目录

[注意 2](#_Toc82074573)

[综述 2](#_Toc82074574)

[虚拟DOM 2](#_Toc82074575)

[监测数据 2](#_Toc82074576)

[vm实例的生命周期 2](#_Toc82074577)

[Vue UI 组件库 3](#_Toc82074578)

[语法 3](#_Toc82074579)

[Vue模板 3](#_Toc82074580)

[vm实例 4](#_Toc82074581)

[事件 6](#_Toc82074582)

[组件化编程 7](#_Toc82074583)

[脚手架 9](#_Toc82074584)

[动画 9](#_Toc82074585)

[代理服务器 10](#_Toc82074586)

[插件 10](#_Toc82074587)

[vuex 10](#_Toc82074588)

[vue-router 12](#_Toc82074589)

[Vue3 14](#_Toc82074590)

[注意 14](#_Toc82074591)

[组合式（Composition）API 14](#_Toc82074592)

[配置项 16](#_Toc82074593)

[自定义hook函数 16](#_Toc82074594)

[新组件 16](#_Toc82074595)

[异步组件 17](#_Toc82074596)

# **注意**

* 1. Vue所管理的函数不能使用箭头函数，会出现this指向的错误；这些函数内若使用不被Vue所管理的函数，用箭头函数使this指向vm
  2. 对象中的属性是字符串，可以不写 ‘’ ，但属性名中出现 . - 时，必须写完整形式
  3. Vue2中，通过 对象.属性=xx 添加属性、delete 对象.属性 删除属性、数组[xx] = aa 修改数组 时，vue无法捕获，无法响应式修改页面
  4. 若通过app中import ‘xx’的方式引入第三方样式库，会进行严格检查，若有资源缺失，则报错
     1. 需要在public中放入文件，通过index.html引入
  5. 脚手架中import xx from ‘目录’，若目录中只有index.js，则会引入index.js不报错
  6. 脚手架中会把import提前解析
  7. 引入图片的几种方式：需要给图片地址用变量代替
     1. 方法一：require img1 from "xx" 方法二：data中img1:require("xx") 方法三：在created钩子中引入
  8. 路径中的@表示src路径：@/component/xxx
  9. vue中引入图片通过let url = require（图片地址）的形式时，require的参数不能为纯变量，可以是 ‘xx/imgs’+变量

# **综述**

* 1. Vue是一套用于构建用户界面的渐进式js框架，自底而上
  2. MVVM和MVC的区别：
     1. MVC：Model、View、Controller。MVC的各部分通信是单向的，MVC并未真正分离页面与数据逻辑到js中
     2. MVVM：Model、ViewModel、View。MVVM的各部分通信是双向的，MVVM真正将页面与数据逻辑分离到js中
  3. 特点：
     1. 采用组件化模式，提高代码复用率、让代码更好维护；
     2. 声明式编码，让编码人员无需直接操作DOM，提高开发效率
     3. 虚拟DOM+优秀的Diff算法，尽量复用

## 虚拟DOM

* 1. 对比算法：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
     1. DOM的差异比较
  2. 比较规则：
     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,使用之前的真实DOM；若虚拟DOM中内容变了, 生成新的真实DOM，替换页面中之前的真实DOM（仅替换不同的部分）
     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。

## 监测数据

* 1. Vue会检测data中所有层次的数据
  2. 对象内数据：通过setter事件监视，这些数据需要在new Vue时传入或者通过$set添加（Observe对象）
  3. 数组内数据：Vue对数组的push、pop、shift、unshift、splice、sort、reverse方法进行包裹，调用后首先调用原方法，然后更新视图
     1. Vue中修改数组中的某个元素需要用以上方法或$set；filter排序完需要通过赋值替换掉原数组

