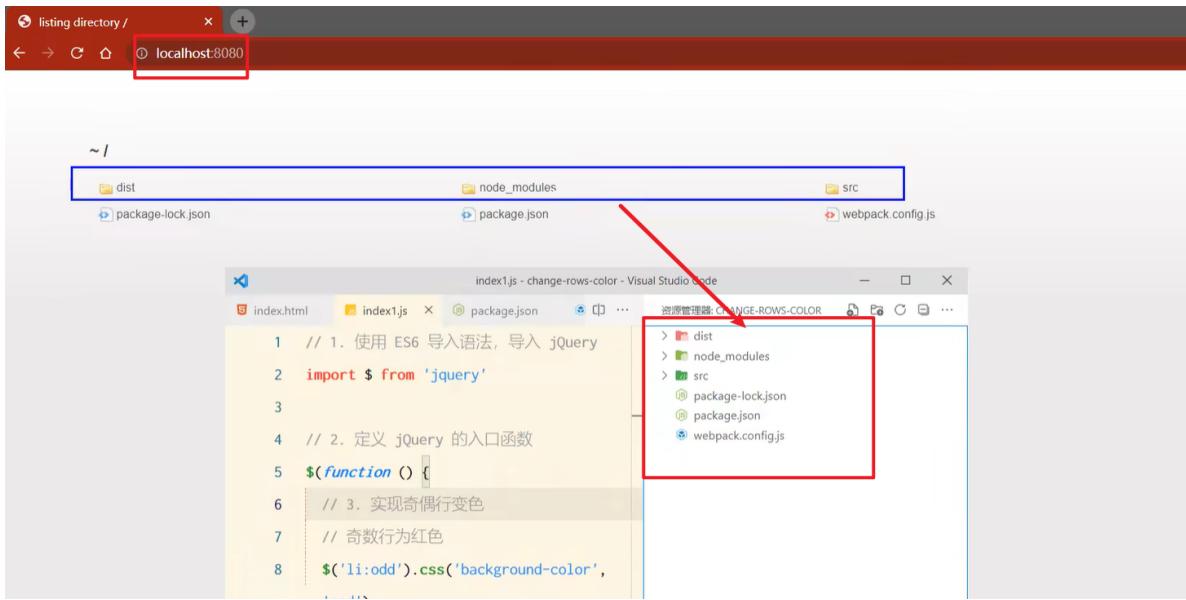
### 配置 webpack-dev-server

① 修改package.json -> scripts 中的dev 命令如下：（之前是dev:webpack,现在加了一个serve）

```
1 "scripts": {  
2     "dev": "webpack serve", // script 节点下的脚本，可以通过 npm run 执行  
3 }
```

② 再次运行npm run dev 命令，重新进行项目的打包

③ 在浏览器中访问<http://localhost:8080> 地址，查看自动打包效果（这个是项目的根目录）



注意：webpack-dev-server 会启动一个实时打包的http 服务器

## 打包生成的文件哪儿去了？

① 不配置webpack-dev-server 的情况下， webpack 打包生成的文件，会存放到实际的物理磁盘上

- 严格遵守开发者在webpack.config.js 中指定配置
- 根据output 节点指定路径进行存放

② 配置了 webpack-dev-server 之后，打包生成的文件存放到内存中

- 不再根据output 节点指定的路径，存放到实际的物理磁盘上
- 提高了实时打包输出的性能，因为内存比物理磁盘速度快很多

## 生成到内存中的文件该如何访问？

webpack-dev-server 生成到内存中的文件， 默认放到了项目的根目录中，而且是虚拟的、不可见的。

- 可以直接用/ 表示项目根目录，后面跟上要访问的文件名称，即可访问内存中的文件
- 例如/bundle.js 就表示要访问webpack-dev-server 生成到内存中的bundle.js 文件

## html-webpack-plugin

html-webpack-plugin 是 webpack 中的HTML 插件，可以通过此插件自定制 index.html 页面的内容。

需求：通过html-webpack-plugin 插件，将 src 目录下的 index.html 首页，复制到项目根目录中一份！

### 安装 html-webpack-plugin

运行如下的命令，即可在项目中安装此插件：

```
npm install html-webpack-plugin@5.3.2 -D
```

### 配置 html-webpack-plugin

下面的配置是在webpack.config.js配置文件中进行配置的

```
1 // 1. 导入 HTML 插件，得到一个构造函数
2 const HtmlWebpackPlugin = require('html-webpack-plugin')
3
4 // 2. 创建 HTML 插件的实例对象
5 const htmlPlugin = new HtmlWebpackPlugin({
6   template: './src/index.html', // 指定原文件的存放路径
7   filename: './index.html', // 指定生成的文件的存放路径
8 })
9
10 module.exports = {
11   mode: 'development',
12   plugins: [htmlPlugin], // 3. 通过 plugins 节点，使 htmlPlugin 插件生效
13 }
```

插件是在webpack指令执行期间会加载并调用这些插件

## 解惑 html-webpack-plugin

- ① 通过HTML 插件复制到项目根目录中的 index.html 页面，也被放到了内存中
- ② HTML 插件在生成的 index.html 页面，自动注入了打包的bundle.js（不一定一定是这个文件，这个文件是我们自己配置的名字）文件

## devServer 节点

在 webpack.config.js 配置文件中，可以通过devServer 节点对webpack-dev-server 插件进行更多的配置，示例代码如下：

```
1 devServer: {
2   open: true, // 初次打包完成后，自动打开浏览器
3   host: '127.0.0.1', // 实时打包所使用的主机地址
4   port: 80, // 实时打包所使用的端口号
5 }
```

修改端口号也是在这修改的

注意：凡是修改了 webpack.config.js 配置文件，或修改了 package.json 配置文件，必须重启实时打包的服务器，否则最新的配置文件无法生效！

## webpack 中的 loader

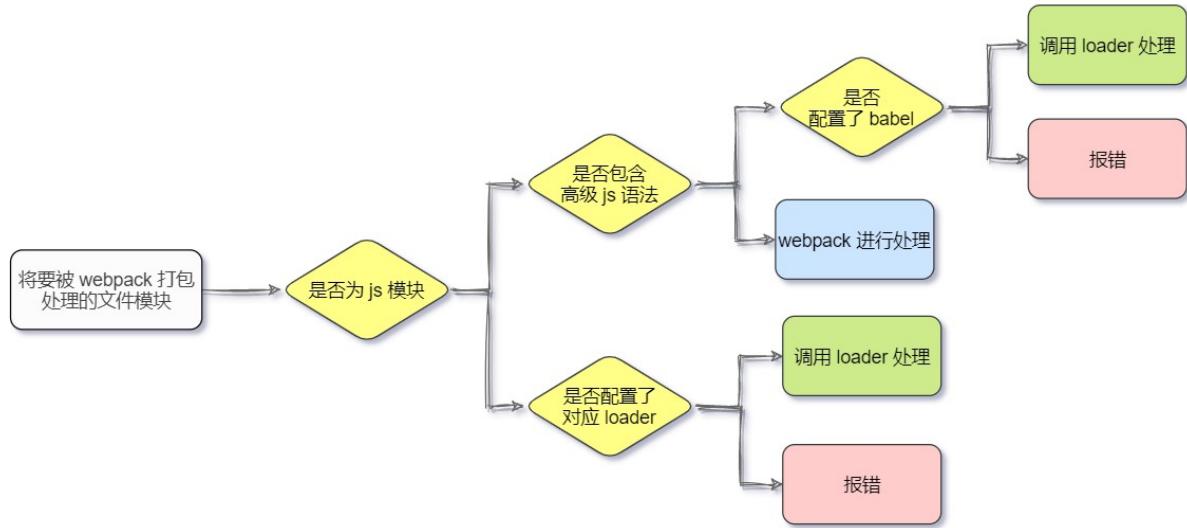
### loader 概述

在实际开发过程中，webpack 默认只能打包处理以.js 后缀名结尾的模块。其他非.js 后缀名结尾的模块，webpack 默认处理不了，需要调用 loader 加载器才可以正常打包，否则会报错！

loader 加载器的作用：协助 webpack 打包处理特定的文件模块。比如：

- css-loader 可以打包处理.css 相关的文件
- less-loader 可以打包处理.less 相关的文件
- babel-loader 可以打包处理webpack 无法处理的高级 JS 语法

## loader 的调用过程



## 打包处理 css 文件

- ① 运行npm i style-loader@3.0.0 css-loader@5.2.6 -D 命令，安装处理css 文件的 loader
- ② 在 webpack.config.js 的module -> rules 数组中，添加 loader 规则如下：

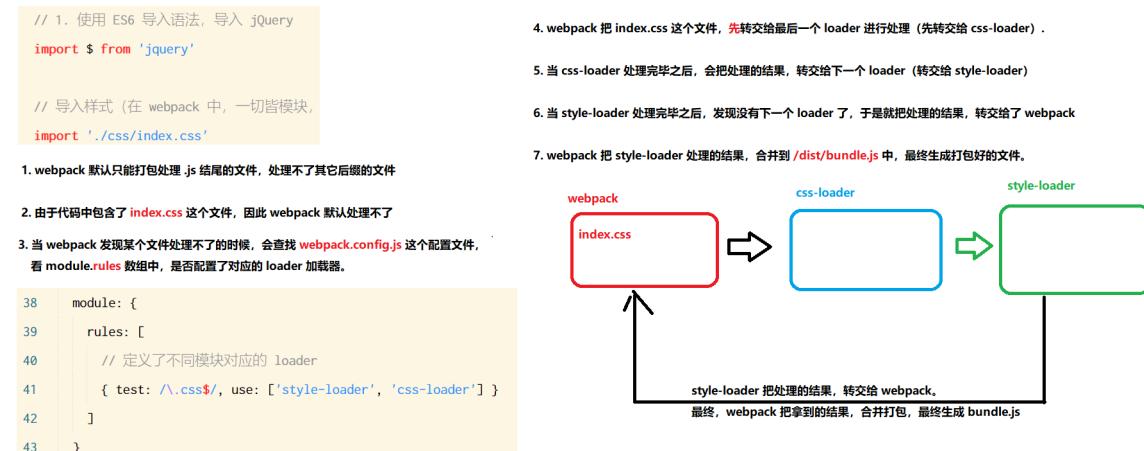
```

1 module: { // 所有第三方文件模块的匹配规则
2   rules: [ // 文件后缀名的匹配规则
3     { test: /\.css$/, use: ['style-loader', 'css-loader'] }
4   ]
5 }
  
```

其中，test 表示匹配的文件类型，use 表示对应要调用的 loader

注意：

- use 数组中指定的 loader 顺序是固定的
- 多个 loader 的调用顺序是：从后往前调用



## 打包处理 less 文件

① 运行 `npm i less-loader@10.0.1 less@4.1.1 -D` 命令

② 在 `webpack.config.js` 的 `module -> rules` 数组中，添加 loader 规则如下：

```
1 module: { // 所有第三方文件模块的匹配规则
2   rules: [ // 文件后缀名的匹配规则
3     { test: /\.less$/, use: ['style-loader', 'css-loader', 'less-loader'] },
4   ]
5 }
```

## 打包处理样式表中与 url 路径相关的文件

首先学一下base64怎么加载图片的：



The screenshot shows a code editor with an HTML file named `test.html`. The code includes a `<img>` tag with a `src` attribute set to a long base64 encoded string. This string represents the image data. A red box highlights this `src` attribute.

```
10
11 <body>
12   
13
14   <!-- 精灵图 -->
15   
17   alt="">
18 </body>
19
20 </html>
```

## 打包处理样式表中与 url 路径相关的文件：

① 运行 `npm i url-loader@4.1.1 file-loader@6.2.0 -D` 命令

② 在 `webpack.config.js` 的 `module -> rules` 数组中，添加 loader 规则如下：

```
1 module: { // 所有第三方文件模块的匹配规则
2   rules: [ // 文件后缀名的匹配规则
3     { test: /\.jpg|png|gif$/, use: 'url-loader?limit=22229' },
4   ]
5 }
```

其中`?之后的是 loader 的参数项：`

- limit 用来指定图片的大小，单位是字节 (byte)
- 只有≤ limit 大小的图片，才会被转为base64 格式的图片，如果大于这个值，不会转成base64格式，还是图片的格式

```

<li>这是第 2 个 li</li>
<li>这是第 3 个 li</li>
<li>这是第 4 个 li</li>
<li>这是第 5 个 li</li>
<li>这是第 6 个 li</li>
<li>这是第 7 个 li</li>
<li>这是第 8 个 li</li>
<li>这是第 9 个 li</li>
</ul>

<!-- 需求：把 /src/images/logo.jpg 设置给 src 属性 -->
<img src="" alt="" class="box">
</body>

</html>

```

4 // 导入样式 (在 webpack 中，一切皆模块，都可以通过 ES6 导入语法进行使用)  
5 import './css/index.css'  
6 import './css/index.less'  
7  
8 // 1. 导入图片，得到图片文件  
9 import logo from './images/logo.jpg'  
10 // 2. 给 img 标签的 src 动态赋值  
11 \$('.box').attr('src', logo)  
12  
13 // 2. 定义 jQuery 的入口函数  
14 \$(function () {  
15 // 3. 实现奇偶行变色  
16 // 奇数行为红色  
17 \$('#list li:odd').css('background-color', 'red')  
18 // 偶数行为黄色  
19 \$('#list li:even').css('background-color', 'yellow')

## 打包处理 js 文件中的高级语法

webpack 只能打包处理一部分高级的 JavaScript 语法。对于那些webpack 无法处理的高级 js 语法，需要借助于babel-loader 进行打包处理。例如webpack 无法处理下面的 JavaScript 代码：

```

1 // 1. 定义了名为 info 的装饰器
2 function info(target) {
3   // 2. 为目标添加静态属性 info
4   target.info = 'Person info'
5 }
6
7 // 3. 为 Person 类应用 info 装饰器
8 @info
9 class Person {}
10
11 // 4. 打印 Person 的静态属性 info
12 console.log(Person.info)

```

## 安装 babel-loader 相关的包

运行如下的命令安装对应的依赖包：

```
npm i babel-loader@8.2.2 @babel/core@7.14.6 @babel/plugin-proposal-decorators@7.14.5
-D
```

在 webpack.config.js 的 module -> rules 数组中，添加 loader 规则如下：

```

1 // 注意：必须使用 exclude 指定排除项；因为 node_modules 目录下的第三方包不需要被打包
2 { test: /\.js$/, use: 'babel-loader', exclude: /node_modules/ }

```

至于为什么要指定排除项：因为第三方包中的JS兼容性，不需要我们关心，人家已经处理好了…

## 配置 babel-loader

这里其实是配置插件的插件

在项目根目录下，创建名为 babel.config.js 的配置文件，定义 Babel 的配置项如下：

```
module.exports = {  
    // 声明 babel 可用的插件  
    plugins: [[ '@babel/plugin-proposal-decorators', { legacy: true } ]]  
}
```

详情请参考Babel 的官网<https://babeljs.io/docs/en/babel-plugin-proposal-decorators>

# 打包发布

## 为什么要打包发布

项目开发完成之后，需要使用webpack 对项目进行打包发布，主要原因有以下两点：

- ① 开发环境下，打包生成的文件存放于内存中，无法获取到最终打包生成的文件
- ② 开发环境下，打包生成的文件不会进行代码压缩和性能优化

为了让项目能够在生产环境中高性能的运行，因此需要对项目进行打包发布。

## 配置 webpack 的打包发布

在 package.json 文件的 scripts 节点下，新增build 命令如下：

```
1 "scripts": {  
2     "dev": "webpack serve", // 开发环境中，运行 dev 命令  
3     "build": "webpack --mode production" // 项目发布时，运行 build 命令  
4 }
```

--model 是一个参数项，用来指定webpack 的运行模式。production 代表生产环境，会对打包生成的文件进行代码压缩和性能优化（记不得之前我们在webpack.config.js中配置模式为development，这里的--model就是覆盖那里的配置文件，改成了生产环境）

注意：通过 --model 指定的参数项，会覆盖webpack.config.js 中的model 选项。

## 把 JavaScript 文件统一生成到 js 目录中

在 webpack.config.js 配置文件的output 节点中，进行如下的配置：

```
1 output: {
2   path: path.join(__dirname, 'dist'),
3   // 明确告诉 webpack 把生成的 bundle.js 文件存放到 dist 目录下的 js 子目录中
4   filename: 'js/bundle.js',
5 }
```

## 把图片文件统一生成到 image 目录中

修改webpack.config.js 中的 url-loader 配置项，新增outputPath 选项即可指定图片文件的输出路径：

```
1 {
2   test: /\.jpg|png|gif$/,
3   use: {
4     loader: 'url-loader',
5     options: {
6       limit: 22228,
7       // 明确指定把打包生成的图片文件，存储到 dist 目录下的 image 文件夹中
8       outputPath: 'image',
9     },
10   },
11 }
```

## 自动清理 dist 目录下的旧文件

为了在每次打包发布时自动清理掉dist 目录中的旧文件，可以安装并配置clean-webpack-plugin 插件：

```
1 // 1. 安装清理 dist 目录的 webpack 插件
2 npm install clean-webpack-plugin@3.0.0 -D
3
4 // 2. 按需导入插件、得到插件的构造函数之后，创建插件的实例对象
5 const { CleanWebpackPlugin } = require('clean-webpack-plugin')
6 const cleanPlugin = new CleanWebpackPlugin()
7
8 // 3. 把创建的 cleanPlugin 插件实例对象，挂载到 plugins 节点中
9 plugins: [htmlPlugin, cleanPlugin], // 挂载插件
```

## Source Map

### 生产环境遇到的问题

前端项目在投入生产环境之前，都需要对 JavaScript 源代码进行压缩混淆，从而减小文件的体积，提高文件的加载效率。此时就不可避免的产生了另一个问题：

对压缩混淆之后的代码除错（debug）是一件极其困难的事情，下面是压缩过的代码：

```

20tAAAAAASUVORK5CYII="},t={};function n(r){if(t[r])return t[r].exports;var o=t[r]={id:r,exports:{}},return e[r].call(o,exports,o,o.exports,n),o.exports}n.n=e=>{var t=e&&e.__esModule?()=>e.default:()=>e;return n.d(t,{a:t}),t},n.d=(e,t)=>{for(var r in t)n.o(t,r)&&!n.o(e,r)&&Object.defineProperty(e,r,{enumerable:!0,get:t[r]}),n.o=(e,t)=>Object.prototype.hasOwnProperty.call(e,t),(()=>{"use strict";var e=n(755),t=n.n(e),r=n(379),o=n.n(r),i=n(340);o()(i.Z,{insert:"head",singleton:!1}),i.Z.locals;var a,s,u,l=n(543);o()(l.Z,{insert:"head",singleton:!1}),l.Z.locals,t()((function(){t("li:odd").css("backgroundColor","pink"),t("li:even").css("backgroundColor","red"))));class c{}u="person info",s="info")in(a=c)?Object.defineProperty(a,s,{value:u,enumerable:!0,configurable:!0,writable:!0}):a.info=u,console.log(c.info))()})();

```

有如下的问题：

- 变量被替换成没有任何语义的名称
- 空行和注释被剔除

## 什么是 Source Map

Source Map 就是一个信息文件，里面储存着位置信息。也就是说，Source Map 文件中存储着压缩混淆后的代码，所对应的转换前的位置。

有了它，出错的时候，除错工具将直接显示原始代码，而不是转换后的代码，能够极大的方便后期的调试。

## webpack 开发环境下的 Source Map

在开发环境下，webpack 默认启用了 Source Map 功能。当程序运行出错时，可以直接在控制台提示错误行的位置，并定位到具体的源代码：

The screenshot shows a browser's developer tools. On the left, there's a tree view of files: 'top', '127.0.0.1' (with 'index' and 'bundle.js' expanded), and 'change-rows-color'. In the center, a file named 'index.js' is open, showing code from line 8 to 21. Line 20 contains the error: 'console.log(Person.info);'. A red box highlights the line number '20'. On the right, a stack trace is displayed in a pink box, starting with 'Uncaught ReferenceError: consle is not defined' and pointing to 'index.js:20'. Another red box highlights 'index.js:20'.

```

✖ Uncaught ReferenceError: consle is not defined
  at eval (index.js:20)
  at Module../src/index.js (bundle.js:50)
  at __webpack_require__ (bundle.js:600)
  at bundle.js:674
  at bundle.js:677

```

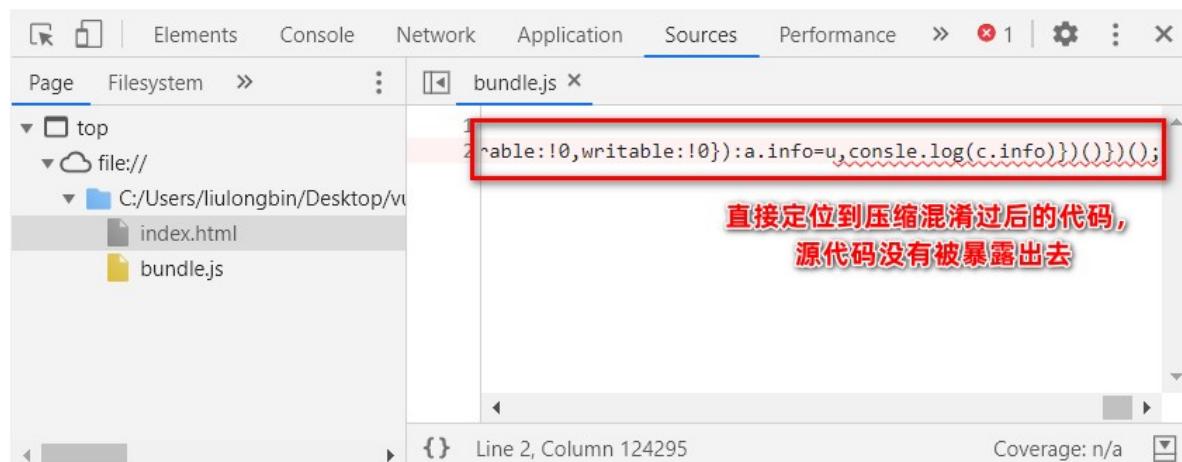
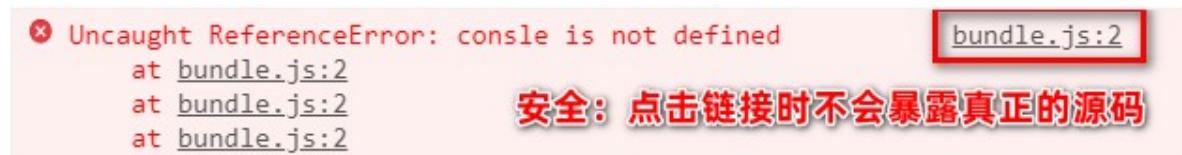
## 解决默认 Source Map 的问题

开发环境下，推荐在webpack.config.js 中添加如下的配置，即可保证运行时报错的行数与源代码的行数保持一致：

```
1 module.exports = {
2   mode: 'development',
3   // eval-source-map 仅限在"开发模式"下使用，不建议在"生产模式"下使用。
4   // 此选项生成的 Source Map 能够保证"运行时报错的行数"与"源代码的行数"保持一致
5   devtool: 'eval-source-map',
6   // 省略其它配置项...
7 }
```

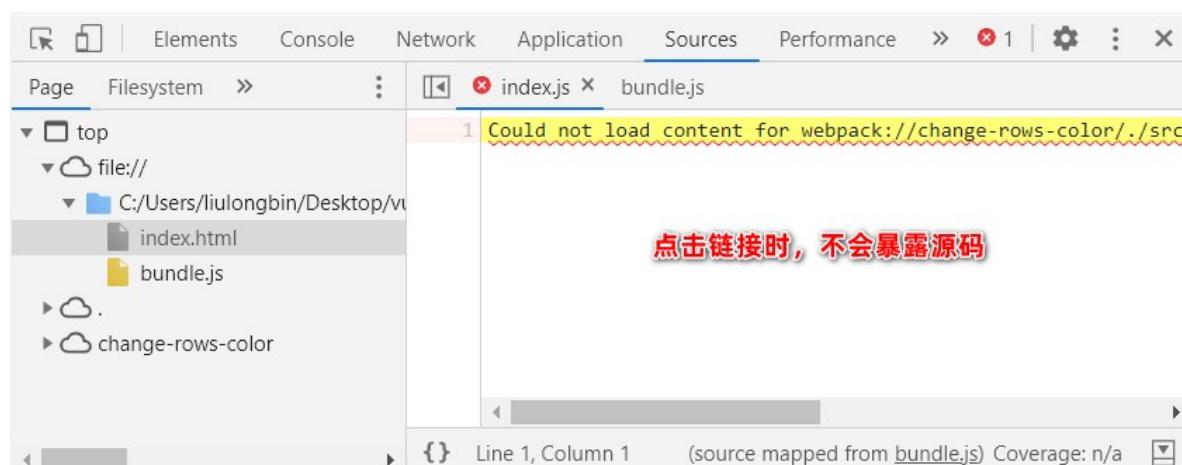
## webpack 生产环境下的 Source Map

在生产环境下，如果省略了devtool 选项，则最终生成的文件中不包含 Source Map。这能够防止原始代码通过 Source Map 的形式暴露给别有所图之人。



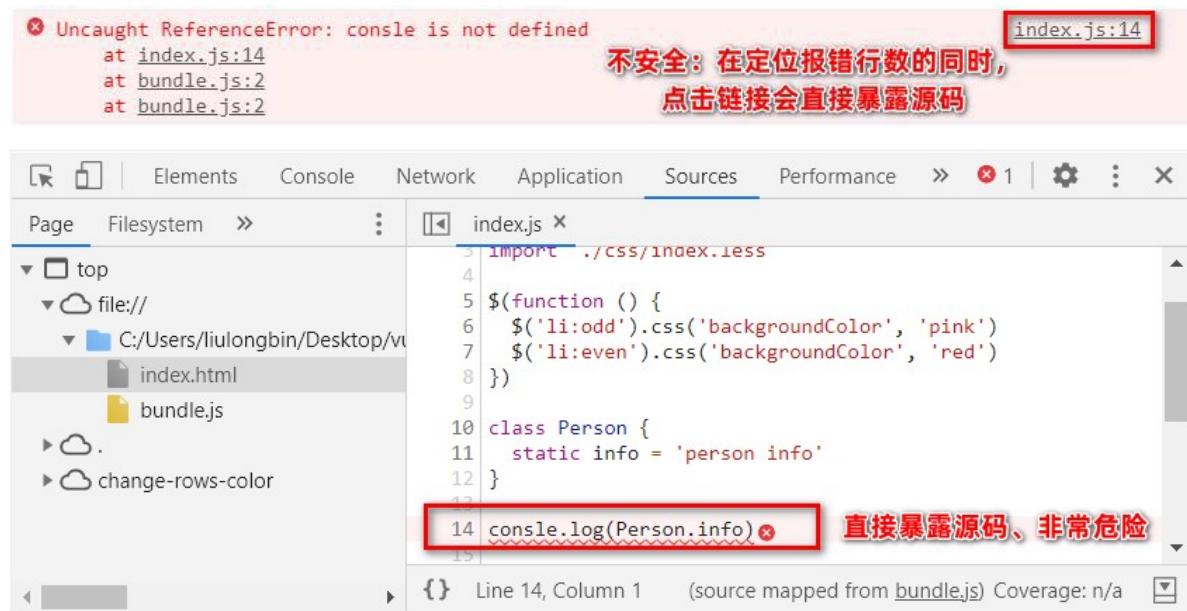
## 只定位行数不暴露源码

在生产环境下，如果只想定位报错的具体行数，且不想暴露源码。此时可以将 devtool 的值设置为 nosources-source-map。实际效果如图所示：



## 定位行数且暴露源码

在生产环境下，如果想在定位报错行数的同时，展示具体报错的源码。此时可以将devtool的值设置为source-map。实际效果如图所示：



采用此选项后：你应该将你的服务器配置为，不允许普通用户访问 source map 文件！

## Source Map 的最佳实践

① 开发环境下：

- 建议把devtool的值设置为eval-source-map
- 好处：可以精准定位到具体的错误行

② 生产环境下：

- 建议关闭 Source Map 或将devtool的值设置为nosources-source-map
- 好处：防止源码泄露，提高网站的安全性

## 实际开发中需要自己配置 webpack 吗？

答案：不需要！

- 实际开发中会使用命令行工具（俗称 CLI）一键生成带有 webpack 的项目
- 开箱即用，所有 webpack 配置项都是现成的！
- 我们只需要知道 webpack 中的基本概念即可！

## @符号的作用



配置@指向的位置，但是在学了vue后他会默认给我们配置了

```
dex1.js 1  msg.js  info.js  webpack.config.js ×
2/   entry: path.join(__dirname, './src/index1.js'),
28 // 指定生成的文件要存放到哪里
29 > output: { ...
34 },
35 // 3. 插件的数组，将来 webpack 在运行时，会加载并调用这些插件
36 plugins: [htmlPlugin, new CleanWebpackPlugin()],
37 > devServer: { ...
44 },
45 > module: { ...
60 },
61 resolve: {
62   alias: {
63     // 告诉 webpack，程序员写的代码中，@ 符号表示 src 这一层目录
64     '@': path.join(__dirname, './src/')
65   }
66 }
67 }
```

# Vue2

## vue2 基础入门

### Vue的特性

vue 框架的特性，主要体现在如下两方面：

- ① 数据驱动视图
- ② 双向数据绑定

### 数据驱动视图

在使用了vue 的页面中，vue 会监听数据的变化，从而自动重新渲染页面的结构。示意图如下：



好处：当页面数据发生变化时，页面会自动重新渲染！

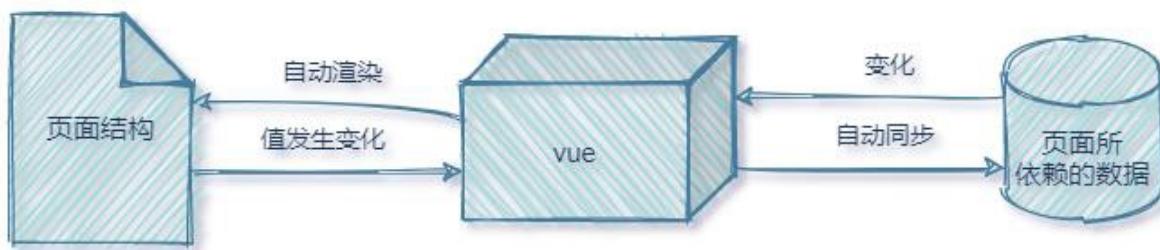
注意：数据驱动视图是单向的数据绑定。

## 双向数据绑定

在填写表单时，双向数据绑定可以辅助开发者在不操作 DOM 的前提下，自动把用户填写的内容同步到数据源中。示意图如下：

```
// javascript:  
var data = {  
    name: 'Bob',  
    email: 'bob@example.com'  
};
```

```
<!-- html -->  
<input name="name" type="text">  
<input name="email" type="text">
```

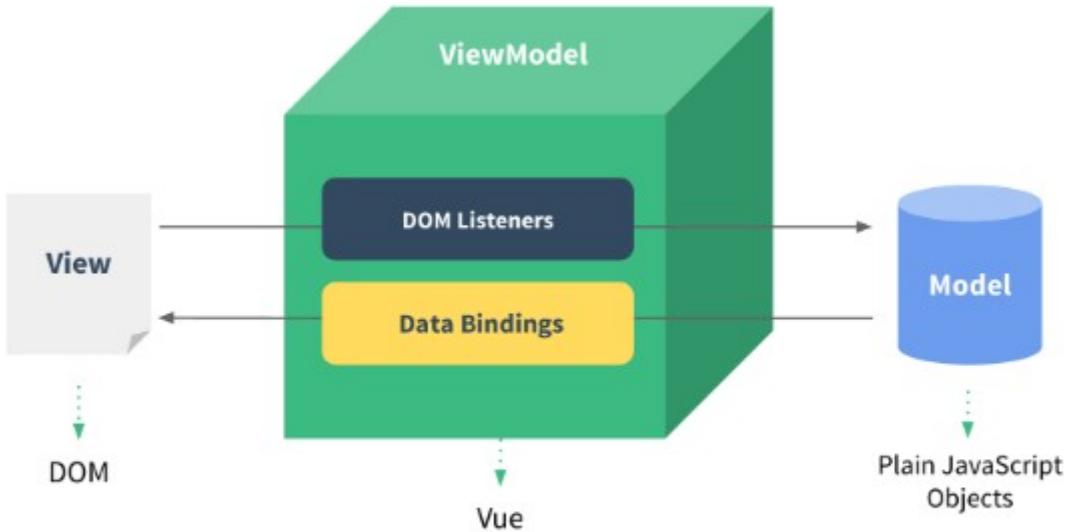


数据驱动视图

好处：开发者不再需要手动操作 DOM 元素，来获取表单元素最新的值！

## MVVM

MVVM 是 vue 实现数据驱动视图和双向数据绑定的核心原理。MVVM 指的是 Model、View 和 ViewModel，它把每个HTML 页面都拆分成了这三个部分，如图所示：



MVVM 示意图

在 MVVM 概念中：

Model 表示当前页面渲染时所依赖的数据源（也就是数据对象）。

View 表示当前页面所渲染的 DOM 结构（页面）。

ViewModel 表示 vue 的实例，它是MVVM 的核心。

## MVVM 的工作原理

ViewModel 作为 MVVM 的核心，是它把当前页面的数据源（Model）和页面的结构（View）连接在一起。



当数据源发生变化时，会被 ViewModel 监听到，VM 会根据最新的数据源自动更新页面的结构

当表单元素的值发生变化时，也会被 VM 监听到，VM 会把变化过后最新的值自动同步到 Model 数据源中

## vue 的基本使用

### 基本使用步骤

- ① 导入 vue.js 的 script 脚本文件
- ② 在页面中声明一个将要被 vue 所控制的 DOM 区域
- ③ 创建 vm 实例对象（vue 实例对象）

### 基本代码与 MVVM 的对应关系

```
1 <body>
2   <!-- 2. 在页面中声明一个将要被 vue 所控制的 DOM 区域 -->
3   <div id="app">{{username}}</div> → View 视图区域
4
5   <!-- 1. 导入 vue.js 的 script 脚本文件 -->
6   <script src=".lib/vue-2.6.12.js"></script>
7   <script>
8     // 3. 创建 vm 实例对象（vue 实例对象）
9     const vm = new Vue({
10       // 3.1 指定 当前 vm 实例要控制页面的那个区域
11       el: '#app', → ② el 指向的选择器，就是 View 视图区域
12       // 3.2 指定 Model 数据源
13       data: { → ① data 指向的对象，就是 Model 数据源
14         username: 'zs'
15       }
16     })
17   </script>
18 </body>
```

该代码示例展示了 Vue.js 基本使用与 MVVM 模型的对应关系。第 3 行代码 `<div id="app">{{username}}</div>` 是 View 视图区域。第 9 行代码 `const vm = new Vue({` 是新创建的 vm 实例对象。第 11 行代码 `el: '#app'` 指定了当前 vm 实例要控制的 DOM 区域，即 View 视图区域。第 13 行代码 `data: { username: 'zs' }` 指定了 Model 数据源。注释 ① 指向 `data` 对象，注释 ② 指向 `el` 属性。

注意el的value是一个选择器，这里使用了id选择器选择id为app的节点

配置对象，axios的那个语法其实也是配置对象，这个对于任何的这样形式都是适用的：

```
//创建Vue实例
new Vue({
  el:'#root', //el用于指定当前Vue实例为哪个容器服务，值通常为css选择器字符串。
  data:{ //data中用于存储数据，数据供el所指定的容器去使用，值我们暂时先写成一个对象。
    name:'尚硅谷'
})
学了一个新名词({这里面的是一个配置对象}), 配置对象里面包括挂载点和数据什么的
```

## 第一个vue程序

快捷：html 5导入模板

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>vue基础</title>
</head>
<body>
  <div id="app">
    {{ message }}
  </div>
  <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
  <script>
    var app =new Vue({
      el:"#app",
      data:{
        message:"hello vue!"
      }
    })
  </script>
</body>
</html>
```

## 创建代码片段

文件 => 首选项 => 用户代码片段 => 新建全局代码片段/或文件夹代码片段：

名称为：vue-html.code-snippets

```
{
  "vue html": {
    "scope": "html",
    "prefix": "!v",
    "body": [
      "<!DOCTYPE html>",
      "<html lang=\"en\">",
      "",
```

```

        "<head>",
        " <meta charset=\"UTF-8\">",
        " <meta name=\"viewport\" content=\"width=device-width, initial-
scale=1.0\">",
        " <meta http-equiv=\"X-UA-Compatible\" content=\"ie=edge\">",
        " <title>Document</title>",
        "</head>",
        "",
        "<body>",
        " <div id=\"app\">",
        "",
        " </div>",
        " <script src=\"https://cdn.jsdelivr.net/npm/vue/dist/vue.js\">
</script>",
        " <script>",
        " new Vue({",
        " el: '#app',
        " data: {",
        " $1",
        " }",
        " }",
        " </script>",
        "</body>",
        "",
        "</html>",
    ],
    "description": "my vue template in html"
}
}

```

我设置的快捷键是: `!v` 如果有需要可以自己改

## el挂载点

el是用来设置Vue实例挂载（管理）的元素

### 1.vue的作用范围是什么

在el命中的元素内部可以被渲染

Vue会管理el选项 **命中的元素及其内部的后代元素**

```

<body>
    {{message}}
    <div id="app">
        {{message}}
        <span>{{ message }}</span>
    </div>
    
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
    var app=new Vue({
        el:"#app",
        data:{
            message:"Hello Vue! "
        }
    })
</script>
</body>

```

## 2.这里使用了id选择器，那么是否可以选用其他的选择器

可以，但是建议使用id选择器

## 3.是否可以设置其他的dom元素

可以使用其他的双标签，但是不能使用HTML和BODY标签

## data 数据对象

1.Vue中用到的数据定义在data中

2.data中可以写复杂类型的数据

3.渲染复杂类型数据时，遵循js的语法即可。语法，数组的索引语法

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <div id="app">
        {{ message }}
        <h2>{{school.name}} {{school.mobile}}</h2>
        <ul>
            <li>{{campus[0]}}</li>
            <li>{{campus[1]}}</li>
    
```

```
</ul>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
var app = new Vue({
  el: "#app",
  data: {
    message: "hello, vue",
    school: {
      name: "小黑",
      mobile: "130302030302020"
    },
    campus: ["北京", "上海"]
  }
})
</script>
</body>
</html>
```

hello,vue

**小黑 130302030302020**

- 北京
- 上海

## 回顾Object.defineProperty方法

当我们通过控制台访问data里面的属性，在主界面不会立即显示，需要我们手动点击，当我们手动点击这个，就会自动触发getter方法

VM

```

<- Vue {_uid: 0, _isVue: true, $options: {...}, _renderProxy: Proxy, _self: Vue, ...} ⓘ
    $attrs: (...)
    $children: []
    $createElement: f (a, b, c, d)
    $el: div#root
    $listeners: (...)
    $options: {components: {...}, directives: {...}, filters: {...}, el: "#root", _parent: undefined}
    $parent: undefined
    $refs: {}
    $root: Vue {_uid: 0, _isVue: true, $options: {...}, _renderProxy: Proxy, _self: Vue, ...} ⓘ
    $scopedSlots: {}
    $slots: {}
    $vnode: undefined
    address: (...)
    name: (...) 这里也是vue给我们加的getter
    _c: f (a, b, c, d)
    _data: {__ob__: Observer}
    _directInactive: false
    _events: {}
    _hasHookEvent: false
    _inactive: null
  
```

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>回顾Object.defineProperty方法</title>
  </head>
  <body>
    <script type="text/javascript" >
      let number = 18
      let person = {
        name:'张三',
        sex:'男',
      }

      Object.defineProperty(person, 'age',{
        // value:18,
        // enumerable:true, //控制属性是否可以枚举， 默认值是false
        // writable:true, //控制属性是否可以被修改， 默认值是false
        // configurable:true //控制属性是否可以被删除， 默认值是false

        //当有人读取person的age属性时， get函数(getter)就会被调用，且返回值就是age
        //的值
        get(){
          console.log('有人读取age属性了')
          return number
        },
        //当有人修改person的age属性时， set函数(setter)就会被调用，且会收到修改的
        //具体值
        set(value){
          console.log('有人修改了age属性，且值是',value)
        }
      }
    </script>
  </body>
</html>

```

```

        number = value
    }

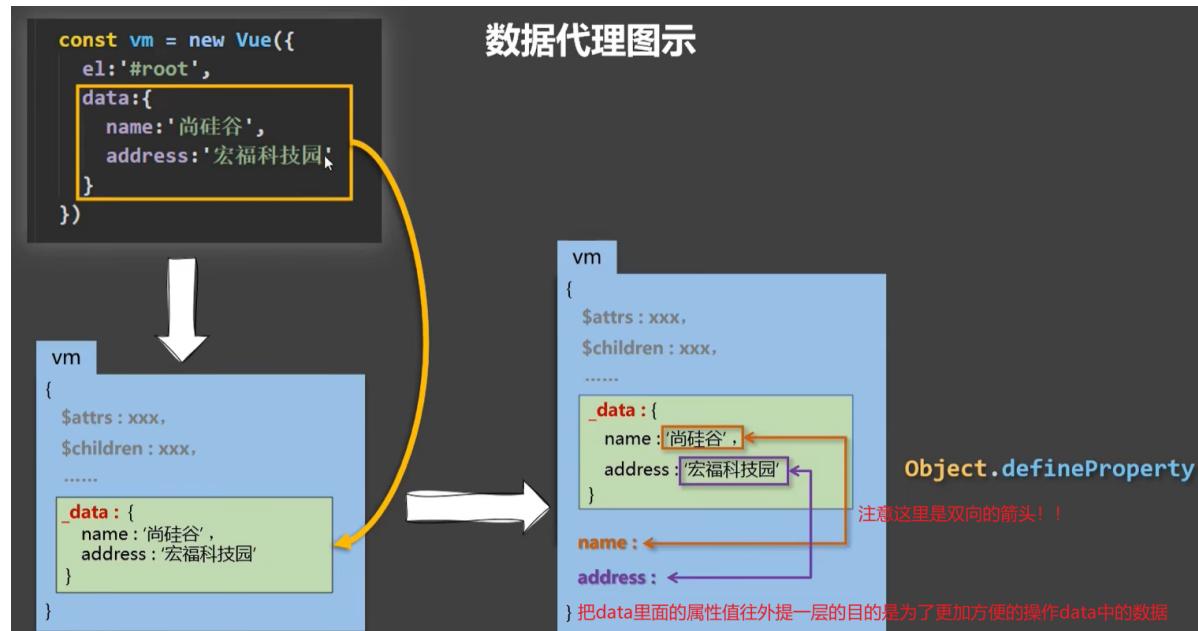
})

// console.log(Object.keys(person))

console.log(person)
</script>
</body>
</html>

```

vue里面的数据代理！！！这也是为什么vue把data里面的数据直接提到外面一层，是为了更方便访问属性



### 1.Vue中的数据代理：

通过vm对象来代理data对象中属性的操作（读/写）

### 2.Vue中数据代理的好处：

更加方便的操作data中的数据

### 3.基本原理：

通过`Object.defineProperty()`把data对象中所有属性添加到vm上。

为每一个添加到vm上的属性，都指定一个getter/setter。

在getter/setter内部去操作（读/写）data中对应的属性。

# vue 的指令

## js表达式

怎么区分 js 表达式 和 js 语句(javascript 代码) 的区别

表达式一定会生成一个值！！！无论是函数的调用还是四则运算，都会生成值

1. 表达式：一个表达式会产生一个值，可以放在任何一个需要值的地方：

- (1). a
- (2). a+b
- (3). demo(1)
- (4). x === y ? 'a' : 'b'

2. js 代码(语句)

- (1). if(){}
- (2). for(){}

## 指令的概念

指令 (Directives) 是 vue 为开发者提供的模板语法，用于辅助开发者渲染页面的基本结构。

vue 中的指令按照不同的用途可以分为如下 6 大类：

- ① 内容渲染指令
- ② 属性绑定指令
- ③ 事件绑定指令
- ④ 双向绑定指令
- ⑤ 条件渲染指令
- ⑥ 列表渲染指令

注意：指令是vue 开发中最基础、最常用、最简单的知识点。

## 内容渲染指令

内容渲染指令用来辅助开发者渲染 DOM 元素的文本内容。常用的内容渲染指令有如下 3 个：

- 1. v-text
- 2. {{ }}
- 3. v-html

### v-text

1.v-text 指令的作用：设置标签的内容 (textContent)

2. 默认写法会替换全部内容，使用差值表达式 {{}} 可以替换指定内容。(也就是说，不管你标签里面写了啥，都会覆盖标签中原有的内容)所以说这种写法基本上都不使用。

### 3. 内部支持写表达式 (如字符串拼接)

```
v-text指令.html > html > head > meta
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5      <meta charset="UTF-8">
6      <meta name="viewport" content="width=device-width,
initial-scale=1.0">
7      <meta http-equiv="X-UA-Compatible" content="ie=edge">
8  <title>v-text指令</title>
9 </head>
10 <body>
11     <div id="app">
12         <h2 v-text="message">深圳</h2>
13         <h2 v-text="info">深圳</h2>
14         <h2>{{ message }}深圳</h2>
15     </div>
16     <!-- 开发环境版本，包含了有帮助的命令行警告 -->
17     <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
18     <script>
19         var app = new Vue({
20             el:"#app",
21             data:{
22                 message:"黑马程序员!!!",
23                 info:"前端与移动教研部"
24             }
25         })
26     </script>
27 </body>
28 </html>
```



## v-text

设置标签的文本值(textContent)

内容全部替换

```
<div id="app">
    <h2 v-text="message"></h2>
    <h2>深圳{{ message }}</h2>
</div>
```

局部替换

```
var app = new Vue({
    el:"#app",
    data:{
        message:"黑马程序员"
    }
})
```

字符的拼接

```
<div id="app">
    <h2 v-text="message+'!'"></h2>
    <h2>深圳{{ message + "!" }}</h2>
</div>
```

只能用单

能用单  
也能用双

```
var app = new Vue({
    el:"#app",
    data:{
        message:"黑马程序员"
    }
})
```

```

1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta name="viewport" content="width=device-width,
7     initial-scale=1.0">
8   <meta http-equiv="X-UA-Compatible" content="ie=edge">
9   <title>v-text指令</title>
10 </head>
11
12 <body>
13   <div id="app">
14     <h2 v-text="message+'!'">深圳</h2>
15     <h2 v-text="info+'!'">深圳</h2>
16     <h2>{{ message + info }}深圳</h2>
17   </div>
18   <!-- 开发环境版本，包含了有帮助的命令行警告 -->
19   <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
20   <script>
21     var app = new Vue({
22       el: "#app",
23       data: {
24         message: "黑马程序员!!!",
25         info: "前端与移动教研部"
26       }
27     })
28   </script>
29 </body>
30 </html>

```

## 插值表达式

在这里面可以写我们创建的Vue对象身上的任何属性，不需要加Vue,也就是说你可以在插值表达式里面通过属性名字访问所有属性，不用加任何前缀

```

Vue {_uid: 0, _isVue: true, $options: {...}, _renderProxy: Proxy(Vue), _self: Vue, ...} ⓘ
  $attrs: (...)