## vm实例的生命周期

* 1. vm有一系列生命周期回调函数（生命周期钩子），**this指向vm或组件实例对象**
  2. 生命周期：
     1. 初始化生命周期、事件 → **beforeCreate**（）｛｝【此时无法通过vm访问data、methods】 → 初始化数据监测、数据代理
     2. → **created**（）｛｝【可以通过vm访问data、methods】 → Vue解析模板，生成虚拟DOM（内存中） ：
        1. 检测el，若没有则在$mount方法调用后继续 → 是否传入template，若有则渲染模板，若无将el所在标签的外部HTML作为模板
     3. → **beforeMount**（）｛｝【此时对DOM的操作最终都不奏效】 → 将内存中的虚拟DOM转成真实DOM插入页面
     4. → **mounted**（）｛｝【一般在此开启定时器、发送网络请求、订阅消息、绑定自定义事件等初始化操作】 → 进入更新循环
     5. → **beforeUpdate**（）｛｝→ 根据新数据，生成新的虚拟DOM，进行比较后完成页面更新 → **updated**（）｛｝
     6. → 回到更新循环开头，直到调用vm.$destroy → **beforeDestroy**（）｛｝【一般关闭定时器、取消订阅消息、解绑自定义事件】
     7. → 销毁vm实例，清理它与其他实例之间的连接，解绑全部指令和自定义事件监听器 → **destroyed**（）｛｝
        1. 销毁后自定义事件会失效，原生DOM事件依然有效；一般不在beforeDestroy操作数据，因为不再触发更新流程
  3. vm.$nextTick（回调函数） 在下一次DOM更新后执行回调

1. Vue UI 组件库：PC端：Element UI、IView UI；移动端：Vant、Cube UI、Mint UI
   1. 按需引入：见官方文档

# **语法**

使用：创建main.js，引入App.vue，并在App中实例化Vue；在index.html中写好根节点，在body的最后引入vue.js和main.js

## 模板语法

* 1. 本质是合法的HTML代码，但能被Vue用于将DOM绑定至底层数据
  2. 插值：
     1. <h1>Hello, {{ name }}</h1> Mustache语法（双大括号），会将双大括号标签替代为对应数据对象上属性的值
        1. 双大括号内可以使用js表达式，此时双大括号被替代为表达式的值
  3. 内置指令：用于标签内，如<div v-bind：xx=“xxxx”>
     1. v-bind：特性="表达式" 或 ：特性=“表达式” 绑定表达式作为特性的值，单向数据绑定
        1. 为class特性绑定：
           1. [‘class1’。。。] 传入字符串数组，会将元素所对应的类应用于该标签
           2. ｛class1：布尔。。。｝ 传入对象，将值为true的类应用于标签
           3. [‘class1’,｛class2：布尔。。。｝。。。] 复合使用
        2. 为style特性绑定：
           1. ｛小驼峰css属性：‘值’｝ 传入对象，若值是变量且与属性同名，可以不写值
           2. [｛。。。｝。。。] 传入对象数组
     2. v-model：特性=“表达式” 双向数据绑定，是语法糖，只能应用在表单类元素上
        1. 简写：v-model：“表达式” 指定value特性
        2. 本质：v-bind: value="xxx" v-on: input="xxx = $event.target.value"
        3. 表单的type相关：
           1. 若type=“text"，则v-model收集的是value值，用户输入的就是value值
     3. 若type="radio"，则v-model收集的是被勾选单选框的value值（需要给单选框value特性）。
        + 1. 若type="checkbox"：

若绑定变量是非数组，那么收集的就是checked特性值（布尔）

若绑定变量是数组，则数组的值由 所有勾选元素的value值的字符串形式 组成

* + - 1. 修饰符：加在v-model的后面来实现指定功能，可以写多个串联，执行顺序从左到右
         1. .lazy【失去焦点再进行数据同步】 .number【输入字符串转为有效的数字】 .trim【输入首尾空格过滤】
    1. v-on：事件=“表达式” 或 @事件=“表达式” 绑定事件，函数不传参时可不写（），传参则需指定第一个参数为$event
       1. 事件修饰符：加在事件名的后面，可以写多个，执行顺序从左到右
          1. .prevent 阻止默认行为 .stop 阻止冒泡 .once 事件只会触发一次 .capture 使用事件的捕获模式（捕获时执行）
          2. .self 只有触发元素是绑定元素时才执行 .passive 事件的默认行为在回调执行前执行 .native 指定为原生事件
       2. 绑定键盘事件：见“事件”
    2. v-show=“表达式” 条件渲染，若表达式的值为false，则隐藏元素（display：none）
    3. v-if=“表达式” / v-else-if=“表达式” / v-else 条件渲染，若表达式的值为false，则删除元素
       1. 操作逻辑同js，if、elseif、else所绑定的元素必须在一块，之间不能有其他元素
       2. template元素只能与v-if配合，不能与v-show配合
    4. v-for=“形参 / （形参，index） in 变量” 列表渲染，需要key属性作为li的唯一标识符（在ul中唯一即可）
       1. 遍历数组：<li v-for=“（a，index）in 数组” :key=“index”></li> a为数组内的元素，index为索引值，从0开始
       2. 遍历对象：<li v-for=“（value，key）in 对象” :key=“key”></li> value值，key属性名
       3. 遍历字符串：<li v-for=“（char，index）in 字符串” :key=“index”></li>
       4. 遍历指定次数：<li v-for=“（num，index）in 整数n” :key=“index”></li> 遍历n次，num从1开始每次递增1
       5. 使用index作为key会在破坏数据顺序时出现问题，**尽量使用每条数据的唯一标识作为key**
    5. v-text=“表达式” 用返回值替换标签内容
    6. v-html=“表达式” 功能同上，支持HTML结构解析，存在cookie泄露的问题
    7. v-cloak css中：[v-cloak]｛display：none｝ 用于防止网速慢时出现未经解析的标签，vue会在解析后移除属性
    8. v-once 所在节点在初次动态渲染后，就视为静态内容了
    9. v-pre 跳过所在节点的编译，可用于加快编译
  1. 特性：HTML或组件标签上的特性
     1. ref=“xx” 给被接管元素或子组件注册引用信息，通过vm.$refs.xx获取真实DOM元素或组件实例对象vc

## Vue实例

* 1. Vue实例并不完全是视图模型（ViewModel，参考MVVM模型），但实例以vm命名
  2. 创建：const vm = new Vue（｛选项｝），可选项如下：
     1. el：‘选择器’ 或 DOM节点 仅在new创建实例时生效。指定一个已存在的元素作为实例的挂载目标
     2. template：`HTML代码` 指定模板，只能有一个根节点，将模板内容直接替换接管的容器
     3. render： h => h(App) 见脚手架
     4. components：｛‘组件名’：组件｝ 注册组件
     5. data：｛属性：值｝ 或 data（）｛返回一个对象｝ 存储数据，对象中的属性都会通过数据代理被绑定到vm上
        1. 数据绑定通过Object.defineproperty实现，调用/修改vm.属性时，getter/setter从vm.\_data.属性中获取/修改
        2. vm中的\_data属性值就是传入的data属性值，可以通过 vm.\_data.属性 进行更改
     6. methods：｛方法名（）｛｝｝ 为实例添加方法，没有数据代理
     7. computed：｛属性：｛get（）｛｝ set（value）｛｝｝｝ 计算属性，也是实例的属性
        1. 简写：computed：｛属性名（）｛｝｝ 若没有setter，可以采用这种简写，函数体视为getter
        2. get函数会在初次读取属性与所依赖数据变化时被调用，并且有缓存机制
     8. watch：｛属性：｛配置项｝｝ 监视属性，指定属性被修改时的动作
        1. immediate：false 设置是否在初始化时调用handler
        2. deep：false 深度监视，开启后，若属性为多级数据，属性内部发生改变后也调用
        3. handler（newValue，oldValue）｛｝ 当属性被修改时调用函数
        4. 简写：watch：｛属性（newValue，oldValue）｛｝｝ 当只需要handler时，可采用简写
        5. 计算属性与监视属性：监视属性可以开启异步任务，且不依赖return（直接修改其他属性的值）；若都能实现，计算属性性能好
     9. filters：｛过滤器名（value，参数1）｛｝｝ 局部过滤器（在当前实例中有效），可用于｛｛｝｝和v-bind
        1. 通过{{属性1 | 过滤器名（参数1）}}使用，首先读取属性，然后作为value参数传给过滤器，过滤器返回值作为表达式的值
        2. 可以通过｛｛属性名 | 过滤器1 | 过滤器2｝｝实现串联
     10. directives：｛对象或函数式｝ 自定义指令，指令名用全小写，通过v-指令名调用，内部函数this为window
         1. 对象内的属性：指令名：｛钩子函数｝ 可以指定任意多个钩子函数作为属性
            1. 钩子函数：