  ▶ $children: []
  ▶ $createElement: f (a, b, c, d)
  ▶ $el: div#root
  $listeners: (...)

  ▶ $options: {components: {...}, directives: {...}, filters: {...}, el: '#root', _base: f, ...}
  $parent: undefined
  ▶ $refs: {}
  ▶ $root: Vue {_uid: 0, _isVue: true, $options: {...}, _renderProxy: Proxy(Vue), _self: Vue, ...}
  ▶ $scopedSlots: {}
  ▶ $slots: {}
  $vnode: undefined

```

vue 提供的 `{{ }}` 语法，专门用来解决v-text 会覆盖默认文本内容的问题。这种 `{{ }}` 语法的专业名称是插值表达式（英文名为：Mustache）。

```

1 <!-- 使用 {{ }} 插值表达式，将对应的值渲染到元素的内容节点中， -->
2 <!-- 同时保留元素自身的默认值 -->
3 <p>姓名: {{username}}</p>
4 <p>性别: {{gender}}</p>

```

注意：相对于v-text 指令来说，插值表达式在开发中更常用一些！因为它不会覆盖元素中默认的文本内容。

## v-html指令

### 1.与插值语法的区别：

(1).v-html会替换掉节点中所有的内容，{{xx}}则不会。

(2).v-html可以识别html结构。

2.严重注意：v-html有安全性问题！！！！

(1).在网站上动态渲染任意HTML是非常危险的，容易导致XSS攻击。

(2).一定要在可信的内容上使用v-html，永不要用在用户提交的内容上！

1. v-html指令的作用是：设置元素的innerHTML

2. 内容中有html结构会被解析为标签

3. v-text指令无论内容是什么，只会解析为文本

4. 解析文本使用v-text

5. 需要解析html结构使用v-html

```
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width,
  initial-scale=1.0" />
  <meta http-equiv="X-UA-Compatible" content="ie=edge" />
  <title>v-html指令</title>
</head>
<body>
  <!-- 2.html结构 -->
  <div id="app">
    <p v-html="content"></p>
    <p v-text="content"></p>
  </div>
  <!-- 1.开发环境版本，包含了有帮助的命令行警告 -->
  <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
  <script>
    // 3.创建Vue实例
    var app = new Vue({
      el:"#app",
      data:{
        // content:"黑马程序员"
        content:<a href='http://www.itheima.com'>黑马程序员</a>
      }
    })
  </script>
</body>
</html>
```

## 属性绑定指令

如果需要为元素的属性动态绑定属性值（注意是属性，之前的插值表达式是写到**标签体**里面的，而这个是写到**标签的属性**上的），则需要用到v-bind属性绑定指令。用法示例如下：

### v-bind指令

加了v-bind指令，引号中的就是一个js表达式了！！！！至于js表达式是什么，在上面提到过！这个必须要注意，用到的有很多

```
<div id="app">
  
  <img v-bind:title="imgTitle+'!!!!'">
  <img v-bind:class="isActive?'active':''">
  <img v-bind:class="{active:isActive}">
</div>
```

两种写法

```
var app = new Vue({
  el:"#app",
  data:{
    imgSrc:"图片地址",
    imgTitle:"黑马程序员",
    isActive:false
  }
})
```

可以直接省略v-bind

```

<div id="app">
  <img :src= "imgSrc" >
  <img :title="imgTitle+'!!!!'">
  <img :class="isActive?'active':''">
  <img :class="{active:isActive}">
</div>

```

```

var app = new Vue({
  el:"#app",
  data:{
    imgSrc:"图片地址",
    imgTitle:"黑马程序员",
    isActive:false
  }
})

```

1.v-bind: 属性名=表达式

2.v-bind指令的作用是:为元素绑定属性

3.完整写法是v-bind:属性名

4.简写的话可以直接省略v-bind,只保留:属性名

5.需要动态的增删class建议使用对象的方式

```

v-bind指令.html
v-bind指令.html > html > body > div#app > img
6   <meta name="viewport" content="width=device-width,
7     initial-scale=1.0">
8   <meta http-equiv="X-UA-Compatible" content="ie=edge">
9   <title>v-bind指令</title>
10  <style>
11    .active{
12      border: 1px solid red;
13    }
14  </style>
15
16 <body>
17  <div id="app">
18    
19    <br>
20    
21    <br>
22    
24  </div>
25  <!-- 开发环境版本，包含了有帮助的命令行警告 -->
26  <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
27  <script>
28    var app = new Vue({
29      el:"#app",
30      data:{
31        imgSrc:"http://www.itheima.com/images/logo.png",
32        imgTitle:"黑马程序员",
33        isActive:false
34      },
35      methods: {
        toggleActive:function(){

```

```

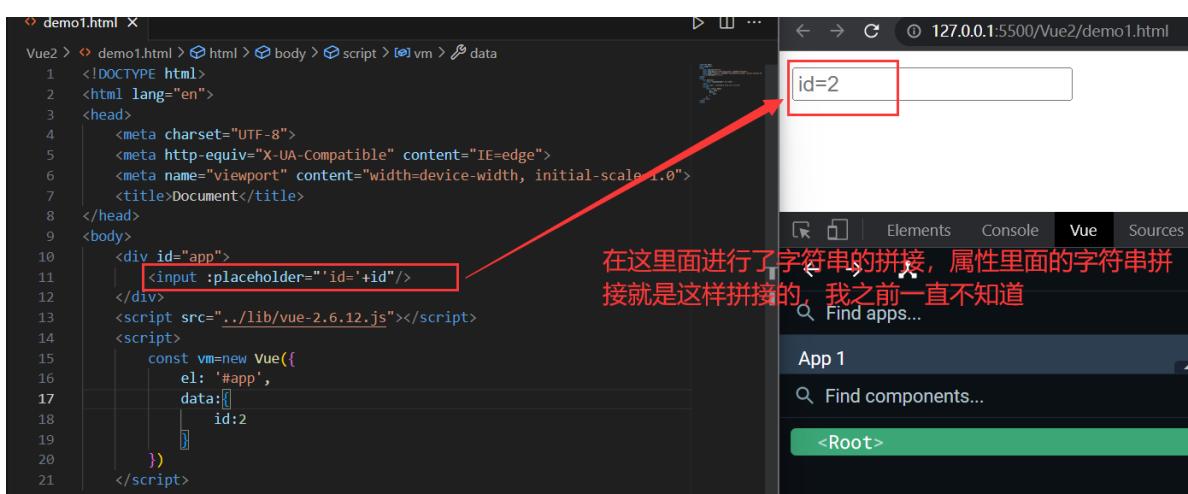
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
      .active{
        border: 2px solid red;
      }
    </style>
  </head>

```

```

<body>
    <div id="app">
        
        <!-- <br>      ?      :   如果是    就怎么 -->
        
        
            <!-- 点击变色                                         active取值依赖于isActive是否取值 -->
        >
    </div>
    <!-- 开发环境版本，包含了有帮助的命令行警告 -->
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <script>
        var app=new Vue({
            el:"#app",
            data:{
                imgSrc:"https://img0.baidu.com/it/u=2753883117,2936626650&fm=26&fmt=auto",
                imgTitle:"壁纸",
                isActive:false
            },
            methods:{
                toggleActive:function(){
                    this.isActive=!this.isActive;
                }
            }
        })
    </script>
</body>
</html>

```



在使用v-bind属性绑定期间，如果绑定内容需要进行动态的字符串拼接，则字符串的外面应该包括单引号，后面可以拼接一个变量，这个变量要去data里面去找

## 事件绑定指令

vue 提供了v-on 事件绑定指令，用来辅助程序员为 DOM 元素绑定事件监听。语法格式如下：

注意：原生 DOM 对象有onclick、oninput、onkeyup 等原生事件，替换为vue 的事件绑定形式后，分别为：v-on:click、v-on:input、v-on:keyup

### v-on指令

示例代码：

```
<div id="app">  
  <input type="button" value="事件绑定" v-on:事件名="方法">  
</div>
```

```
var app = new Vue({  
  el:"#app",  
})
```

举例子

```
<div id="app">  
  <input type="button" value="事件绑定" v-on:click="dolt">  
  <input type="button" value="事件绑定" v-on:mouseenter="dolt">  
  <input type="button" value="事件绑定" v-on:dblclick="dolt">  
  <input type="button" value="事件绑定" @dblclick="dolt">  
</div>
```

```
var app = new Vue({  
  el:"#app",  
  methods:{  
    dolt:function(){  
      // 逻辑  
    }  
  }  
})
```

1. v-on指令的作用是:为元素绑定事件(点击， 移入....)
2. 事件名不需要写on
3. 指令可以简写为@
4. 绑定的方法定义在methods属性中
5. 方法内部通过this关键字可以访问定义在data中数据

```
<meta name="viewport" content="width=device-width, initial-scale=1.0" />  
<meta http-equiv="X-UA-Compatible" content="ie=edge" />  
<title>v-on指令基础</title>  
</head>  
<body>  
  <!-- 2.html结构 -->  
  <div id="app">  
    <input type="button" value="v-on指令" v-on:click="doIt">  
    <input type="button" value="v-on简写" @click="doIt">  
    <input type="button" value="双击事件" @dblclick="doIt">  
    <h2 @click="changeFood">{{ food }}</h2>  
  </div>  
  <!-- 1.开发环境版本，包含了有帮助的命令行警告 -->  
  <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>  
<script>  
  // 3.创建Vue示例  
  var app = new Vue({  
    el: "#app",  
    data:{  
      food:"西兰花炒蛋"  
    },  
    methods: {  
      doIt:function(){  
        alert("做It");  
      },  
      changeFood:function(){  
        // console.log(this.food);  
        this.food+="好好吃!"  
      }  
    }  
  })  
</script>  
</body>  
<html>
```

## 事件参数对象

在原生的 DOM 事件绑定中，可以在事件处理函数的形参处，接收事件参数对象 event。（但是，仅限于没有传参的函数，如果传参了，这个e就不会传进来了）同理，在v-on 指令（简写为@）所绑定的事件处理函数中，同样可以接收到事件参数对象event，（这个e对象就是触发事件对象，比如点击事件对象，e.target就是事件源，也就是那个触发事件的dom元素，这里是button按钮）示例代码如下：

```
1 <h3>count 的值为: {{count}}</h3>
2 <button v-on:click="addCount">+1</button>
3 // -----分割线-----
4 methods: {
5   addCount(e) { // 接收事件参数对象 event，简写为 e
6     const nowBgColor = e.target.style.backgroundColor
7     e.target.style.backgroundColor = nowBgColor === 'red' ? '' : 'red'
8     this.count += 1
9   }
10 }
```

## 绑定事件并传参

在使用v-on 指令绑定事件时，可以使用()进行传参，示例代码如下：

```
1 <h3>count 的值为: {{count}}</h3>
2 <button @click="addNewCount(2)">+2</button>
3 // -----分割线-----
4 methods: {
5   // 在形参处用 step 接收传递过来的参数值
6   addNewCount(step) {
7     this.count += step
8   }
9 }
```

## \$event

*event*是*vue*提供的特殊变量，用来表示原生的事件参数对象*event*。*event*可以解决事件参数对象*event*被覆盖的问题。示例用法如下：

```

1 <h3>count 的值为: {{count}}</h3>
2 <button @click="addNewCount(2, $event)">+2</button>
3 // -----分割线-----
4 methods: {
5   // 在形参处用 e 接收传递过来的原生事件参数对象 $event
6   addNewCount(step, e) {
7     const nowBgColor = e.target.style.backgroundColor
8     e.target.style.backgroundColor = nowBgColor === 'cyan' ? '' : 'cyan'
9     this.count += step
10  }
11 }

```

## 事件修饰符

在原生的JavaScript中，事件处理函数中调用event.preventDefault() 或 event.stopPropagation() 是非常常见的需求。因此，

vue 提供了事件修饰符的概念，来辅助程序员更方便的对事件的触发进行控制。常用的 5 个事件修饰符如下：

事件修饰符	说明
.prevent	阻止默认行为（例如：阻止 a 连接的跳转、阻止表单的提交等）
.stop	阻止事件冒泡
.capture	以捕获模式触发当前的事件处理函数
.once	绑定的事件只触发1次
.self	只有在 event.target 是当前元素自身时触发事件处理函数

修饰符 (Modifiers) 是以半角句号 (.) 指明的特殊后缀，用于指出一个指令应该以特殊方式绑定。

例如：

.prevent 修饰符告诉 v-on 指令对于触发的事件调用 event.preventDefault(),即阻止事件原本的默认行为

语法格式如下：

```

1 <!-- 触发 click 点击事件时，阻止 a 链接的默认跳转行为 -->
2 <a href="https://www.baidu.com" @click.prevent="onLinkClick">百度首页</a>

```

阻止默认行为然后触发onLinkClick函数，这个函数是我们自己定义的。。。

```

<!DOCTYPE html>
<html lang="en">

```

```

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>

<body>
  <div id="app">
    <!--阻止表单的默认提交行为，然后触发onSubmit函数-->
    <form action="localhost:8080/save" v-on:submit.prevent="onSubmit">
      <input type="text" id="name" v-model="user.username" />
      <button type="submit">保存</button>
    </form>
  </div>
  <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  new Vue({
    el: '#app',
    data: {
      user: {
        username: "请输入你的密码"
      }
    },
    methods: {
      onSubmit() {
        if(this.user.username){
          console.log('提交表单')
        }else{
          alert("请输入用户名")
        }
      }
    }
  })
</script>
</body>
</html>

```

按下按钮后，阻止表单提交到指定地址，并且，触发onSubmit()方法

### 按键修饰符

(tab键必须配合keydown使用，因为按下tab键会失去焦点)

在监听键盘事件时，我们经常需要判断详细的按键。此时，可以为键盘相关的事件添加按键修饰符，例如：

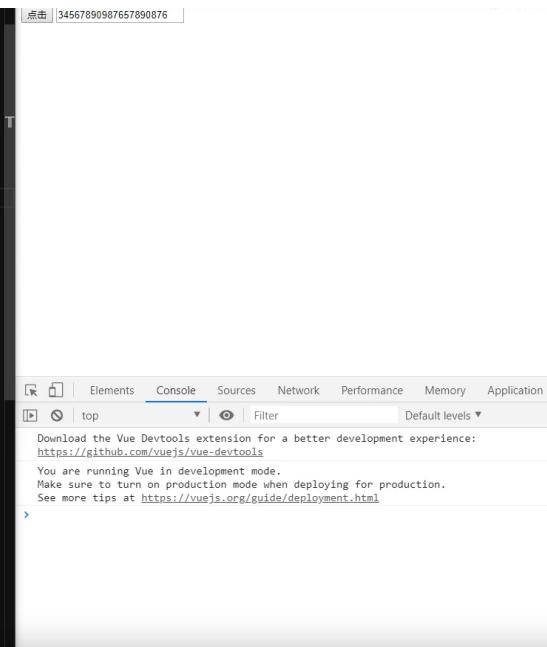
```

1 <!-- 只有在 `key` 是 `Enter` 时调用 `vm.submit()` -->
2 <input @keyup.enter="submit">
3
4 <!-- 只有在 `key` 是 `Esc` 时调用 `vm.clearInput()` -->
5 <input @keyup.esc="clearInput">

```

submit 和 clearInput 这两个函数都是我们自己定义的

## 模板



```

3
4 <head>
5   <meta charset="UTF-8">
6   <meta name="viewport" content="width=device-width,
initial-scale=1.0">
7   <meta http-equiv="X-UA-Compatible" content="IE=edge">
8   <title>v-on补充</title>
9 </head>
10 <body>
11   <div id="app">
12     <input type="button" value="点击" @click="doIt(666,'老铁')">
13     <input type="text" @keyup.enter="sayHi">
14   </div>
15   <!-- 1. 开发环境版本，包含了有帮助的命令行警告 -->
16   <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
17   <script>
18     var app = new Vue({
19       el:"#app",
20       methods: {
21         doIt:function(p1,p2){
22           console.log("点it");
23           console.log(p1);
24           console.log(p2);
25         },
26         sayHi:function(){
27           alert("吃了没");
28         }
29       },
30     })
31   </script>
32 </body>
33

```

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
  </head>
  <body>

    <!-- 2.html结构 -->
    <div id="app">
      <input type="button" value="点击" @click="doIt(666,'老铁')">
      <input type="text" @keyup.enter="sayHi">

    </div>
    <!-- 1. 开发环境版本，包含了有帮助的命令行警告 -->
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <!-- 3. 创建Vue实例 -->
    <script>
      var app=new Vue({
        el:"#app",
        methods:{
          doIt:function(p1,p2){

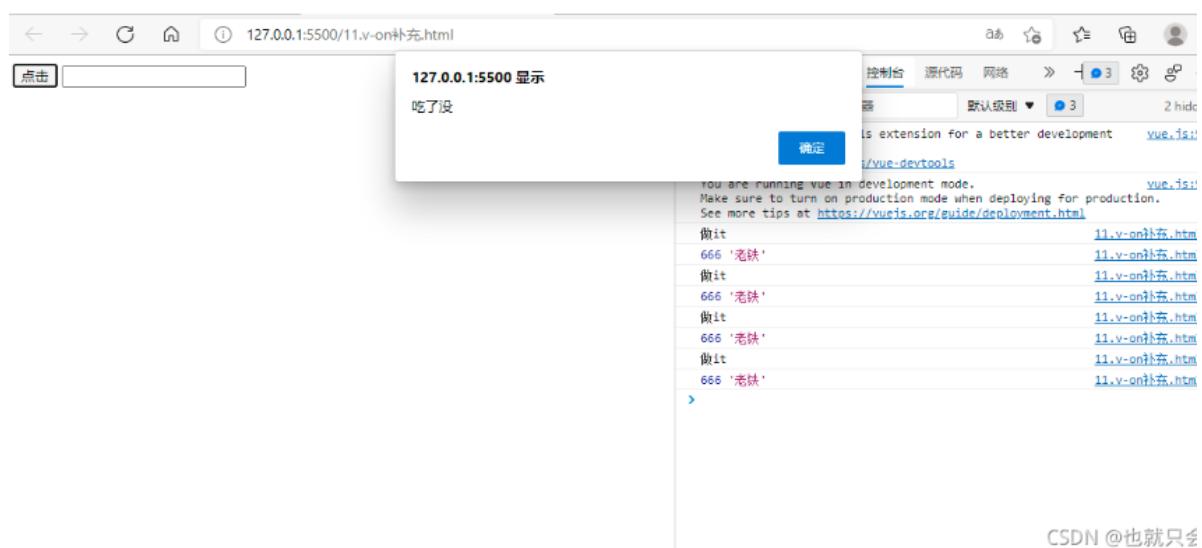
```

```

        console.log("做it");
        console.log(p1,p2);
        alert("吃了没");
        alert(p1,p2);
    },
    sayHi:function(){
        alert("吃了没");
    }
}
})
</script>

</body>
</html>

```



CSDN @也就只会

## 双向绑定指令

Vue 提供了 v-model 双向数据绑定指令，用来辅助开发者在不操作 DOM 的前提下，快速获取表单的数据。

只有表单元素才能使用 v-model 指令，其他标签用它没意义

比如：

- input
- select
- textarea

```

1 <p>用户名是: {{username}}</p>
2 <input type="text" v-model="username" />
3
4 <p>选中的省份是: {{province}}</p>
5 <select v-model="province">
6   <option value="">请选择</option>
7   <option value="1">北京</option>
8   <option value="2">河北</option>
9   <option value="3">黑龙江</option>
10 </select>

```

## v-model

v-model一般都应用在表单类元素上 (input、select这种有value属性的标签上)

```

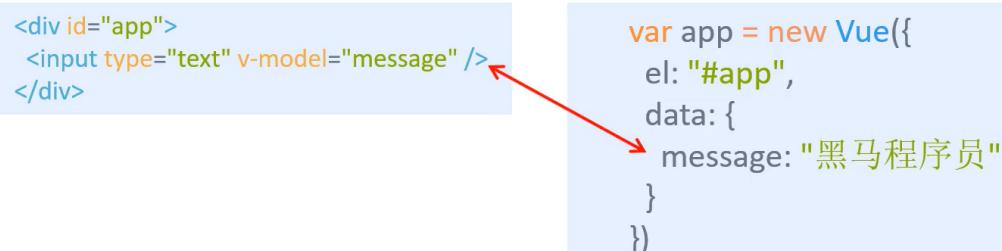
<!-- 普通写法 -->
<!-- 单向数据绑定: <input type="text" v-bind:value="name"><br/>
双向数据绑定: <input type="text" v-model:value="name"><br/> -->

<!-- 简写 --> 这是上面两种方法的简写方式
单向数据绑定: <input type="text" :value="name"><br/>
双向数据绑定: <input type="text" v-model="name"><br/>

```

简单来说双向绑定就是指修改文本框中的message，也会改变data中的message。

1. v-model: 获取和设置表单元素的值(双向数据绑定)



2. v-model指令的作用是：便捷的设置、获取表单元素的值

3. 绑定的数据会和表单元素值相关联

```

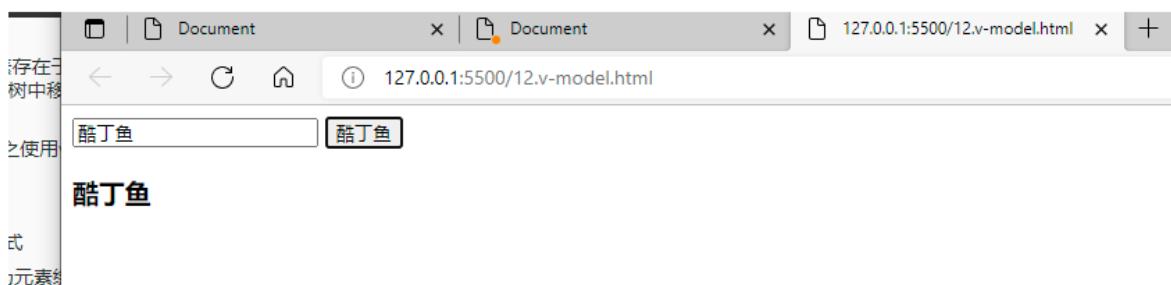
<!-- 2.html结构 -->
<div id="app">
  <input type="text" v-model="message" @keyup.enter="getMessage" />
  <input type="button" v-model="message" @click="setMessage" />
  <h3>{{message}}</h3>
</div>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<!-- 3.创建Vue实例 -->
<script>

```

```

var app=new Vue({
    el:"#app",
    data:{
        message:"沙丁鱼"
    },
    methods:{
        getMessage:function(){
            alert(this.message)
        },
        setMessage:function(){
            this.message="酷丁鱼";
        }
    }
})
</script>

```



### v-model 指令的修饰符

为了方便对用户输入的内容进行处理，vue 为 v-model 指令提供了 3 个修饰符，分别是：

修饰符	作用	示例
.number	自动将用户的输入值转为数值类型	<input type="text"/>
.trim	自动过滤用户输入的首尾空白字符	<input type="text"/>
.lazy	在“change”时而非“input”时更新	<input type="text"/>

```

1 <input type="text" v-model.number="n1"> +
2 <input type="text" v-model.number="n2"> =
3 <span>{{n1 + n2}}</span>

```

### 利用v-model手机表单数据

```

<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>收集表单数据</title>
        <script type="text/javascript" src="../js/vue.js"></script>
    </head>
    <body>

```

```

<!--
收集表单数据:
若: <input type="text"/>, 则v-model收集的是value值, 用户输入的就是
value值。
若: <input type="radio"/>, 则v-model收集的是value值, 且要给标签配
置value值。
若: <input type="checkbox"/>
1. 没有配置input的value属性, 那么收集的就是checked (勾选 or
未勾选, 是布尔值)
2. 配置input的value属性:
(1)v-model的初始值是非数组, 那么收集的就是checked
(勾选 or 未勾选, 是布尔值)
(2)v-model的初始值是数组, 那么收集的就是value组成的数组

备注: v-model的三个修饰符:
lazy: 失去焦点再收集数据
number: 输入字符串转为有效的数字
trim: 输入首尾空格过滤
-->
<!-- 准备好一个容器-->
<div id="root">
  <form @submit.prevent="demo">
    账号: <input type="text" v-model.trim="userInfo.account"> <br/>
<br/>
    密码: <input type="password" v-model="userInfo.password"> <br/>
<br/>
    年龄: <input type="number" v-model.number="userInfo.age"> <br/>
<br/>
    性别:
      男<input type="radio" name="sex" v-model="userInfo.sex"
value="male">
      女<input type="radio" name="sex" v-model="userInfo.sex"
value="female"> <br/><br/>
    爱好:
      学习<input type="checkbox" v-model="userInfo.hobby"
value="study">
      打游戏<input type="checkbox" v-model="userInfo.hobby"
value="game">
      吃饭<input type="checkbox" v-model="userInfo.hobby" value="eat">
<br/><br/>
    所属校区
    <select v-model="userInfo.city">
      <option value="">请选择校区</option>
      <option value="beijing">北京</option>
      <option value="shanghai">上海</option>
      <option value="shenzhen">深圳</option>
      <option value="wuhan">武汉</option>
    </select>
<br/><br/>
    其他信息:
    <textarea v-model.lazy="userInfo.other"></textarea> <br/><br/>
    <input type="checkbox" v-model="userInfo.agree">阅读并接受<a
href="http://www.atguigu.com">《用户协议》</a>
    <button>提交</button>
  </form>

```

```

        </div>
    </body>

    <script type="text/javascript">
        Vue.config.productionTip = false

        new Vue({
            el: '#root',
            data: {
                userInfo: {
                    account: '',
                    password: '',
                    age: 18,
                    sex: 'female',
                    hobby: [],
                    city: 'beijing',
                    other: '',
                    agree: ''
                }
            },
            methods: {
                demo() {
                    console.log(JSON.stringify(this.userInfo))
                }
            }
        })
    </script>
</html>

```

## 条件渲染指令

条件渲染指令用来辅助开发者按需控制 DOM 的显示与隐藏。条件渲染指令有如下两个，分别是：

- v-if
- v-show

示例用法如下：

下面的networkState也是我们自己定义的一个变量

```

1 <div id="app">
2     <p v-if="networkState === 200">请求成功 --- 被 v-if 控制</p>
3     <p v-show="networkState === 200">请求成功 --- 被 v-show 控制</p>
4 </div>

```

### v-show指令

1. show指令的作用

根据真假进行切换元素的显示

**状态原理**是修改元素的**display**,实现显示隐藏

2. 指令后面的内容,最终都会解析为布尔值
3. true元素显示, 值为false元素隐藏
4. 改变之后, 对应元素的显示状态会同步更新

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <div id="app">
        <input type="button" value="切换显示状态" @click="changeIsShow">
        <input type="button" value="累加年龄" @click="addAge">
        
        <img v-show="age>=18" src="./1.jpg">
    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <script>
        var app =new Vue({
            el: "#app",
            data:{
                isShow:false,
                age:17
            },
            methods:{
                changeIsShow:function(){
                    this.isShow=!this.isShow;
                },
                addAge:function(){
                    this.age++;
                }
            }
        })
    </script>

</body>
</html>

```

## v-if指令

1. v-if指令的作用是:根据表达式的真假切换元素的显示状态
2. 本质是通过操纵dom元素来切换显示状态
3. 表达式的值为true,把dom元素添加到dom树中。如果值为false,则从dom树中移除
4. 频繁的切换使用v-show。反之使用v-if, v-show如果多次进行切换消耗的资源小

v-if和v-show的区别: v-show直接修改display , 而v-if是直接抹除dom标签

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">

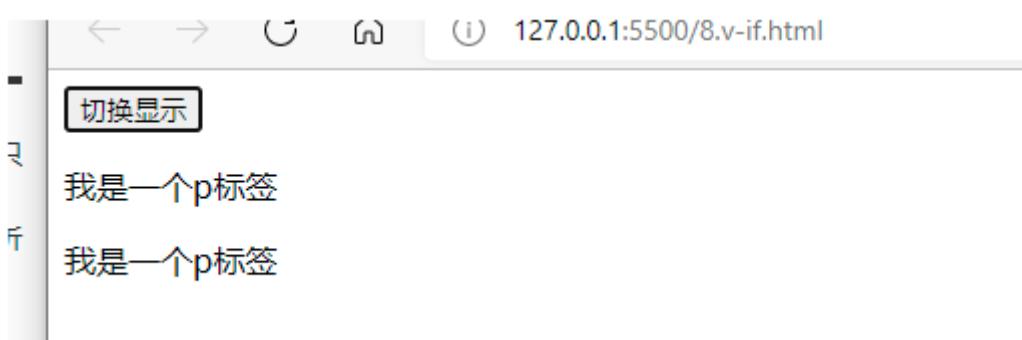
```

```

<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
</head>
<body>
    <div id="app">
        <input type="button" value="切换显示" @click="change">
        <p v-if="true">我是一个p标签</p>
        <p v-if="isShow">我是一个p标签</p>

    </div>
    <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
    <script>
        var app=new Vue({
            el:"#app",
            data:{
                isShow:false
            },
            methods:{
                change:function(){
                    this.isShow=!this.isShow;
                }
            }
        })
    </script>
</body>
</html>

```



在实际开发中，绝大多数情况，不用考虑性能，直接使用v-if就行了

v-if和template的配合使用

```
<!-- v-if与template的配合使用 -->
<template v-if="n === 1">
  <h2>你好</h2>
  <h2>尚硅谷</h2>
  <h2>北京</h2>
</template>
```

渲染到页面上只有三个h2标签（不过只能配合v-if使用）

```
<h2>你好</h2> == $0
<h2>尚硅谷</h2>
<h2>北京</h2>
```

### v-else

v-if 可以单独使用，或配合v-else 指令一起使用：

```
1 <div v-if="Math.random() > 0.5">
2   随机数大于 0.5
3 </div>
4 <div v-else>
5   随机数小于或等于 0.5
6 </div>
```

注意：v-else 指令必须配合v-if 指令一起使用，否则它将不会被识别！

### v-else-if

v-else-if 指令，顾名思义，充当v-if 的“else-if 块”，可以连续使用：

```
1 <div v-if="type === 'A'">优秀</div>
2 <div v-else-if="type === 'B'">良好</div>
3 <div v-else-if="type === 'C'">一般</div>
4 <div v-else>差</div>
```

注意：v-else-if 指令必须配合v-if 指令一起使用，否则它将不会被识别！

这玩意一般都不用。。。

## 列表渲染指令

vue 提供了 v-for 列表渲染指令，用来辅助开发者基于一个数组来循环渲染一个列表结构。v-for 指令需要使用 item in items 形式的特殊语法，其中：

- items 是待循环的数组
- item 是被循环的每一项

```
1 data: {
2   list: [ // 列表数据
3     { id: 1, name: 'zs' },
4     { id: 2, name: 'ls' }
5   ]
6 }
7 // -----分割线-----
8 <ul>
9   <li v-for="item in list">姓名是: {{item.name}}</li>
10 </ul>
```

### v-for

```
1 <!DOCTYPE html>
2 <html lang="en">
3
4
5   <meta charset="UTF-8">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <meta http-equiv="X-UA-Compatible" content="ie=edge">
8   <title>Document</title>
9
10
11
12 <div id="app">
13
14   <ul>
15     <li v-for="n in 10">{{n}}</li>
16   </ul>
17   <ol>
18     <li v-for="(n,index) in 10">{{n}}----{{index}}</li>
19   </ol>
20 </div>
21 <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js">
22 </script>
23 <new Vue({
24   el: '#app',
25   data: {
26     }
27 })
28 </script>
29 </div>
30 >
31 >
```

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>

<body>
  <div id="app">

    <ul>
```

```

        <li v-for="n in 10">{{n}}</li>
    </ul>
<ol>
    <li v-for="(n,index) in 10">{{n}}----{{index}}</li>
</ol>
</div>
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
    new Vue({
        el: '#app',
        data: {
            ...
        }
    })
</script>
</body>

</html>

```

遍历表格 (其中的user可以随便起名字。改成haohao都行)

The screenshot shows a table with three rows:

1 helen	18
2 peter	28
3 andy	38

Next to it is a code editor displaying the following code:

```


|             |                   |              |
|-------------|-------------------|--------------|
| {{user.id}} | {{user.username}} | {{user.age}} |
|-------------|-------------------|--------------|


```

Below the table is a code editor with a script block containing:

```

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
    new Vue({
        el: '#app',
        data: {
            userList: [
                { id: 1, username: 'helen', age: 1 },
                { id: 2, username: 'peter', age: 2 },
                { id: 3, username: 'andy', age: 38 }
            ]
        }
    })
</script>

```

The screenshot shows a list of cities:

- 黑马程序员校区:北京
- 黑马程序员校区:上海
- 黑马程序员校区:广州
- 黑马程序员校区:深圳

Next to it is a code editor displaying the following code:

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
    initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>v-for指令</title>
</head>
<body>    这个可以随机变化，跟下面的对应    这个in是固定的
<div id="app">
    <ul>
        <li v-for="item in arr">
            黑马程序员校区:{{ item }}
        </li>
    </ul>
</div>
<!-- 1. 开发环境版本，包含了有帮助的命令行警告 -->
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
    var app = new Vue({
        el: "#app",
        data: {
            arr:["北京", "上海", "广州", "深圳"]
        }
    })
</script>

```

The screenshot shows a Vue.js application with the following code structure:

```

10 <body>
11   <div id="app">
12     <input type="button" value="添加数据" @click="add" />
13     <input type="button" value="移除数据" @click="remove" />
14
15     <ul>
16       <li v-for="(it, index) in arr">
17         {{ index+1 }}黑马程序员校区:{{ it }}
18       </li>
19     </ul>
20     <h2 v-for="item in vegetables" v-bind:title="item.name">
21       {{ item.name }}
22     </h2>
23   </div>
24   
25   <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
26   <script>
27     var app = new Vue({
28       el: "#app",
29       data: {
30         arr: ["北京", "上海", "广州", "深圳"],
31         vegetables: [
32           {name: "西兰花炒蛋"},
33           {name: "蛋炒西蓝花"}
34         ]
35       },
36       methods: {
37         add: function() {
38           this.vegetables.push({ name: "花菜炒蛋" });
39         },
40         remove: function() {
41           this.vegetables.shift();
42         }
43       }
44     })
45   </script>

```

The application displays a list of cities and a list of vegetable dishes. A dropdown menu on the right lists the cities: Beijing, Shanghai, Guangzhou, and Shenzhen. The vegetable list includes "西兰花炒蛋" and "蛋炒西蓝花".