bind（）｛｝【指令第一次绑定到元素时调用】，inserted（）｛｝【被绑元素插入父节点时调用】，

update（）｛｝【所在组件VNode更新时调用】，unbind（）｛｝【指令与元素解绑时调用】，

componentUpdated（）｛｝【指令所在组件的VNode及其子VNode全部更新后调用】

* + - * 1. 钩子函数的参数：**element，binding**，vnode，oldVnode

element：指令所绑定的DOM元素 binding：一个对象，包含name、value、oldValue等属性

* + - 1. 函数式：指令名（element，binding）｛｝ 指令与元素成功绑定时（bind）或所在模板被重新解析时（update）调用
    1. 生命周期函数名（）｛｝ 在创建vue实例的某个时刻调用
  1. 实例属性：
     1. vm.$refs 获取数组，内含有ref特性的真实DOM元素或组件实例对象vc
     2. vm.$attrs 组件从外部传入未被props接收的属性保存在该属性中，无法做类型限制
     3. vm.$slot 组件从外部传入的插槽内容保存在该属性中
  2. 实例方法：
     1. vm.$mount（‘选择器’） 指定el的值
     2. vm.$watch（‘属性’，｛配置项｝） 添加监视属性
        1. vm.$watch（‘属性’，fn（newValue，oldValue）｛｝） 当只需要handler时，可采用简写
     3. vm.$set（vm.\_data.目标，‘属性’，‘值’） 向目标身上添加响应式属性（\_data. 可省略），可用于数组元素的修改
        1. 目标不能是vm或vm的根数据对象（\_data）
     4. vm.$delete（vm.\_data.目标，‘属性’，‘值’） 从目标身上移除属性（响应式），同上
     5. vm.$destroy（） 执行vm实例的销毁流程
     6. vm.$emit（‘事件名’） 触发当前实例上的指定事件
     7. vm.$on（‘事件名’，回调函数） 监听当前实例上的自定义事件
     8. vm.$once（‘事件名’，回调函数） 监听当前实例上的自定义事件，只触发一次
     9. vm.$off（‘事件名’ 或 [‘事件名’]） 解绑自定义事件，若不传参，表示全解绑
     10. vm.$nextTick（回调函数） 在下一次DOM更新后执行回调
  3. Vue的方法：
     1. Vue.set（vm.\_data.目标，‘属性’，‘值’） 同vm.$set
     2. Vue.filter（‘过滤器名’，fn（）｛｝） 创建全局过滤器，需在创建vm实例前绑定
     3. Vue.directive（‘指令名’，函数或配置对象） 创建全局自定义指令，在创建vm实例前绑定
     4. Vue.component（‘组件名’，组件） 全局注册组件
     5. Vue.mixin（xx） 全局混入
     6. Vue.use（插件对象） 应用插件

## 事件

* 1. 绑定事件到按键：@keyup/keydown.按键别名=“表达式”
     1. 别名：enter、delete、esc、space、tab、up、down、left、right、caps-loack。。。
     2. tab在按下时会默认切换到下一个元素，因此只能配合keydown使用，不能配合keyup
     3. ctrl、alt、shift、meta绑定keyup时，按下后再按下其他键，释放其他键后才能触发；配合keydown正常使用
     4. 上述四键可以在按键别名后追加 .按键别名2 ，表示同时按下两键时触发
     5. Vue.config.KeyCodes.自定义键名=键码 自定义键名【不推荐】
  2. 组件自定义事件：可用于子组件给父组件传递数据
     1. 给子组件绑定事件的本质是绑定到其实例上。
     2. 触发事件（子组件中）：this.$emit（‘事件名’[,参数]） 参数会作为回调函数的入参
     3. 监听（绑定）自定义事件：
        1. 方法一：父组件中的子组件标签中添加 @事件名=“回调函数”
        2. 方法二：父组件中this.$refs.子组件.$on（‘事件名’，回调函数） 更灵活，可以设置异步；若想只触发一次，使用$once
     4. 解绑自定义事件：子组件实例.$off（‘事件名’ 或 [‘事件名’,…]） 不传参表示全解绑
     5. 给组件绑定DOM原生事件时，默认视为自定义事件，除非添加 .native 修饰符
  3. 全局事件总线：用于任意组件间通信
     1. 创建：在Vue实例的创建中添加选项：beforeCreate（）｛Vue.prototype.$bus=this｝
     2. 使用：
        1. 发送数据组件中：mounted（）｛this.$bus.$on（‘xx’，回调）｝ beforeDestroy（）｛this.$bus.$off（‘xx’）｝
        2. 接收数据组件中：this.$bus.$emit（‘xx’[，参数]）
  4. 消息订阅与发布：利用pubsub-js库（适用任何框架），用于任意组件间通信
     1. 引入库：import pubsub from ‘pubsub-js’
     2. 订阅消息：
        1. mounted（）｛this.pubId=pubsub.subscribe（‘消息名’，fn（msgName，参数...）｛｝）｝
        2. beforeDestroy（）｛pubsub.unsubscribe（this.pubId）｝
     3. 发布消息：pubsub.publish（‘消息名’[，参数]）

## 组件

* 1. 说明：
     1. 传统方式编写应用相较组件化编程：依赖关系混乱，不好维护；代码复用率不高
     2. 组件化编程流程：实现静态组件 → 展现动态数据 → 交互绑定
     3. 组件是实现应用中局部功能代码和资源的集合。组件中函数的this指向VueComponent实例对象（vc，也称组件实例对象）
     4. 重要的内置关系：VueComponent.prototype.\_ \_proto\_ \_ === Vue.prototype
     5. fn VueComponent（options）：是Vue.extend的返回值，是一个构造函数；若写了组件标签，Vue会执行new VueComponent（options）；
        1. 每次调用Vue.extend，返回的都是全新的构造函数；vc的属性与vm大致相同（vc无el）
  2. 非单文件组件：一个文件中包含多个组件
     1. 创建组件：const xx = Vue.extend（｛选项｝） 【简写：const xx = ｛选项｝】
        1. 大部分选项同Vue（｛选项｝），但没有el属性，有name属性，data属性的值必须是函数式，可用template配置组件结构
        2. name：‘abc’【组件名可采用 短横线分隔命名（推荐） 或 大驼峰命名】
     2. 注册组件：
        1. 局部注册：Vue或组件实例对象添加属性 components：｛‘组件名’：xx｝
        2. 全局注册组件：Vue.component（‘组件名’，xx）
     3. 编写组件标签：HTML中：<组件名>...</组件名> 或 <组件名/>（脚手架环境下）
        1. 组件名：使用短横线分隔命名【注册的组件名若为大驼峰，也要在DOM中使用时改为短横线】 或 大驼峰（仅在脚手架环境中）
  3. 单文件组件：一个文件中包含一个组件。文件名：xxx.vue（文件名规则与组件名一致，一般用大驼峰）
     1. 文件中有三个标签：<template></template ><script></script><style></style>
        1. 三个标签分别代表组件的结构、组件交互代码、组件的样式。注释的方式同html、js、css
        2. script中需要暴露组件；若使用extend的简写模式，最好写name属性
  4. 组件的嵌套：子组件在父组件中注册，且要在父组件创建之前创建；编写组件标签也要在父组件中
     1. **创建app组件，将所有组件注册为它的子组件，管理所有组件**
  5. 组件接收外部数据（props）：**这些数据是只读的，若需修改，复制到data中再修改副本**
     1. 传递数据：<组件名 属性=“xx” 属性=“xx”。。。> **此时属性都为字符串型**，可用 :属性 传递其他类型
     2. 接收数据：组件实例选项中添加props：值 数据将被存放在组件实例中
        1. 简单声明接收：props:['name']
        2. 限制类型：props:{name:String}
        3. 限制类型、限制必要性、指定默认值：props：｛name：｛type：String，required：true，default：‘xx’｝｝
  6. 子组件向父组件发送数据：父组件定义函数receive（x）｛｝，将函数传给子组件，子组件使用函数传递
  7. mixin（混入）：把多个组件共用的配置提取成一个混入对象
     1. 定义混合：新建js文件，export const xx = ｛ data(){}, methods:{}。。。｝
     2. 使用混入：先将js文件引入，Vue.mixin（xx）【全局混入】 组件中mixins：[xx]【局部混入】
     3. 当混入与配置项冲突时，以配置项为主，但生命周期钩子冲突时，混入与配置项都会生效
  8. 组件插槽：用于父组件向子组件的指定位置放入HTML结构
     1. 默认插槽：若父组件中的**子组件标签体**内有结构，会将这些结构解析到该子组件内template中的所有<slot></slot>处
        1. <slot>默认值</slot>：当使用者没有传递结构时，显示默认值
        2. 样式可以写在组件标签体所在父组件的style中，也可以写在组件内的style中
     2. 具名插槽：子组件标签体内，指定元素添加slot=“xx”；组件内的slot标签添加name=“xx”
        1. 会把子组件标签体内 所有slot属性为xx的元素 解析到 子组件内name值为xx的slot标签处
        2. <template slot=”xx”>可以简写为<template v-slot：footer>，仅对template元素有效（使用v-slot语法，slot语法已被废弃）
     3. 作用域插槽：用于子组件插槽向父组件传递数据
        1. 子组件的slot标签添加属性 :变量1=“变量1” 子组件标签体内放置<template slot-scope =“变量2”></template>，所有结构放于template中。此时 变量1 被传为 变量2 的属性，template内的结构可以通过 变量2.变量1 使用，也能用解构赋值｛xx｝代替aa
        2. scope=“xx” 同 slot-scope=“xx” 后者为新api