1. v-for指令的作用是:根据数据生成列表结构
2. 数组经常和v-for结合使用
3. 语法是( item,index ) in 数据
4. item和index可以结合其他指令一起使用
5. 数组长度的更新会同步到页面上，是响应式的

## 利用v-for遍历各种东西

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>基本列表</title>
    <script type="text/javascript" src="../js/vue.js"></script>
  </head>
  <body>
    <!--
      v-for指令：
      1. 用于展示列表数据
      2. 语法: v-for="(item, index) in xxx" :key="yyy"
      3. 可遍历：数组、对象、字符串（用的很少）、指定次数（用的很少）
    -->
    <!-- 准备好一个容器-->
    <div id="root">
      <!-- 遍历数组 -->
      <h2>人员列表（遍历数组）</h2>
      <ul>
        <li v-for="(p, index) of persons" :key="index">
          {{p.name}}-{{p.age}}
        </li>
      </ul>

      <!-- 遍历对象 -->
      <h2>汽车信息（遍历对象）</h2>
      <ul>

```

```

<li v-for="(value,k) of car" :key="k">
    {{k}}-{{value}}
</li>
</ul>

<!-- 遍历字符串 -->
<h2>测试遍历字符串（用得少）</h2>
<ul>
    <li v-for="(char,index) of str" :key="index">
        {{char}}-{{index}}
    </li>
</ul>

<!-- 遍历指定次数 -->
<h2>测试遍历指定次数（用得少）</h2>
<ul>
    <li v-for="(number,index) of 5" :key="index">
        {{index}}-{{number}}
    </li>
</ul>
</div>

<script type="text/javascript">
    Vue.config.productionTip = false

    new Vue({
        el:'#root',
        data:{
            persons:[
                {id:'001',name:'张三',age:18},
                {id:'002',name:'李四',age:19},
                {id:'003',name:'王五',age:20}
            ],
            car:{
                name:'奥迪A8',
                price:'70万',
                color:'黑色'
            },
            str:'hello'
        }
    })
</script>
</html>

```

## v-for 中的索引

v-for 指令还支持一个可选的第二个参数，即当前项的索引。语法格式为(item, index) in items，示例代码如下：

```
1 data: {
2   list: [ // 列表数据
3     { id: 1, name: 'zs' },
4     { id: 2, name: 'ls' }
5   ]
6 }
7 // -----分割线-----
8 <ul>
9   <li v-for="(item, index) in list">索引是: {{index}}, 姓名是: {{item.name}}</li>
10 </ul>
```

注意: v-for 指令中的 item 项和 index 索引都是形参, 可以根据需要进行重命名。例如(user, i) in userlist

### 使用 key 维护列表的状态

当列表的数据变化时, 默认情况下, vue 会尽可能的复用已存在的 DOM 元素, 从而提升渲染的性能。但这种默认的性能优化策略, 会导致有状态的列表无法被正确更新。

为了给vue一个提示, 以便它能跟踪每个节点的身份, 从而在保证有状态的列表被正确更新的前提下, 提升渲染的性能。此时, 需要为每项提供一个唯一的key 属性。

官方建议: 只要用到了v-for指令, 那么一定要绑定一个 :key 属性

```
1 <!-- 用户列表区域 -->
2 <ul>
3   <!-- 加 key 属性的好处: -->
4   <!-- 1. 正确维护列表的状态 -->
5   <!-- 2. 复用现有的 DOM 元素, 提升渲染的性能 -->
6   <li v-for="user in userlist" :key="user.id">
7     <input type="checkbox" />
8     姓名: {{user.name}}
9   </li>
10 </ul>
```

### key 的注意事项

- ① key 的值只能是字符串或数字类型
- ② key 的值必须具有唯一性 (即: key 的值不能重复)
- ③ 建议把数据项 id (我们data里面的数据中的key) 属性的值作为key 的值 (因为 id 属性的值具有唯一性)
- ④ 使用 index 的值当作 key 的值没有任何意义 (因为 index 的值不具有唯一性)
- ⑤ 建议使用v-for 指令时一定要指定key 的值 (既提升性能、又防止列表状态紊乱)

## key的原理

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>key的原理</title>
    <script type="text/javascript" src="../js/vue.js"></script>
  </head>
  <body>
    <!--
      面试题：react、vue中的key有什么作用？（key的内部原理）
    -->
```

### 1. 虚拟DOM中key的作用：

key是虚拟DOM对象的标识，当数据发生变化时，Vue会根据【新数据】生成【新的虚拟DOM】，随后Vue进行【新虚拟DOM】与【旧虚拟DOM】的差异比较，比较规则如下：

### 2. 对比规则：

#### (1). 旧虚拟DOM中找到了与新虚拟DOM相同的key：

①. 若虚拟DOM中内容没变，直接使用之前的真实DOM！  
②. 若虚拟DOM中内容变了，则生成新的真实DOM，随后替换掉页面中之前的真实DOM。

#### (2). 旧虚拟DOM中未找到与新虚拟DOM相同的key

创建新的真实DOM，随后渲染到到页面。

### 3. 用index作为key可能会引发的问题：

1. 若对数据进行：逆序添加、逆序删除等破坏顺序操作：

会产生没有必要的真实DOM更新 ==> 界面效果没问题，但效率低。

#### 2. 如果结构中还包含输入类的DOM：

会产生错误DOM更新 ==> 界面有问题。

### 4. 开发中如何选择key?：

1. 最好使用每条数据的唯一标识作为key，比如id、手机号、身份证号、学号等唯一值。  
2. 如果不存在对数据的逆序添加、逆序删除等破坏顺序操作，仅用于渲染列表用于展示，

使用index作为key是没有问题的。

```
-->
<!-- 准备好一个容器-->
<div id="root">
  <!-- 遍历数组 -->
  <h2>人员列表（遍历数组）</h2>
  <button @click.once="add">添加一个老刘</button>
  <ul>
    <li v-for="(p, index) of persons" :key="index">
      {{p.name}}-{{p.age}}
      <input type="text">
```

```

        </li>
    </ul>
</div>

<script type="text/javascript">
Vue.config.productionTip = false

new Vue({
    el:'#root',
    data:{
        persons:[
            {id:'001',name:'张三',age:18},
            {id:'002',name:'李四',age:19},
            {id:'003',name:'王五',age:20}
        ]
    },
    methods: {
        add(){
            const p = {id:'004',name:'老刘',age:40}
            this.persons.unshift(p)
        }
    }
})
</script>
</html>

```

## 其他指令

### v-cloak

这个指令是防止js文件没有加载完，然后浏览器页面出现一些用户不想看到的页面

```

<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>v-cloak指令</title>
        <style>
            /*
                让有v-cloak指令的标签全不显示
                等js全部加载完，vue对象创建后就会自动删除标签里面的v-cloak指令
            */
            [v-cloak]{
                display:none;
            }
        </style>
        <!-- 引入Vue -->
    </head>
    <body>
        <!--
            v-cloak指令（没有值）：
            1. 本质是一个特殊属性，Vue实例创建完毕并接管容器后，会删掉v-cloak属性。
            2. 使用css配合v-cloak可以解决网速慢时页面展示出{{xxx}}的问题。
        -->
    </body>
</html>

```

```

<!-- 准备好一个容器-->
<div id="root">
    <h2 v-cloak>{{name}}</h2>
</div>
<script type="text/javascript"
src="http://localhost:8080/resource/5s/vue.js"></script>
</body>

<script type="text/javascript">
    console.log(1)
    Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。

    new Vue({
        el: '#root',
        data: {
            name: '尚硅谷'
        }
    })
</script>
</html>

```

### v-once

当我们有需求，只渲染初始时候的数据，后面数据不管怎么变，前端页面都不会变的需求可以用这个指令

```

<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>v-once指令</title>
        <!-- 引入Vue -->
        <script type="text/javascript" src="../js/vue.js"></script>
    </head>
    <body>
        <!--
            v-once指令:
            1.v-once所在节点在初次动态渲染后，就视为静态内容了。
            2.以后数据的改变不会引起v-once所在结构的更新，可以用于优化性能。
        -->
        <!-- 准备好一个容器-->
        <div id="root">
            <h2 v-once>初始化的n值是:{{n}}</h2>
            <h2>当前的n值是:{{n}}</h2>
            <button @click="n++">点我n+1</button>
        </div>
    </body>

    <script type="text/javascript">
        Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。

        new Vue({
            el: '#root',
            data: {
                n:1
            }
        })
    </script>

```

```
        }
    })
</script>
</html>
```

## v-pre

这个玩意可以提升vue编译的性能（因为跳过这个节点vue对他不解析了）。但是最好不要用

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>v-pre指令</title>
    <!-- 引入Vue -->
    <script type="text/javascript" src="../../js/vue.js"></script>
  </head>
  <body>
    <!--
      v-pre指令:
      1. 跳过其所在节点的编译过程。
      2. 可利用它跳过：没有使用指令语法、没有使用插值语法的节点，会加快编译。
    -->
    <!-- 准备好一个容器-->
    <div id="root">
      <h2 v-pre>Vue其实很简单</h2>
      <h2>当前的n值是:{{n}}</h2>
      <button @click="n++">点我n+1</button>
    </div>
  </body>

  <script type="text/javascript">
    vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。

    new Vue({
      el:'#root',
      data:{
        n:1
      }
    })
  </script>
</html>
```

## 自定义指令

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>自定义指令</title>
    <script type="text/javascript" src="../../js/vue.js"></script>
  </head>
  <body>
    <!--
      需求1： 定义一个v-big指令，和v-text功能类似，但会把绑定的数值放大10倍。
    -->
```

需求2：定义一个v-fbind指令，和v-bind功能类似，但可以让其所绑定的input元素默认获取焦点。

自定义指令总结：

一、定义语法：

(1). 局部指令：

```
new Vue({  
    directives:{指令名:配置对象}  
})  
或  
new Vue({  
    directives{指令名:回调函数}  
})  
或  
Vue.directive(指令名,回调函数)
```

(2). 全局指令：

```
Vue.directive(指令名,配置对象)
```

二、配置对象中常用的3个回调：

(1).bind: 指令与元素成功绑定时调用。

(2).inserted: 指令所在元素被插入页面时调用。

(3).update: 指令所在模板结构被重新解析时调用。

三、备注：

1. 指令定义时不加v-，但使用时要加v-；

2. 指令名如果是多个单词，要使用kebab-case命名方式，

不要用camelCase命名。

```
-->  
<!-- 准备好一个容器-->  
<div id="root">  
    <h2>{{name}}</h2>  
    <h2>当前的n值是: <span v-text="n"></span> </h2>  
    <!-- <h2>放大10倍后的n值是: <span v-big-number="n"></span> </h2> -->  
    <h2>放大10倍后的n值是: <span v-big="n"></span> </h2>  
    <button @click="n++">点我n+1</button>  
    <hr/>  
    <input type="text" v-fbind:value="n">  
</div>  
</body>  
  
<script type="text/javascript">  
    Vue.config.productionTip = false  
  
    //定义全局指令  
    /* Vue.directive('fbind',{
        //指令与元素成功绑定时（一上来）  
        bind(element,binding){  
            element.value = binding.value  
        },  
        //指令所在元素被插入页面时  
        inserted(element,binding){  
            element.focus()  
        },  
        //指令所在的模板被重新解析时  
        update(element,binding){  
            element.value = binding.value  
        }  
    }) */
```

```

new Vue({
    el:'#root',
    data:{
        name:'尚硅谷',
        n:1
    },
    directives:{
        //big函数何时会被调用? 1.指令与元素成功绑定时(一上来)。2.指令所在的模板被
        //重新解析时。
        /* 'big-number'(element,binding){
            // console.log('big')
            element.innerText = binding.value * 10
        }, */
        big(element,binding){
            console.log('big',this) //注意此处的this是window,这里需要注意别取
            //错了
            // console.log('big')
            element.innerText = binding.value * 10
        },
        fbind:{
            //指令与元素成功绑定时(一上来)
            bind(element,binding){
                element.value = binding.value
            },
            //指令所在元素被插入页面时
            inserted(element,binding){
                element.focus()
            },
            //指令所在的模板被重新解析时
            update(element,binding){
                element.value = binding.value
            }
        }
    }
})

</script>
</html>

```

## vue中的数据检测

```

<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>总结数据监视</title>
        <style>
            button{
                margin-top: 10px;
            }
        </style>
        <!-- 引入Vue -->

```

```

<script type="text/javascript" src="../js/vue.js"></script>
</head>
<body>
    <!--
        Vue监视数据的原理:
        1. vue会监视data中所有层次的数据。
        2. 如何监测对象中的数据?
            通过setter实现监视, 且要在new Vue时就传入要监测的数据。
            (1).对象中后追加的属性, Vue默认不做响应式处理
            (2).如需给后添加的属性做响应式, 请使用如下API:
                Vue.set(target,
propertyName/index, value) 或
                    vm.$set(target,
propertyName/index, value)
    3. 如何监测数组中的数据?
        通过包裹数组更新元素的方法实现, 本质就是做了两件事:
            (1).调用原生对应的方法对数组进行更新。
            (2).重新解析模板, 进而更新页面。
    4. 在vue修改数组中的某个元素一定要用如下方法:
        1. 使用这些API:push()、pop()、shift()、unshift()、
splice()、sort()、reverse()
        2.Vue.set() 或 vm.$set()
    特别注意: Vue.set() 和 vm.$set() 不能给vm 或 vm的根数据对象 添加属
性! ! !
-->
<!-- 准备好一个容器-->
<div id="root">
    <h1>学生信息</h1>
    <button @click="student.age++">年龄+1岁</button> <br/>
    <button @click="addSex">添加性别属性, 默认值: 男</button> <br/>
    <button @click="student.sex = '未知'">修改性别</button> <br/>
    <button @click="addFriend">在列表首位添加一个朋友</button> <br/>
    <button @click="updateFirstFriendName">修改第一个朋友的名字为: 张三
</button> <br/>
    <button @click="addHobby">添加一个爱好</button> <br/>
    <button @click="updateHobby">修改第一个爱好为: 开车</button> <br/>
    <button @click="removeSmoke">过滤掉爱好中的抽烟</button> <br/>
    <h3>姓名: {{student.name}}</h3>
    <h3>年龄: {{student.age}}</h3>
    <h3 v-if="student.sex">性别: {{student.sex}}</h3>
    <h3>爱好: </h3>
    <ul>
        <li v-for="(h,index) in student.hobby" :key="index">
            {{h}}
        </li>
    </ul>
    <h3>朋友们: </h3>
    <ul>
        <li v-for="(f,index) in student.friends" :key="index">
            {{f.name}}--{{f.age}}
        </li>
    </ul>
</div>

```

```

-->
<!-- 准备好一个容器-->
<div id="root">
    <h1>学生信息</h1>
    <button @click="student.age++">年龄+1岁</button> <br/>
    <button @click="addSex">添加性别属性, 默认值: 男</button> <br/>
    <button @click="student.sex = '未知'">修改性别</button> <br/>
    <button @click="addFriend">在列表首位添加一个朋友</button> <br/>
    <button @click="updateFirstFriendName">修改第一个朋友的名字为: 张三
</button> <br/>
    <button @click="addHobby">添加一个爱好</button> <br/>
    <button @click="updateHobby">修改第一个爱好为: 开车</button> <br/>
    <button @click="removeSmoke">过滤掉爱好中的抽烟</button> <br/>
    <h3>姓名: {{student.name}}</h3>
    <h3>年龄: {{student.age}}</h3>
    <h3 v-if="student.sex">性别: {{student.sex}}</h3>
    <h3>爱好: </h3>
    <ul>
        <li v-for="(h,index) in student.hobby" :key="index">
            {{h}}
        </li>
    </ul>
    <h3>朋友们: </h3>
    <ul>
        <li v-for="(f,index) in student.friends" :key="index">
            {{f.name}}--{{f.age}}
        </li>
    </ul>
</div>

```

```
</u]>
</div>
</body>

<script type="text/javascript">
Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。

const vm = new Vue({
  el:'#root',
  data:{
    student:{
      name:'tom',
      age:18,
      hobby:['抽烟','喝酒','烫头'],
      friends:[
        {name:'jerry',age:35},
        {name:'tony',age:36}
      ]
    }
  },
  methods: {
    addSex(){
      // vue.set(this.student,'sex','男')
      this.$set(this.student,'sex','男')
    },
    addFriend(){
      this.student.friends.unshift({name:'jack',age:70})
    },
    updateFirstFriendName(){
      this.student.friends[0].name = '张三'
    },
    addHobby(){
      this.student.hobby.push('学习')
    },
    updateHobby(){
      // this.student.hobby.splice(0,1,'开车')
      // vue.set(this.student.hobby,0,'开车')
      this.$set(this.student.hobby,0,'开车')
    },
    removeSmoke(){
      this.student.hobby = this.student.hobby.filter((h)=>{
        return h !== '抽烟'
      })
    }
  }
)
</script>
</html>
```

# vue动态绑定样式

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>绑定样式</title>
    <style>
      .basic{
        width: 400px;
        height: 100px;
        border: 1px solid black;
      }

      .happy{
        border: 4px solid red;;
        background-color: rgba(255, 255, 0, 0.644);
        background: linear-gradient(30deg,yellow,pink,orange,yellow);
      }

      .sad{
        border: 4px dashed rgb(2, 197, 2);
        background-color: gray;
      }

      .normal{
        background-color: skyblue;
      }

      .atguigui{
        background-color: yellowgreen;
      }

      .atguigu2{
        font-size: 30px;
        text-shadow:2px 2px 10px red;
      }

      .atguigu3{
        border-radius: 20px;
      }
    </style>
    <script type="text/javascript" src="../js/vue.js"></script>
  </head>
  <body>
    <!--
        绑定样式:
        1. class样式
          写法: class="xxx" xxx可以是字符串、对象、数组。
          字符串写法适用于: 类名不确定, 要动态获取。
          对象写法适用于: 要绑定多个样式, 个数不确定, 名字
        也不确定。
        数组写法适用于: 要绑定多个样式, 个数确定, 名字也
        确定, 但不确定用不用。
        2. style样式
          :style="{fontsize: xxx}"其中xxx是动态值。
          :style="[a,b]"其中a、b是样式对象。
    -->
    <!-- 准备好一个容器-->
```

```
<div id="root">
    <!-- 绑定class样式--字符串写法，适用于：样式的类名不确定，需要动态指定 -->
    <div class="basic" :class="mood" @click="changeMood">{{name}}</div>
<br/><br/>

    <!-- 绑定class样式--数组写法，适用于：要绑定的样式个数不确定、名字也不确定 -->
    <div class="basic" :class="classArr">{{name}}</div> <br/><br/>

    <!-- 绑定class样式--对象写法，适用于：要绑定的样式个数确定、名字也确定，但要动态决定用不用 -->
    <div class="basic" :class="classObj">{{name}}</div> <br/><br/>

    <!-- 绑定style样式--对象写法 -->
    <div class="basic" :style="styleObj">{{name}}</div> <br/><br/>
    <!-- 绑定style样式--数组写法 -->
    <div class="basic" :style="styleArr">{{name}}</div>
</div>
</body>

<script type="text/javascript">
    Vue.config.productionTip = false

    const vm = new Vue({
        el:'#root',
        data:{
            name:'尚硅谷',
            mood:'normal',
            classArr:['atguigu1','atguigu2','atguigu3'],
            classObj:{
                atguigu1:false,
                atguigu2:false,
            },
            styleObj:{
                fontSize: '40px',
                color: 'red',
            },
            styleObj2:{
                backgroundColor: 'orange'
            },
            styleArr:[
                {
                    fontSize: '40px',
                    color: 'blue',
                },
                {
                    backgroundColor: 'gray'
                }
            ]
        },
        methods: {
            changeMood(){
                const arr = ['happy','sad','normal']
                const index = Math.floor(Math.random()*3)
                this.mood = arr[index]
            }
        }
    })

```

```
        },
    })
</script>

</html>
```

## 过滤器

### 过滤器的使用

过滤器 (Filters) 是 vue 为开发者提供的功能，常用于文本的格式化。过滤器可以用在两个地方：插值表达式和 v-bind 属性绑定。

过滤器应该被添加在 JavaScript 表达式的尾部，由“管道符”进行调用，示例代码如下：

下面的capitalize和formatId都是需要我们自定义的函数，至于格式，往下看

```
1 <!-- 在双花括号中通过“管道符”调用 capitalize 过滤器，对 message 的值进行格式化 -->
2 <p>{{ message | capitalize }}</p>
3
4 <!-- 在 v-bind 中通过“管道符”调用 formatId 过滤器，对 rawId 的值进行格式化 -->
5 <div v-bind:id="rawId | formatId"></div>
```

示例代码：

```
<!DOCTYPE html>
<html lang="en">

    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Document</title>
    </head>

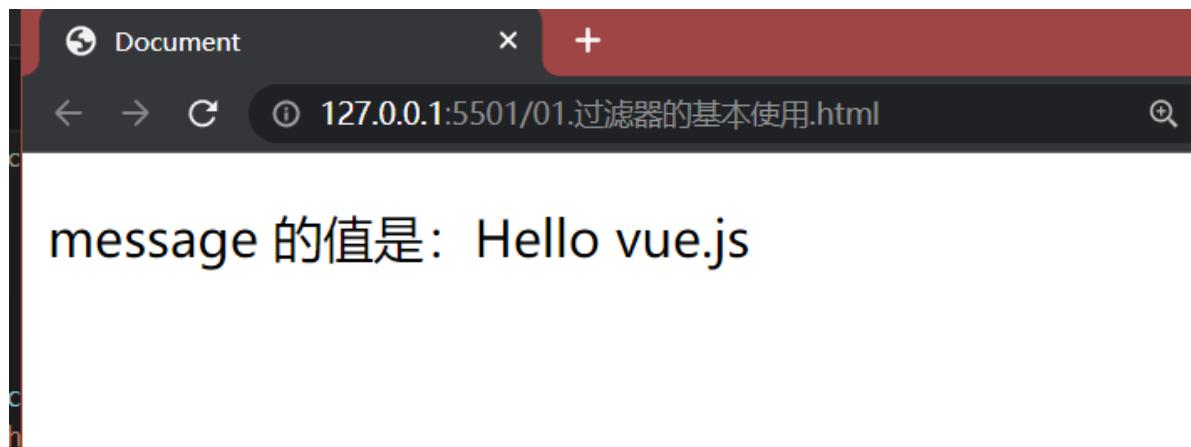
    <body>
        <div id="app">
            <p>message 的值是: {{ message | capi }}</p>
        </div>
        <script src="./lib/vue-2.6.12.js"></script>
        <script>
            const vm = new Vue({
                el: '#app',
                data: {
                    message: 'hello vue.js'
                },
                // 过滤器函数，必须被定义到 filters 节点之下
                // 过滤器本质上是函数
                filters: {
                    // 注意：过滤器函数形参中的 val，永远都是“管道符”前面的那个值
                }
            })
        </script>
    </body>
</html>
```

```

    //这个函数的作用是把首字母变为大写
    capi(val) {
        // 字符串有 charAt 方法，这个方法接收索引值，表示从字符串中把索引
        对应的字符，获取出来
        // val.charAt(0)
        const first = val.charAt(0).toUpperCase()
        // 字符串的 slice 方法，可以截取字符串，从指定索引往后截取
        const other = val.slice(1)
        // 强调：过滤器中，一定要有一个返回值
        return first + other
    }
}
</script>
</body>

</html>

```



## 私有过滤器和全局过滤器

在 filters 节点下定义的过滤器，也就是上面我们自己定义的过滤器，称为“私有过滤器”，因为它只能在当前vm 实例所控制的el 区域内使用。如果希望在多个vue 实例之间共享过滤器，则可以按照如下的格式定义全局过滤器：

```

1 // 全局过滤器 - 独立于每个 vm 实例之外
2 // Vue.filter() 方法接收两个参数：
3 // 第 1 个参数，是全局过滤器的“名字”
4 // 第 2 个参数，是全局过滤器的“处理函数”
5 Vue.filter('capitalize', (str) => {
6     return str.charAt(0).toUpperCase() + str.slice(1) + '~~'
7 })

```

示例代码：

```

<!DOCTYPE html>
<html lang="en">

```

```
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>

<body>
  <div id="app">
    <p>message 的值是: {{ message | capi }}</p>
  </div>

  <div id="app2">
    <p>message 的值是: {{ message | capi }}</p>
  </div>

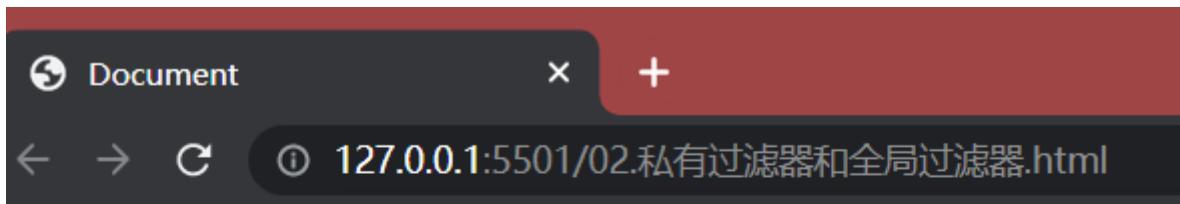
<script src="./lib/vue-2.6.12.js"></script>
<script>
  // 使用 vue.filter() 定义全局过滤器
  // 如果全局过滤器和私有过滤器名字一致，此时就按照就近原则，调用的是私有过滤器
  vue.filter('capi', function (str) {
    const first = str.charAt(0).toUpperCase()
    const other = str.slice(1)
    return first + other + '~~~'
  })

  const vm = new Vue({
    el: '#app',
    data: {
      message: 'hello vue.js'
    },
    filters: {
      capi(val) {
        const first = val.charAt(0).toUpperCase()
        const other = val.slice(1)
        return first + other
      }
    }
  })

  // -----
  // 第二个vue示例，这里没有配置过滤器
  const vm2 = new Vue({
    el: '#app2',
    data: {
      message: 'Hao Hao'
    }
  })
</script>
</body>

</html>
```

运行效果：第一个调用了私有过滤器，第二个调用了全局过滤器



message 的值是: Hello vue.js

message 的值是: Haohao~~~

## 连续调用多个过滤器

过滤器可以串联地进行调用，例如：

```
1 <!-- 把 message 的值，交给 filterA 进行处理 -->
2 <!-- 把 filterA 处理的结果，再交给 filterB 进行处理 -->
3 <!-- 最终把 filterB 处理的结果，作为最终的值渲染到页面上 -->
4 {{ message | filterA | filterB }}
```

## 过滤器传参

过滤器的本质是 JavaScript 函数，因此可以接收参数，格式如下：

```
1 <!-- arg1 和 arg2 是传递给 filterA 的参数 -->
2 <p>{{ message | filterA(arg1, arg2) }}</p>
3
4 // 过滤器处理函数的形参列表中：
5 // 第一个参数：永远都是“管道符”前面待处理的值
6 // 从第二个参数开始，才是调用过滤器时传递过来的 arg1 和 arg2 参数
7 Vue.filter('filterA', (msg, arg1, arg2) => {
8   // 过滤器的代码逻辑...
9 })
```

## 过滤器的兼容性

过滤器仅在 vue 2.x 和 1.x 中受支持，在 vue 3.x 的版本中剔除了过滤器相关的功能。在企业级项目开发中：

如果使用的是 2.x 版本的 vue，则依然可以使用过滤器相关的功能

如果项目已经升级到了 3.x 版本的 vue，官方建议使用计算属性或方法代替被剔除的过滤器功能

具体的迁移指南，请参考vue 3.x 的官方文档给出的说明：<https://v3.vuejs.org/guide/migration/filter.html#migration-strategy>

## watch 倾听器

### 什么是 watch 倾听器

倾听器watch 倾听器允许开发者监视数据的变化，从而针对数据的变化做特定的操作。语法格式如下：

```
1 const vm = new Vue({
2   el: '#app',
3   data: { username: '' },
4   watch: {
5     // 监听 username 值的变化
6     // newVal 是“变化后的值”，oldVal 是“变化之前的旧值”
7     username(newVal, oldVal) {
8       console.log(newVal, oldVal)
9     }
10   }
11 })
```

举例：

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>

<body>
  <div id="app">
    <input type="text" v-model="username">
  </div>
  <script src="./lib/vue-2.6.12.js"></script>
  <script>
    const vm = new Vue({
      el: '#app',
      data: {
        username: 'admin'
      },
      // 所有的倾听器，都应该被定义到 watch 节点下
      watch: {
        // 倾听器本质上是一个函数，要监视哪个数据的变化，就把数据名作为方法名即可
        // 新值在前，旧值在后
      }
    })
  </script>
</body>
</html>
```

```

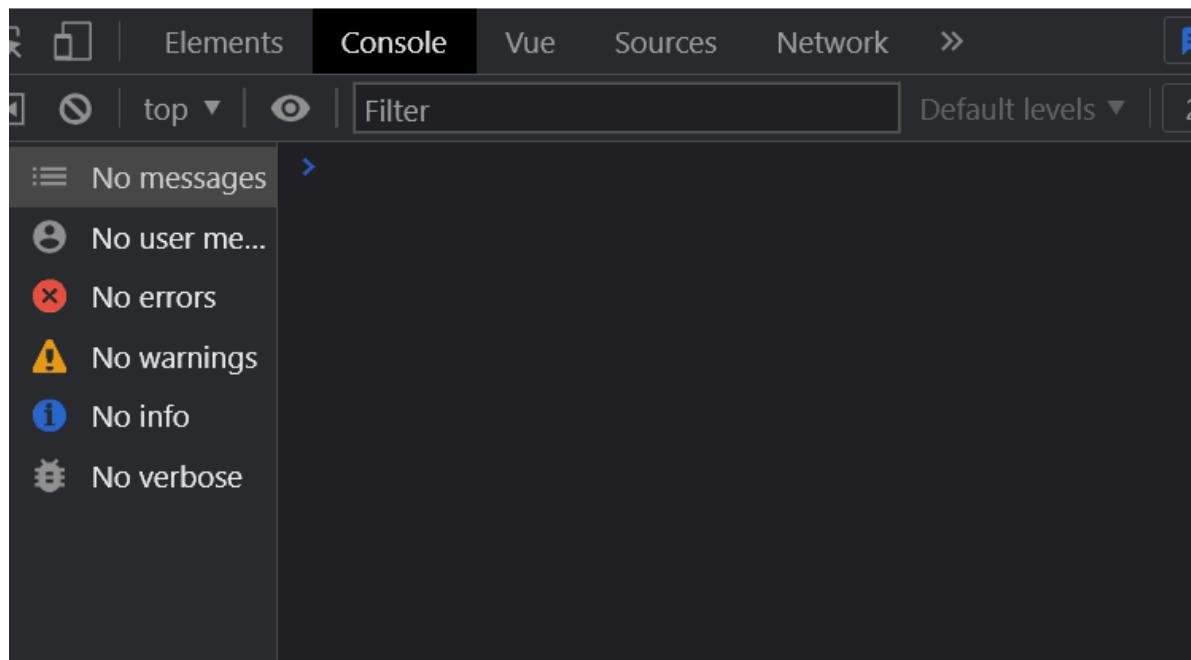
username(newVal,oldVal) {
  console.log("检测到了数值的变化");
  console.log(`output->newVal`,newVal)
  console.log(`output->oldVal`,oldVal)
}
}

</script>
</body>

</html>

```

实现效果：



## 监听器的应用场景

可以用来检测用户名是否被占用，每一次修改都发送一次axios请求去看用户名是否被占用。

### immediate 选项

默认情况下，组件在初次加载完毕后不会调用 watch 侦听器。如果想让 watch 侦听器立即被调用，则需要使用 immediate 选项。示例代码如下：

```

<!DOCTYPE html>
<html lang="en">

<head>

```

```

<meta charset="UTF-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Document</title>
</head>

<body>
<div id="app">
  <input type="text" v-model="username">
</div>

<script src="./lib/vue-2.6.12.js"></script>
<script src="./lib/jquery-v3.6.0.js"></script>

<script>
  const vm = new Vue({
    el: '#app',
    data: {
      username: 'admin'
    },
    // 所有的侦听器，都应该被定义到 watch 节点下
    watch: {
      // 定义对象格式的侦听器
      username: {
        // 侦听器的处理函数
        handler(newVal, oldVal) {
          console.log(newVal, oldVal)
        },
        // immediate 选项的默认值是 false
        // immediate 的作用是：控制侦听器是否自动触发一次！
        // 如果为真，则说明是进入浏览器默认触发一次handler
        immediate: true
      }
    }
  })
</script>
</body>

</html>

```

## deep 选项

如果 watch 侦听的是一个对象，如果对象中的属性值发生了变化，则无法被监听到。此时需要使用 deep 选项，代码示例如下：

```

<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>

```

```

<body>
  <div id="app">
    <input type="text" v-model="info.username">
    <input type="text" v-model="info.address.city">
  </div>

  <script src="./lib/vue-2.6.12.js"></script>
  <script src="./lib/jquery-v3.6.0.js"></script>

  <script>
    const vm = new Vue({
      el: '#app',
      data: {
        // 用户的信息对象
        info: {
          username: 'admin',
          address: {
            city: '北京'
          }
        },
        // 所有的侦听器，都应该被定义到 watch 节点下
        watch: {
          info: {
            handler(newVal) {
              console.log(newVal)
            },
            // 开启深度监听，只要对象中任何一个属性变化了，都会触发“对象的侦听器”
            deep: true
          },
        }
      }
    })
  </script>
</body>
</html>

```

## 监听对象单个属性的变化

如果只想监听对象中单个属性的变化，则可以按照如下的方式定义 watch 侦听器：

```

<!DOCTYPE html>
<html lang="en">

  <head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
  </head>

  <body>
    <div id="app">
      <input type="text" v-model="info.username">
      <input type="text" v-model="info.address.city">
    </div>
  </body>

```

```

<script src="./lib/vue-2.6.12.js"></script>
<script src="./lib/jquery-v3.6.0.js"></script>

<script>
  const vm = new Vue({
    el: '#app',
    data: {
      // 用户的信息对象
      info: {
        username: 'admin',
        address: {
          city: '北京'
        }
      },
      watch: {
        // 如果要侦听的是对象的子属性（只监听这一个的）的变化，则必须包裹一层单引号
        // 这里只监听了info对象的username属性
        'info.username'(newVal) {
          console.log(newVal)
        }
      }
    }
  })
</script>
</body>

</html>

```

## 对象监听器和方法监听器的区别

### 1. 方法格式的侦听器

- 缺点1：无法在刚进入页面的时候，自动触发！！！
- 缺点2：如果侦听的是一个对象，如果对象中的属性发生了变化，不会触发侦听器！！！
- 好处1：定义简单，使用简单。。。

### 2. 对象格式的侦听器

- 好处1：可以通过 **immediate** 选项，让侦听器自动触发！！！
- 好处2：可以通过 **deep** 选项，让侦听器深度监听对象中每个属性的变化！！！

## 计算属性

### 什么是计算属性

计算属性指的是通过一系列运算之后，最终得到一个属性值。

这个动态计算出来的属性值可以被模板结构或methods方法使用。

计算属性也是属性，计算属性的返回值也会成为data的一部分，不过和data又有一些不一样，下面的computed方法里面的就是计算属性，计算属性会以：方法名为key,返回值为value挂载到Vue对象(vm)身上。访问这个属性也是通过this.方法名，就可以访问到这个属性了，这个就是计算属性。

下面是三种语法实现计算姓名拼接的功能，注意区别：！！

## 插值语法实现

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>姓名案例_插值语法实现</title>
    <!-- 引入Vue -->
    <script type="text/javascript" src="../js/vue.js"></script>
  </head>
  <body>
    <!-- 准备好一个容器-->
    <div id="root">
      姓: <input type="text" v-model="firstName"> <br/><br/>
      名: <input type="text" v-model="lastName"> <br/><br/>
      全名: <span>{{firstName}}-{{lastName}}</span>
    </div>
  </body>

  <script type="text/javascript">
    Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。

    new Vue({
      el: '#root',
      data: {
        firstName: '张',
        lastName: '三'
      }
    })
  </script>
</html>
```

## methods实现

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>姓名案例_methods实现</title>
    <!-- 引入Vue -->
    <script type="text/javascript" src="../js/vue.js"></script>
  </head>
  <body>
    <!-- 准备好一个容器-->
    <div id="root">
      姓: <input type="text" v-model="firstName"> <br/><br/>
      名: <input type="text" v-model="lastName"> <br/><br/>
      全名: <span>{{fullName()}}</span>
    </div>
  </body>

  <script type="text/javascript">
    Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。

    new Vue({
```

```

    el:'#root',
    data:{
        firstName:'张',
        lastName:'三'
    },
    methods: {
        fullName(){
            console.log('@---fullName')
            return this.firstName + ' - ' + this.lastName
        }
    },
})
</script>
</html>

```

## 计算属性实现

```

<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>姓名案例_计算属性实现</title>
        <!-- 引入Vue -->
        <script type="text/javascript" src="../js/vue.js"></script>
    </head>
    <body>
        <!--
            计算属性:
            1. 定义: 要用的属性不存在, 要通过已有属性计算得来。
            2. 原理: 底层借助了Object.defineProperty方法提供的getter和setter。
            3.get函数什么时候执行?
                (1).初次读取时会执行一次。
                (2).当依赖的数据发生改变时会被再次调用。
            4.优势: 与methods实现相比, 内部有缓存机制(复用), 效率更高, 调试方便。
            5.备注:
                1.计算属性最终会出现在vm上, 直接读取使用即可。
                2.如果计算属性要被修改, 那必须写set函数去响应修改, 且set中要引起计算时依赖的数据发生改变。
        -->
        <!-- 准备好一个容器-->
        <div id="root">
            姓: <input type="text" v-model="firstName"> <br/><br/>
            名: <input type="text" v-model="lastName"> <br/><br/>
            测试: <input type="text" v-model="x"> <br/><br/>
            全名: <span>{{fullName}}</span> <br/><br/>
            <!-- 全名: <span>{{fullName}}</span> <br/><br/>
            全名: <span>{{fullName}}</span> <br/><br/>
            全名: <span>{{fullName}}</span> -->
        </div>
    </body>

    <script type="text/javascript">
        Vue.config.productionTip = false //阻止vue在启动时生成生产提示。
        const vm = new Vue({

```

```

    el:'#root',
    data:{
        firstName:'张',
        lastName:'三',
        x:'你好'
    },
    methods: {
        demo(){
            ...
        }
    },
    computed:{
        fullName:{
            //get有什么作用? 当有人读取fullName时, get就会被调用, 且返回值就作为
            //fullName的值
            get(){
                console.log('get被调用了')
                // console.log(this) //此处的this是vm
                return this.firstName + '-' + this.lastName
            },
            //set什么时候调用? 当fullName被修改时。
            set(value){
                console.log('set',value)
                const arr = value.split('-')
                this.firstName = arr[0]
                this.lastName = arr[1]
            }
        }
    }
)
</script>
</html>

```

## 监听器实现

```

<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>姓名案例_watch实现</title>
        <!-- 引入Vue -->
        <script type="text/javascript" src="../js/vue.js"></script>
    </head>
    <body>
        <!--
            computed和watch之间的区别:
            1.computed能完成的功能, watch都可以完成。
            2.watch能完成的功能, computed不一定能完成, 例如: watch可以进行异步操作。
        -->
        两个重要的小原则:
        1.所被Vue管理的函数, 最好写成普通函数, 这样this的指向才是vm 或 组件实例对象。
    </body>

```

2. 所有不被Vue所管理的函数（定时器的回调函数、ajax的回调函数等、Promise的回调函数），最好写成箭头函数，

这样this的指向才是vm 或 组件实例对象。

```
-->
<!-- 准备好一个容器-->
<div id="root">
    姓: <input type="text" v-model="firstName"> <br/><br/>
    名: <input type="text" v-model="lastName"> <br/><br/>
    全名: <span>{{fullName}}</span> <br/><br/>
</div>
</body>

<script type="text/javascript">
Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。

const vm = new Vue({
    el:'#root',
    data:{
        firstName:'张',
        lastName:'三',
        fullName:'张-三'
    },
    watch:{
        firstName(val){
            setTimeout(()=>{
                console.log(this)
                this.fullName = val + '-' + this.lastName
            },1000);
        },
        lastName(val){
            this.fullName = this.firstName + '-' + val
        }
    }
})
</script>
</html>
```

## 计算属性简写

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="UTF-8" />
    <title>姓名案例_计算属性实现</title>
    <!-- 引入Vue -->
    <script type="text/javascript" src="../../js/vue.js"></script>
</head>

<body>
```

```

<!-- 准备好一个容器-->
<div id="root">
    姓: <input type="text" v-model="firstName"> <br /><br />
    名: <input type="text" v-model="lastName"> <br /><br />
    全名: <span>{{fullName}}</span> <br /><br />
</div>
</body>

<script type="text/javascript">
    vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。

    const vm = new Vue({
        el: '#root',
        data: {
            firstName: '张',
            lastName: '三',
        },
        computed: {
            /* fullName:{
                get(){
                    console.log('get被调用了')
                    return this.firstName + '-' + this.lastName
                },
                set(value){
                    console.log('set',value)
                    const arr = value.split('-')
                    this.firstName = arr[0]
                    this.lastName = arr[1]
                }
            }*/
            //这个函数就当那个getter使用
            //只有考虑读取，不考虑修改的情况下才能使用简写形式
            fullName() {
                console.log('get被调用了')
                return this.firstName + '-' + this.lastName
            }
        }
    })
</script>

</html>

```

## 计算属性的特点

- ① 虽然计算属性在声明的时候被定义为方法，但是计算属性的本质是一个属性
- ② 计算属性会缓存计算的结果，只有计算属性依赖的数据变化时，才会重新进行运算

## 利用计算属性编写小案例

示例代码如下：

```
1 var vm = new Vue({
2   el: '#app',
3   data: {
4     r: 0, g: 0, b: 0
5   },
6   computed: {
7     rgb() { return `rgb(${this.r}, ${this.g}, ${this.b})` }
8   },
9   methods: {
10   show() { console.log(this.rgb) }
11 },
12 })
```

案例代码：

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
  <script src="./lib/vue-2.6.12.js"></script>
  <style>
    .box {
      width: 200px;
      height: 200px;
      border: 1px solid #ccc;
    }
  </style>
</head>

<body>
  <div id="app">
    <div>
      <span>R: </span>
      <input type="text" v-model.number="r">
    </div>
    <div>
      <span>G: </span>
      <input type="text" v-model.number="g">
    </div>
    <div>
      <span>B: </span>
      <input type="text" v-model.number="b">
    </div>
  </div>
</body>
```

```
<hr>

<!-- 专门用户呈现颜色的 div 盒子 -->
<!-- 在属性身上，: 代表 v-bind: 属性绑定 -->
<!-- 这里的style里面写的是一个js对象，里面是backgroundColor为key，后面的为值 -->
<div class="box" :style="{ backgroundColor: `rgb(${r}, ${g}, ${b})` }">
    <!-- 这里面写的是js表达式，也就是说可以用js的语法 -->
    {{ `rgb(${r}, ${g}, ${b})` }}
</div>
<button @click="show">按钮</button>
</div>

<script>
// 创建 Vue 实例，得到 viewModel
var vm = new Vue({
    el: '#app',
    data: {
        // 红色
        r: 0,
        // 绿色
        g: 0,
        // 蓝色
        b: 0
    },
    methods: {
        // 点击按钮，在终端显示最新的颜色
        show() {
            console.log(`rgb(${this.r}, ${this.g}, ${this.b})`)
        }
    }
);
</script>
</body>

</html>
```

实现效果：

R:	0
G:	0
B:	0



按钮

需求：使用计算属性改造案例，不改变原有的效果。

改造后的代码：

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
  <script src="./lib/vue-2.6.12.js"></script>
  <style>
    .box {
      width: 200px;
      height: 200px;
      border: 1px solid #ccc;
    }
  </style>
</head>

<body>
  <div id="app">
    <div>
      <span>R: </span>
      <input type="text" v-model.number="r">
    </div>
  </div>
</body>
```

```
</div>
<div>
  <span>G: </span>
  <input type="text" v-model.number="g">
</div>
<div>
  <span>B: </span>
  <input type="text" v-model.number="b">
</div>
<hr>

<!-- 专门用户呈现颜色的 div 盒子 --&gt;
<!-- 在属性身上，: 代表 v-bind: 属性绑定 --&gt;
<!-- :style 代表动态绑定一个样式对象，它的值是一个 { } 样式对象 --&gt;
<!-- 当前的样式对象中，只包含 backgroundColor 背景颜色 --&gt;
&lt;div class="box" :style="{ backgroundColor: rgb }"&gt;
  {{ rgb }}
&lt;/div&gt;
&lt;button @click="show"&gt;按钮&lt;/button&gt;
&lt;/div&gt;

&lt;script&gt;
// 创建 Vue 实例，得到 ViewModel
var vm = new Vue({
  el: '#app',
  data: {
    // 红色
    r: 0,
    // 绿色
    g: 0,
    // 蓝色
    b: 0
  },
  methods: {
    // 点击按钮，在终端显示最新的颜色
    show() {
      console.log(this.rgb)
    }
  },
  // 所有的计算属性，都要定义到 computed 节点之下
  // 计算属性在定义的时候，要定义成“方法格式”
  computed: {
    // rgb 作为一个计算属性，被定义成了方法格式，
    // 最终，在这个方法中，要返回一个生成好的 rgb(x,x,x) 的字符串

    // 这个返回的字符串，会默认的成为Vue对象身上的一个属性
    // 比如说你方法名为rgb 返回值为rgb(0,0,0) 那Vue对象就会有一个属性叫rgb，值为
    // rgb(0,0,0)
    rgb() {
      return `rgb(${this.r}, ${this.g}, ${this.b})`
    }
  }
);
console.log(vm)</pre>
```

```
</script>
</body>

</html>
```

## Vue-Cli

---

### 单页面应用程序

#### 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：



以下页面的功能，都是基于一个html页面实现的：



## 什么是 vue-cli

vue-cli 是 Vue.js 开发的标准工具。它简化了程序员基于webpack 创建工程化的 Vue 项目的过程。

引用自 vue-cli 官网上的一句话：程序员可以专注在撰写应用上，而不必花好几天去纠结 webpack 配置的问题。

中文官网：<https://cli.vuejs.org/zh/>

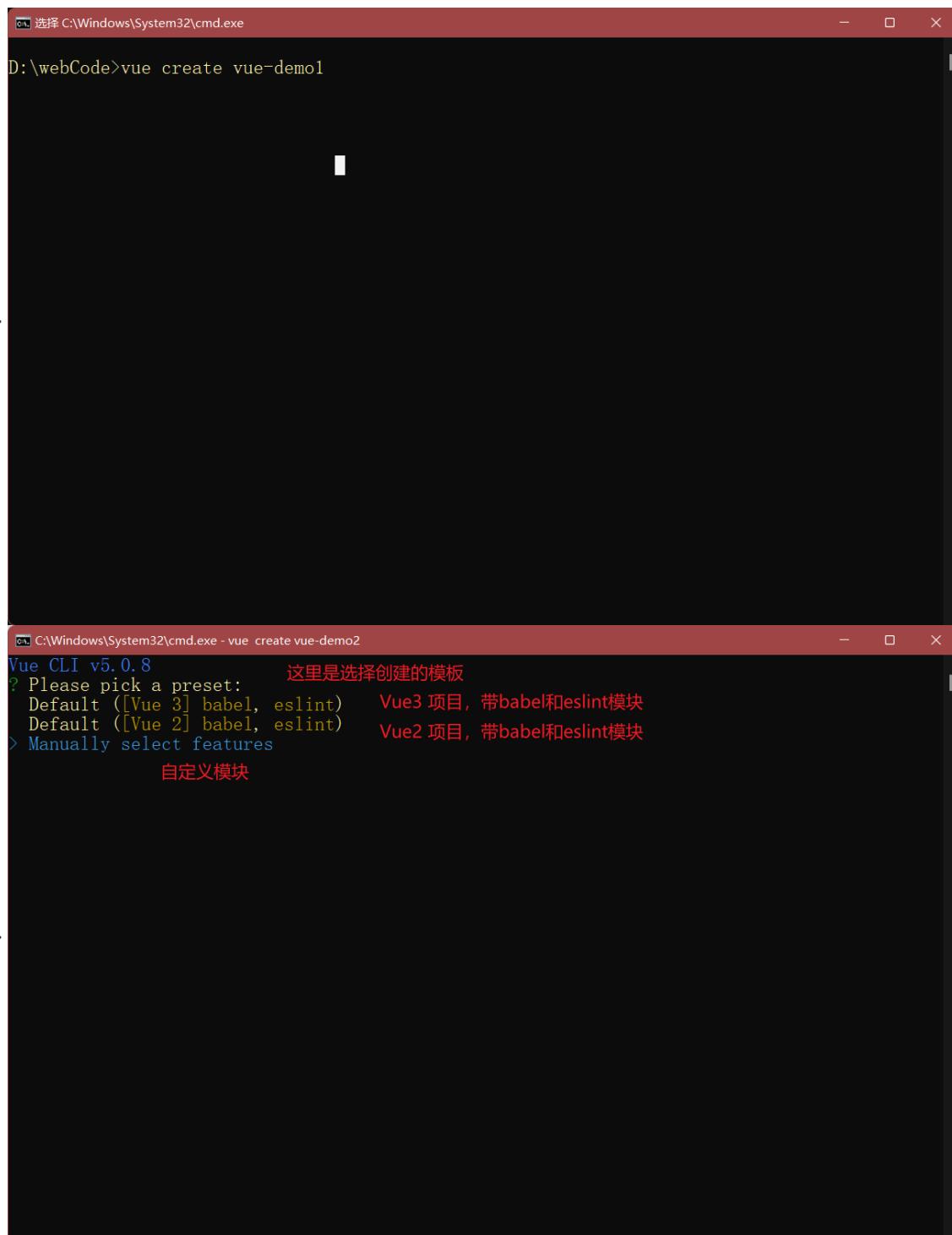
## 安装和使用

vue-cli 是 npm 上的一个全局包，使用 npm install 命令，即可方便的把它安装到自己的电脑上：

```
npm install -g @vue/cli
```

基于 vue-cli 快速生成工程化的 Vue 项目：

```
vue create 项目的名称(项目名称尽量中文)
```



```
C:\Windows\System32\cmd.exe - vue create vue-demo2
Vue CLI v5.0.8
? Please pick a preset: Manually select features
? Check the features needed for your project: (Press <space> to select, <a> to toggle all, <i> to invert selection, and <enter> to proceed)
  (*) Babel      babel
  ( ) TypeScript
  ( ) Progressive Web App (PWA) Support
  ( ) Router
  ( ) Vuex
> (*) CSS Pre-processors  css样式预处理器, 我们之前学的less就是这个玩意
  ( ) Linter / Formatter 注意一定要把这个formatter取消,要不然会莫名其妙的报错
  ( ) Unit Testing
  ( ) E2E Testing
```

3.

```
C:\Windows\System32\cmd.exe - vue create vue-demo2
Vue CLI v5.0.8
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, CSS Pre-processors
? Choose a version of Vue.js that you want to start the project with (Use arrow keys)
> 3.x          选择需要创建Vue的版本
  2.x
```

4.

```
C:\Windows\System32\cmd.exe - vue create vue-demo2
Vue CLI v5.0.8
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, CSS Pre-processors
? Choose a version of Vue.js that you want to start the project with 2.x
? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules are supported by default):
  Sass/SCSS (with dart-sass)
> Less
  Stylus      选择css预处理器，目前只学了less....
```

5.

```
选择 C:\Windows\System32\cmd.exe - vue create vue-demo2
Vue CLI v5.0.8
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, CSS Pre-processors
? Choose a version of Vue.js that you want to start the project with 2.x
? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules are supported by default): Less
? Where do you prefer placing config for Babel, ESLint, etc.? (Use arrow keys)
> In dedicated config files 这里最好选择单独存放，          选择babel和eslint的配置文件的存放位置
  In package.json      不要和其他模块混着存储
```

6.

C:\Windows\System32\cmd.exe - vue create vue-demo2

Vue CLI v5.0.8

? Please pick a preset: Manually select features

? Check the features needed for your project: Babel, CSS Pre-processors

? Choose a version of Vue.js that you want to start the project with 2.x

? Pick a CSS pre-processor (PostCSS, Autoprefixer and CSS Modules are supported by default): Less

? Where do you prefer placing config for Babel, ESLint, etc.? In dedicated config files

? Save this as a preset for future projects? (y/N) 否把我们之前配置的东西保存为一个模板

7.

## vue 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

The screenshot shows a code editor on the left and a file explorer on the right. The code editor contains the following main.js code:

```
// 导入 vue 这个包, 得到 Vue 构造函数
import Vue from 'vue'
// 导入 App.vue 根组件, 将来要把 App.vue 中的模板结构, 渲染到 HTML 页面中
import App from './App.vue'

Vue.config.productionTip = false

// 创建 Vue 的实例对象
new Vue({
  el: '#app',
  // 把 render 函数指定的组件, 渲染到 HTML 页面中
  render: h => h(App)
})
```

The file explorer on the right shows the project structure:

- node\_modules
- public
- favicon.ico
- index.html (highlighted with a red box)
- src
- assets
- components
- App.vue (highlighted with a red box)
- main.js
- .browserslistrc
- .gitignore
- babel.config.js
- package-lock.json
- package.json
- README.md

Annotations in the file explorer:

- A red arrow points from the 'App.vue' entry in the components folder to the 'App.vue' entry in the code editor, with the text '把这个渲染到 占位标签的位置'.
- A red box highlights the 'index.html' file in the public folder, with the text '这里面有个id为app的占位标签'.

此外，关于el挂载点，还有一种书写方式，名叫：`$mount('')`

```
JS main.js M X
vue-demo2 > src > JS main.js > ...
1 import Vue from 'vue'
2 import App from './App.vue'
3
4 Vue.config.productionTip = false
5
6 new Vue({
7   render: h => h(App),
8 }).$mount('#app')
9
10 // 需要注意的是，这两种方式都是可以使用的。是等价的
11 // new Vue({
12 //   el: "#app",
13 //   render: h => h(App),
14 // })
15 |
```

对于render函数的理解：

## 脚手架文件结构

```
|── node_modules
|── public
|   ├── favicon.ico: 页签图标
|   └── index.html: 主页面
└── src
    ├── assets: 存放静态资源
    |   └── logo.png
    ├── component: 存放组件
    |   └── HelloWorld.vue
    ├── App.vue: 汇总所有组件
    └── main.js: 入口文件
├── .gitignore: git版本管制忽略的配置
├── babel.config.js: babel的配置文件
├── package.json: 应用包配置文件
├── README.md: 应用描述文件
└── package-lock.json: 包版本控制文件
```

## 关于不同版本的Vue

1. vue.js与vue.runtime.xxx.js的区别：

1. vue.js是完整版的Vue，包含：核心功能 + 模板解析器。
2. vue.runtime.xxx.js是运行版的Vue，只包含：核心功能；没有模板解析器。

2. 因为vue.runtime.xxx.js没有模板解析器，所以不能使用template这个配置项，需要使用render函数接收到的createElement函数去指定具体内容。

## vue.config.js配置文件

1. 使用`vue inspect > output.js`可以查看到Vue脚手架的默认配置。
2. 使用`vue.config.js`可以对脚手架进行个性化定制，详情见：<https://cli.vuejs.org/zh>

## vue 组件

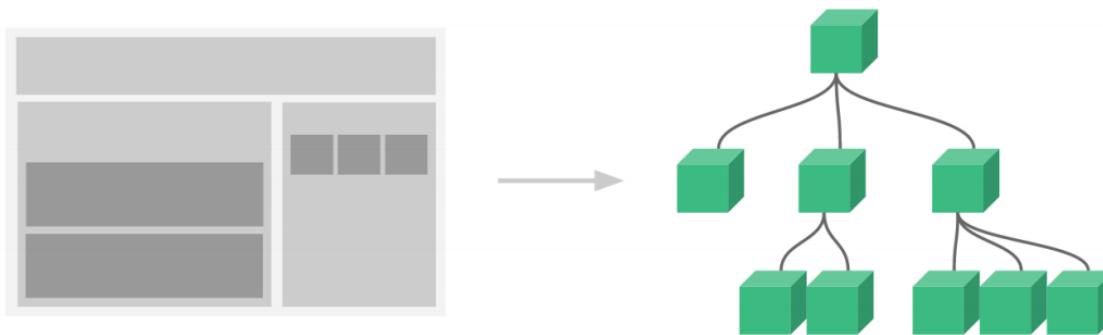
### 什么是组件化开发

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

组件（Component）是 Vue.js 最强大的功能之一。

组件可以扩展 HTML 元素，封装可重用的代码。

组件系统让我们可以用独立可复用的小组件来构建大型应用，几乎任意类型的应用的界面都可以抽象为一个组件树：



### vue 中的组件化开发

vue 是一个支持组件化开发的前端框架。

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。

### vue 组件的三个组成部分

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

#### template

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

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。

```
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

template 中只能包含唯一的根节点，也就是说必须在里面先写一个root根标签...

## script

vue 规定：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>
```

文件为.vue为结尾的，组件中的 data 必须是函数，而不能是对象

```
//data的第一种写法：对象式
/* data: {
  name: '尚硅谷'
} */


//data的第二种写法：函数式
data(){
  console.log('@@@',this) //此处的this是Vue实例对象
  return{
    name: '尚硅谷'
  }
}
```

并且以vue管理的函数，一定不要写箭头函数，要不然会出错，因为箭头函数没有自己的this，框架内部调用会出现错误！！！

vue 规定：.vue 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 }
```

```
//下面的写法是正确的
//组件中的data必须是一个函数
data() {
  //在这里定义我们需要的数据
  return {
    username:'admin'
  }
}
```

## style

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>
2 h1 {
3   font-weight: normal;
4 }
5 </style>
```

让 style 中支持 less 语法

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   span {
5     color: red;
6   }
7 }
8 </style>
```

## 组件之间的父子关系



组件在被封装好之后，彼此之间是相互独立的，不存在父子关系



在使用组件的时候，根据彼此的嵌套关系，形成了父子关系、兄弟关系

### 使用组件的三个步骤

```
<div class="box">  
  <Left></Left>  
</div>  
  
import Left from '@/components/Left.vue'  
  
export default {  
  components: {  
    Left  
  }  
}
```

步骤3：以标签形式使用刚才注册的组件

步骤1：使用 `import` 语法导入需要的组件

步骤2：使用 `components` 节点注册组件

App.vue 组件

还有一个例子，下面这个例子是直接把简单的模板写到components对象中了，但是一般我们不这样：

```

前端 > vue > ▶ 局部组件.html > html > body > script > components
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5      <meta charset="UTF-8">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <meta http-equiv="X-UA-Compatible" content="ie=edge">
8      <title>Document</title>
9  </head>
10
11 <body>
12     <div id="app">
13         <haohao></haohao>
14     </div>
15     <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js">
16     </script>
17     new Vue({
18         el: '#app',
19         // 定义局部组件，这里可以定义多个局部组件
20         components: {
21             //组件的名字
22             'haohao': {
23                 //组件的内容
24                 template: '<ul><li>首页</li><li>学员管理</li></ul>'
25             }
26         }
27     })
28     </script>
29 </body>
30 </html>

```

## 定义组件

```

new Vue({
    el: '#app',
    // 定义局部组件，这里可以定义多个局部组件
    components: {
        //组件的名字
        'haohao': {
            //组件的内容
            template: '<ul><li>首页</li><li>学员管理</li></ul>'
        }
    }
})

```

## 使用组件

```
<haohao></haohao>
```

## 私有组件

通过 components 注册的是私有子组件

例如：

在组件A 的components 节点下，注册了组件F。则组件F 只能用在组件A 中；不能被用在组件C 中。

**上面的例子都是注册的是私有组件，下面的例子全都是注册的全局组件**

## 注册全局组件

第一种方法：

创建js文件

The screenshot shows the VS Code interface with the following details:

- Resource Manager:** Shows the project structure: `> 打开的编辑器`, `< 前端 (工...)`, `< 前端`, `> demo1`, `> es01demo`, `< vue`, `< components`, and `JS Navbar.js` (highlighted with a red box).
- Editor:** The file `JS Navbar.js` contains the following code:

```
// 定义全局组件
Vue.component('haohao', {
  template: '<ul><li>首页</li><li>学员管理</li><li>讲师管理</li></ul>'})
```
- Output Panel:** Shows the transpiled JavaScript code:

```
// 定义全局组件
Vue.component('haohao', {
  template: '<ul><li>首页</li><li>学员管理</li><li>讲师管理</li></ul>'})
```

```
body>
  <div id="app">
    |   <haohao></haohao> ②
  </div>
  <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
  <script src="components/Navbar.js"></script>
```

引入并使用

第二种方法：

还有一种方法：

在 vue 项目的 `main.js` 入口文件中，通过 `Vue.component()` 方法，可以注册全局组件。示例代码如下：

下面的代码是写在 `main.js` 中的：

```
1 // 导入需要全局注册的组件
2 import Count from '@/components/Count.vue'
3
4 // 参数1：字符串格式，表示组件的“注册名称”
5 // 参数2：需要被全局注册的那个组件
6 Vue.component('MyCount', Count)
```

## 组件的 props

props 是组件的自定义属性，在封装通用组件的时候，合理地使用 props 可以极大的提高组件的复用性！它的语法格式如下：

```
1 export default {
2   // 组件的自定义属性
3   props: ['自定义属性A', '自定义属性B', '其它自定义属性...'],
4
5   // 组件的私有数据
6   data() {
7     return {}
8   }
9 }
```

### props 是只读的

vue 规定：组件中封装的自定义属性是只读的，程序员不能直接修改 props 的值。否则会直接报错：

```
[HMR] Waiting for update signal from WDS...                                     log.js?1afcd:24
vue-devtools Detected Vue v2.6.14                                         backend.js:2237
✖ [Vue warn]: Avoid mutating a prop directly since the value will vue.runtime.esm.js?2b0e:619
be overwritten whenever the parent component re-renders. Instead, use a data or computed
property based on the prop's value. Prop being mutated: "init"
found in
--> <MyCount> at src/components/Count.vue
    <Left> at src/components/Left.vue
    <App> at src/App.vue
    <Root>
```

要想修改props 的值，可以把 props 的值转存到 data 中，因为 data 中的数据都是可读可写的！

```
1 props: ['init'],
2 data() {
3   return {
4     count: this.init // 把 this.init 的值转存到 count
5   }
6 }
```

可以这样理解，props的值我们只做**初始值**，如果想修改他，就把他添加到data里面就可以了

## props 的 default 默认值

在声明自定义属性时，可以通过default 来定义属性的默认值。示例代码如下：

```
1 export default {
2   props: {
3     init: {
4       // 用 default 属性定义属性的默认值
5       default: 0
6     }
7   }
8 }
```

## props 的 type 值类型

在声明自定义属性时，可以通过type 来定义属性的值类型。示例代码如下：

```
1 export default {
2   props: {
3     init: {
4       // 用 default 属性定义属性的默认值
5       default: 0,
6       // 用 type 属性定义属性的值类型，
7       // 如果传递过来的值不符合此类型，则会在终端报错
8       type: Number
9     }
10   }
11 }
```

## props 的 required 必填项

在声明自定义属性时，可以通过 required 选项，将属性设置为必填项，强制用户必须传递属性的值。示例代码如下：

```
1 export default {
2   props: {
3     init: {
4       // 值类型为 Number 数字
5       type: Number,
6       // 必填项校验
7       required: true
8     }
9   }
10 }
```

## 具体代码

封装者：

Count.vue

```
<template>
<div>
  <h5>Count 组件</h5>
  <p>count 的值是: {{ count }}</p>
  <button @click="count += 1">+1</button>

  <button @click="show">打印 this</button>
</div>
</template>

<script>
export default {
  // props 是"自定义属性", 允许使用者通过自定义属性, 为当前组件指定初始值
  // 自定义属性的名字, 是封装者自定义的(只要名称合法即可)
  // props 中的数据, 可以直接在模板结构中被使用
  // 注意: props 是只读的, 不要直接修改 props 的值, 否则终端会报错!
  // props: ['init'],
  props: {
    // 自定义属性A : { /* 配置选项 */ },
    // 自定义属性B : { /* 配置选项 */ },
    // 自定义属性C : { /* 配置选项 */ },
    init: {
      // 如果外界使用 Count 组件的时候, 没有传递 init 属性, 则默认值生效
      default: 0,
      // init 的值类型必须是 Number 数字
      // 这里 vue 帮我们做了校验, 如果传参不正确就会终端报错!
      type: Number,
      // 必填项校验, 就算有默认值, 但是调用者没传参, 还是会报错
      required: true
    },
    },
    data() {
      return {
        // 把 props 中的 init 值, 转存到 count 上
        count: this.init
      }
    },
    methods: {
      show() {
        console.log(this)
      }
    }
}
</script>

<style lang="less"></style>
```

使用者：

Left.vue

```
<template>
  <div class="left-container">
    <h3>Left 组件</h3>
    <hr />
    <!-- 这里使用了v-bind， 保证传参是个数字 -->
    <MyCount :init="9"></MyCount>
  </div>
</template>
```

## 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，都会渲染到唯一的index.html 页面进行呈现
- ② 每个组件中的样式，都会影响整个index.html 页面中的DOM 元素

### 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>
```

### 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（原理就是上面的加一个自定义属性。。。可以在开发者页面上面看到）：

```
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>
```

## /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>
```

上面红框里面一个是并集选择器，一个是子代选择器

这个玩意可以修改组件库的样式，等到后面用到了我再回来填坑。。。

## 组件的构造函数

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>VueComponent</title>
    <script type="text/javascript" src="../js/vue.js"></script>
  </head>
  <body>
    <!--
        关于VueComponent:
        1. school组件本质是一个名为VueComponent的构造函数，且不是程序员定义的，是Vue.extend生成的。
    -->
```

2. 我们只需要写<school/>或<school></school>, vue解析时会帮我们创建school组件的实例对象，

即vue帮我们执行的: new VueComponent(options)。

3. 特别注意: 每次调用Vue.extend, 返回的都是一个全新的VueComponent!!!!

4. 关于this指向:

(1). 组件配置中:

data函数、methods中的函数、watch中的函数、computed中的函数 它们的this均是【VueComponent实例对象】。

(2). new Vue(options)配置中:

data函数、methods中的函数、watch中的函数、computed中的函数 它们的this均是【Vue实例对象】。

5. VueComponent的实例对象, 以后简称vc (也可称之为: 组件实例对象)。

Vue的实例对象, 以后简称vm。

```
-->
<!-- 准备好一个容器-->
<div id="root">
  <school></school>
  <hello></hello>
</div>
</body>

<script type="text/javascript">
  vue.config.productionTip = false

  //定义school组件
  const school = Vue.extend({
    name:'school',
    template: `
      <div>
        <h2>学校名称: {{name}}</h2>
        <h2>学校地址: {{address}}</h2>
        <button @click="showName">点我提示学校名</button>
      </div>
    `,
    data(){
      return {
        name:'尚硅谷',
        address:'北京'
      }
    },
    methods: {
      showName(){
        console.log('showName',this)
      }
    }
  })

  const test = Vue.extend({
    template: `<span>atguigu</span>`
  })
```

```

// 定义hello组件
const hello = Vue.extend({
  template: `
    <div>
      <h2>{{msg}}</h2>
      <test></test>
    </div>
  `,
  data() {
    return {
      msg: '你好啊！'
    }
  },
  components:{test}
})

// console.log('@',school)
// console.log('#',hello)

// 创建vm
const vm = new Vue({
  el:'#root',
  components:{school,hello}
})
</script>
</html>

```

## Vue和VueComponent的内置关系

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>一个重要的内置关系</title>
    <!-- 引入Vue -->
    <script type="text/javascript" src="../../js/vue.js"></script>
  </head>
  <body>
    <!--
        1. 一个重要的内置关系: VueComponent.prototype.__proto__ ===
        Vue.prototype
        2. 为什么要有个关系: 让组件实例对象 (vc) 可以访问到 Vue 原型上的属性、方法。
    -->
    <!-- 准备好一个容器-->
    <div id="root">
      <school></school>
    </div>
  </body>

```

```
<script type="text/javascript">
Vue.config.productionTip = false //阻止 vue 在启动时生成生产提示。
Vue.prototype.x = 99

//定义school组件
const school = Vue.extend({
  name:'school',
  template:`
    <div>
      <h2>学校名称: {{name}}</h2>
      <h2>学校地址: {{address}}</h2>
      <button @click="showX">点我输出x</button>
    </div>
  `,
  data(){
    return {
      name:'尚硅谷',
      address:'北京'
    }
  },
  methods: {
    showX(){
      console.log(this.x)
    }
  },
})

//创建一个vm
const vm = new Vue({
  el:'#root',
  data:{
    msg:'你好'
  },
  components:{school}
})

//定义一个构造函数
/* function Demo(){
  this.a = 1
  this.b = 2
}
//创建一个Demo的实例对象
const d = new Demo()

console.log(Demo.prototype) //显示原型属性

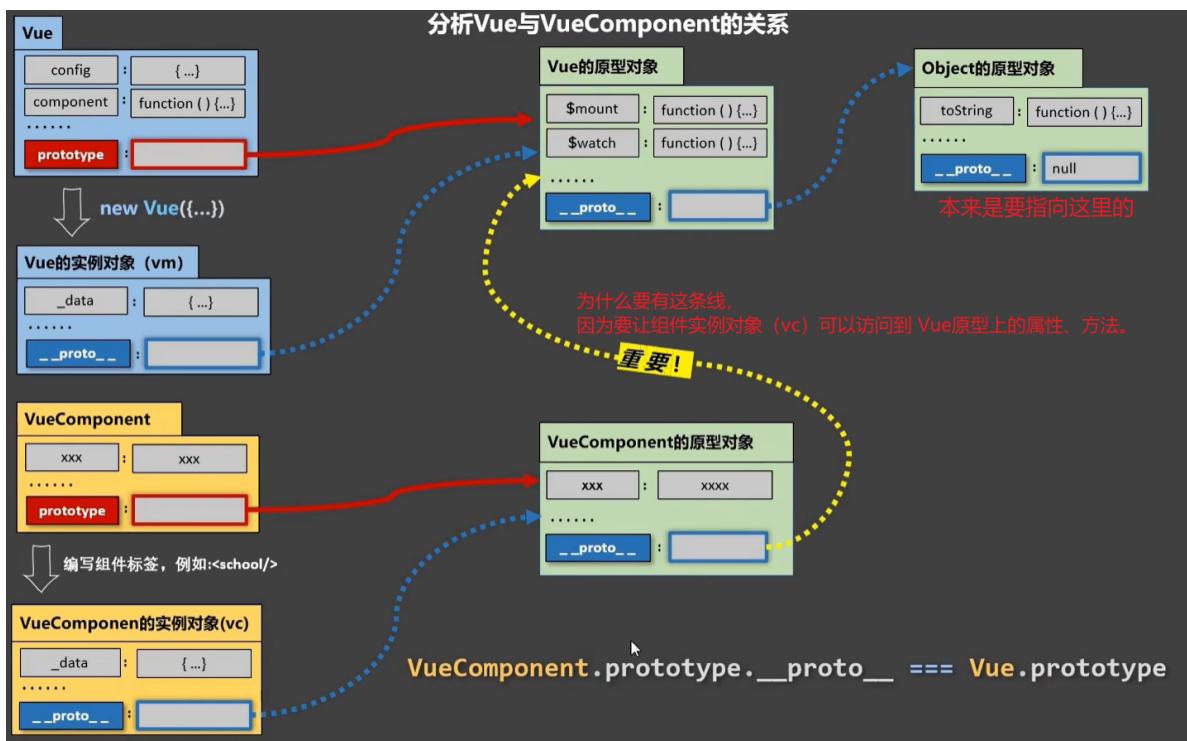
console.log(d.__proto__) //隐式原型属性

console.log(Demo.prototype === d.__proto__)

//程序员通过显示原型属性操作原型对象，追加一个x属性，值为99
Demo.prototype.x = 99

console.log('@',d) */
```

```
</script>
</html>
```



## 生命周期

- 生命周期 (Life Cycle) 是指一个组件从创建 -> 运行-> 销毁的整个阶段，强调的是一个时间段。
- 生命周期函数：是由 vue 框架提供的内置函数，会伴随着组件的生命周期，自动按次序执行。

注意：生命周期强调的是时间段，生命周期函数强调的是时间点。

## 实例生命周期

我们主要了解的是在渲染页面之前会执行 `created()` 方法，在渲染完成后执行 `mounted()` 方法，那么如何查看呢，我们了解了这两个方法就可以了，因为主要用的就是这两个。

前端也有debug模式，通过 `debugger` 关键字来使用

```
<body>
  <div id="app">
    <h1>hahahahaha</h1>
  </div>
  <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
  <script>
    new Vue({
      el: '#app',
      data: {

      },
      created() {
        debugger
      }
    })
  </script>

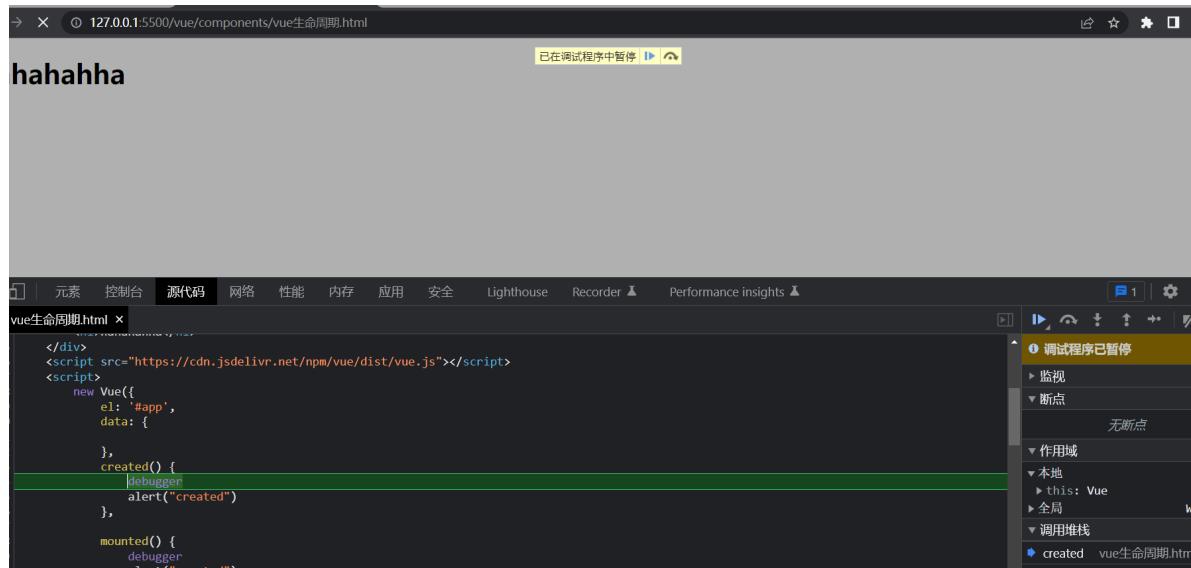
```

```

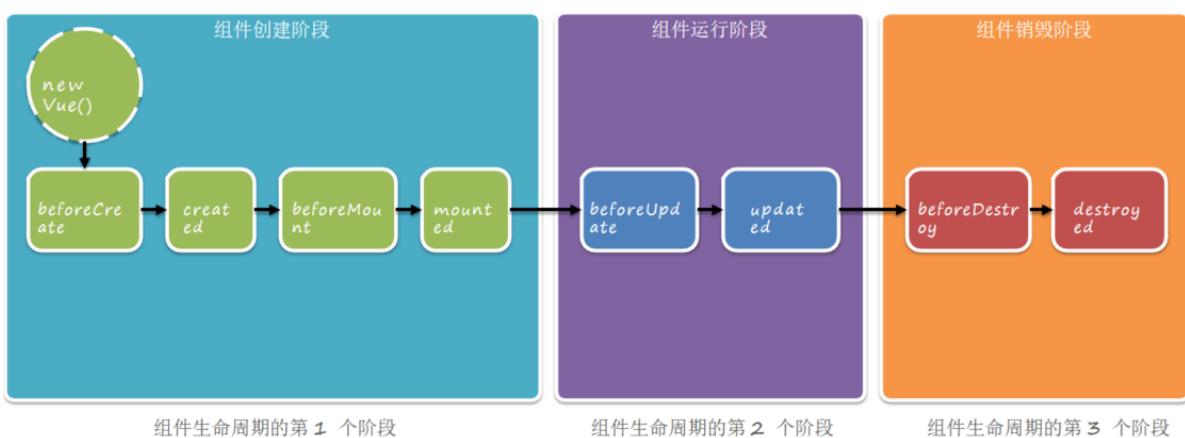
        alert("created")
    },
    mounted() {
        debugger
        alert("mounted")
    }
})
</script>
</body>

```

直接启动之后运行一下点刷新就可以进入debug模式了。



## 组件生命周期函数的分类



## 生命周期的八大钩子函数

什么是钩子函数，官方文档：

每个 Vue 实例在被创建时都要经过一系列的初始化过程——例如，需要设置数据监听、编译模板、将实例挂载到 DOM 并在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做**生命周期钩子**的函数，这给了用户在不同阶段添加自己的代码的机会。

简单来说，钩子函数就算创建Vue在初始化、更新数据、销毁时会自动被调用的函数

[API — Vue.js (vuejs.org)](https://v2.cn.vuejs.org/v2/api/#选项-生命周期钩子)

八大钩子函数分别是：

- beforeCreate,

- created,

- beforeMount,

- mounted,

- beforeUpdate,

- updated,

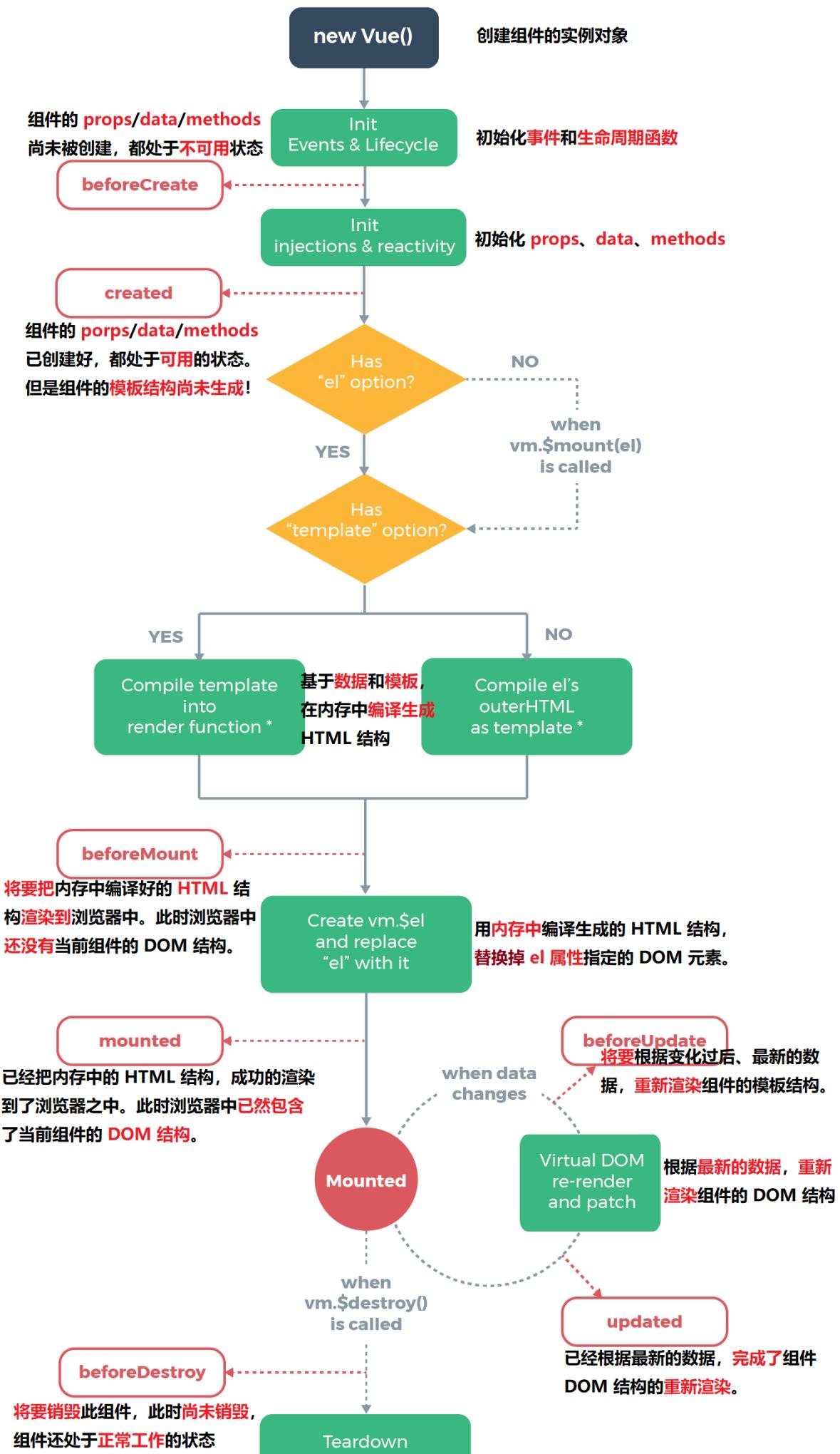
- beforeDestory,

- destoryed

官网的还多了几个其他的，但是目前我们用不太到，只需要了解即可

The screenshot shows a browser window displaying the Vue.js API documentation at <https://v2.cn.vuejs.org/v2/api/#选项-生命周期钩子>. The page title is "API — Vue.js". The left sidebar contains a tree view of the API structure, with "选项 / 生命周期钩子" expanded. The main content area is titled "选项 / 生命周期钩子". It includes a note about the "this" context for all lifecycle hooks and detailed descriptions for "#beforeCreate" and "#created". The "#beforeCreate" section notes that it runs after component initialization and before data and event listeners are configured. The "#created" section notes that it runs after the component has been created.