## 动画

* 1. 使用：
     1. css中：.v-enter-active｛｝ .v-leave-active｛｝ animation属性添加于这两个选择器中
     2. html中：将需要添加动画的元素加上v-show属性，并包裹在<transition></transition>标签中，在隐藏/显示前播放动画
  2. transition标签：内部只能有一个元素，特性如下：
     1. tag=“xx” 该标签会被解析成xx标签，不写默认解析为span
     2. name=“xx” 定义动画时需用 .xx-enter-active｛｝，若不写该特性，则需用.v-enter-active｛｝
     3. appear 在页面加载时播放进入动画
     4. enter/leave-active-class=“xx” 自定义过渡的类名，配合第三方库使用
  3. 过渡：定义.v-enter｛｝、.v-enter-to｛｝、.v-leave｛｝、.v-leave-to｛｝ 分别表示进入的起点/终点、离开的起点/终点
     1. 内部用transfrom设置位置，需要添加transition属性于.v-enter/leave-active｛｝或元素上
  4. transition-group标签：用于包含多个元素，属性同transition，内部元素需要有值唯一的key属性

## 代理服务器

* 1. 用于解决跨域问题。原理：添加一个代理服务器在网页所在端口，由代理服务器向真实服务器发起请求，获得数据后转交vue（代理服务器不是浏览器，没有同源策略）。项目根目录/public中的所有文件都可以通过 网页所在url/文件名 访问
  2. 添加单个url代理：vue.config.js中添加属性：devServer：｛proxy：‘服务器url’｝ 开启后要向网页所在端口发送请求
     1. 当代理服务器可以找到文件时，不会向目标服务器发送请求；请求直接发给前端（8081）即可
  3. 添加多个url代理：vue.config.js中添加属性：devServer：｛proxy：｛选项｝｝
     1. ‘/xxx’：｛target：‘目标服务器url’，pathRewrite：｛‘^/xxx’：‘’｝，ws：true，changeOrigin：true｝或‘/foo’：｛target：‘url’｝
        1. ‘/xxx’为请求前缀，放于端口号后方，当请求附带请求前缀时，才代理发送请求
        2. pathRewrite可以在转发请求时清掉请求前缀；ws为是否支持websocket；changeOrigin控制请求头中的host值为目标服务器
        3. ws与changeOrigin默认值为true，可不写
     2. 使用时向后端发送请求，通过是否加请求前缀表示是否转发请求