用图片来表示，下图新增了两个新的方法：beforeUpdate()和updated()方法





\* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

## 一、beforeCreate,created

beforeCreate可以简单的理解为在数据初始化的之前被调用，这时候data和methods尚未没有数据。

created可以理解为在数据初始化之后被调用，这时候data和methods已经被填充了相应的数据。

```
<body>
  <div id="app"></div>
  <script>
    var vm=new Vue({
      el: "#app",
      data: {
        msg:"在这之间"          //添加msg数据
      },
      beforeCreate() {
        console.log("this= "+this)
        console.log("this.msg= "+this.msg)
        console.log("this.md= "+this.md)
        console.log("")
      },
      created() {
        console.log("this= "+this)
        console.log("this.msg= "+this.msg)
        console.log("this.md= "+this.md)
        console.log("")
      },
      methods: {
        md: function(){},           //空方法
      }
    });
  </script>
</body>
```

结果：

```
this= [object Object]
this.msg= undefined 可见beforeCreate方法执行时：该Vue对象已创建，data中没有数据，methods中没有数据
this.md= undefined
```

```
this= [object Object]
this.msg= 在这之间 可见created方法执行时：该Vue对象已创建，data中有数据，methods中有数据
this.md= function () { [native code] }
```

在beforeCreate方法与create方法之间完成了资源的注入

## 二、beforeMount,mounted

上面实验已经证明Vue中数据已经注入完毕，beforeMount,mounted则是与页面渲染有关

beforeMount在页面尚未被渲染时使用，也就是Vue的数据没有传到页面。

mounted在页面渲染完成之后使用，也就是此时页面已完全取出Vue中的数据。

实验测试：

```
<body>
  <div id="app">
    <h1 id="ren">{{msg}}</h1>
  </div>
  <script>
    var vm=new Vue({
      el: "#app",
      data: {
        msg:"在这之间" //添加msg数据
      },
      beforeMount() {
        let doc = document.querySelector("#ren");//查询到id名为ren的节点
        console.log(doc)
        console.log("")
      },
      mounted() {
        let doc = document.querySelector("#ren");
        console.log(doc)
      },
    });
  </script>
</body>
```

结果如下：

<h1 id="ren">{{msg}}</h1> msg的数据此时尚未取出

<h1 id="ren">在这之间</h1> msg的值已经渲染到页面

此时，Vue对象中资源已注入完毕，页面也已经渲染完毕，上述四个方法在页面被加载时自动被执行

## 三、beforeUpdate,updated

- beforeUpdate在页面更新渲染完成后，DOM树发生改变前被调用
- updated在页面DOM树改变后被调用

需要注意的是如果只是改变了dom中的数据（data），未对页面造成任何影响，就不会触发beforeUpdate,updated方法。

```

<body>
  <div id="app">
    <h1 id="ren">
      <p v-if="msg"></p>
    </h1>
  </div>
  <script>
    var vm=new Vue({
      el: "#app",
      data: {
        msg:true          //添加msg数据
      },
      beforeupdate() {
        let a = document.getElementById("ren");
        console.log(a.childElementCount)
        console.log("")
      },
      updated() {
        let a = document.getElementById("ren");
        console.log(a.childElementCount)
      },
    });
  </script>
</body>

```

结果显示：

> vm.msg=false	
1	dom树还未删除节点
0	dom树已经删除节点

#### 四、beforeDestory,destroyed

**beforeDestory**是在Vue组件销毁之前被调用

**destroyed**在Vue组件销毁之后被调用

这里为了搭建环境，引入了组件的概念（注意由于解析时自上而下，所以组件写在Vue对象前）

```

<body>
    <div id="app">
        <mytest></mytest>          自定义组件就是模板，可以被多次引用
    </div>
    <script>
        let myname = Vue.component('mytest', {
            template: '<li>{{msg}}</li>',           在Vue中引入组件
            data :function(){
                return{
                    msg : "这是我的测试"
                }
            },
            methods: {
                m1:function(){}
            },
        });
        var vm=new Vue({
            el: "#app",
            data: {},
            components:{             组件标签名，自定义
                "mytest" : myname,
            }
        });

    </script>
</body>

```

[https://blog.csdn.net/a\\_killer\\_](https://blog.csdn.net/a_killer_)

```

<body>
    <div id="app">
        <mytest id="child" v-if="flag">
        </mytest>
    </div>
    <script>

        let myname = Vue.component('mytest', {
            template: '<p>yes</p>',

            beforeDestroy() {
                console.log("beforeDestroy被执行")
            },
            destroyed() {
                console.log("destroyed被执行")
            },
        });
        var vm=new Vue({
            el: "#app",
            data: {
                flag: true
            },
            components:{
                "mytest" : myname,
            },
        });

    </script>
</body>

```

```

> vm.flag=false
beforeDestroy被执行
destroyed被执行
< false

```

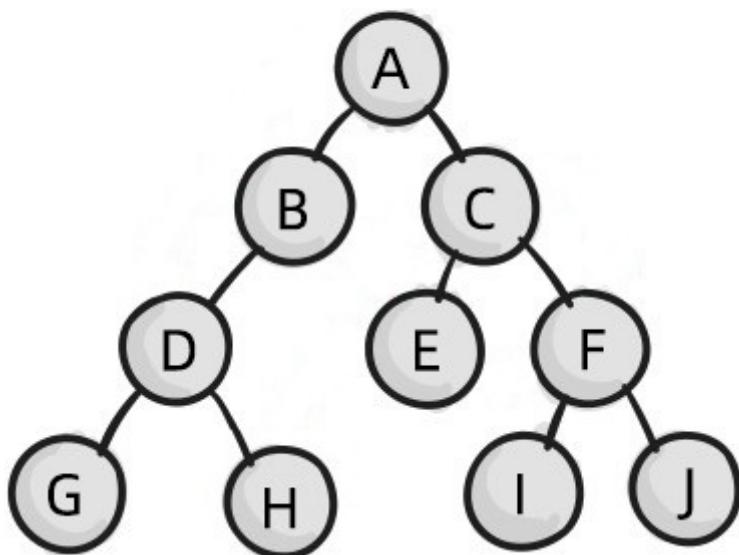
# 组件之间的数据共享

## 组件之间的关系

在项目开发中，组件之间的最常见的关系分为如下两种：

- ① 父子关系
- ② 兄弟关系

## 组件之间的关系



## 父子组件之间的数据共享

父子组件之间的数据共享又分为：

- ① 父 -> 子共享数据
- ② 子 -> 父共享数据

## 父组件向子组件共享数据

父组件向子组件共享数据需要使用自定义属性。示例代码如下：

```
1 // 父组件
2 <Son :msg="message" :user="userinfo"></Son>
3
4 data() {
5   return {
6     message: 'hello vue.js',
7     userinfo: { name: 'zs', age: 20 }
8   }
9 }
```

```
1 <template>
2   <div>
3     <h5>Son 组件</h5>
4     <p>父组件传递过来的 msg 值是: {{ msg }}</p>
5     <p>父组件传递过来的 user 值是: {{ user }}</p>
6   </div>
7 </template>
8
9 props: ['msg', 'user']
```

## 子组件向父组件共享数据

子组件向父组件共享数据使用自定义事件。示例代码如下：



## 子组件

```
1 export default {  
2   data() {  
3     return { count: 0 }  
4   },  
5   methods: {  
6     add() {  
7       this.count += 1  
8       // 修改数据时，通过 $emit() 触发自定义事件  
9       this.$emit('numchange', this.count)  
10    }  
11  }  
12 }
```

父组件

```
1 <Son @numchange="getNewCount"></Son>
2
3 export default {
4   data() {
5     return { countFromSon: 0 }
6   },
7   methods: {
8     getNewCount(val) {
9       this.countFromSon = val
10    }
11  }
12 }
```

### 组件的自定义事件

1. 一种组件间通信的方式，适用于：**子组件 ==> 父组件**
2. 使用场景：A是父组件，B是子组件，B想给A传数据，那么就要在A中给B绑定自定义事件（**事件的回调在A中**）。
3. 绑定自定义事件：

1. 第一种方式，在父组件中：`<Demo @atguigu="test"/>` 或 `<Demo v-on:atguigu="test"/>`

2. 第二种方式，在父组件中：

```
<Demo ref="demo"/>
.....
mounted(){
  this.$refs.demo.$on('atguigu', this.test)
}
```

3. 若想让自定义事件只能触发一次，可以使用 `once` 修饰符，或 `$once` 方法。
4. 触发自定义事件：`this.$emit('atguigu', 数据)`
5. 解绑自定义事件 `this.$off('atguigu')`
6. 组件上也可以绑定原生DOM事件，需要使用 `native` 修饰符。
7. 注意：通过 `this.$refs.xxx.$on('atguigu', 回调)` 绑定自定义事件时，回调**要么配置在 methods 中，要么用箭头函数**，否则 `this` 指向会出现问题！

## 兄弟组件之间的数据共享

在 vue2.x 中，兄弟组件之间数据共享的方案是 EventBus。

```
import bus from './eventBus.js'

export default {
  data() {
    return {
      msg: 'hello vue.js'
    }
  },
  methods: {
    sendMsg() {
      bus.$emit('share', this.msg)
    }
  }
}
```

兄弟组件 A (数据发送方)

```
import Vue from 'vue'
// 向外共享 Vue 的实例对象
export default new Vue()
```

eventBus.js

```
import bus from './eventBus.js'

export default {
  data() {
    return {
      msgFromLeft: ''
    }
  },
  created() { bus.$on('share', val => {
    this.msgFromLeft = val
  })
  }
}
```

兄弟组件 C (数据接收方)

EventBus 的使用步骤

- ① 创建eventBus.js 模块，并向外共享一个 Vue 的实例对象
- ② 在数据发送方，调用bus.  
`emit('事件名称', 要发送的数据)`方法触发自定义事件
- ③ 在数据接收方，调用`bus.on('事件名称', 事件处理函数)`方法注册一个自定义事件

## 全局事件总线

这个玩意可以实现任意组件间的通信 (GlobalEventBus)

1. 一种组件间通信的方式，适用于**任意组件间通信**。
2. 安装全局事件总线：

```
new Vue({
  .....
  beforeCreate() {
    Vue.prototype.$bus = this //安装全局事件总线，$bus就是当前应用的vm
  },
  .....
})
```

3. 使用事件总线：

1. 接收数据：A组件想接收数据，则在A组件中给\$bus绑定自定义事件，事件的**回调留在A组件自身**。

```
methods(){
  demo(data){.....}
}

.....
mounted() {
  this.$bus.$on('xxxx', this.demo)
}
```

2. 提供数据：`this.$bus.$emit('xxxx', 数据)`

4. 最好在beforeDestroy钩子中，用\$off去解绑**当前组件所用到的**事件。

具体代码示例：

main.js

```
<template>
<div class="student">
  <h2>学生姓名: {{ name }}</h2>
  <h2>学生性别: {{ sex }}</h2>
  <button @click="sendStudentName">把学生名给School组件</button>
</div>
</template>

<script>
export default {
  name: "Student",
  data() {
    return {
      name: "张三",
      sex: "男",
    };
  },
  methods: {
    sendStudentName() {
      this.$bus.$emit("hello", this.name);
    },
  },
}
</script>

<style lang="less" scoped>
.student {
  background-color: pink;
  padding: 5px;
  margin-top: 30px;
}
</style>
```

school.vue

```
<template>
<div class="school">
  <h2>学校名称: {{ name }}</h2>
  <h2>学校地址: {{ address }}</h2>
</div>
</template>

<script>
export default {
  name: "School",
```

```

data() {
  return {
    name: "尚硅谷",
    address: "北京",
  };
},
//在数据初始化完成阶段，给$bus绑定自定义事件
mounted() {
  this.$bus.$on("hello", (data) => {
    console.log("我是School组件，收到了数据", data);
  });
},
//在组件将要被销毁的时候需要把这个自定义事件解绑，正如我轻轻的来，轻轻的走，不带走一片云彩
beforeDestroy() {
  this.$bus.$off("hello");
},
}
</script>

<style scoped>
.school {
  background-color: skyblue;
  padding: 5px;
}
</style>

```

student.vue

```

<template>
<div class="student">
  <h2>学生姓名: {{ name }}</h2>
  <h2>学生性别: {{ sex }}</h2>
  <button @click="sendStudentName">把学生名给School组件</button>
</div>
</template>

<script>
export default {
  name: "Student",
  data() {
    return {
      name: "张三",
      sex: "男",
    };
  },
  methods: {
    sendStudentName() {
      this.$bus.$emit("hello", this.name);
    },
  },
};
</script>

<style lang="less" scoped>
.student {

```

```
background-color: pink;
padding: 5px;
margin-top: 30px;
}
</style>
```

## 消息订阅与发布 (pubsub)

1. 一种组件间通信的方式，适用于任意组件间通信。

2. 使用步骤：

1. 安装pubsub: `npm i pubsub-js`

2. 引入: `import pubsub from 'pubsub-js'`

3. 接收数据：A组件想接收数据，则在A组件中订阅消息，订阅的回调留在A组件自身。

```
methods(){
    demo(data){.....}
}

.....
mounted() {
    this.pid = pubsub.subscribe('xxx',this.demo) //订阅消息
}
```

4. 提供数据: `pubsub.publish('xxx', 数据)`

5. 最好在beforeDestroy钩子中，用 `PubSub.unsubscribe(pid)` 去取消订阅。

School.vue

```
<template>
<div class="school">
    <h2>学校名称: {{name}}</h2>
    <h2>学校地址: {{address}}</h2>
</div>
</template>

<script>
    import pubsub from 'pubsub-js'
    export default {
        name:'School',
        data() {
            return {
                name:'尚硅谷',
                address:'北京',
            }
        },
        mounted() {
            //这里把pubid挂载到vc对象身上，但是我感觉这种方法不太优雅，因为这个组件可能有多个
            //消息
            this.pubId = pubsub.subscribe('hello',(msgName,data)=>{
                console.log(this)//这里是vc对象，因为这个是箭头函数，并且是外部的api进行
                调用的，所以vc->外部api->这个函数，所以这个this就是vc对象
            })
        }
    }
</script>
```

```

        console.log('有人发布了hello消息, hello消息的回调执行了', msgName, data)
    })
},
beforeDestroy() {
    //这里通过id进行解绑对象, 很想定时器
    pubsub.unsubscribe(this.pubId)
},
}
</script>

<style scoped>
.school{
    background-color: skyblue;
    padding: 5px;
}
</style>

```

student.vue

```

<template>
<div class="student">
    <h2>学生姓名: {{name}}</h2>
    <h2>学生性别: {{sex}}</h2>
    <button @click="sendStudentName">把学生名给School组件</button>
</div>
</template>

<script>
import pubsub from 'pubsub-js'
export default {
    name:'Student',
    data() {
        return {
            name:'张三',
            sex:'男',
        }
    },
    methods: {
        sendStudentName(){
            //这里写你要输入的消息
            pubsub.publish('hello',666)
        }
    }
}
</script>

<style lang="less" scoped>
.student{
    background-color: pink;
    padding: 5px;
    margin-top: 30px;
}
</style>

```

# Vue封装的过度与动画

1. 作用：在插入、更新或移除 DOM元素时，在合适的时候给元素添加样式类名。

2. 写法：

- 元素在进入的时候会加三个class：

3. v-enter：进入的起点

2. v-enter-active：进入过程中

3. v-enter-to：进入的终点

- 元素在离开的时候会加三个class：

1. v-leave：离开的起点

2. v-leave-active：离开过程中

3. v-leave-to：离开的终点

- 不过需要注意的是，v-leave和v-enter这两个class只会存在一帧，就会被vue删除，因为已经达到使用样式的效果了

1. 使用 `<transition>` 包裹要过度的元素，并配置name属性：

```
<transition name="hello">
  <h1 v-show="isShow">你好啊！</h1>
</transition>
```

2. 备注：若有很多个元素需要过度，则需要使用：`<transition-group>`，且每个元素都要指定 key 值。

## 动画

```
<template>
  <div>
    <button @click="isShow = !isShow">显示/隐藏</button>
    <!-- 这里添加一个appear属性是初始的时候需要加载一个enter动画 -->
    <transition name="hello" appear>
      <h1 v-show="isShow">你好啊！</h1>
    </transition>
  </div>
</template>

<script>
export default {
  name: "Test",
  data() {
    return {
      isShow: true,
    };
  },
};
</script>

<style scoped>
h1 {
```

```

background-color: orange;
}
/*
下面这两个样式的命名规则是，如果你上面的标签加了name属性，这里的名字就是
name属性值-enter-active
name属性值-leave-active
*/
.hello-enter-active {
  animation: atguigu 0.5s linear;
}

.hello-leave-active {
  animation: atguigu 0.5s linear reverse;
}

@keyframes atguigu {
  from {
    transform: translateX(-100%);
  }
  to {
    transform: translateX(0px);
  }
}

```

## 过度

利用过度也可以实现同样的效果

```

<template>
  <div>
    <button @click="isShow = !isShow">显示/隐藏</button>
    <transition name="hello" appear>
      <h1 v-show="isShow">尚硅谷! </h1>
    </transition>
  </div>
</template>

<script>
export default {
  name: "Test",
  data() {
    return {
      isShow: true,
    };
  },
};
</script>

<style scoped>
h1 {
  background-color: orange;
}
/* 进入的起点、离开的终点 */

```

```

.hello-enter,
.hello-leave-to {
  transform: translateX(-100%);
}
/* 进入的终点、离开的起点 */
.hello-enter-to,
.hello-leave {
  transform: translateX(0);
}

/* 给这两个阶段设置样式，离开花费0.5s，并且效果是匀速 */
.hello-enter-active,
.hello-leave-active {
  transition: 0.5s linear;
}

```

## 引入第三方库完成过度和动画

```

<template>
<div>
  <button @click="isShow = !isShow">显示/隐藏</button>
  <!-- 引入第三方库，就不用我们自己写进入动画和出动画的效果了 直接在属性上配置就行 -->
  <!-- enter-active-class 进入动画 -->
  <!-- leave-active-class 离开动画 -->
  <transition-group
    appear
    name="animate__animated animate__bounce"
    enter-active-class="animate__swing"
    leave-active-class="animate__backoutUp"
  >
    <h1 v-show="!isShow" key="1">你好啊！</h1>
    <h1 v-show="isShow" key="2">尚硅谷！</h1>
  </transition-group>
</div>
</template>

<script>
import "animate.css";
export default {
  name: "Test",
  data() {
    return {
      isShow: true,
    };
  },
};
</script>

<style scoped>
h1 {
  background-color: orange;
}

```

## nextTick

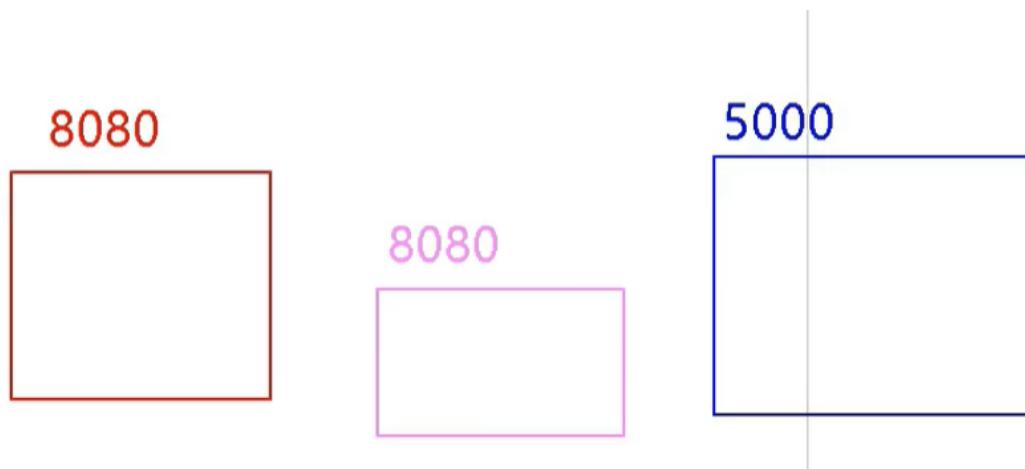
1. 语法: `this.$nextTick(回调函数)`
2. 作用: 在下一次 DOM 更新结束后执行其指定的回调。
3. 什么时候用: 当改变数据后, 要基于更新后的DOM进行某些操作时, 要在nextTick所指定的回调函数中执行。

比如说你要显示一个输入框并获取焦点, 首先在函数里面你先把显示输入框改为true, 这时候dom还没更新, 你就调用focus是不好使的, 所以这个api就是让渲染完dom才会执行这里面的语句

```
//编辑
handleEdit(todo){
  if(todo.hasOwnProperty('isEdit')){
    todo.isEdit = true
  }else{ ...
  }
  // 等渲染完dom才会执行这个语句
  this.$nextTick(function(){
    this.$refs.inputTitle.focus()
  })
},
```

## 利用代理解决跨域问题

[配置参考 | Vue CLI \(vuejs.org\)](#)



## 方法一

在`vue.config.js`中添加如下配置:

```
//下面这个是代理服务器需要转发到的地址, 如果发请求就直接发送给前端的端口就行了
devServer:{
  proxy:"http://localhost:5000"
}
```

说明：

1. 优点：配置简单，请求资源时直接发给前端（8080）即可。
2. 缺点：不能配置多个代理，不能灵活的控制请求是否走代理。
3. 工作方式：若按照上述配置代理，当请求了前端不存在的资源时，那么该请求会转发给服务器（优先匹配前端资源）

## 方法二

编写vue.config.js配置具体代理规则：

```
module.exports = {
  devServer: {
    proxy: {
      '/api1': { // 匹配所有以 '/api1' 开头的请求路径
        target: 'http://localhost:5000', // 代理目标的基础路径
        changeOrigin: true,
        pathRewrite: {'^/api1': ''}
      },
      '/api2': { // 匹配所有以 '/api2' 开头的请求路径
        target: 'http://localhost:5001', // 代理目标的基础路径
        changeOrigin: true,
        pathRewrite: {'^/api2': ''}
      }
    }
  }
/*
  changeOrigin 设置为 true 时，服务器收到的请求头中的 host 为： localhost:5000 撒谎
  changeOrigin 设置为 false 时，服务器收到的请求头中的 host 为： localhost:8080 不撒谎
  changeOrigin 默认值为 true
*/
}
```

说明：

1. 优点：可以配置多个代理，且可以灵活的控制请求是否走代理。
2. 缺点：配置略微繁琐，请求资源时必须加前缀。

## ref 引用

### 什么是 ref 引用

ref 应用在 html 标签上获取的是真实 DOM 元素，应用在组件标签上是组件实例对象（vc）

ref 用来辅助开发者在不依赖于 jQuery 的情况下，获取 DOM 元素或组件的引用。

每个 vue 的组件实例上，都包含一个 `refs` 对象，里面存储着对应的 DOM 元素或组件的引用。默认情况下，组件的 `refs` 指向一个空对象。

```
1 <template>
2   <div>
3     <h3>MyRef 组件</h3>
4     <button @click="getRef">获取 $refs 引用</button>
5   </div>
6 </template>
7
8 export default {
9   methods: {
10     getRef() { console.log(this) } // this 是当前组件的实例对象, this.$refs 默认指向空对象
11   }
12 }
```

上面打印的this，就是当前的vue实例对象，里面有一个\$refs属性，默认为空，如果我们需要用到他，需要自己定义

## 使用 ref 引用 DOM 元素

如果想要使用 ref 引用页面上的DOM 元素，则可以按照如下的方式进行操作：

```
1 <!-- 使用 ref 属性，为对应的 DOM 添加引用名称 -->
2 <h3 ref="myh3">MyRef 组件</h3>
3 <button @click="getRef">获取 $refs 引用</button>
4
5 methods: {
6   getRef() {
7     // 通过 this.$refs.引用的名称 可以获取到 DOM 元素的引用
8     console.log(this.$refs.myh3)
9     // 操作 DOM 元素，把文本颜色改为红色
10    this.$refs.myh3.style.color = 'red'
11  },
12 }
```

## 使用 ref 引用组件实例

不光能获得html的标签，还能获得我们自定义的标签（也就是组件实例）

如果想要使用 ref 引用页面上的组件实例，则可以按照如下的方式进行操作：

```
1 <!-- 使用 ref 属性，为对应的“组件”添加引用名称 -->
2 <my-counter ref="counterRef"></my-counter>
3 <button @click="getRef">获取 $refs 引用</button>
4
5 methods: {
6   getRef() {
7     // 通过 this.$refs.引用的名称 可以引用组件的实例
8     console.log(this.$refs.counterRef)
9     // 引用到组件的实例之后，就可以调用组件上的 methods 方法
10    this.$refs.counterRef.add()
11  },
12 }
```

## 控制文本框和按钮的按需切换

下面是一个案例，要求点击按钮显示输入框并选中输入框，然后按钮消失，如果失去焦点就删除输入框，出现按钮

通过布尔值inputVisible 来控制组件中的文本框与按钮的按需切换。示例代码如下：

```
1 <template>
2   <input type="text" v-if="inputVisible">
3   <button v-else @click="showInput">展示input输入框</button>
4 </template>
```

```
1 <script>
2 export default {
3   data() {
4     return {
5       // 控制文本框和按钮的按需切换
6       inputVisible: false,
7     }
8   },
9   methods: {
10    showInput() { // 切换布尔值，显示文本框
11      this.inputVisible = true
12    },
13  },
14 }
15 </script>
```

## 让文本框自动获得焦点

当文本框展示出来之后，如果希望它立即获得焦点，则可以为其添加 ref 引用，并调用原生DOM 对象的.focus() 方法即可。示例代码如下：

但是这种写法是不正确的，因为v-if是通过修改增加dom元素节点来完成显示的，如果像下面的代码一样，还没有进行重新渲染页面就取不到this.\$refs.ipt这个对象，所以我们要延迟到渲染页面之后才能执行下面这段代码

```
1 <input type="text" v-if="inputVisible" ref="ipt">
2 <button v-else @click="showInput">展示input输入框</button>
3
4 methods: {
5   showInput() {
6     this.inputVisible = true
7     // 获取文本框的 DOM 引用，并调用 .focus() 使其自动获得焦点
8     this.$refs.ipt.focus()
9   },
10 }
```

## this.\$nextTick(cb) 方法

组件的\$nextTick(cb)方法，会把cb 回调推迟到下一个DOM 更新周期之后执行。通俗的理解是：等组件的DOM 更新完成之后，再执行cb 回调函数。从而能保证cb 回调函数可以操作到最新的DOM 元素。

```
1 <input type="text" v-if="inputVisible" ref="ipt">
2 <button v-else @click="showInput">展示input输入框</button>
3
4 methods: {
5   showInput() {
6     this.inputVisible = true
7     // 把对 input 文本框的操作，推迟到下次 DOM 更新之后。否则页面上根本不存在文本框元素
8     this.$nextTick(() => {
9       this.$refs.ipt.focus()
10    })
11  },
12 }
```

你可能会问，为啥不用生命周期里面的update()呢：

因为每一次数据的改变都会触发update()方法，如果到时候失去焦点的时候，我们就没有那个元素了，也就为null了，但是这个可以在逻辑代码里面自己判断，所以update总的来说还是可以的，但是使用的话比较复杂而已

## 动态组件

### 什么是动态组件

动态组件指的是动态切换组件的显示与隐藏。

### 如何实现动态组件渲染

vue 提供了一个内置的 `<component>` 组件，专门用来实现动态组件的渲染。示例代码如下：

```
1 data() {
2   // 1. 当前要渲染的组件名称
3   return { comName: 'Left' }
4 }
5
6 <!-- 2. 通过 is 属性，动态指定要渲染的组件 -->
7 <component :is="comName"></component>
8
9 <!-- 3. 点击按钮，动态切换组件的名称 -->
10 <button @click="comName = 'Left'">展示 Left 组件</button>
11 <button @click="comName = 'Right'">展示 Right 组件</button>
```

is属性里面填的是组件的名字，注意，这两个图片是不一样的，上面的使用了动态绑定，下面的是初始的版本

```
5     <div class="box">
6
7         <!-- 渲染 Left 组件和 Right 组件 -->
8         <component is="Left"></component>
9     </div>
10    </div>
11
12 </template>
13
14 <script>
15     import Left from '@/components/Left.vue'
16     import Right from '@/components/Right.vue'
17
18
19 <export default {>
```

## 使用 keep-alive 保持状态

默认情况下，切换动态组件时无法保持组件的状态（也就是说这里面的组件如果切换了，就会重新渲染页面（其实底层是组件的创建和销毁，这个东西是Vue帮我们实现的），包括里面的数据也会重置为原始的数据）。此时可以使用vue 内置的 `<keep-alive>` 组件保持动态组件的状态。示例代码如下：

```
1 <keep-alive>
2   <component :is="comName"></component>
3 </keep-alive>
```

keep-alive可以把内部的组件进行缓存（可以通过Vue的调试工具看到，标记为inactive了），而不是销毁！！

## keep-alive 对应的生命周期函数

当组件被缓存时，会自动触发组件的deactivated 生命周期函数。当组件被激活时，会自动触发组件的activated 生命周期函数。

```
1 export default {
2   created() { console.log('组件被创建了') },
3   destroyed() { console.log('组件被销毁了') },
4
5   activated() { console.log('Left 组件被激活了！') },
6   deactivated() { console.log('Left 组件被缓存了！') }
7 }
```

当组件第一次被创建的时候，既会执行created 生命周期，也会执行activated 生命周期

当组件被激活的时候，只会触发activated 生命周期，不再触发created。因为组件没有被重新创建

## keep-alive 的 include 属性

默认的情况下，被keep-alive标签包括的组件都会被缓存，但是，我们可以指定只缓存哪几个组件，这样就比较灵活了

include 属性用来指定：只有名称匹配的组件会被缓存。多个组件名之间使用英文的逗号分隔：

```
1 <keep-alive include="MyLeft,MyRight">
2   <component :is="comName"></component>
3 </keep-alive>
```

keep-alive标签还有一个exclude属性来标识不缓存哪个组件，用法和include一样，但是，这两个属性不能同时使用

还有一点是这个include属性里面填的是组件的名称，是在这里定义的



```
1 <template>
2   <div class="right-container">
3     <h3>Right 组件</h3>
4   </div>
5 </template>
6
7 <script>
8 export default {
9   // 当提供了 name 属性之后，组件的名称，就是 name 属性的值
10  name: 'MyRight'[red box]
11 }
12 </script>
13
```

## mixin(混入)

说简单点混入的作用就是复用代码

1. 功能：可以把多个组件共用的配置提取成一个混入对象

2. 使用方式：

第一步定义混合：

```
{  
    data(){....},  
    methods:{....}  
    ....  
}
```

第二步使用混入：

全局混入： `vue.mixin(xxx)`

局部混入： `mixins:['xxx']`

第一步：定义混合：

下面是mixin.js的代码

```
export const hunhe = {  
    methods: {  
        showName(){  
            alert(this.name)  
        }  
    },  
    mounted() {  
        console.log('你好啊！')  
    },  
}  
export const hunhe2 = {  
    data() {  
        return {  
            x:100,  
            y:200  
        }  
    },  
}
```

第二步：下面使用mixin

```
<template>  
    <div>  
        <h2 @click="showName">学校名称: {{name}}</h2>  
        <h2>学校地址: {{address}}</h2>  
    </div>  
</template>  
  
<script>  
    //引入mixin  
    import {hunhe,hunhe2} from '../mixin'  
  
    export default {  
        name:'School',  
        data() {  
            return {  
                name:'尚硅谷',  
                address:'北京',  
            }  
        }  
    }  
</script>
```

```

        x:666
    }
},
mixins:[hunhe,hunhe2],
}
</script>

```

或者使用全局引入

```

4_src_mixin混入(合) > JS main.js > render
1 //引入Vue
2 import Vue from 'vue'
3 //引入App
4 import App from './App.vue'
5 import {hunhe,hunhe2} from './mixin'
6 //关闭Vue的生产提示
7 Vue.config.productionTip = false
8
9 Vue.mixin(hunhe)
10 Vue.mixin(hunhe2)
11
12
13 //创建vm
14 new Vue({
15   el: '#app',
16   render: h => h(App)
17 })

```

或者全局使用mixin，这样的话会全局挂载所有的vm和vc

不过需要注意的是，如果在mixin文件里面服用了生命周期函数，但是你在引用的组件里面仍然编写了生命周期函数，这两个不会发生覆盖，这两个都会执行。。。但是如果是普通的函数或者属性的话，以你引用的组件里面的为准

## 自定义插件

1. 功能：用于增强Vue
2. 本质：包含install方法的一个对象，install的第一个参数是Vue，第二个以后的参数是插件使用者传递的数据。
3. 定义插件：

```

对象.install = function (vue, options) {
    // 1. 添加全局过滤器
    vue.filter(...)

    // 2. 添加全局指令
    vue.directive(...)

    // 3. 配置全局混入(合)
}

```

```

Vue.mixin(...)

// 4. 添加实例方法
Vue.prototype.$myMethod = function () {...}
Vue.prototype.$myProperty = xxxx
}

```

示例：

第一步：定义插件

plugins.js，里面的内容就是我们要实现的功能

```

export default {
  install(Vue,x,y,z){
    console.log(x,y,z)
    //全局过滤器
    Vue.filter('myslice',function(value){
      return value.slice(0,4)
    })

    //定义全局指令
    Vue.directive('fbind',{
      //指令与元素成功绑定时（一上来）
      bind(element,binding){
        element.value = binding.value
      },
      //指令所在元素被插入页面时
      inserted(element,binding){
        element.focus()
      },
      //指令所在的模板被重新解析时
      update(element,binding){
        element.value = binding.value
      }
    })

    //定义混入
    Vue.mixin({
      data(){
        return {
          x:100,
          y:200
        }
      },
    })
  }

  //给Vue原型上添加一个方法（vm和vc就都能用了）
  Vue.prototype.hello = ()=>{alert('你好啊')}
}

```

第二步：使用插件

## main.js

```
//引入Vue
import Vue from 'vue'
//引入App
import App from './App.vue'
//引入插件
import plugins from './plugins'

//应用（使用）插件，这里还可以进行传参
Vue.use(plugins,1,2,3)
//创建vm
new Vue({
  el: '#app',
  render: h => h(App)
})
```

## vue-resource

注意，这个东西用的不是很多，了解即可

这个玩意是对xhr的封装

安装 `npm i vue-resource`

用法其实和axios一模一样，只是引入的方式有点不一样

```
methods: {
  searchUsers(){
    //请求前更新List的数据
    this.$bus.$emit('updateListData',{isLoading:true,errMsg:'',users:[],isFirst:false})
    this.$http.get(`https://api.github.com/search/users?q=${this.keyWord}`).then(
      response => {
        console.log('请求成功了')
    })
  }
}
```

## 插槽

### 什么是插槽

插槽（Slot）是 vue 为组件的封装者提供的能力。允许开发者在封装组件时，把不确定的、希望由用户指定的部分定义为插槽。



可以把插槽认为是组件封装期间，为用户预留的内容的占位符。

## 体验插槽的基础用法

在封装组件时，可以通过`<slot>`元素定义插槽，从而为用户预留内容占位符。示例代码如下：

```
这个是MyCom1组件，也就是我们将要使用的组件
1 <template>
2   <p>这是 MyCom1 组件的第一个 p 标签</p>
3   <!-- 通过 slot 标签，为用户预留内容占位符（插槽） -->
4   <slot></slot>
5   <p>这是 MyCom1 组件最后一个 p 标签</p>
6 </template>
```

我们在使用这个组件的时候，可以自定义内容到插槽里面，如下代码：

```
1 <my-com-1>
2   <!-- 在使用 MyCom1 组件时，为插槽指定具体的内容 -->
3   <p>~~~用户自定义的内容~~~</p>
4 </my-com-1>
```

## 没有预留插槽的内容会被丢弃

如果在封装组件时没有预留任何`<slot>`插槽，则用户提供的任何自定义内容都会被丢弃。示例代码如下：

```
1 <template>
2   <p>这是 MyCom1 组件的第一个 p 标签</p>
3   <!-- 封装组件时吗，没有预留任何插槽 -->
4   <p>这是 MyCom1 组件最后一个 p 标签</p>
5 </template>
```

如果没有预留插槽，我们的内容会丢失，如下代码使用：

```
1 <my-com-1>
2   <!-- 自定义的内容会被丢弃 -->
3   <p>~~~用户自定义的内容~~~</p>
4 </my-com-1>
```

## 后备内容

封装组件时，可以为预留的 `<slot>` 插槽提供后备内容（默认内容）。如果组件的使用者没有为插槽提供任何内容，则后备内容会生效。示例代码如下：

```
1 <template>
2   <p>这是 MyCom1 组件的第一个 p 标签</p>
3   <slot>这是后备内容</slot> 可以在插槽标签里面定义默认显示的内容
4   <p>这是 MyCom1 组件最后一个 p 标签</p>
5 </template>
```

## 具名插槽

如果在封装组件时需要预留多个插槽节点，则需要为每个 `<slot>` 插槽指定具体的name 名称。这种带有具体名称的插槽叫做“具名插槽”。示例代码如下：

```
1 <div class="container">
2   <header>
3     <!-- 我们希望把页头放这里 -->
4     <slot name="header"></slot>
5   </header>
6   <main>
7     <!-- 我们希望把主要内容放这里 -->
8     <slot></slot>
9   </main>
10  <footer>
11    <!-- 我们希望把页脚放这里 -->
12    <slot name="footer"></slot>
13  </footer>
14 </div>
```

注意：没有指定 name 名称的插槽，会有隐含的名称叫做 “default”。

## 为具名插槽提供内容

在向具名插槽提供内容的时候，我们可以在一个 `<template>` 元素上使用 `v-slot` 指令（但是注意： `v-slot` 指令只能用在 `template` 标签上或者是自定义的组件上！！！），并以 `v-slot` 的参数的形式提供其名称。示例代码如下：

```
1 <my-com-2>
2   <template v-slot:header>
3     <h1>滕王阁序</h1>
4   </template>
5
6   <template v-slot:default>
7     <p>豫章故郡，洪都新府。</p>
8     <p>星分翼轸，地接衡庐。</p>
9     <p>襟三江而带五湖，控蛮荆而引瓯越。</p>
10  </template>
11
12  <template v-slot:footer>
13    <p>落款：王勃</p>
14  </template>
15 </my-com-2>
```

`template`这个标签，它是一个虚拟的标签，不会渲染到页面上，只起到标识渲染到的位置的作用

## 具名插槽的简写形式

跟 `v-on` 和 `v-bind` 一样，`v-slot` 也有缩写，即把参数之前的所有内容(`v-slot:`) 替换为字符#。

例如`v-slot:header`可以被重写为`#header`，所以上面的代码可以改成：

```
1 <my-com-2>
2   <template #header>
3     <h1>滕王阁序</h1>
4   </template>
5
6   <template #default>
7     <p>豫章故郡，洪都新府。</p>
8     <p>星分翼轸，地接衡庐。</p>
9     <p>襟三江而带五湖，控蛮荆而引瓯越。</p>
10  </template>
11
12  <template #footer>
13    <p>落款：王勃</p>
14  </template>
15 </my-com-2>
```

## 作用域插槽

在封装组件的过程中，可以为预留的 `<slot>` 插槽绑定 `props` 数据，这种带有 `props` 数据的 `<slot>` 叫做“作用域插槽”。示例代码如下：

The diagram illustrates the binding of a slot prop. On the left, a parent component's template shows a slot named #content="obj" with a red box around it. An arrow points from this slot to a child component's template on the right. In the child component, a slot named #content with a red box around it has its msg prop set to "hello vue.js", also with a red box around it. This visualizes how the parent's slot prop is passed down to the child's slot.

```
<template #content="obj">
  <div>
    <p>啊, 大海, 全是水。</p>
    <p>啊, 蜈蚣, 全是腿。</p>
    <p>啊, 辣椒, 静辣嘴。</p>
    <p>{{ obj.msg }}</p>
  </div>
</template>

<template #author>
  <!-- 文章的内容 -->
  <div class="content-box">
    <!-- 在封装组件时, 为预留的 <slot> 提供属性对应的值, 这种方法, 叫做 “作用域插槽” -->
    <slot name="content" msg="hello vue.js"></slot>
  </div>
</template>
```

A screenshot of a code editor showing a `<tbody>` component. It contains a comment indicating the slot is an "作用域插槽". Below it, a `<slot v-for="item in list" :user="item"></slot>` directive is shown, also with a red box around it. This demonstrates how the parent's slot prop is used within the child component's template.

```
1 <tbody>
2   <!-- 下面的 slot 是一个作用域插槽 -->
3   <slot v-for="item in list" :user="item"></slot>
4 </tbody>
```

## 使用作用域插槽

可以使用 `v-slot:` 的形式，**接收**作用域插槽对外提供的数据。示例代码如下：

A screenshot of a code editor showing a `<my-com-3>` component. It includes a template section with a `<template v-slot:default="scope">` directive. Inside this template, there's a table row (`<tr>`) and a table cell (`<td>{{ scope }}</td>`). This demonstrates how the parent's slot prop is received and used within the child component's template.

```
1 <my-com-3>
2   <!-- 1. 接收作用域插槽对外提供的数据 -->
3   <template v-slot:default="scope">
4     <tr>
5       <!-- 2. 使用作用域插槽的数据 -->
6       <td>{{ scope }}</td>
7     </tr>
8   </template>
9 </my-com-3>
```

## 解构插槽

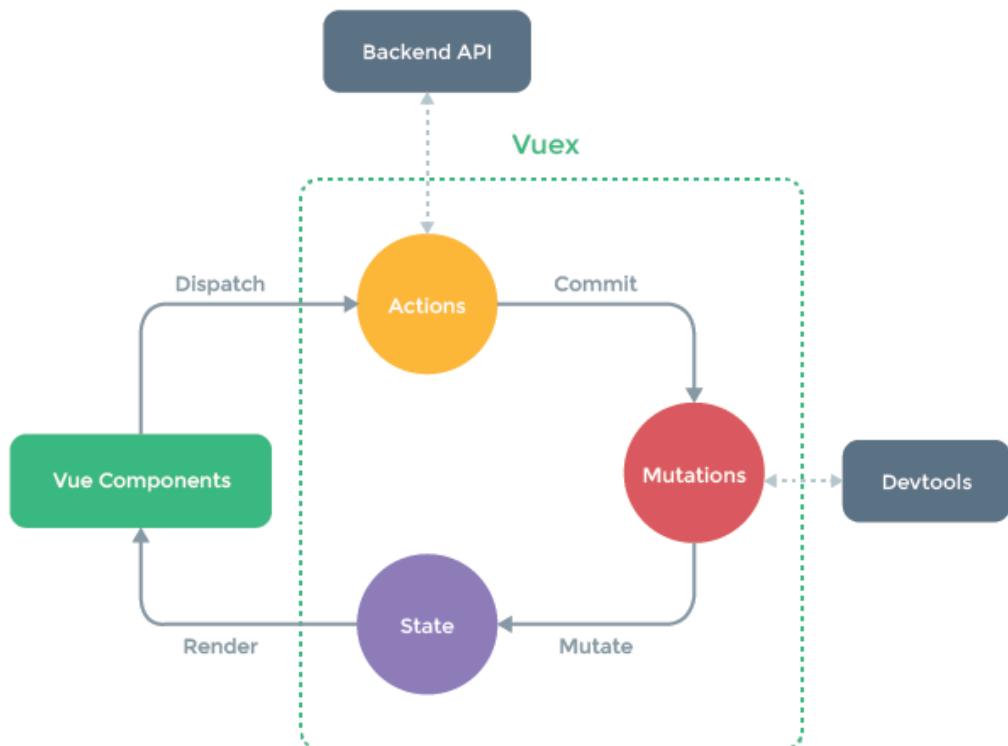
Prop作用域插槽对外提供的数据对象，可以使用解构赋值简化数据的接收过程。示例代码如下：

```
1  <my-com-3>
2    <!-- v-slot: 可以简写成 # -->
3    <!-- 作用域插槽对外提供的数据对象，可以通过“解构赋值”简化接收的过程 -->
4    <template #default="{user}">
5      <tr>
6        <td>{{user.id}}</td>
7        <td>{{user.name}}</td>
8        <td>{{user.state}}</td>
9      </tr>
10     </template>
11   </my-com-3>
```

## Vuex

涉及到多组件共享数据（既读又写）的话，全局事件总线就显得有点麻烦了

所以我们需要Vuex了



## 1.概念

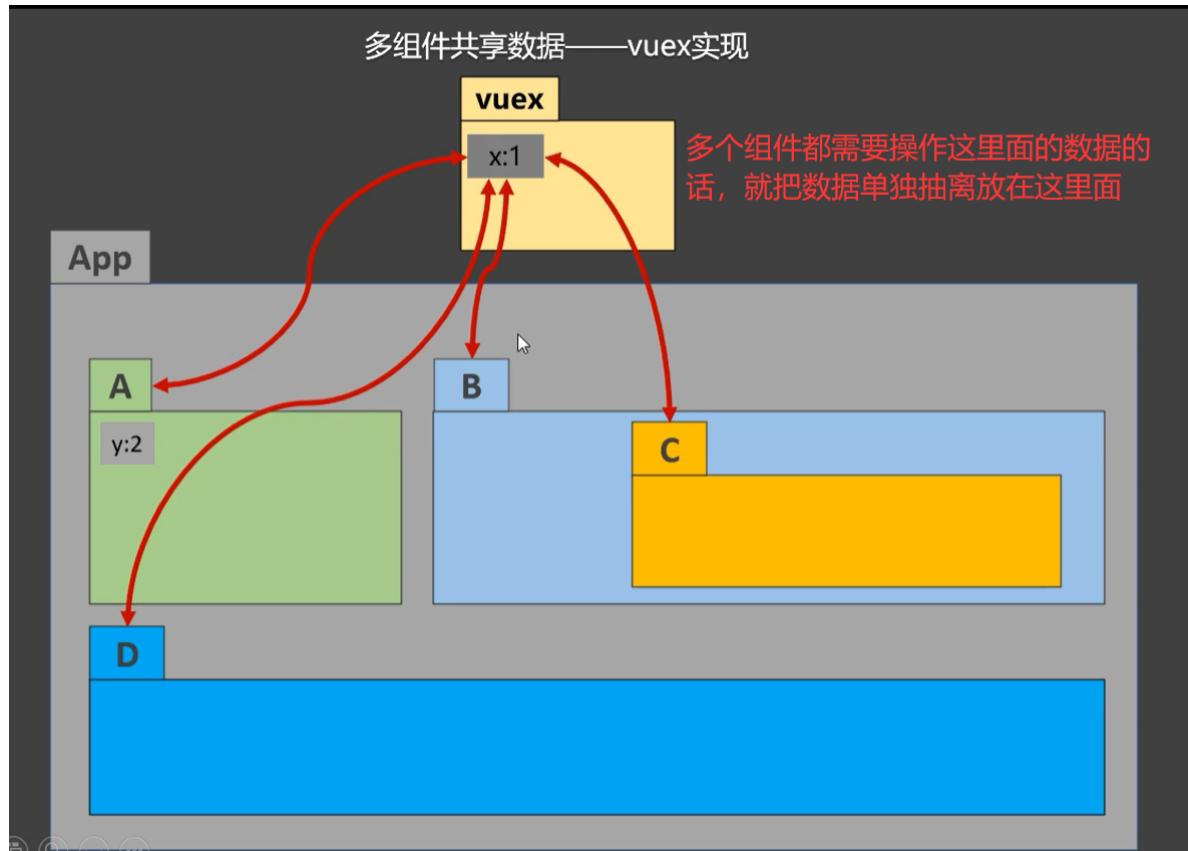
1. 专门在 Vue 中实现集中式状态（数据）管理的一个 Vue 插件，对 vue 应用中多个组件的共享状态进行集中式的管理（读/写），也是一种组件间通信的方式，且适用于任意组件间通信。
2. Github 地址: <https://github.com/vuejs/vuex>

这里需要注意的是

- vue2中，要用vuex的3版本
- vue3中，要用vuex的4版本

## 2.什么时候使用 Vuex

1. 多个组件依赖于同一状态
2. 来自不同组件的行为需要变更同一状态



## 3.搭建vuex环境

1. 创建文件: `src/store/index.js`

```
//引入Vue核心库
import Vue from 'vue'
//引入Vuex
import Vuex from 'vuex'
//应用Vuex插件
Vue.use(Vuex)

//准备actions对象--响应组件中用户的动作
const actions = []
//准备mutations对象--修改state中的数据
const mutations = []
//准备state对象--保存具体的数据
const state = []

//创建并暴露store
export default new Vuex.Store({
  //这里面是配置对象的内容
  actions,
  mutations,
  state
})
```

```
    mutations,  
    state  
)
```

2. 在 `main.js` 中创建 `vm` 时传入 `store` 配置项

```
.....  
//引入store  
import store from './store'  
.....  
  
//创建vm  
new Vue({  
  el:'#app',  
  render: h => h(App),  
  store  
)
```

## 4. 基本使用

1. 初始化数据、配置 `actions`、配置 `mutations`，操作文件 `store.js`

```
//引入Vue核心库  
import Vue from 'vue'  
//引入Vuex  
import Vuex from 'vuex'  
//引用vuex  
Vue.use(Vuex)  
  
const actions = {  
  //响应组件中加的动作  
  //这里面有两个参数，一个叫context，是一个ministore  
  jia(context,value){  
    //注意这里是大写！！  
    context.commit('JIA',value)  
  },  
}  
  
const mutations = {  
  //执行加  
  JIA(state,value){  
    // console.log('mutations中的JIA被调用了',state,value)  
    state.sum += value  
  }  
}  
  
//初始化数据  
const state = {  
  sum:0  
}  
  
//创建并暴露store  
export default new Vuex.Store({  
  actions,
```

```
    mutations,  
    state,  
})
```

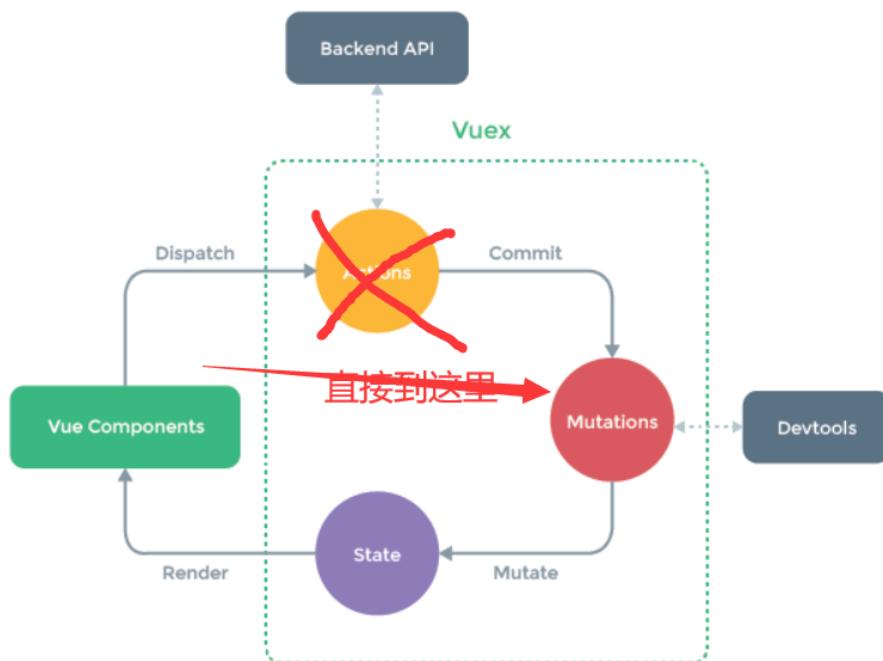
2. 组件中读取vuex中的数据: `$store.state.sum`

components > v-count.vue > template > div

```
<template>  
<div>  
  <h1>当前求和为: {{ $store.state.sum }}</h1>  
  <select v-model.number="n">  
    <option value="1">1</option>  
    <option value="2">2</option>
```

3. 组件中修改vuex中的数据: `$store.dispatch('action中的方法名', 数据)` 或 `$store.commit('mutations中的方法名', 数据)`

备注: 若没有网络请求或其他业务逻辑, 组件中也可以越过actions, 即不写`dispatch`, 直接编写`commit`



比如下面的代码:

store/index.js

```
// 该文件用于创建Vuex中最为核心的store  
import Vue from 'vue'  
// 引入vuex  
import Vuex from 'vuex'  
// 应用vuex插件  
Vue.use(Vuex)  
  
// 准备actions—用于响应组件中的动作  
const actions = {  
  /* jia(context,value){  
    console.log('actions中的jia被调用了')  
    context.commit('JIA',value)  
  } */}
```

```

    },
    jian(context,value){
        console.log('actions中的jian被调用了')
        context.commit('JIAN',value)
    },
    /* 
    jiaOdd(context,value){
        console.log('actions中的jiaOdd被调用了')
        if(context.state.sum % 2){
            context.commit('JIA',value)
        }
    },
    jiawait(context,value){
        console.log('actions中的jiawait被调用了')
        setTimeout(()=>{
            context.commit('JIA',value)
        },500)
    }
}
//准备mutations--用于操作数据 (state)
const mutations = {
    JIA(state,value){
        console.log('mutations中的JIA被调用了')
        state.sum += value
    },
    JIAN(state,value){
        console.log('mutations中的JIAN被调用了')
        state.sum -= value
    }
}
//准备state--用于存储数据
const state = {
    sum:0 //当前的和
}

//创建并暴露store
export default new Vuex.Store({
    actions,
    mutations,
    state,
})

```

count.vue

```

<template>
<div>
    <h1>当前求和为: {{$store.state.sum}}</h1>
    <select v-model.number="n">
        <option value="1">1</option>
        <option value="2">2</option>
        <option value="3">3</option>
    </select>
    <button @click="increment">+</button>
    <button @click="decrement">-</button>
    <button @click="incrementodd">当前求和为奇数再加</button>
    <button @click="incrementwait">等一等再加</button>

```

```

        </div>
    </template>

    <script>
        export default {
            name:'Count',
            data() {
                return {
                    n:1, //用户选择的数字
                }
            },
            methods: {
                //这里直接跳过了服务员，直接到了后厨
                increment(){
                    this.$store.commit('JIA',this.n)
                },
                decrement(){
                    this.$store.commit('JIAN',this.n)
                },
                incrementOdd(){
                    this.$store.dispatch('jiaodd',this.n)
                },
                incrementWait(){
                    this.$store.dispatch('jiawait',this.n)
                },
            },
            mounted() {
                console.log('Count',this)
            },
        }
    </script>

    <style lang="css">
        button{
            margin-left: 5px;
        }
    </style>

```

## 5.getters的使用

这个东西和计算属性的区别，就是计算属性只能在自己的单个组件里面使用，而getters配置的计算每一个组件都能使用，可以理解为就是vuex里面的计算属性

1. 概念：当state中的数据需要经过加工后再使用时，可以使用getters加工。
2. 在 `store/index.js` 中追加 `getters` 配置

```

.....
const getters = {
  bigSum(state){
    return state.sum * 10
  }
}

//创建并暴露store
export default new Vuex.Store({
  .....
  getters
})

```

3. 组件中读取数据: `$store.getters.bigSum`

## 6. 四个map方法的使用

这个东西只是为了简化我们写变量的功夫

1. **mapState方法**: 用于帮助我们映射 `state` 中的数据为计算属性

```

computed: {
  //借助mapState生成计算属性: sum、school、subject (对象写法)
  ...mapState({sum:'sum',school:'school',subject:'subject'}),

  //借助mapState生成计算属性: sum、school、subject (数组写法)
  ...mapState(['sum','school','subject']),
},

```

2. **mapGetters方法**: 用于帮助我们映射 `getters` 中的数据为计算属性

```

computed: {
  //借助mapGetters生成计算属性: bigSum (对象写法)
  ...mapGetters({bigSum:'bigSum'}),

  //借助mapGetters生成计算属性: bigSum (数组写法)
  ...mapGetters(['bigSum'])
},

```

```

<template>
<div>
  <h1>当前求和为: {{sum}}</h1>
  <h3>当前求和放大10倍为: {{bigSum}}</h3>
  <h3>我在{{school}}, 学习{{subject}}</h3>
  <select v-model.number="n">
    <option value="1">1</option>
    <option value="2">2</option>
    <option value="3">3</option>
  </select>
  <button @click="increment">+</button>
  <button @click="decrement">-</button>
  <button @click="incrementOdd">当前求和为奇数再加</button>
</div>

```

```
<button @click="incrementwait">等一等再加</button>
</div>
</template>

<script>
    import {mapState, mapGetters} from 'vuex'
    export default {
        name:'Count',
        data() {
            return {
                n:1, //用户选择的数字
            }
        },
        computed:{ 
            //靠程序员自己亲自去写计算属性
            /* sum(){
                return this.$store.state.sum
            },
            school(){
                return this.$store.state.school
            },
            subject(){
                return this.$store.state.subject
            }, */ 

            //借助mapstate生成计算属性，从state中读取数据。（对象写法）
            // ...mapState({he:'sum',xuexiao:'school',xueke:'subject'}), 

            //借助mapstate生成计算属性，从state中读取数据。（数组写法）
            ...mapState(['sum','school','subject']), 

            /*
            ****
            */

            /* bigSum(){
                return this.$store.getters.bigSum
            }, */ 

            //借助mapGetters生成计算属性，从getters中读取数据。（对象写法）
            // ...mapGetters({bigSum:'bigSum'}) 

            //借助mapGetters生成计算属性，从getters中读取数据。（数组写法）
            ...mapGetters(['bigSum'])

        },
        methods: {
            increment(){
                this.$store.commit('JIA',this.n)
            },
            decrement(){
                this.$store.commit('JIAN',this.n)
            },
            incrementOdd(){
                this.$store.dispatch('jiaodd',this.n)
            },
        }
    }

```

```

        incrementWait(){
            this.$store.dispatch('jiawait',this.n)
        },
    },
    mounted() {
        const x = mapState({he:'sum',xuexiao:'school',xueke:'subject'})
        console.log(x)
    },
}
</script>

<style lang="css">
button{
    margin-left: 5px;
}
</style>

```

1. **mapActions方法**: 用于帮助我们生成与 actions 对话的方法, 即: 包含  
\$store.dispatch(xxx) 的函数

```

methods:{
    //靠mapActions生成: incrementOdd、incrementwait (对象形式)
    ...mapActions({incrementOdd:'jiaOdd',incrementwait:'jiawait'})

    //靠mapActions生成: incrementOdd、incrementwait (数组形式)
    ...mapActions(['jiaOdd','jiawait'])
}

```

2. **mapMutations方法**: 用于帮助我们生成与 mutations 对话的方法, 即: 包含  
\$store.commit(xxx) 的函数

```

methods:{
    //靠mapActions生成: increment、decrement (对象形式)
    ...mapMutations({increment:'JIA',decrement:'JIAN'}),

    //靠mapMutations生成: JIA、JIAN (数组形式)
    ...mapMutations(['JIA','JIAN']),
}

```

备注: mapActions与mapMutations使用时, 若需要传递参数需要: 在模板中绑定事件时传递好参数, 否则参数是事件对象。

具体使用的时候需要先引入 import {mapState} from 'vuex'

```

<template>
<div>
    <h1>当前求和为: {{sum}}</h1>
    <h3>当前求和放大10倍为: {{bigSum}}</h3>

```

```

<h3>我在{{school}}, 学习{{subject}}</h3>
<select v-model.number="n">
    <option value="1">1</option>
    <option value="2">2</option>
    <option value="3">3</option>
</select>
<button @click="increment(n)">+</button>
<button @click="decrement(n)">-</button>
<button @click="incrementOdd(n)">当前求和为奇数再加</button>
<button @click="incrementWait(n)">等一等再加</button>
</div>
</template>

<script>
    import {mapState, mapGetters, mapMutations, mapActions} from 'vuex'
    export default {
        name: 'Count',
        data() {
            return {
                n: 1, // 用户选择的数字
            }
        },
        computed: {
            // 借助mapState生成计算属性，从state中读取数据。（对象写法）
            // ...mapState({he: 'sum', xuexiao: 'school', xueke: 'subject'}),

            // 借助mapState生成计算属性，从state中读取数据。（数组写法）
            ...mapState(['sum', 'school', 'subject']),
        },
        /*
        *****
        */
        // 借助mapGetters生成计算属性，从getters中读取数据。（对象写法）
        // ...mapGetters({bigSum: 'bigSum'})

        // 借助mapGetters生成计算属性，从getters中读取数据。（数组写法）
        ...mapGetters(['bigSum'])

    },
    methods: {
        // 程序员亲自写方法
        /* increment(){
            this.$store.commit('JIA', this.n)
        },
        decrement(){
            this.$store.commit('JIAN', this.n)
        }, */

        // 借助mapMutations生成对应的方法，方法中会调用commit去联系mutations(对象写法)
        ...mapMutations({increment: 'JIA', decrement: 'JIAN'}),

        // 借助mapMutations生成对应的方法，方法中会调用commit去联系mutations(数组写法)
        // ...mapMutations(['JIA', 'JIAN']),
    }
</script>

```

```

/* **** */
//程序员亲自写方法
/* incrementOdd(){
    this.$store.dispatch('jiaOdd',this.n)
},
incrementWait(){
    this.$store.dispatch('jiawait',this.n)
}, */

//借助mapActions生成对应的方法，方法中会调用dispatch去联系actions(对象写法)
...mapActions({incrementOdd:'jiaOdd',incrementWait:'jiawait'})

//借助mapActions生成对应的方法，方法中会调用dispatch去联系actions(数组写法)
// ...mapActions(['jiaOdd','jiawait'])

},
mounted() {
    const x = mapState({he:'sum',xuexiao:'school',xueke:'subject'})
    console.log(x)
},
}

```

</script>

<style lang="css">

```

button{
    margin-left: 5px;
}

```

</style>

## 7.模块化+命名空间

1. 目的：让代码更好维护，让多种数据分类更加明确。

2. 修改 `store.js`

```

const countAbout = {
    namespaced:true,//开启命名空间
    state:{x:1},
    mutations: { ... },
    actions: { ... },
    getters: {
        bigSum(state){
            return state.sum * 10
        }
    }
}

const personAbout = {
    namespaced:true,//开启命名空间
    state:{ ... },
    mutations: { ... },

```

```
actions: { ... }

}

const store = new Vuex.Store({
  modules: {
    countAbout,
    personAbout
  }
})
```

3. 开启命名空间后，组件中读取state数据：

```
//方式一：自己直接读取
this.$store.state.personAbout.list
//方式二：借助mapState读取：
...mapState('countAbout', ['sum', 'school', 'subject']),
```

4. 开启命名空间后，组件中读取getters数据：

```
//方式一：自己直接读取
this.$store.getters['personAbout/firstPersonName']
//方式二：借助mapGetters读取：
...mapGetters('countAbout', ['bigSum'])
```

5. 开启命名空间后，组件中调用dispatch

```
//方式一：自己直接dispatch
this.$store.dispatch('personAbout/addPersonwang', person)
//方式二：借助mapActions：
...mapActions('countAbout', {incrementOdd: 'jiaOdd', incrementWait: 'jiawait'})
```

6. 开启命名空间后，组件中调用commit

```
//方式一：自己直接commit
this.$store.commit('personAbout/ADD_PERSON', person)
//方式二：借助mapMutations：
...mapMutations('countAbout', {increment: 'JIA', decrement: 'JIAN'}),
```

## 自定义指令

### 什么是自定义指令

vue 官方提供了v-text、v-for、v-model、v-if 等常用的指令。除此之外vue 还允许开发者自定义指令。

### 自定义指令的分类

vue 中的自定义指令分为两类，分别是：

1. 私有自定义指令
2. 全局自定义指令

## 私有自定义指令

在每个vue 组件中，可以在directives 节点（这个节点和data, methods节点都是平级的）下声明私有自定义指令。示例代码如下：

```
1 directives: {
2   color: {
3     // 为绑定到的 HTML 元素设置红色的文字
4     bind(el) {
5       // 形参中的 el 是绑定了此指令的、原生的 DOM 对象
6       el.style.color = 'red'
7     }
8   }
9 }
```

这个bind函数的触发时机就是当使用这个自定义指令的时候会执行

## 使用自定义指令

在使用自定义指令时，需要加上v- 前缀。示例代码如下：

```
1 <!-- 声明自定义指令时，指令的名字是 color -->
2 <!-- 使用自定义指令时，需要加上 v- 指令前缀 -->
3 <h1 v-color>App 组件</h1>
```

## 为自定义指令动态绑定参数值

在 template 结构中使用自定义指令时，可以通过等号 (=) 的方式，为当前指令动态绑定参数值：

```
1 data() {
2   return {
3     color: 'red' // 定义 color 颜色值
4   }
5 }
6
7 <!-- 在使用指令时，动态为当前指令绑定参数值 color -->
8 <h1 v-color="
```

## 通过 binding 获取指令的参数值

在声明自定义指令时，可以通过形参中的第二个参数，来接收指令的参数值：

```
1 directives: {
2   color: {
3     bind(el, binding) {
4       // 通过 binding 对象的 .value 属性，获取动态的参数值
5       el.style.color = binding.value
6     }
7   }
8 }
```

## update 函数

之前我们不是把颜色的值放在了data里面嘛，如果我们想改这个值，那我们可以正常改，但是发现颜色并不会改变，因为bind函数里面的逻辑只在初始化的时候执行

bind 函数只在初始化的时候调用 1 次：当指令第一次绑定到元素时调用，当 DOM 更新时bind 函数不会被触发。 update 函数（这个update函数跟生命周期那个函数不一样）会在每次 DOM 更新时被调用。示例代码如下：

```
1 directives: {
2   color: {
3     // 当指令第一次被绑定到元素时被调用
4     bind(el, binding) {
5       el.style.color = binding.value
6     },
7     // 每次 DOM 更新时被调用
8     update(el, binding) {
9       el.style.color = binding.value
10    }
11  }
12 }
```

## 函数简写

如果 bind和update 函数中的逻辑完全相同，则对象格式的自定义指令可以简写成函数格式：

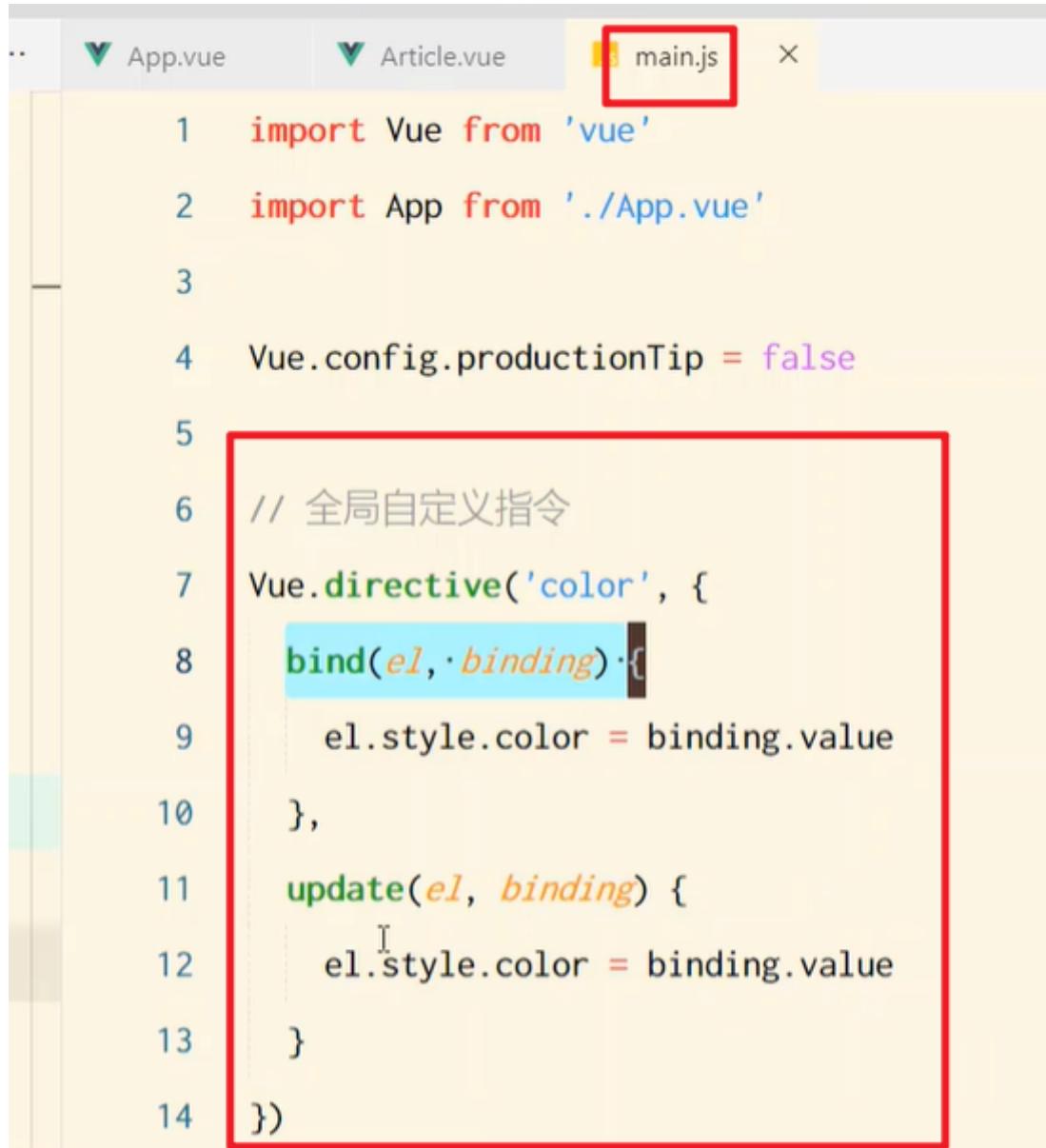
```
1 directives: {
2   // 在 insert 和 update 时，会触发相同的业务逻辑
3   color(el, binding) {
4     el.style.color = binding.value
5   }
6 }
```

## 全局自定义指令

全局共享的自定义指令需要通过“Vue.directive()”进行声明，示例代码如下：

```
1 // 参数1：字符串，表示全局自定义指令的名字  
2 // 参数2：对象，用来接收指令的参数值  
3 Vue.directive('color', function(el, binding) {  
4   el.style.color = binding.value  
5 })
```

上面是简化写法，这个是完整写法



```
..  App.vue Article.vue main.js ×  
1 import Vue from 'vue'  
2 import App from './App.vue'  
3  
4 Vue.config.productionTip = false  
5  
6 // 全局自定义指令  
7 Vue.directive('color', {  
8   bind(el, binding){  
9     el.style.color = binding.value  
10  },  
11  update(el, binding) {  
12    el.style.color = binding.value  
13  }  
14})
```

这个和定义全局过滤器差不多

axios补充，还没弄完

The screenshot shows a code editor interface with multiple tabs at the top: App.vue, Left.vue, Right.vue, and main.js. The main.js tab is highlighted with a red box. The code in main.js is as follows:

```
1 import Vue from 'vue'
2 import App from './App.vue'
3 import axios from 'axios' import axios from 'axios'
4
5 Vue.config.productionTip = false
6
7 Vue.prototype.axios = axios
8
9 new Vue({
10   render: h => h(App)
11 }).$mount('#app')
12
13 export default {
14   methods: {
15     async getInfo() {
16       const { data: res } = await this.axios.get('http://www.liulongbin.top:3006/api/get')
17       console.log(res)
18     }
19   }
20 }
```

The code editor has syntax highlighting and some parts are annotated with red boxes: the import statement for axios in line 3, the assignment of axios to Vue.prototype.axios in line 7, and the entire file body from line 1 to line 20.

The screenshot shows a Visual Studio window with several files open in tabs at the top: App.vue M, Left.vue U, Right.vue U, and main.js M. The main.js file is the active tab, indicated by a red box around its tab and the code area. The code in main.js is as follows:

```
1 import Vue from 'vue'
2 import App from './App.vue'
3 import axios from 'axios'
4
5 Vue.config.productionTip = false
6
7 // 全局配置 axios 的请求根路
8 axios.defaults.baseURL = 'http://www.liulongbin.top:3006'
9 // 把 axios 挂载到 Vue.prototype 上，供每个 .vue 组件的实例直接使用
10 Vue.prototype.$http = axios
11
12 // 今后，在每个 .vue 组件中要发起请求，直接调用 this.$http.xxx
```

Annotations: A red box highlights the configuration code from line 7 to line 10, which sets the baseURL for axios and attaches it to the Vue prototype. Another red box highlights the line 'Vue.prototype.\$http = axios'.

## 路由

vue-router4只能用在vue3中

vue-router3只能用在vue2中

## 前端路由的概念与原理

路由（英文：router）就是页面hash与组件的对应关系。

## SPA 与前端路由

SPA 指的是一个web 网站只有唯一的一个HTML 页面，所有组件的展示与切换都在这唯一的一个页面内完成。此时，不同组件之间的切换需要通过前端路由来实现。

结论：在 SPA 项目中，不同功能之间的切换，要依赖于前端路由来完成！

## 锚链接实现

```
锚链接:  
<!DOCTYPE html>  
<html lang="en">  
  
<head>  
  <meta charset="UTF-8">  
  <meta http-equiv="X-UA-Compatible" content="IE=edge">  
  <meta name="viewport" content="width=device-width, initial-scale=1.0">  
  <title>Document</title>  
  <style>  
    .box {  
      height: 800px;  
    }  
  
    #b1 {  
      background-color: pink;  
    }  
  
    #b2 {  
      background-color: red;  
    }  
  
    #b3 {  
      background-color: orange;  
    }  
  
    #b4 {  
      background-color: skyblue;  
    }  
  
    .side-bar {  
      position: fixed;  
      top: 0;  
      right: 0;  
      background: white;  
    }  
  </style>  
</head>  
  
<body>  
  <div class="side-bar">  
    <a href="#b1">b1</a>  
    <a href="#b2">b2</a>  
    <a href="#b3">b3</a>  
    <a href="#b4">b4</a>  
  </div>  
  
  <div class="box" id="b1"></div>  
  <div class="box" id="b2"></div>  
  <div class="box" id="b3"></div>  
  <div class="box" id="b4"></div>  
</body>
```

```
</html>
```

## 前端路由的工作方式

- ① 用户点击了页面上的路由链接
- ② 导致了 URL 地址栏中的Hash 值发生了变化
- ③ 前端路由监听到了 Hash 地址的变化
- ④ 前端路由把当前 Hash 地址对应的组件渲染到浏览器中



结论：前端路由，指的是Hash 地址与组件之间的对应关系！

## 实现简易的前端路由

步骤1：通过 `<component :is="comName"></component>` 标签，结合 `comName` 动态渲染组件。示例代码如下：

```
1 <!-- 通过 is 属性，指定要展示的组件的名称 -->
2 <component :is="comName"></component>
3
4 export default {
5   name: 'App',
6   data() {
7     return {
8       // 要展示的组件的名称
9       comName: 'Home'
10    }
11  }
12 }
```

步骤2：在App.vue 组件中，为 `<a>` 链接添加对应的hash 值：

```
1 <a href="#/home">Home</a>&nbsp;;
2 <a href="#/movie">Movie</a>&nbsp;;
3 <a href="#/about">About</a>
```

步骤3：在 `created` 生命周期函数中，监听浏览器地址栏中hash 地址的变化，动态切换要展示的组件的名称：

```
1 created() {
2   window.onhashchange = () => {
3     switch (location.hash) {
4       case '#/home': // 点击了“首页”的链接
5         this.comName = 'Home'
6         break
7       case '#/movie': // 点击了“电影”的链接
8         this.comName = 'Movie'
9         break
10      case '#/about': // 点击了“关于”的链接
11        this.comName = 'About'
12        break
13    }
14  }
15 }
```

代码实现：

```
<template>
<div class="app-container">
  <h1>App 根组件</h1>

  <a href="#/home">首页</a>
  <a href="#/movie">电影</a>
  <a href="#/about">关于</a>
  <hr />

  <component :is="comName"></component>
</div>
</template>

<script>
// 导入组件
import Home from "@/components/Home.vue";
import Movie from "@/components/Movie.vue";
import About from "@/components/About.vue";

export default {
  name: "App",
  data() {
    return {
      // 在动态组件的位置，要展示的组件的名字，值必须是字符串
      comName: "Home",
    };
  },
  created() {
    // 只要当前的 App 组件一被创建，就立即监听 window 对象的 onhashchange 事件
    window.onhashchange = () => {
      console.log("监听到了 hash 地址的变化", location.hash);
      switch (location.hash) {
        case "#/home":
```

```
        this.comName = "Home";
        break;
    case "#/movie":
        this.comName = "Movie";
        break;
    case "#/about":
        this.comName = "About";
        break;
    }
};

},
// 注册组件
components: {
    Home,
    Movie,
    About,
},
};

</script>

<style lang="less" scoped>
.app-container {
    background-color: #eefefef;
    overflow: hidden;
    margin: 10px;
    padding: 15px;
    > a {
        margin-right: 10px;
    }
}
</style>
```

## vue-router 的基本用法

### 在项目中安装 vue-router

在 vue2 的项目中，安装vue-router 的命令如下：



```
1 npm i vue-router@3.5.2 -S
```

### 创建路由模块

在 src 源代码目录下，新建router/index.js 路由模块，并初始化如下的代码：

```
1 // 1. 导入 Vue 和 VueRouter 的包
2 import Vue from 'vue'
3 import VueRouter from 'vue-router'
4
5 // 2. 调用 Vue.use() 函数，把 VueRouter 安装为 Vue 的插件
6 Vue.use(VueRouter)
7
8 // 3. 创建路由的实例对象
9 const router = new VueRouter()
10
11 // 4. 向外共享路由的实例对象
12 export default router
```

## 导入并挂载路由模块

在 src/main.js 入口文件中，导入并挂载路由模块。示例代码如下：

```
1 import Vue from 'vue'
2 import App from './App.vue'
3 // 1. 导入路由模块
4 import router from '@/router'
5
6 new Vue({
7   render: h => h(App),
8   // 2. 挂载路由模块
9   router: router
10 }).$mount('#app')
```

## 声明路由链接和占位符

在 src/App.vue 组件中，使用 vue-router 提供的 `<router-link>` 和 `<router-view>` 声明路由链接和占位符：

```
1 <template>
2   <div class="app-container">
3     <h1>App 组件</h1>
4     <!-- 1. 定义路由链接 -->
5     <router-link to="/home">首页</router-link>
6     <router-link to="/movie">电影</router-link>
7     <router-link to="/about">关于</router-link>
8
9     <hr />
10
11    <!-- 2. 定义路由的占位符 -->
12    <router-view></router-view>
13  </div>
14 </template>
```

## 声明路由的匹配规则

在 src/router/index.js 路由模块中，通过routes 数组声明路由的匹配规则。示例代码如下：

```
1 // 导入需要使用路由切换展示的组件
2 import Home from '@/components/Home.vue'
3 import Movie from '@/components/Movie.vue'
4 import About from '@/components/About.vue'
5
6 // 2. 创建路由的实例对象
7 const router = new VueRouter({
8   routes: [ // 在 routes 数组中，声明路由的匹配规则
9     // path 表示要匹配的 hash 地址； component 表示要展示的路由组件
10    { path: '/home', component: Home },
11    { path: '/movie', component: Movie },
12    { path: '/about', component: About }
13  ],
14 })
```

## vue-router 的常见用法

### 路由重定向

路由重定向指的是：用户在访问地址 A 的时候，强制用户跳转到地址C，从而展示特定的组件页面。通过路由规则的redirect 属性，指定一个新的路由地址，可以很方便地设置路由的重定向：

```
1 const router = new VueRouter({
2   // 在 routes 数组中，声明路由的匹配规则
3   routes: [
4     // 当用户访问 / 的时候，通过 redirect 属性跳转到 /home 对应的路由规则
5     { path: '/', redirect: '/home' },
6     { path: '/home', component: Home },
7     { path: '/movie', component: Movie },
8     { path: '/about', component: About }
9   ]
10 })
```

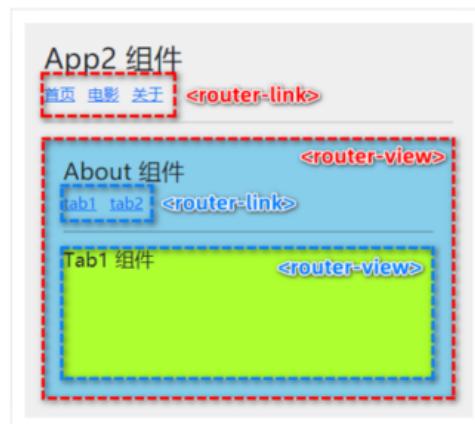
## 嵌套路由

通过路由实现组件的嵌套展示，叫做嵌套路由。

简单来说就是路由里面又有路由（套娃）



点击父级路由链接显示模板内容



① 模板内容中又有子级路由链接

② 点击子级路由链接显示子级模板内容

## 声明子路由链接和子路由占位符

在 About.vue 组件中，声明 tab1 和 tab2 的子路由链接以及子路由占位符。示例代码如下：

```
1 <template>
2   <div class="about-container">
3     <h3>About 组件</h3>
4     <!-- 1. 在关于页面中，声明两个子路由链接 -->
5     <router-link to="/about/tab1">tab1</router-link>
6     <router-link to="/about/tab2">tab2</router-link>
7
8     <hr />
9
10    <!-- 2. 在关于页面中，声明子路由的占位符 -->
11    <router-view></router-view>
12  </div>
13 </template>
```

### 通过 children 属性声明子路由规则

在 src/router/index.js 路由模块中，导入需要的组件，并使用children 属性声明子路由规则：

```
1 import Tab1 from '@/components/tabs/Tab1.vue'
2 import Tab2 from '@/components/tabs/Tab2.vue'
3
4 const router = new VueRouter({
5   routes: [
6     { // about 页面的路由规则（父级路由规则）
7       path: '/about',
8       component: About,
9       children: [ // 1. 通过 children 属性，嵌套声明子级路由规则
10         { path: 'tab1', component: Tab1 }, // 2. 访问 /about/tab1 时，展示 Tab1 组件
11         { path: 'tab2', component: Tab2 } // 2. 访问 /about/tab2 时，展示 Tab2 组件
12       ]
13     }
14   ]
15 })
```

配置子路由的默认跳转页面：

```

  path: '/movie', component: Movie,
}

{
  path: '/about',
  component: About,
  // redirect: '/about/tab1',          这两种方法都可以默认跳转到tab1页面
  children: [
    // 子路由规则
    // 默认子路由: 如果 children 数组中, 某个路由规则的 path 值为空字符串, 则这条路由规则叫做“默认子路由”
    { path: '', component: Tab1 },
    { path: 'tab2', component: Tab2 }
  ]
}

```

## 动态路由匹配

思考：有如下 3 个路由链接：

```

1 <router-link to="/movie/1">电影1</router-link>
2 <router-link to="/movie/2">电影2</router-link>
3 <router-link to="/movie/3">电影3</router-link>

```

定义如下 3 个路由规则，是否可行？

- 是可行的，就是复用性太差了，所以我们需要动态路由

```

1 { path: '/movie/1', component: Movie }
2 { path: '/movie/2', component: Movie }
3 { path: '/movie/3', component: Movie }

```

## 动态路由的概念

动态路由指的是：把 Hash 地址中可变的部分定义为参数项，从而提高路由规则的复用性。在 vue-router 中使用英文的冒号（:）来定义路由的参数项。所以需要修改 router/index.js 中的文件

### \$route.params 参数对象

首先要改 router/index.js 中的代码，注意，下面的代码只需要写第二行一行就可以了

router/index.js

```
1 // 路由中的动态参数以 : 进行声明，冒号后面的是动态参数的名称
2 { path: '/movie/:id', component: Movie }
3
4 // 将以下 3 个路由规则，合并成了一个，提高了路由规则的复用性
5 { path: '/movie/1', component: Movie }
6 { path: '/movie/2', component: Movie }
7 { path: '/movie/3', component: Movie }
```

传参的话需要在

```
<router-link :to="/movie/1">
    movie1
<router-link/>
<router-link :to="/movie/2">
    movie2
<router-link/>
<router-link :to="/movie/3">
    movie3
<router-link/>
```

在动态路由渲染出来的组件中，可以使用this.\$route.params 对象访问到动态匹配的参数值。

```
1 <template>
2   <div class="movie-container">
3     <!-- this.$route 是路由的“参数对象” -->
4     <h3>Movie 组件 -- {{ this.$route.params.id }}</h3>
5   </div>
6 </template>
7
8 <script>
9 export default {
10   name: 'Movie'
11 }
12 </script>
```

但是这样取到的值很麻烦，下面还有一种方法可以简便的获取传过来的参数值

### query进行传参

```
<!-- 跳转路由并携带query参数，to的字符串写法 -->
<!-- <router-link :to="`/home/message/detail?id=${m.id}&title=${m.title}`">{{m.title}}</router-link>&ampnbsp&nbs
<!-- 跳转路由并携带query参数，to的对象写法 -->
<router-link :to={
  path:'/home/message/detail',
  query:{
    id:m.id,
    title:m.title
  }
}>
  {{m.title}}
</router-link>
```

这种也是通过this.\$router.query.xxx进行接收的。。。

## 使用 props 接收路由参数

为了简化路由参数的获取形式，vue-router 允许在路由规则中开启props 传参。示例代码如下：

```
1 // 1、在定义路由规则时，声明 props: true 选项，  
2 // 即可在 Movie 组件中，以 props 的形式接收到路由规则匹配到的参数项  
3 { path: '/movie/:id', component: Movie, props: true}  
4  
5 <template>  
6   <!-- 3、直接使用 props 中接收的路由参数 -->  
7   <h3>MyMovie组件 --- {{id}}</h3>  
8 </template>  
9  
10 <script>  
11 export default {  
12   props: ['id'] // 2、使用 props 接收路由规则中匹配到的参数项  
13 }  
14 </script>
```

在这里需要补充一点，万一url后面还加的有查询参数（?name=zs&age=18这种）的话，可以通过这种方式来进行接收

```
<!-- 注意2：在 hash 地址中，？后面的参数项，叫做“查询参数” -->  
<!-- 在路由“参数对象”中，需要使用 this.$route.query 来访问查询参数 -->  
<router-link to="/movie/1">洛基</router-link>  
<router-link to="/movie/2?name=zs&age=20">雷神</router-link>
```

## 声明式导航 & 编程式导航

在浏览器中，点击链接实现导航的方式，叫做声明式导航。例如：

- 普通网页中点击 `<a>` 链接、vue 项目中点击 `<router-link>` 都属于声明式导航

在浏览器中，调用 API 方法实现导航的方式，叫做编程式导航。例如：

- 普通网页中调用 `location.href` 跳转到新页面的方式，属于编程式导航

### vue-router 中的编程式导航 API

vue-router 提供了许多编程式导航的API，其中最常用的导航 API 分别是：

- ① `this.$router.push('hash 地址')`
- 跳转到指定hash 地址，并增加一条历史记录

- ② this.\$router.replace('hash 地址')
- 跳转到指定的hash 地址，并替换掉当前的历史记录

- ③ this.\$router.go(数值 n)

- 实现导航历史前进、后退

### \$router.push

调用this.\$router.push() 方法，可以跳转到指定的hash 地址，从而展示对应的组件页面。这个push里面也可以写配置对象，可以写里面可以携带什么参数，具体跳转到那里，name属性是什么等等。示例代码如下：

```
1 <template>
2   <div class="home-container">
3     <h3>Home 组件</h3>
4     <button @click="gotoMovie">跳转到 Movie 页面</button>
5   </div>
6 </template>
7
8 <script>
9 export default {
10   methods: {
11     gotoMovie() { this.$router.push('/movie/1') }
12   }
13 }
14 </script>
```

### \$router.replace

调用this.\$router.replace() 方法，可以跳转到指定的hash 地址，从而展示对应的组件页面。

push 和 replace 的区别：

- push 会增加一条历史记录
- replace 不会增加历史记录，而是替换掉当前的历史记录（replace是替换掉栈顶的那一条记录，所以不会增加历史记录）

### \$router.go

调用this.\$router.go() 方法，可以在浏览历史中前进和后退。示例代码如下：

```
1 <template>
2   <h3>MyMovie组件 --- {{id}}</h3>
3   <button @click="goBack">后退</button>
4 </template>
5
6 <script>
7 export default {
8   props: ['id'],
9   methods: {
10     goBack() { this.$router.go(-1) } // 后退到之前的组件页面
11   },
12 }
13 </script>
```

注意：如果后退的次数超过上限，会原地不动。。。所以一般都后退一步或者前进一步

### \$router.go 的简化用法

在实际开发中，一般只会前进和后退一层页面。因此vue-router 提供了如下两个便捷方法：

① \$router.back()

在历史记录中，后退到上一个页面

② \$router.forward()

在历史记录中，前进到下一个页面

## 缓存路由组件

很明显，组件的切换就会发生销毁和创建的过程

用keep-alive标签直接包住就行了。



```
<keep-alive include="News">
  <router-view></router-view>
</keep-alive>

<!-- 缓存多个路由组件 -->
<!-- <keep-alive :include="['News', 'Message']"> -->

<!-- 缓存一个路由组件 -->
<keep-alive include="News">
  <router-view></router-view>
</keep-alive>
```

也可以不写include这个属性，里面的组件会全部缓存！！

## 两个新的生命周期钩子

1. 作用：路由组件所独有的两个钩子，用于捕获路由组件的激活状态。|

2. 具体名字：

1. `activated` 路由组件被激活时触发。

2. `deactivated` 路由组件失活时触发。

```
activated() []
```

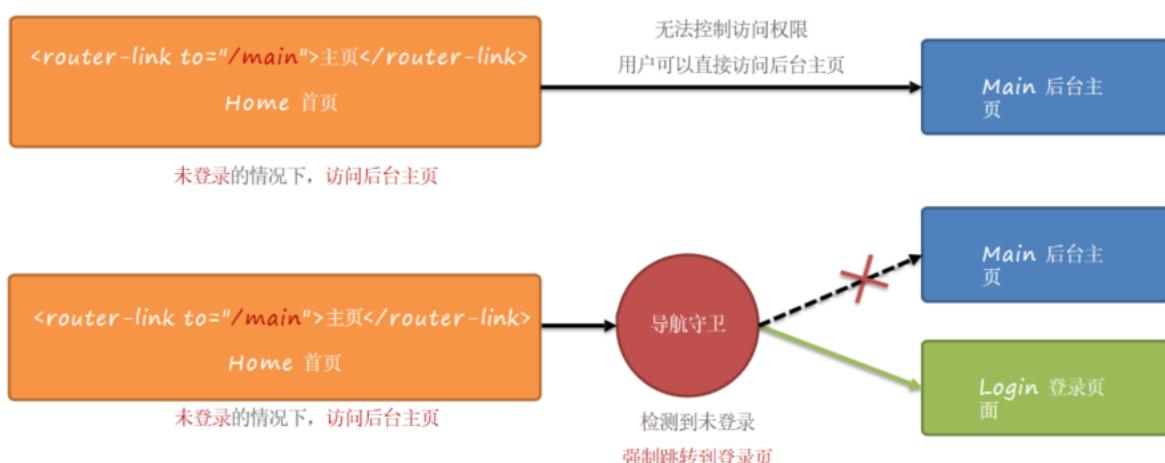
```
deactivated() {}
```

一个失活，一个激活

这个搭配着缓存路由组件比较有用

## 导航守卫

导航守卫可以控制路由的访问权限。示意图如下：



## 全局前置守卫

每次发生路由的导航跳转时，都会触发全局前置守卫。因此，在全局前置守卫中，程序员可以对每个路由进行访问权限的控制：

```
1 // 创建路由实例对象
2 const router = new VueRouter({ ... })
3
4 // 调用路由实例对象的 beforeEach 方法，即可声明“全局前置守卫”
5 // 每次发生路由导航跳转的时候，都会自动触发 fn 这个“回调函数”
6 router.beforeEach(fn)
```