# Vue CLI

* 1. Vue CLI（command line interface），即Vue脚手架。安装：npm install -g @vue/cli
  2. 注意：
     1. <%= BASE\_URL %>xx.xx 表示public目录下的xx.xx（仅public/index.html或其它通过html-webpack-plugin作模板的HTML文件有效）
     2. import Vue from "vue"; 引入精简版的vue，无法解析template属性，需用render。template标签可以解析
     3. import Vue from "vue/dist/vue"; 引入完整版vue
     4. <style scoped> 让样式在局部生效，防止冲突
        1. 使用>>>【::v-deep】实现样式穿透，less使用/deep/
     5. <style lang=“less”> 指定样式所用的语言（需要安装loader），默认为css
  3. 重要命令：
     1. 指定目录下执行 vue create 项目名 创建项目
     2. 项目根目录后执行 npm run serve 运行服务
     3. npm run build 编译为html、css、js文件
        1. 编译后默认需要在服务器上运行，可用以下方法解决：
           1. 使用serve库创建服务器；更改vue.config.js中的publicPath为相对路径“./”
     4. vue inspect > output.js 导出webpack配置
  4. 通过render解析模板：
     1. render（createElement）｛return createElement（‘HTML标签名’[，‘内容’] 或 组件实例）｝
     2. 创建App的简写：render： h => h(App)
  5. vue.config.js文件：可选的配置文件，放于项目根目录，详见文档
     1. 通过commonjs的形式导出对象（module.exports=｛选项｝），与webpack的配置文件整合
        1. pages：配置入口等 lintOnSave：false关闭语法检查
  6. 设置全局less文件：npm i style-resources-loader vue-cli-plugin-style-resources-loader -D
     1. vue.config.js中添加属性：
        1. pluginOptions: { 'style-resources-loader': {preProcessor: 'less',
        2. patterns: [path.resolve(\_\_dirname, "src/assets/less/variable.less")] // 指定路径
        3. }}

# **插件**

1. 插件：用于增强Vue，本质是包含install方法的一个对象
   1. 定义插件：｛install（Vue，options）｛｝｝ install中可以添加全局过滤器、全局指令、配置全局混合、添加实例方法等
   2. 应用插件：Vue.use（插件对象，参数1，参数2。。。） 若有多个参数，将参数传给options 需要在Vue实例化之前调用
2. vue-resource：ajax插件
   1. 引入：import vueResource from ‘vue-resource’ vue.use（vueResource）
   2. 发送请求：this.$http.get（‘url’） 返回一个Promise对象，同axios.get（）