## 守卫方法的 3 个形参

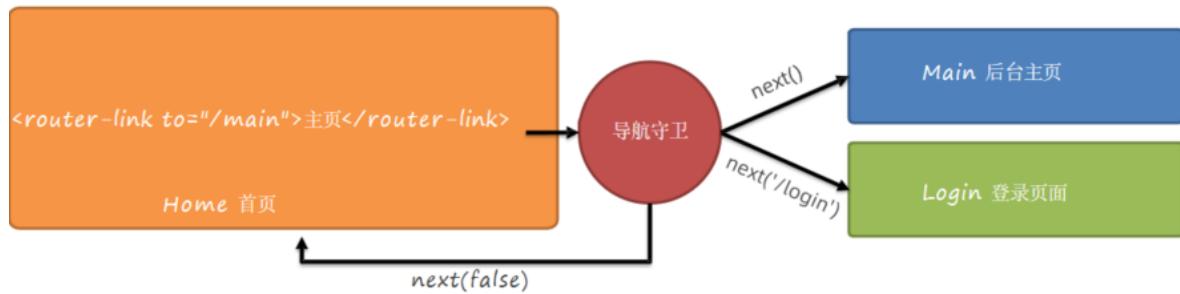
全局前置守卫的回调函数中接收 3 个形参，格式为：

```
1 // 创建路由实例对象
2 const router = new VueRouter({ ... })
3
4 // 全局前置守卫
5 router.beforeEach((to, from, next) => {
6   // to 是将要访问的路由的信息对象
7   // from 是将要离开的路由的信息对象
8   // next 是一个函数，调用 next() 表示放行，允许这次路由导航
9 })
```

由于这个是全局的前置守卫，意思就是每次要访问的路由都需要调用这个函数来判断！

### next 函数的 3 种调用方式

参考示意图，分析next 函数的 3 种调用方式最终导致的结果：



- 当前用户拥有后台主页的访问权限，直接放行：next()
- 当前用户没有后台主页的访问权限，强制其跳转到登录页面：next('/login')
- 当前用户没有后台主页的访问权限，不允许跳转到后台主页：next(false)

### 控制后台主页的访问权限

查看to和from的有关属性可以直接console.log进行打印查看，这个玩意是记不住的，不如手动打印出来看看

```
1 router.beforeEach((to, from, next) => {
2   if (to.path === '/main') {
3     const token = localStorage.getItem('token')
4     if (token) {
5       next() // 访问的是后台主页，且有 token 的值
6     } else {
7       next('/login') // 访问的是后台主页，但是没有 token 的值
8     }
9   } else {
10   next() // 访问的不是后台主页，直接放行
11 }
12 })
```

但是这样写的比较繁琐。我们上面是根据路径来判断的，下面我们可以根据路由里面新增一个属性，如

果to里面有这个属性就可以进这里面的逻辑

```
path: 'news',
component: News,
meta: {isAuth: false}
},
```

之后就可以直接

```
if(to.meta.isAuth){
//XX逻辑
}
```

直接这样判断就可以了，比较方便

## 全局后置守卫

每次路由切换之后调用

使用的场景比如说当切换页面过后，改个标题就行了

## 独享路由守卫

在路由规则里直接配置beforeEnter 为独享路由守卫

注意独享路由守卫只有前置,没有后置！！！！！！！！！！！！！！

要完成后置就可以配置全局的后置路由守卫一起使用

这个路由守卫写在路由的配置里面就可以了

router/index.js

```
const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      component: Foo,
      beforeEnter: (to, from, next) => {
        // ...
      }
    }
  ]
})
```

```
beforeEnter(to,from,next){
  console.log('beforeEnter',to,from)
  if(to.meta.isAuth){ //判断当前路由是否需要进行权限控制
    if(localStorage.getItem('school') === 'atguigu'){
      next()
    }else{
      alert('暂无权限查看')
      // next({name:'guanyu'})
    }
  }else{
    next()
  }
}
```

## 组件内路由守卫

```
index.js M About.vue ●
src > pages > About.vue > {} "About.vue" > script > default
3  </template>
4
5  <script>
6    export default {
7      name:'About',
8      /* beforeDestroy() {
9        console.log('About组件即将被销毁了')
10     }*/
11     /* mounted() {
12       console.log('About组件挂载完毕了',this)
13       window.aboutRoute = this.$route
14       window.aboutRouter = this.$router
15     }*/
16     mounted() {
17       console.log('%%%',this.$route)
18     },
19   },          这两个是写在组件内的
20   //通过路由规则，进入该组件时被调用
21   beforeRouteEnter (to, from, next) {
22     // ...
23   },
24
25   //通过路由规则，离开该组件时被调用
26   beforeRouteLeave (to, from, next) {
27     // ...
28   }
29 }
30 </script>
```

注意这里是通过**路由规则**，这个路由规则的意思是你必须经过路由访问的话才会执行这个方法，如果是你直接引用这个组件的话，是不会调用这个方法的！

### history模式和hash模式

```
9
0  //创建并暴露一个路由器
1  const router = new VueRouter([
2    mode:'h'
3    routes: [
4      { 
5        name:'guanyu',
```

默认是hash工作模式

① localhost:8080/#/home/message

这个符号就叫hash

反正就用hash模式就可以了，省事！

总结：

1. 对于一个url来说，什么是hash值？—— #及其后面的内容就是hash值。
2. hash值不会包含在 HTTP 请求中，即：hash值不会带给服务器。
3. hash模式：
  1. 地址中永远带着#号，不美观。
  2. 若以后将地址通过第三方手机app分享，若app校验严格，则地址会被标记为不合法。
  3. 兼容性较好。
4. history模式：
  1. 地址干净，美观。
  2. 兼容性和hash模式相比略差。
  3. 应用部署上线时需要后端人员支持，解决刷新页面服务端404的问题。

## 完整的导航解析流程

1. 导航被触发。
2. 在失活的组件里调用 `beforeRouteLeave` 守卫。
3. 调用全局的 `beforeEach` 守卫。
4. 在重用的组件里调用 `beforeRouteUpdate` 守卫 (2.2+)。
5. 在路由配置里调用 `beforeEnter`。
6. 解析异步路由组件。
7. 在被激活的组件里调用 `beforeRouteEnter`。
8. 调用全局的 `beforeResolve` 守卫 (2.5+)。
9. 导航被确认。
10. 调用全局的 `afterEach` 钩子。
11. 触发 DOM 更新。
12. 调用 `beforeRouteEnter` 守卫中传给 `next` 的回调函数，创建好的组件实例会作为回调函数的参数传入。

Vue3

ES6模块化与异步编程高级用法

# ES6 模块化

## 回顾：node.js 中如何实现模块化

node.js 遵循了 CommonJS 的模块化规范。其中：

- 导入其它模块使用 require() 方法
- 模块对外共享成员使用 module.exports 对象

模块化的好处：

大家都遵守同样的模块化规范写代码，降低了沟通的成本，极大方便了各个模块之间的相互调用，利人利己。

## 前端模块化规范的分类

在 ES6 模块化规范诞生之前，JavaScript 社区已经尝试并提出了AMD、CMD、CommonJS 等模块化规范。但是，这些由社区提出的模块化标准，还是存在一定的差异性与局限性、并不是浏览器与服务器通用的模块化标准，例如：

- AMD 和 CMD 适用于浏览器端的 Javascript 模块化
- CommonJS 适用于服务器端的 Javascript 模块化

太多的模块化规范给开发者增加了学习的难度与开发的成本。因此，大一统的ES6 模块化规范诞生了！

## 什么是 ES6 模块化规范

ES6 模块化规范是浏览器端与服务器端通用的模块化开发规范。它的出现极大的降低了前端开发者的模块化学习成本，开发者不需再额外学习AMD、CMD 或 CommonJS 等模块化规范。

ES6 模块化规范中定义：

- 每个 js 文件都是一个独立的模块
- 导入其它模块成员使用 import 关键字
- 向外共享模块成员使用 export 关键字

## 在 node.js 中体验 ES6 模块化

node.js 中默认仅支持 CommonJS 模块化规范，若想基于node.js 体验与学习ES6 的模块化语法，可以按照如下两个步骤进行配置：

- ① 确保安装了v14.15.1 或更高版本的node.js
- ② 在 package.json 的根节点中添加 "type": "module" 节点

## ES6 模块化的基本语法

ES6 的模块化主要包含如下 3 种用法：

- ① 默认导出与默认导入
- ② 按需导出与按需导入

### ③ 直接导入并执行模块中的代码

#### 默认导出

默认导出的语法： export default 默认导出的成员

```
1 let n1 = 10 // 定义模块私有成员 n1
2 let n2 = 20 // 定义模块私有成员 n2（外界访问不到 n2，因为它没有被共享出去）
3 function show() {} // 定义模块私有方法 show
4
5 export default { // 使用 export default 默认导出语法，向外共享 n1 和 show 两个成员
6   n1,
7   show
8 }
```

#### 默认导入

默认导入的语法： import 接收名称 from '模块标识符'

```
1 // 从 01_m1.js 模块中导入 export default 向外共享的成员
2 // 并使用 m1 成员进行接收
3 import m1 from './01_m1.js'
4
5 // 打印输出的结果为：
6 // { n1: 10, show: [Function: show] }
7 console.log(m1)
```

#### 默认导出的注意事项

每个模块中，只允许使用唯一的一次export default，否则会报错！

```
1 let n1 = 10 // 定义模块私有成员 n1
2 let n2 = 20 // 定义模块私有成员 n2（外界访问不到 n2，因为它没有被共享出去）
3 function show() {} // 定义模块私有方法 show
4
5 export default { // 使用 export default 默认导出语法，向外共享 n1 和 show 两个成员
6   n1,
7   show
8 }
9
10 // SyntaxError: Identifier 'default' has already been declared
11 export default {
12   n2
13 }
```

## 默认导入的注意事项

默认导入时的接收名称可以任意名称，只要是合法的成员名称即可：

```
1 // m1 是合法的名称
2 import m1 from './01_m1.js'
3
4 // 123m 不是合法的名称，因为成员名称不能以数字开头
5 import 123m from './01_m1.js'
```

## 按需导出

按需导出的语法： export 按需导出的成员

```
1 // 当前模块为 03_m2.js
2
3 // 向外按需导出变量 s1
4 export let s1 = 'aaa'
5 // 向外按需导出变量 s2
6 export let s2 = 'ccc'
7 // 向外按需导出方法 say
8 export function say() {}
```

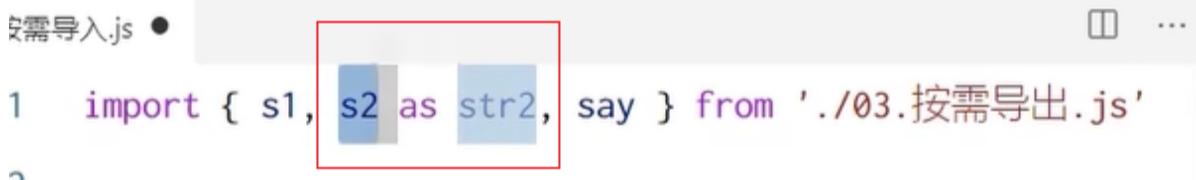
## 按需导入

按需导入的语法： import { s1 } from '模块标识符'

```
1 // 导入模块成员
2 import { s1, s2, say } from './03_m2.js'
3
4 console.log(s1) // 打印输出 aaa
5 console.log(s2) // 打印输出 ccc
6 console.log(say) // 打印输出 [Function: say]
```

## 按需导出与按需导入的注意事项

- ① 每个模块中可以使用多次按需导出
- ② 按需导入的成员名称必须和按需导出的名称保持一致
- ③ 按需导入时，可以使用 as 关键字进行重命名



- ④ 按需导入可以和默认导入一起使用

## 直接导入并执行模块中的代码

如果只想单纯地执行某个模块中的代码，并不需要得到模块中向外共享的成员。此时，可以直接导入并执行模块代码，示例代码如下：

```
1 // 当前文件模块为 05_m3.js
2
3 // 在当前模块中执行一个 for 循环操作
4 for (let i = 0; i < 3; i++) {
5   console.log(i)
6 }
7
8 // -----分割线-----
9
10 // 直接导入并执行模块代码，不需要得到模块向外共享的成员
11 import './05_m3.js'
```

## Promise

### 回调地狱

多层回调函数的相互嵌套，就形成了回调地狱。示例代码如下：

```
1 setTimeout(() => { // 第 1 层回调函数
2   console.log('延时 1 秒后输出')
3
4   setTimeout(() => { // 第 2 层回调函数
5     console.log('再延时 2 秒后输出')
6
7     setTimeout(() => { // 第 3 层回调函数
8       console.log('再延时 3 秒后输出')
9     }, 3000)
10    }, 2000)
11  }, 1000)
```

回调地狱的缺点：

- 代码耦合性太强，牵一发而动全身，难以维护
- 大量冗余的代码相互嵌套，代码的可读性变差

## 如何解决回调地狱的问题

为了解决回调地狱的问题，ES6（ECMAScript 2015）中新增了Promise的概念。

### Promise 的基本概念

#### ① Promise 是一个构造函数

- 我们可以创建 Promise 的实例const p = new Promise()
- new 出来的 Promise 实例对象，代表一个异步操作

#### ② Promise.prototype 上包含一个.then() 方法

- 每一次new Promise() 构造函数得到的实例对象，都可以通过原型链的方式访问到.then() 方法，例如p.then()

#### ③ .then() 方法用来预先指定成功和失败的回调函数

- p.then(成功的回调函数，失败的回调函数)
- p.then(result => {}, error => {})
- 调用.then() 方法时，成功的回调函数是必选的、失败的回调函数是可选的

## 基于回调函数按顺序读取文件内容

```
1 // 读取文件 1.txt
2 fs.readFile('./files/1.txt', 'utf8', (err1, r1) => {
3   if (err1) return console.log(err1.message) // 读取文件 1 失败
4   console.log(r1) // 读取文件 1 成功
5   // 读取文件 2.txt
6   fs.readFile('./files/2.txt', 'utf8', (err2, r2) => {
7     if (err2) return console.log(err2.message) // 读取文件 2 失败
8     console.log(r2) // 读取文件 2 成功
9     // 读取文件 3.txt
10    fs.readFile('./files/3.txt', 'utf8', (err3, r3) => {
11      if (err3) return console.log(err3.message) // 读取文件 3 失败
12      console.log(r3)// 读取文件 3 成功
13    })
14  })
15 })
```

## 基于 then-fs 读取文件内容

由于node.js 官方提供的fs 模块仅支持以回调函数的方式读取文件，不支持 Promise 的调用方式。因此，需要先运行如下的命令，安装then-fs 这个第三方包，从而支持我们基于 Promise 的方式读取文件的内容：

```
1 npm install then-fs
```

## then-fs 的基本使用

调用then-fs 提供的readFile() 方法，可以异步地读取文件的内容，它的返回值是 Promise 的实例对象。因此可以调用.then() 方法为每个 Promise 异步操作指定成功和失败之后的回调函数。示例代码如下：

```
1 /**
2 * 基于 Promise 的方式读取文件
3 */
4 import thenFs from 'then-fs'
5 // 注意: .then() 中的失败回调是可选的，可以被省略
6 thenFs.readFile('./files/1.txt', 'utf8').then(r1 => { console.log(r1) }, err1 => { console.log(err1.message) })
7 thenFs.readFile('./files/2.txt', 'utf8').then(r2 => { console.log(r2) }, err2 => { console.log(err2.message) })
8 thenFs.readFile('./files/3.txt', 'utf8').then(r3 => { console.log(r3) }, err3 => { console.log(err3.message) })
```

注意：上述的代码无法保证文件的读取顺序(也就是说没有办法保证先读文件1，再读文件2，再读文件3。。)，需要做进一步的改进！

### .then() 方法的特性

如果上一个 .then() 方法中返回了一个新的 Promise 实例对象，则可以通过下一个 .then() 继续进行处理。通过 .then() 方法的链式调用，就解决了回调地狱的问题。

### 基于 Promise 按顺序读取文件的内容

Promise 支持链式调用，从而来解决回调地狱的问题。示例代码如下：

```
1 thenFs.readFile('./files/1.txt', 'utf8') // 1. 返回值是 Promise 的实例对象
2   .then((r1) => { // 2. 通过 .then 为第一个 Promise 实例指定成功之后的回调函数
3     console.log(r1)
4     return thenFs.readFile('./files/2.txt', 'utf8') // 3. 在第一个 .then 中返回一个新的 Promise 实例对象
5   })
6   .then((r2) => { // 4. 继续调用 .then，为上一个 .then 的返回值（新的 Promise 实例）指定成功之后的回调函数
7     console.log(r2)
8     return thenFs.readFile('./files/3.txt', 'utf8') // 5. 在第二个 .then 中再返回一个新的 Promise 实例对象
9   })
10  .then((r3) => { // 6. 继续调用 .then，为上一个 .then 的返回值（新的 Promise 实例）指定成功之后的回调函数
11    console.log(r3)
12  })
```

解决了回调里面套回调的问题。

### 通过 .catch 捕获错误

在 Promise 的链式操作中如果发生了错误，可以使用 Promise.prototype.catch 方法进行捕获和处理：

```
1 thenFs.readFile('./files/11.txt', 'utf8') // 文件不存在导致读取失败，后面的 3 个 .then 都不执行
2   .then(r1 => {
3     console.log(r1)
4     return thenFs.readFile('./files/2.txt', 'utf8')
5   })
6   .then(r2 => {
7     console.log(r2)
8     return thenFs.readFile('./files/3.txt', 'utf8')
9   })
10  .then(r3 => {
11    console.log(r3)
12  })
13  .catch(err => { // 捕获第 1 行发生的错误，并输出错误的消息
14    console.log(err.message)
15  })
```

如果不希望前面的错误导致后续的 .then 无法正常执行，则可以将 .catch 的调用提前，示例代码如下：

```
1 thenFs.readFile('./files/11.txt', 'utf8')
2   .catch(err => {           // 捕获第 1 行发生的错误，并输出错误的消息
3     console.log(err.message) // 由于错误已被及时处理，不影响后续 .then 的正常执行
4   })
5   .then(r1 => {
6     console.log(r1) // 输出 undefined
7     return thenFs.readFile('./files/2.txt', 'utf8')
8   })
9   .then(r2 => {
10    console.log(r2) // 输出 222
11    return thenFs.readFile('./files/3.txt', 'utf8')
12  })
13  .then(r3 => {
14    console.log(r3) // 输出 333
15  })
```

## Promise.all() 方法

Promise.all() 方法会发起并行的 Promise 异步操作，等所有的异步操作全部结束后才会执行下一步的 .then 操作（等待机制）。示例代码如下：

```
1 // 1. 定义一个数组，存放 3 个读文件的异步操作
2 const promiseArr = [
3   thenFs.readFile('./files/11.txt', 'utf8'),
4   thenFs.readFile('./files/2.txt', 'utf8'),
5   thenFs.readFile('./files/3.txt', 'utf8'),
6 ]
7 // 2. 将 Promise 的数组，作为 Promise.all() 的参数
8 Promise.all(promiseArr)
9   .then(([r1, r2, r3]) => { // 2.1 所有文件读取成功（等待机制）
10     console.log(r1, r2, r3)
11   })
12   .catch(err => { // 2.2 捕获 Promise 异步操作中的错误
13     console.log(err.message)
14   })
```

注意：数组中 Promise 实例的顺序，就是最终结果的顺序！

### Promise.race() 方法

Promise.race() 方法会发起并行的 Promise 异步操作，只要任何一个异步操作完成，就立即执行下一步的.then 操作（赛跑机制）。示例代码如下：

```
1 // 1. 定义一个数组，存放 3 个读文件的异步操作
2 const promiseArr = [
3   thenFs.readFile('./files/1.txt', 'utf8'),
4   thenFs.readFile('./files/2.txt', 'utf8'),
5   thenFs.readFile('./files/3.txt', 'utf8'),
6 ]
7 // 2. 将 Promise 的数组，作为 Promise.race() 的参数
8 Promise.race(promiseArr)
9   .then((result) => { // 2.1 只要任何一个异步操作完成，就立即执行成功的回调函数（赛跑机制）
10     console.log(result)
11   })
12   .catch(err => { // 2.2 捕获 Promise 异步操作中的错误
13     console.log(err.message)
14   })
```

## 基于 Promise 封装读文件的方法

方法的封装要求：

- ① 方法的名称要定义为 getFile
- ② 方法接收一个形参 fpath，表示要读取的文件的路径
- ③ 方法的返回值为 Promise 实例对象

## getFile 方法的基本定义

```
1 // 1. 方法的名称为 getFile  
2 // 2. 方法接收一个形参 fpath, 表示要读取的文件的路径  
3 function getFile(fpath) {  
4     // 3. 方法的返回值为 Promise 的实例对象  
5     return new Promise()  
6 }
```

注意：第 5 行代码中的new Promise() 只是创建了一个形式上的异步操作。

## 创建具体的异步操作

如果想要创建具体的异步操作，则需要在new Promise() 构造函数期间，传递一个function 函数，将具体的异步操作定义到function 函数内部。示例代码如下：

```
1 // 1. 方法的名称为 getFile  
2 // 2. 方法接收一个形参 fpath, 表示要读取的文件的路径  
3 function getFile(fpath) {  
4     // 3. 方法的返回值为 Promise 的实例对象  
5     return new Promise(function() {  
6         // 4. 下面这行代码，表示这是一个具体的、读文件的异步操作  
7         fs.readFile(fpath, 'utf8', (err, dataStr) => { })  
8     })  
9 }
```

## 获取 .then 的两个实参

通过.then() 指定的成功和失败的回调函数，可以在function 的形参中进行接收，示例代码如下：

```
1 function getFile(fpath) {  
2     // resolve 形参是：调用 getFile() 方法时，通过 .then 指定的“成功的”回调函数  
3     // reject 形参是：调用 getFile() 方法时，通过 .then 指定的“失败的”回调函数  
4     return new Promise(function(resolve, reject) {  
5         fs.readFile(fpath, 'utf8', (err, dataStr) => { })  
6     })  
7 }  
8  
9 // getFile 方法的调用过程：  
10 getFile('./files/1.txt').then(成功的回调函数, 失败的回调函数)
```

## 调用 resolve 和 reject 回调函数

Promise 异步操作的结果，可以调用 resolve 或 reject 回调函数进行处理。示例代码如下：

```
1 function getFile(fpath) {
2   // resolve 是“成功的”回调函数; reject 是“失败的”回调函数
3   return new Promise(function(resolve, reject) {
4     fs.readFile(fpath, 'utf8', (err, dataStr) => {
5       if(err) return reject(err) // 如果读取失败，则调用“失败的回调函数”
6       resolve(dataStr)         // 如果读取成功，则调用“成功的回调函数”
7     })
8   })
9 }
10
11 // getFile 方法的调用过程:
12 getFile('./files/1.txt').then(成功的回调函数, 失败的回调函数)
```

## async/await

### 什么是 async/await

async/await 是 ES8 (ECMAScript 2017) 引入的新语法，用来简化 Promise 异步操作。在 async/await 出现之前，开发者只能通过链式.then() 的方式处理 Promise 异步操作。示例代码如下：

```
1 thenFs.readFile('./files/1.txt', 'utf8')
2   .then(r1 => {
3     console.log(r1)
4     return thenFs.readFile('./files/2.txt', 'utf8')
5   })
6   .then(r2 => {
7     console.log(r2)
8     return thenFs.readFile('./files/3.txt', 'utf8')
9   })
10  .then(r3 => {
11    console.log(r3)
12  })
```

.then 链式调用的优点：解决了回调地狱的问题

.then 链式调用的缺点：代码冗余、阅读性差、不易理解

### async/await 的基本使用

使用 async/await 简化 Promise 异步操作的示例代码如下：

```
1 import thenFs from 'then-fs'
2
3 // 按照顺序读取文件 1, 2, 3 的内容
4 async function getAllFile() {
5   const r1 = await thenFs.readFile('./files/1.txt', 'utf8')
6   console.log(r1)
7   const r2 = await thenFs.readFile('./files/2.txt', 'utf8')
8   console.log(r2)
9   const r3 = await thenFs.readFile('./files/3.txt', 'utf8')
10  console.log(r3)
11 }
12
13 getAllFile()
```

## async/await 的使用注意事项

- ① 如果在function 中使用了 await, 则 function 必须被 async 修饰
- ② 在 async 方法中, 第一个 await 之前的代码会同步执行, await 之后的代码会异步执行 (也就会退出方法的执行, 然后执行主线程后面的代码)

```
1 console.log('A')
2 async function getAllFile() {
3   console.log('B')
4   const r1 = await thenFs.readFile('./files/1.txt', 'utf8')
5   const r2 = await thenFs.readFile('./files/2.txt', 'utf8')
6   const r3 = await thenFs.readFile('./files/3.txt', 'utf8')
7   console.log(r1, r2, r3)
8   console.log('D')
9 }
10
11 getAllFile()
12 console.log('C')
```

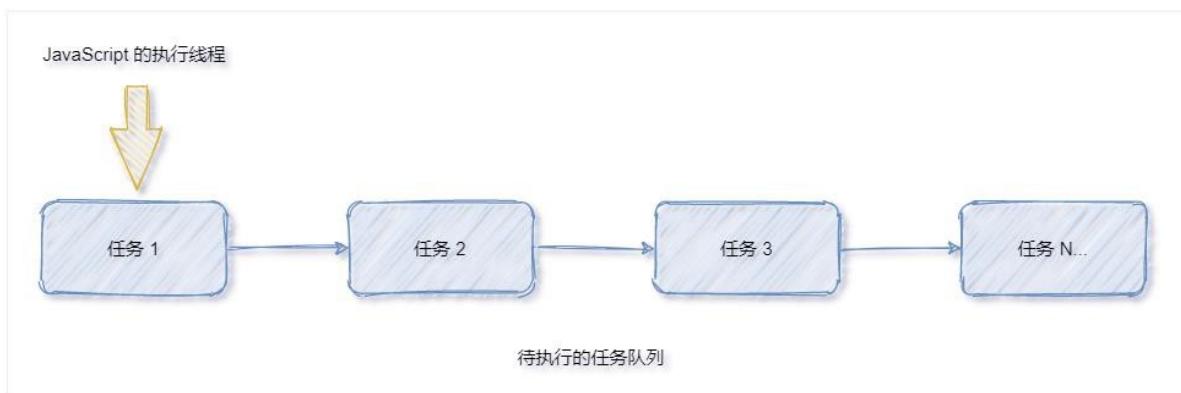
输出结果:

```
● ● ●  
1 // 最终输出的顺序  
2 A  
3 B  
4 C  
5 111 222 333  
6 D
```

## EventLoop

### JavaScript 是单线程的语言

JavaScript 是一门单线程执行的编程语言。也就是说，同一时间只能做一件事情。



单线程执行任务队列的问题：

如果前一个任务非常耗时，则后续的任务就不得不一直等待，从而导致程序假死的问题。

### 同步任务和异步任务

为了防止某个耗时任务导致程序假死的问题，JavaScript 把待执行的任务分为了两类：

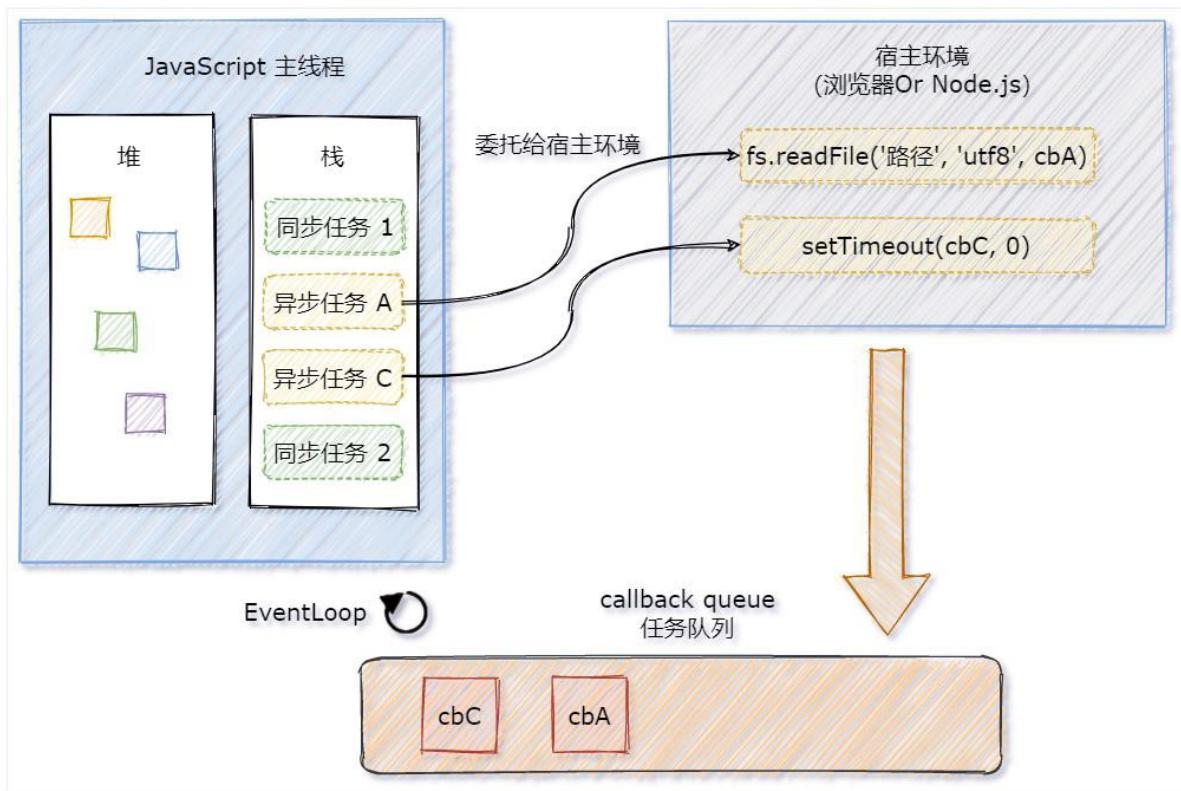
① 同步任务 (synchronous)

- 又叫做非耗时任务，指的是在主线程上排队执行的那些任务
- 只有前一个任务执行完毕，才能执行后一个任务

② 异步任务 (asynchronous)

- 又叫做耗时任务，异步任务由 JavaScript 委托给宿主环境（浏览器/node）进行执行
- 当异步任务执行完成后，会通知 JavaScript 主线程执行异步任务的回调函数

## 同步任务和异步任务的执行过程



- ① 同步任务由 JavaScript 主线程次序执行
- ② 异步任务委托给宿主环境执行
- ③ 已完成的异步任务对应的回调函数，会被加入到任务队列中等待执行
- ④ JavaScript 主线程的执行栈被清空后，会读取任务队列中的回调函数，次序执行
- ⑤ JavaScript 主线程不断重复上面的第 4 步

## EventLoop 的基本概念

JavaScript 主线程从“任务队列”中读取异步任务的回调函数，放到执行栈中依次执行。这个过程是循环不断的，所以整个的这种运行机制又称为EventLoop（事件循环）。

## 结合 EventLoop 分析输出的顺序

```
1 import thenFs from 'then-fs'  
2  
3 console.log('A')  
4 thenFs.readFile('./files/1.txt', 'utf8').then(dataStr => {  
5   console.log('B')  
6 })  
7 setTimeout(() => {  
8   console.log('C')  
9 }, 0)  
10 console.log('D')
```

正确的输出结果：ADCB。其中：

- A 和 D 属于同步任务。会根据代码的先后顺序依次被执行
- C 和 B 属于异步任务。它们的回调函数会被加入到任务队列中，等待主线程空闲时再执行

## 宏任务和微任务

### 什么是宏任务和微任务

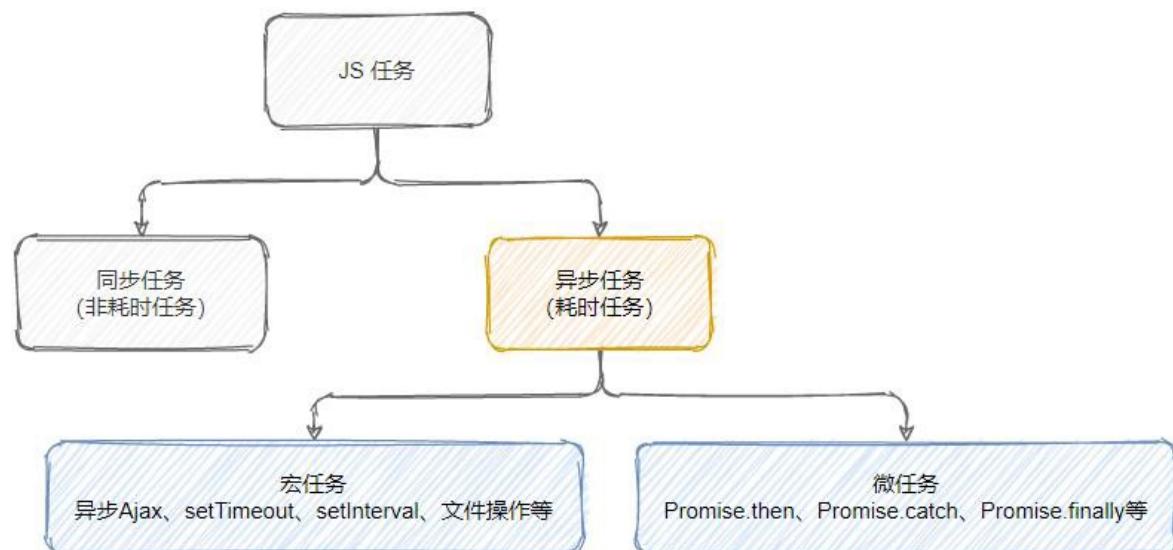
JavaScript 把异步任务又做了进一步的划分，异步任务又分为两类，分别是：

#### ① 宏任务 (macrotask)

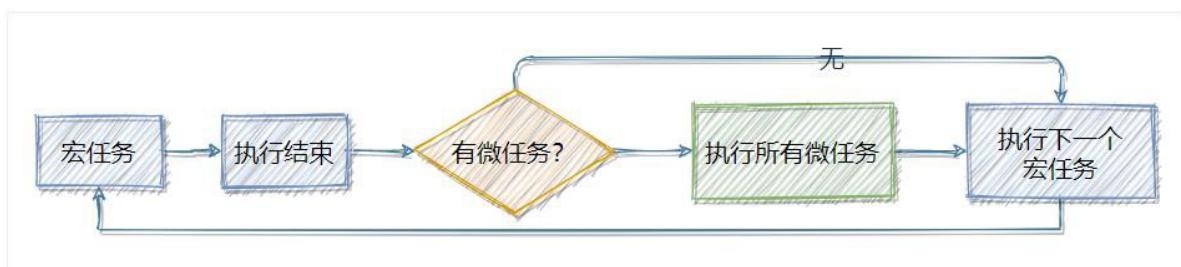
- 异步Ajax 请求、
- setTimeout、setInterval、
- 文件操作
- 其它宏任务

#### ② 微任务 (microtask)

- Promise.then、.catch 和 .finally
- process.nextTick
- 其它微任务



### 宏任务和微任务的执行顺序



每一个宏任务执行完之后，都会检查是否存在待执行的微任务，如果有，则执行完所有微任务之后，再继续执行下一个宏任务。

## 去银行办业务的场景

- ① 小云和小腾去银行办业务。首先，需要取号之后进行排队
  - 宏任务队列
- ② 假设当前银行网点只有一个柜员，小云在办理存款业务时，小腾只能等待

- 单线程，宏任务按次序执行
- ③ 小云办完存款业务后，柜员询问他是否还想办理其它业务？
- 当前宏任务执行完，检查是否有微任务
- ④ 小云告诉柜员：想要买理财产品、再办个信用卡、最后再兑换点马年纪念币？
- 执行微任务，后续宏任务被推迟
- ⑤ 小云离开柜台后，柜员开始为小腾办理业务
- 所有微任务执行完毕，开始执行下一个宏任务

## 分析以下代码输出的顺序

```
1 setTimeout(function () {
2   console.log('1')
3 })
4
5 new Promise(function (resolve) {
6   console.log('2')
7   resolve()
8 }).then(function () {
9   console.log('3')
10 })
11
12 console.log('4')
```

正确的输出顺序是：2431

分析：

- ① 先执行所有的同步任务
- 执行第 6 行、第 12 行代码
- ② 再执行微任务
- 执行第 9 行代码
- ③ 再执行下一个宏任务
- 执行第 2 行代码

## 经典面试题

请分析以下代码输出的顺序（代码较长，截取成了左中右3个部分）：

```
1 console.log('1') //
2 setTimeout(function () {
3   console.log('2') //
4   new Promise(function (resolve) {
5     console.log('3') //
6     resolve()
7   }).then(function () {
8     console.log('4') //
9   })
10 })
```

```
1 new Promise(function (resolve) {
2   console.log('5') //
3   resolve()
4 }).then(function () {
5   console.log('6') //
6 })
7 )
```

```
1 setTimeout(function () {
2   console.log('7') //
3   new Promise(function (resolve) {
4     console.log('8') //
5     resolve()
6   }).then(function () {
7     console.log('9') //
8   })
9 })
```

正确的输出顺序是：156234789

## Vue引入第三方

awesome-vue，一些vue的第三方组件。

[GitHub - vuejs/awesome-vue: 🎉 A curated list of awesome things related to Vue.js](https://github.com/vuejs/awesome-vue)

下面介绍swiper

安装swiper `cnpm install --save swiper`

使用

```
<template>
  <div>
    <Swiper>
      <Swiperslide>
        
      </Swiperslide>
      <Swiperslide>
        
      </Swiperslide>
      <Swiperslide>
        
      </Swiperslide>
    </Swiper>
  </div>
</template>
<script>
import { Swiper,Swiperslide } from 'swiper/vue';
import 'swiper/css';

export default{
  name:'SwiperDemo',
  components:{
    Swiper,
    Swiperslide
  }
}
</script>
```

# Vue3快速上手

---



## 1.Vue3简介

---

- 2020年9月18日，Vue.js发布3.0版本，代号：One Piece（海贼王）
- 耗时2年多、[2600+次提交](#)、[30+个RFC](#)、[600+次PR](#)、[99位贡献者](#)
- github上的tags地址：<https://github.com/vuejs/vue-next/releases/tag/v3.0.0>

## 2.Vue3带来了什么

---

### 1.性能的提升

- 打包大小减少41%
- 初次渲染快55%，更新渲染快133%
- 内存减少54%

.....

### 2.源码的升级

- 使用Proxy代替defineProperty实现响应式
- 重写虚拟DOM的实现和Tree-Shaking

.....

### 3.拥抱TypeScript

- Vue3可以更好的支持TypeScript

## 4.新的特性

### 1. Composition API (组合API)

- setup配置
- ref与reactive
- watch与watchEffect
- provide与inject
- .....

### 2. 新的内置组件

- Fragment

- Teleport
- Suspense

### 3. 其他改变

- 新的生命周期钩子
- data 选项应始终被声明为一个函数
- 移除keyCode支持作为 v-on 的修饰符
- .....

# 一、创建Vue3.0工程

## 1. 使用 vue-cli 创建

官方文档: <https://cli.vuejs.org/zh/guide/creating-a-project.html#vue-create>

```
## 查看@vue/cli版本，确保@vue/cli版本在4.5.0以上
vue --version
## 安装或者升级你的@vue/cli
npm install -g @vue/cli
## 创建
vue create vue_test
## 启动
cd vue_test
npm run serve
```

## 2. 使用 vite 创建

官方文档: <https://v3.cn.vuejs.org/guide/installation.html#vite>

vite官网: <https://vitejs.cn>

- 什么是vite? —— 新一代前端构建工具。
- 优势如下：
  - 开发环境中，无需打包操作，可快速的冷启动。
  - 轻量快速的热重载（HMR）。
  - 真正的按需编译，不再等待整个应用编译完成。
- 传统构建与vite构建对比图

```
## 创建工程
npm init vite-app <project-name>
## 进入工程目录
cd <project-name>
## 安装依赖
npm install
## 运行
npm run dev
```

## 分析工程结构

main.js

```
//引入的不再是vue构造函数了，引入的是一个名为createApp的工厂函数
import { createApp } from 'vue'
import App from './App.vue'

//创建应用实例对象--app(类似于之前Vue2中的vm，但app比vm更“轻”)
const app = createApp(App)

//挂载
app.mount('#app')
```

App.vue

```
<template>
    <!--只有这里发生了改变！！-->
    <!-- Vue3组件中的模板结构可以没有根标签 -->
    
    <HelloWorld msg="welcome to Your Vue.js App"/>
</template>

<script>
    import HelloWorld from './components/HelloWorld.vue'
    export default {
        name: 'App',
        components: {
            HelloWorld
        }
    }
</script>
```

Vue3

## 二、常用 Composition API

常用的组合式API

官方文档: <https://v3.cn.vuejs.org/guide/composition-api-introduction.html>

### 1. 拉开序幕的setup

1. 理解：Vue3.0中一个新的配置项，值为一个函数。
2. setup是所有**Composition API (组合API)** “表演的舞台”。
3. 组件中所用到的：数据、方法等等，均要配置在setup中。
4. setup函数的两种返回值：
  1. 若返回一个对象，则对象中的属性、方法，在模板中均可以直接使用。（重点关注！）
  2. 若返回一个渲染函数：则可以自定义渲染内容。（了解）
5. 注意点：
  1. 尽量不要与Vue2.x配置混用
    - Vue2.x配置（data、methos、computed...）中**可以访问到**setup中的属性、方法。

- 但在setup中**不能访问到**Vue2.x配置 (data、methos、computed...)。
  - 如果有重名, setup优先。
2. setup不能是一个async函数, 因为返回值不再是return的对象, 而是promise, 模板看不到return对象中的属性。 (后期也可以返回一个Promise实例, 但需要Suspense和异步组件的配合)

```

<template>
    <h1>一个人的信息</h1>
    <h2>姓名: {{name}}</h2>
    <h2>年龄: {{age}}</h2>
    <h2>性别: {{sex}}</h2>
</template>

<script>
    // import {h} from 'vue'
    export default {
        name: 'App',
        //此处只是测试一下setup, 暂时不考虑响应式的问题。
        setup(){
            //数据
            let name = '张三'
            let age = 18
            let a = 200

            //方法
            function sayHello(){
                alert(`我叫${name}, 我${age}岁了, 你好啊!`)
            }
            function test2(){
                console.log(name)
                console.log(age)
                console.log(sayHello)
                console.log(this.sex)
                console.log(this.saywelcome)
            }

            //返回一个对象 (常用)
            return {
                name,
                age,
                sayHello,
                test2,
                a
            }

            //返回一个函数 (渲染函数)
            // return ()=> h('h1','尚硅谷')
        }
    }
</script>

```

## 2.ref函数

- 作用: 定义一个响应式的数据
- 语法: `const xxx = ref(initialValue)`
  - 创建一个包含响应式数据的引用对象 (reference对象, 简称ref对象)。
  - JS中操作数据: `xxx.value`
  - 模板中读取数据: 不需要.value, 直接: `<div>{{xxx}}</div>`
- 备注:
  - 接收的数据可以是: 基本类型、也可以是对象类型。
  - 基本类型的数据: 响应式依然是靠 `Object.defineProperty()` 的 `get` 与 `set` 完成的。
  - 对象类型的数据: 内部 “**求助**” 了Vue3.0中的一个新函数—— `reactive` 函数。

```

<template>
  <h1>一个人的信息</h1>
  <h2>姓名: {{name}}</h2>
  <h2>年龄: {{age}}</h2>
  <h3>工作种类: {{job.type}}</h3>
  <h3>工作薪水: {{job.salary}}</h3>
  <button @click="changeInfo">修改人的信息</button>
</template>

<script>
  import {ref} from 'vue'
  export default {
    name: 'App',
    setup() {
      //数据
      let name = ref('张三')
      let age = ref(18)
      let job = ref({
        type: '前端工程师',
        salary: '30K'
      })

      //方法
      function changeInfo() {
        name.value = '李四'
        age.value = 48
        //这里注意, 不需要再在后面加value了, 因为这个是对象类型, 这个借助了reactive
        //函数
        job.value.type = 'UI设计师'
        job.value.salary = '60K'
        console.log(name, age)
        console.log(job.value)
      }

      //返回一个对象 (常用)
      return {
        name,
        age,
        job,
        changeInfo
      }
    }
  }

```

```
</script>
```

## 3.reactive函数

- 作用: 定义一个**对象类型 (或者数组)** 的响应式数据 (基本类型不要用它, 要用 `ref` 函数)
- 语法: `const 代理对象 = reactive(源对象)` 接收一个对象 (或数组) , 返回一个**代理对象 (Proxy的实例对象, 简称proxy对象)**
- `reactive`定义的响应式数据是“深层次的”。
- 内部基于 ES6 的 `Proxy` 实现, 通过代理对象操作源对象内部数据进行操作。

```
<template>
    <h1>一个人的信息</h1>
    <h2>姓名: {{person.name}}</h2>
    <h2>年龄: {{person.age}}</h2>
    <h3>工作种类: {{person.job.type}}</h3>
    <h3>工作薪水: {{person.job.salary}}</h3>
    <h3>爱好: {{person.hobby}}</h3>
    <h3>测试的数据c: {{person.job.a.b.c}}</h3>
    <button @click="changeInfo">修改人的信息</button>
</template>

<script>
    import {reactive} from 'vue'
    export default {
        name: 'App',
        setup() {
            //数据
            //这里包裹成了一个大对象, 当然你也可以分开写,
            let person = reactive({
                name:'张三',
                age:18,
                job:{
                    type:'前端工程师',
                    salary:'30K',
                    a:{
                        b:{
                            c:666
                        }
                    }
                },
                hobby:['抽烟','喝酒','烫头']
            })
            //方法
            function changeInfo(){
                person.name = '李四'
                person.age = 48
                person.job.type = 'UI设计师'
                person.job.salary = '60K'
                person.job.a.b.c = 999
                person.hobby[0] = '学习'
            }
        }
    }
</script>
```

```
//返回一个对象（常用）
return {
    person,
    changeInfo
}
}

</script>
```

## 4.Vue3.0中的响应式原理

### vue2.x的响应式

- 实现原理：
  - 对象类型：通过 `Object.defineProperty()` 对属性的读取、修改进行拦截（数据劫持）。
  - 数组类型：通过重写更新数组的一系列方法来实现拦截。（对数组的变更方法进行了包裹）。

```
Object.defineProperty(data, 'count', {
    get () {},
    set () {}
})
```

- 存在问题：
  - 新增属性、删除属性，界面不会更新。
  - 直接通过下标修改数组，界面不会自动更新。

### Vue3.0的响应式

- 实现原理：
  - 通过Proxy（代理）：拦截对象中任意属性的变化，包括：属性值的读写、属性的添加、属性的删除等。
  - 通过Reflect（反射）：对源对象的属性进行操作。
  - MDN文档中描述的Proxy与Reflect：
    - Proxy: [https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global\\_Objects/Proxy](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Proxy)
    - Reflect: [https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global\\_Objects/Reflect](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Reflect)

```
new Proxy(data, {
    // 拦截读取属性值
    get (target, prop) {
        return Reflect.get(target, prop)
    },
    // 拦截设置属性值或添加新属性
    set (target, prop, value) {
```

```

        return Reflect.set(target, prop, value)
    },
    // 拦截删除属性
    deleteProperty (target, prop) {
        return Reflect.deleteProperty(target, prop)
    }
}

proxy.name = 'tom'

```

## 5.reactive对比ref

- 从定义数据角度对比：
  - ref用来定义：**基本类型数据**。
  - reactive用来定义：**对象（或数组）类型数据**。
  - 备注：ref也可以用来定义**对象（或数组）类型数据**, 它内部会自动通过 reactive 转为**代理对象**。
- 从原理角度对比：
  - ref通过 Object.defineProperty() 的 get 与 set 来实现响应式（数据劫持）。
  - reactive通过使用**Proxy**来实现响应式（数据劫持），并通过**Reflect**操作**源对象**内部的数据。
- 从使用角度对比：
  - ref定义的数据：操作数据**需要 .value**，读取数据时模板中直接读取**不需要 .value**。
  - reactive定义的数据：操作数据与读取数据：**均不需要 .value**。

## 6.setup的两个注意点

- setup执行的时机
  - 在beforeCreate之前执行一次，this是undefined。
- setup的参数
  - props: 值为对象，包含：组件外部传递过来，且组件内部声明接收了的属性。
  - context: 上下文对象
    - attrs: 值为对象，包含：组件外部传递过来，但没有在props配置中声明的属性, 相当于 `this.$attrs`。
    - slots: 收到的插槽内容, 相当于 `this.$slots`。
    - emit: 分发自定义事件的函数, 相当于 `this.$emit`。

## 7.计算属性与监视

### 1.computed函数

- 与Vue2.x中computed配置功能一致
- 写法

```

import {computed} from 'vue'

setup(){
    ...
    //计算属性--简写
    let fullName = computed(()=>{

```

```

        return person.firstName + '-' + person.lastName
    })
//计算属性--完整
let fullName = computed({
    get(){
        return person.firstName + '-' + person.lastName
    },
    set(value){
        const nameArr = value.split('-')
        person.firstName = nameArr[0]
        person.lastName = nameArr[1]
    }
})
}

```

## 2.watch函数

- 与Vue2.x中watch配置功能一致
- 两个小“坑”：
  - 监视reactive定义的响应式数据时： oldValue无法正确获取、强制开启了深度监视（deep配置失效）。
  - 监视reactive定义的响应式数据中某个属性时： deep配置有效。

```

setup(){
    //数据
    let sum = ref(0)
    let msg = ref('你好啊')
    let person = reactive({
        name:'张三',
        age:18,
        job:{
            j1:{
                salary:20
            }
        }
    })
    //情况一： 监视ref定义的响应式数据，注意，这里不需要在sum.value了
    watch(sum,(newValue,oldValue)=>{
        console.log('sum变化了',newValue,oldValue)
    },{immediate:true})

    //情况二： 监视多个ref定义的响应式数据
    watch([sum,msg],(newValue,oldValue)=>{//这里的newValue和oldValue都是数组了
        console.log('sum或msg变化了',newValue,oldValue)
    })

    /* 情况三： 监视reactive定义的响应式数据
       若watch监视的是reactive定义的响应式数据，则无法正确获得oldValue！！
       若watch监视的是reactive定义的响应式数据，则强制开启了深度监视
    */
    watch(person,(newValue,oldValue)=>{
        console.log('person变化了',newValue,oldValue)
    },{immediate:true,deep:false}) //此处的deep配置不再奏效
}

```

```

//情况四：监视reactive定义的响应式数据中的某个属性
watch(()=>person.job, (newValue, oldValue)=>{
    console.log('person的job变化了', newValue, oldValue)
})

//情况五：监视reactive定义的响应式数据中的某些属性
watch([()=>person.job, ()=>person.name], (newValue, oldValue)=>{
    console.log('person的job变化了', newValue, oldValue)
})

//特殊情况
watch(()=>person.job, (newValue, oldValue)=>{
    console.log('person的job变化了', newValue, oldValue)
}, {deep:true}) //此处由于监视的是reactive素定义的对象中的某个属性，所以deep配置有效
}

```

### 3.watchEffect函数

我去，这个东西好智能

- watch的套路是：既要指明监视的属性，也要指明监视的回调。
- watchEffect的套路是：不用指明监视哪个属性，监视的回调中用到哪个属性，那就监视哪个属性。
- watchEffect有点像computed：
  - 但computed注重的计算出来的值（回调函数的返回值），所以必须要写返回值。
  - 而watchEffect更注重的是过程（回调函数的函数体），所以不用写返回值。

```

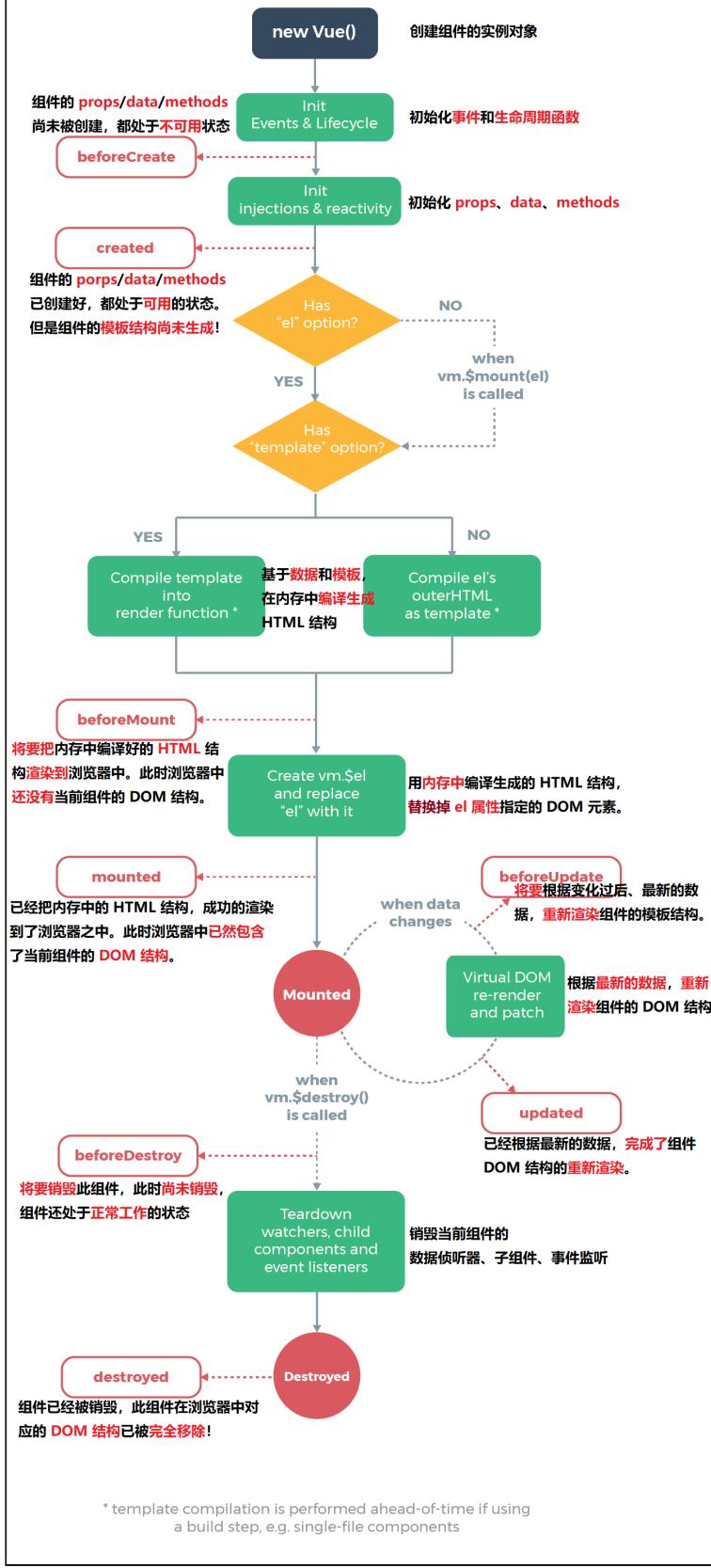
//watchEffect所指定的回调中用到的数据只要发生变化，则直接重新执行回调。
watchEffect(()=>{
    const x1 = sum.value
    const x2 = person.age
    console.log('watchEffect配置的回调执行了')
})

```

## 8.生命周期

---

## vue2.x的生命周期

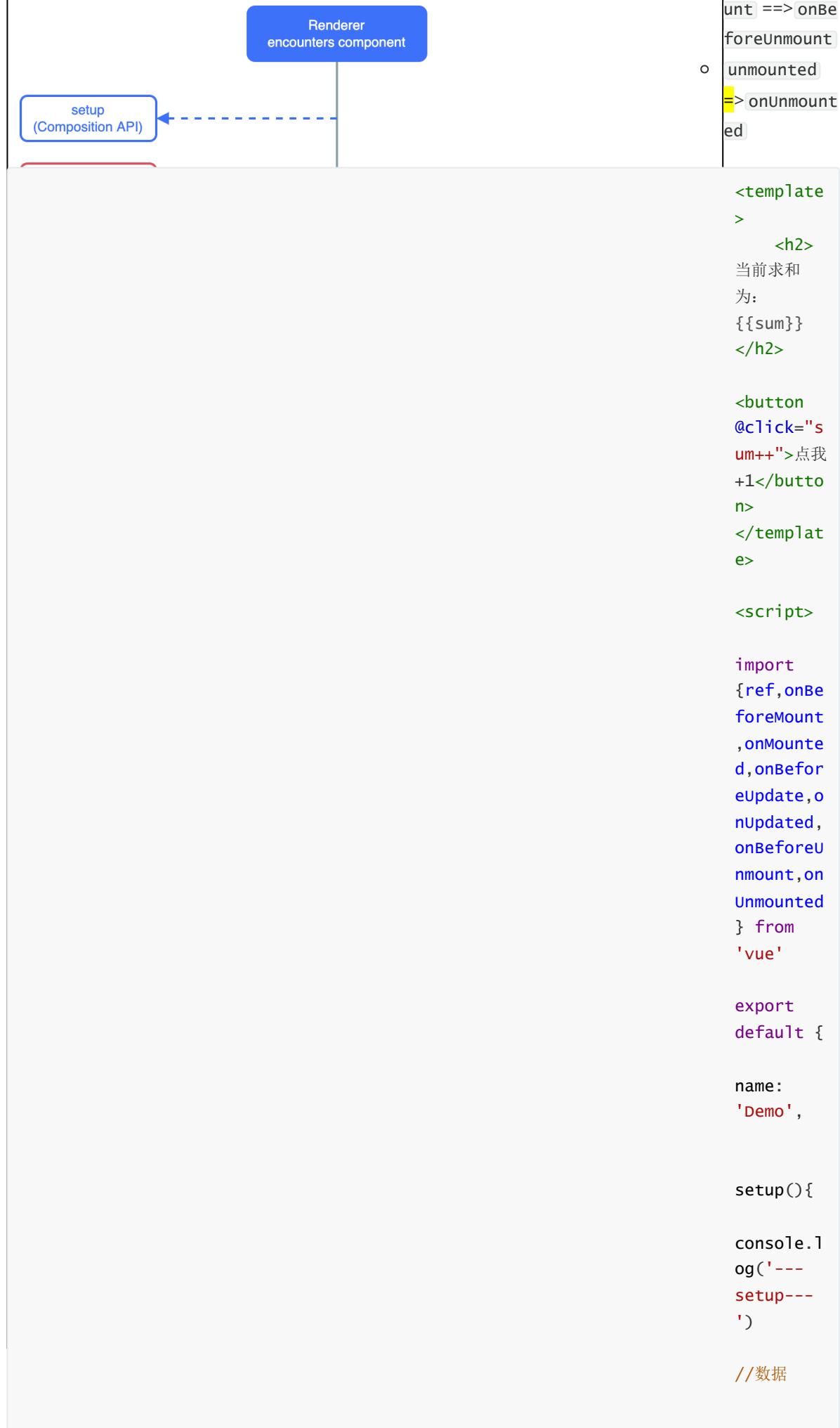


- Vue3.0中可以继续使用Vue2.x中的生命周期钩子，但有有两个被重名：
  - `beforeDestroy` 改名为 `beforeUnmount`
  - `destroyed` 改名为 `unmounted`

- Vue3.0也提供了 Composition API 形式的生命周期钩子，与Vue2.x中钩子对应关系如下：

- `beforeCreate` ==> `setup()`
- `created` ==> `setup()`
- `beforeMount` ==> `onBeforeMount`
- `mounted` ==> `onMounted`
- `beforeUpdate` ==> `onBeforeUpdate`
- `updated` ==> `onUpdated`

## vue3.0的生命周期



```
let sum =  
ref(0)
```

//通过组合  
式API的形  
式去使用生  
命周期钩子

```
onBeforeM  
ount(()=>  
{
```

```
console.l  
og('---  
onBeforeM  
ount---')
```

```
)
```

```
onMounted  
(()=>{
```

```
console.l  
og('---  
onMounted  
---')
```

```
)
```

```
onBeforeU  
pdate(()=>  
>{
```

```
console.l  
og('---  
onBeforeU  
pdate---  
')
```

```
)
```

```
onUpdated  
(()=>{
```

```
console.l  
og('---  
onUpdated  
---')
```

```
)
```

```
onBeforeU  
nmount()
```

```
=>{
```

```
    console.l  
og('---  
onBeforeU  
nmount---  
' )
```

```
})
```

```
onUnmount  
ed()=>{
```

```
    console.l  
og('---  
onUnmount  
ed---')
```

```
})
```

//返回一个  
对象（常  
用）

```
return  
{sum}
```

```
},
```

//通过配置  
项的形式使  
用生命周期  
钩子

```
//#region
```

```
beforeCre  
ate() {
```

```
    console.l  
og('---  
beforeCre  
ate---')
```

```
},
```

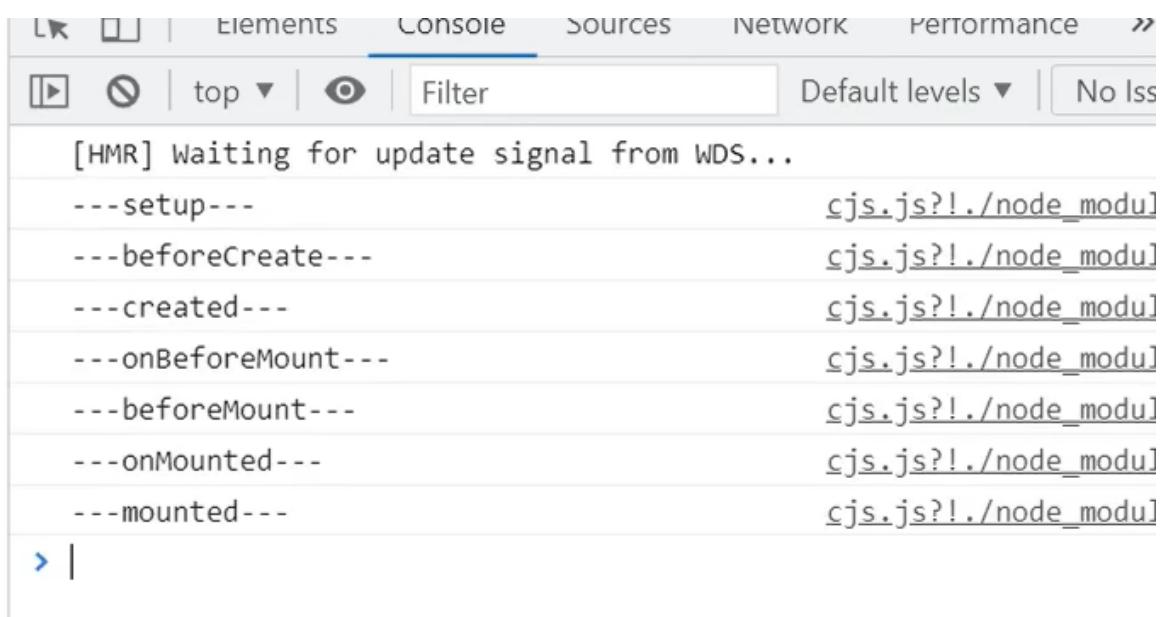
```
created()  
{
```

```
    console.log('---  
    created--  
    -')  
  
,  
  
beforeMount() {  
  
    console.log('---  
    beforeMount---')  
  
,  
  
mounted() {  
  
    console.log('---  
    mounted--  
    -')  
  
,  
  
beforeUpdated() {  
  
    console.log('---  
    beforeupdated---')  
  
,  
  
updated() {  
  
    console.log('---  
    updated--  
    -')  
  
,  
  
beforeUnmount() {  
  
    console.log('---  
    beforeunmount---')
```

```

        },
        unmounted
        () {
            console.log('---')
            unmounted
            ---')
        },
        // #endreg
        ion
    }
</script>

```



## 9.自定义hook函数

- 什么是hook? —— 本质是一个函数，把setup函数中使用的Composition API进行了封装。
- 类似于vue2.x中的mixin。
- 自定义hook的优势: 复用代码, 让setup中的逻辑更清楚易懂。

新建hooks文件夹

hooks/usePoint.js

```

import {reactive, onMounted, onBeforeUnmount} from 'vue'
export default function () {
    //实现鼠标“打点”相关的数据
    let point = reactive({
        x: 0,
        y: 0
    })
}

```

```

//实现鼠标“打点”相关的方法
function savePoint(event){
    point.x = event.pageX
    point.y = event.pageY
    console.log(event.pageX, event.pageY)
}

//实现鼠标“打点”相关的生命周期钩子
onMounted(()=>{
    window.addEventListener('click', savePoint)
})

onBeforeUnmount(()=>{
    window.removeEventListener('click', savePoint)
})

return point
}

```

引入这个hook，实现代码复用

demo.vue

引入这个hook就可以实现一个点击哪里就显示出坐标的功能，但是这个编写的是有些问题的，因为是在windows上绑定的。。。但是不影响

这样做的好处是，不用关心hook里面内部的细节，直接用就可以了

```

<template>
    <h2>当前求和为: {{sum}}</h2>
    <button @click="sum++">点我+1</button>
    <hr>
    <h2>当前点击时鼠标的坐标为: x: {{point.x}}, y: {{point.y}}</h2>
</template>

<script>
    import {ref} from 'vue'
    import usePoint from '../hooks/usePoint'
    export default {
        name: 'Demo',
        setup(){
            //数据
            let sum = ref(0)
            let point = usePoint()

            //这里也返回了point
            return {sum, point}
        }
    }
</script>

```

## 10.toRef

- 作用：创建一个 ref 对象，其 value 值指向另一个对象中的某个属性。
- 语法：`const name = toRef(person, 'name')`
- 应用：要将响应式对象中的某个属性单独提供给外部使用时。
- 扩展：`toRefs` 与 `toRef` 功能一致，但可以批量创建多个 ref 对象，语法：`toRefs(person)`

这个东西给我的感觉是 Java 里面的 String 类里面的 intern 方法。返回引用地址

```
<template>
  <h4>{{person}}</h4>
  <h2>姓名： {{name}}</h2>
  <h2>年龄： {{age}}</h2>
  <h2>薪资： {{job.j1.salary}}K</h2>
  <button @click="name+='~'">修改姓名</button>
  <button @click="age++">增长年龄</button>
  <button @click="job.j1.salary++">涨薪</button>
</template>

<script>
  import {ref, reactive, toRef, toRefs} from 'vue'
  export default {
    name: 'Demo',
    setup() {
      // 数据
      let person = reactive({
        name: '张三',
        age: 18,
        job: {
          j1: {
            salary: 20
          }
        }
      })

      // const name1 = person.name
      // console.log('%%%', name1)

      // const name2 = toRef(person, 'name')
      // console.log('####', name2)

      const x = toRefs(person)
      console.log('*****', x)

      // 返回一个对象（常用）
      return {
        person,
        // name: toRef(person, 'name'),
        // age: toRef(person, 'age'),
        // salary: toRef(person.job.j1, 'salary'), // 这里可以直接用 salary 取
        ... toRefs(person) // 解构了，但是这里只能拆第一层
      }
    }
  }
</script>
```

```
</script>
```

## 三、其它 Composition API

### 1.shallowReactive 与 shallowRef

- shallowReactive：只处理对象最外层属性的响应式（浅响应式）。
- shallowRef：只处理基本数据类型的响应式，不进行对象的响应式处理。
- 什么时候使用？
  - 如果有一个对象数据，结构比较深，但变化时只是外层属性变化 ==> shallowReactive。
  - 如果有一个对象数据，后续功能不会修改该对象中的属性，而是生成新的对象来替换 ==> shallowRef。

```
<template>
    <h4>当前的x.y值是: {{x.y}}</h4>
    <button @click="x={y:888}">点我替换x</button>
    <button @click="x.y++">点我x.y++</button>
    <hr>
    <h4>{{person}}</h4>
    <h2>姓名: {{name}}</h2>
    <h2>年龄: {{age}}</h2>
    <h2>薪资: {{job.j1.salary}}</h2>
    <button @click="name+='~'">修改姓名</button>
    <button @click="age++">增长年龄</button>
    <button @click="job.j1.salary++">涨薪</button>
</template>

<script>
    import {ref,reactive,toRef,toRefs,shallowReactive,shallowRef} from 'vue'
    export default {
        name: 'Demo',
        setup(){
            //数据
            // let person = shallowReactive({ //只考虑第一层数据的响应式
            let person = reactive({
                name:'张三',
                age:18,
                job:{
                    j1:{
                        salary:20
                    }
                }
            })
            let x = shallowRef({
                y:0
            })
            console.log('*****',x)

            return {
                x,
                person,
```

```
    ...toRefs(person)
  }
}
</script>
```

## 2.readonly 与 shallowReadonly

- readonly: 让一个响应式数据变为只读的（深只读）。
- shallowReadonly: 让一个响应式数据变为只读的（浅只读）。
- 应用场景: 不希望数据被修改时，而且有可能这个数据是别人传过来的。

```
<template>
  <h4>当前求和为: {{sum}}</h4>
  <button @click="sum++">点我++</button>
  <hr>
  <h2>姓名: {{name}}</h2>
  <h2>年龄: {{age}}</h2>
  <h2>薪资: {{job.j1.salary}}K</h2>
  <button @click="name+='~'">修改姓名</button>
  <button @click="age++">增长年龄</button>
  <button @click="job.j1.salary++">涨薪</button>
</template>

<script>
  import {ref,reactive,toRefs,readonly,shallowReadonly} from 'vue'
  export default {
    name: 'Demo',
    setup() {
      //数据
      let sum = ref(0)
      let person = reactive({
        name:'张三',
        age:18,
        job:{
          j1:{
            salary:20
          }
        }
      })
      //这里面的person从现在开始就不能再改变了
      person = readonly(person)
      //浅层次的不能再改变，如果是深层次的还可以改
      // person = shallowReadonly(person)

      // sum = readonly(sum)
      // sum = shallowReadonly(sum)

      //返回一个对象（常用）
      return {
    }
  }
}
```

```
        sum,
        ...toRefs(person)
    }
}
</script>
```

## 3.toRaw 与 markRaw

- toRaw:
  - 作用：将一个由 `reactive` 生成的 **响应式对象** 转为 **普通对象**。
  - 使用场景：用于读取响应式对象对应的普通对象，对这个普通对象的所有操作，不会引起页面更新。
- markRaw:
  - 作用：标记一个对象，使其永远不会再成为响应式对象（如果是直接在响应式对象上新增属性的话，默认的话也是响应式的，但是有些需求我们不想让他是响应式的）。
  - 应用场景：
    1. 有些值不应被设置为响应式的，例如复杂的第三方类库等。
    2. 当渲染具有不可变数据源的大列表时，跳过响应式转换可以提高性能。

## 4.customRef

- 作用：创建一个自定义的 ref，并对其依赖项跟踪和更新触发进行显式控制。
- 实现防抖效果：

```
<template>
  <input type="text" v-model="keyword">
  <h3>{{keyword}}</h3>
</template>

<script>
  import {ref,customRef} from 'vue'
  export default {
    name:'Demo',
    setup(){
      // let keyword = ref('hello') //使用vue准备好的内置ref
      //自定义一个myRef
      function myRef(value,delay){
        let timer
        //通过customRef去实现自定义
        return customRef((track,trigger)=>{
          return{
            get(){
              track() //告诉Vue这个value值是需要被“追踪”的（提前跟
get说，这个value是有用的）
              return value
            },
            set(newValue){

```

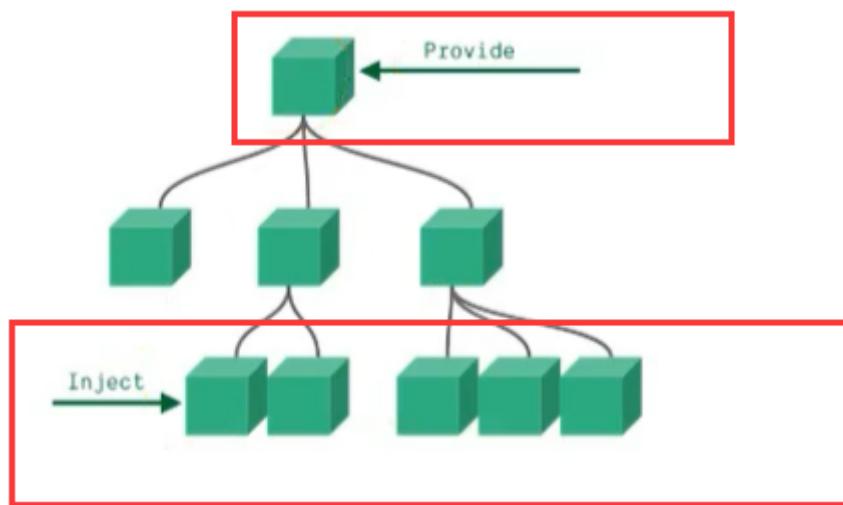
```

        clearTimeout(timer)
        timer = setTimeout(()=>{
            value = newValue
            trigger() //告诉Vue去更新界面
        },delay)
    }
}
}
}
let keyword = myRef('hello',500) //使用程序员自定义的ref
return {
    keyword
}
}
</script>

```

## 5.provide 与 inject

提供数据和注入数据



- 作用：实现**祖与后代（跨级）组件间**通信，但是其实子组件也可以用，不过我们一般用props来实现父子间传递数据
- 套路：父组件有一个 `provide` 选项来提供数据，后代组件有一个 `inject` 选项来开始使用这些数据
- 具体写法：

1. 祖组件中：

```

setup(){
    .....
    let car = reactive({name:'奔驰',price:'40万'})
    provide('car',car)
    .....
}

```

2. 后代（孙及后代）组件中：

```
setup(props, context){  
    ....  
    const car = inject('car')  
    return {car}  
    ....  
}
```

## 6.响应式数据的判断

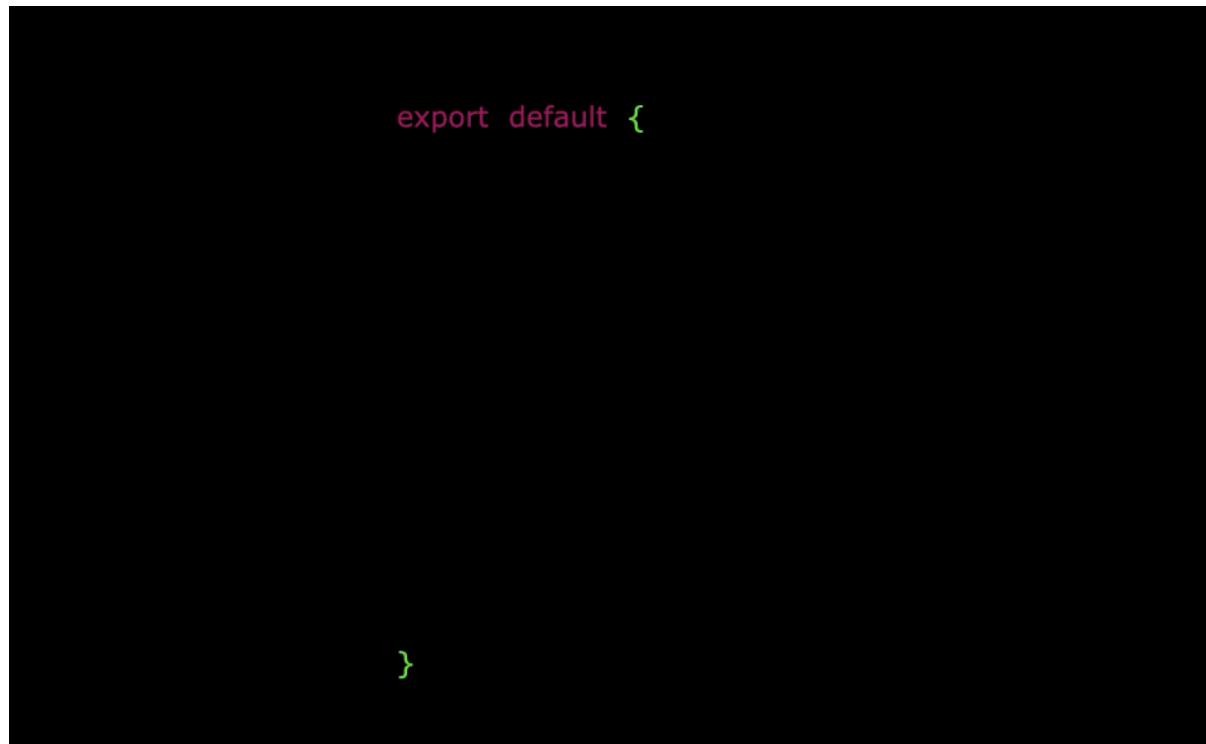
- isRef: 检查一个值是否为一个 ref 对象
- isReactive: 检查一个对象是否是由 `reactive` 创建的响应式代理
- is\_READONLY: 检查一个对象是否是由 `readonly` 创建的只读代理
- isProxy: 检查一个对象是否是由 `reactive` 或者 `readonly` 方法创建的代理

```
<template>  
    <h3>我是App组件</h3>  
</template>  
  
<script>  
    import {ref, reactive,toRefs,readonly,isRef,isReactive,is_READONLY,isProxy }  
from 'vue'  
    export default {  
        name:'App',  
        setup(){  
            let car = reactive({name:'奔驰',price:'40W'})  
            let sum = ref(0)  
            let car2 = readonly(car)  
  
            console.log(isRef(sum))//true  
            console.log(isReactive(car))//true  
            console.log(is_READONLY(car2))//true  
            console.log(isProxy(car))//true  
            console.log(isProxy(sum))//false  
  
            return {...toRefs(car)}  
        }  
    }  
</script>  
  
<style>  
.app{  
    background-color: gray;  
    padding: 10px;  
}  
</style>
```

## 四、Composition API 的优势

## 1.Options API 存在的问题

使用传统OptionsAPI中，新增或者修改一个需求，就需要分别在data, methods, computed里修改。



## 2.Composition API 的优势

我们可以更加优雅的组织我们的代码，函数。让相关功能的代码更加有序的组织在一起。

同一个颜色意思就是我们在一个功能里面需要用到的东西

主要的思想就是：相关的data、method、computed、watch组合成一个hook

## 五、新的组件

Options API		
data		
methods		
computed		
watch		
Composition API		
		<ul style="list-style-type: none"><li>• 在Vue2中: 组件必须有一个根标签</li><li>• 在Vue3中: 组件可以没有根标签, 内部会将多个标签包含在一个Fragment虚拟元素中</li><li>• 好处: 减少标签层级, 减小内存占用</li></ul>
		<h3>2.Telport</h3> <p>Teleport是传送的意思</p> <ul style="list-style-type: none"><li>• 什么是Teleport? —— <code>teleport</code> 是一种能够将我们的组件html结构移动到指定位置的技术。</li></ul>

```
<template>
  <div>
    <button
      @click="isShow = true">点我弹个窗
    </button>
    <!-- 这里的body的意思是移动到body标签里面 -->
    <teleport
      to="body">
      <div v-if="isShow"
        class="mask">
        <div
          class="dialog">
          <h3>我是一个
          弹窗</h3>
          <h4>一些内容
        </h4>
          <h4>一些内容
        </h4>
    
```

<h4>一些內容

```
</h4>
<button
@click="isShow =
false">关闭弹窗
</button>
</div>
</div>
</teleport>
</div>
</template>
```

```
<script>
import { ref } from
"vue";
export default {
  name: "Dialog",
  setup() {
    let isShow =
ref(false);
    return { isShow
  };
  },
};
</script>
```

```
<style>
.mask {
  position:
absolute;
  top: 0;
  bottom: 0;
  left: 0;
  right: 0;
  background-color:
rgba(0, 0, 0, 0.5);
}
.dialog {
  position:
absolute;
  top: 50%;
  left: 50%;
  transform:
translate(-50%,
-50%);
  text-align:
center;
  width: 300px;
  height: 300px;
  background-color:
green;
}
</style>
```

### 3.Suspense

- 等待异步组件时渲染一些额外内容，让应用有更好的用户体验
- 使用步骤：
  - 异步引入组件

```
import {defineAsyncComponent} from 'vue'  
const Child = defineAsyncComponent(()=>import('./components/child.vue'))
```

- 使用 `suspense` 包裹组件，并配置好 `default` 与 `fallback`

```
<template>  
  <div class="app">  
    <h3>我是App组件</h3>  
    <suspense>  
      <template v-slot:default>  
        <child/>  
      </template>  
      <template v-slot:fallback>  
        <h3>加载中.....</h3>  
      </template>  
    </suspense>  
  </div>  
</template>
```

总代码：

App.vue

```
<template>  
  <div class="app">  
    <h3>我是App组件</h3>  
    <suspense>  
      <template v-slot:default>  
        <child/>  
      </template>  
      <template v-slot:fallback>  
        <h3>稍等，加载中...</h3>  
      </template>  
    </suspense>  
  </div>  
</template>  
  
<script>  
  // import Child from './components/child'//静态引入  
  import {defineAsyncComponent} from 'vue'  
  const Child = defineAsyncComponent(()=>import('./components/child')) //异步引入  
  export default {  
    name:'App',  
    components:{Child},  
  }  
</script>
```

```
<style>
    .app{
        background-color: gray;
        padding: 10px;
    }
</style>
```

Child.vue

```
<template>
    <div class="child">
        <h3>我是child组件</h3>
        {{sum}}
    </div>
</template>

<script>
    import {ref} from 'vue'
    export default {
        name:'Child',
        async setup(){
            let sum = ref(0)
            <!--这里可以设置3秒后在进行渲染child这个组件-->
            let p = new Promise((resolve,reject)=>{
                setTimeout(()=>{
                    resolve({sum})
                },3000)
            })
            return await p
        }
    }
</script>

<style>
    .child{
        background-color: skyblue;
        padding: 10px;
    }
</style>
```

## 六、其他

### 1.全局API的转移

- Vue 2.x 有许多全局 API 和配置。
  - 例如：注册全局组件、注册全局指令等。

```

//注册全局组件
Vue.component('MyButton', {
  data: () => ({
    count: 0
  }),
  template: '<button @click="count++">Clicked {{ count }} times.
</button>'
})

//注册全局指令
Vue.directive('focus', {
  inserted: el => el.focus()
})

```

- Vue3.0中对这些API做出了调整：
  - 将全局的API，即：`vue.xxx` 调整到应用实例（`app`）上

2.x 全局 API ( vue )	3.x 实例 API ( app )
<code>Vue.config.xxxx</code>	<code>app.config.xxxx</code>
<code>Vue.config.productionTip</code>	<b>移除</b>
<code>Vue.component</code>	<code>app.component</code>
<code>Vue.directive</code>	<code>app.directive</code>
<code>Vue.mixin</code>	<code>app.mixin</code>
<code>Vue.use</code>	<code>app.use</code>
<code>Vue.prototype</code>	<code>app.config.globalProperties</code>

## 2.其他改变

- `data`选项应始终被声明为一个函数。
- 过渡动画类名的更改（其实就是改了个名字）：
  - Vue2.x写法

```

.v-enter,
.v-leave-to {
  opacity: 0;
}
.v-leave,
.v-enter-to {
  opacity: 1;
}

```

- Vue3.x写法

```
.v-enter-from,  
.v-leave-to {  
  opacity: 0;  
}  
  
.v-leave-from,  
.v-enter-to {  
  opacity: 1;  
}
```

- 移除 `keyCode` 作为 `v-on` 的修饰符，同时也不再支持 `config.keyCodes`
- 移除 `v-on.native` 修饰符（自定义事件上用的）
  - 父组件中绑定事件

```
<my-component  
  v-on:close="handleComponentEvent"  
  v-on:click="handleNativeClickEvent"  
/>
```

- 子组件中声明自定义事件

```
<script>  
  export default {  
    emits: ['close']  
  }  
</script>
```

- 移除 过滤器（filter）

过滤器虽然这看起来很方便，但它需要一个自定义语法，打破大括号内表达式是“只是 JavaScript”的假设，这不仅有学习成本，而且有实现成本！建议用方法调用或计算属性去替换过滤器。