## Vuex

* 1. 用于在Vue中实现集中式状态（数据）管理。适用于：多个组件依赖于同一状态；来自不同组件的行为需要变更同一状态
  2. 工作原理：
     1. Vue组件 → Actions：通过dispatch（‘动作名’，值）
        1. Vue组件可以跳过Actions，通过调用commit（‘动作名’，值）直接到达Mutations
        2. Actions：用于响应组件中的动作，是一个对象，一般用于与后端API通信。可以包含任意异步操作
     2. Actions → Mutations：通过commit（‘动作名’，值）
        1. Mutations：用于操作数据，是一个对象，可以与开发者调试工具通信。用于直接更新state中的数据
     3. Mutations → State：Mutate自动实现
        1. State：用于存储数据，一个对象
     4. State → Vue组件：Render
  3. 引入：npm i vuex
     1. src中创建store目录，store中创建index.js（用于创建store）：
        1. import Vue from ‘vue’ ； import Vuex from ‘vuex’ ； Vue.use（Vuex）
        2. const actions = ｛｝ ； const mutations = ｛｝ ； const state = ｛｝ ； [const getters = ｛｝]
        3. export default new Vuex.Store（｛actions，mutations，state [， getters] ｝）
     2. main.js中：import store from ‘./store/index.js’ 、 Vue实例化时添加选项store：store
  4. store：用于管理Actions、Mutations、State。 actions、mutations等都是store上的属性
     1. actions配置：
        1. 动作名（context，value）｛｝ store.dispatch（）后会调用函数，获取的context是精简版store的实例
           1. 函数内需调用context.commit（‘mutations中对应的动作名’，value）
           2. 该函数内可以通过调用dispatch触发其他动作
     2. mutations配置：
        1. 动作名（state，value）｛｝ mutations中的动作名一般用全大写，函数内对属性进行操作
     3. state配置：属性：值 可以通过 state.属性 访问对应属性
     4. getters配置：可选配置，用于将state中的数据进行加工
        1. 属性名（）｛return xx｝ 类似计算属性，通过getters.属性名 访问
  5. mapState与mapGetters：mapGetters将下面写法中mapState替换即可
     1. 引入：组件中：import｛mapState｝ from ‘Vuex’； computed：｛...mapState（｛属性名：‘state中的属性名’｝）｝
        1. 若mapState声明属性名同state中的属性名同名，可用数组形式：...mapState（[‘属性名’]）
     2. 使用：属性名会被添加到计算属性中，值同state中的指定属性名
  6. mapMutations与mapActions：
     1. 引入：组件中：import｛mapMutations｝ from ‘Vuex’； methods：｛...mapMutations（｛方法名：‘Mutations中的动作名’｝）｝
        1. 当方法名同动作名时，可用数组形式：...mapMutations（[‘方法名’]）
     2. 使用：调用 方法名（value） 时，自动调用commit（‘mutations中的动作名’，value）
  7. 模块化编码：const 模块1 = ｛namespaced：true，actions：｛｝，mutations：｛｝。。。｝const 模块2 = ｛。。。｝【一般写在不同文件引入】
     1. 创建并暴露时：export default new Vuex.Store（｛modules：｛模块1：模块1，模块2：模块2｝｝）
     2. 使用时：
        1. state中没有模块内的数据，需要通过 模块.属性 进行访问：
           1. ...mapState（‘模块1’，[‘模块1内的属性’。。。]） 使用时无需通过 模块.属性 访问，需要开启模块的namespaced属性
           2. ...mapMutations（‘模块1’，｛方法名：‘模块1内的事件名’｝） 同上
        2. 不使用map时，getters中的属性名、mutations和actions中的事件名变为：模块名/事件名
           1. commit（‘模块1/事件名’，value） 不建议直接写，建议用map写
           2. dispatch（‘模块1/事件名’，value）
           3. this.$store.getters.模块名/属性名
  8. mutation-types.js文件：export const INCREASE = 'INCREASE' ，在mutation.js文件中引入，通过[INCREASE]（）｛｝的方式定义函数

## Vue-Router

* + 1. 用于实现SPA应用 npm i vue-router 路由组件一般放在src/pages中
    2. 引入：
       1. src目录中创建目录router，router中创建index.js：
          1. import VueRouter from ‘vue-router’
          2. import 组件 from ‘路径’
          3. export default new VueRouter（｛选项｝）

mode：‘history’ 指定模式，默认为hash模式

hash模式：通过url中的#实现路由跳转，且不会将数据传到后端

history模式：利用HTML5中的pushState()和replaceState()方法。它们虽然改变了当前的URL，但不会立即向后端发送请求。该模式直接刷新页面时，仍会向后端传递请求，需要在后端进行配置以确保正常使用。

routes：[｛选项｝，。。。]

name：‘xx’ 可选，命名路由，在router-link的to特性的对象写法中不写path，写name：‘xx’即可跳转

path：‘/xx’ 路由路径

component：组件 对应路由组件

redirect：‘/xxxx’ 可选，自动跳转到指定路由

* + - 1. main.js中：import VueRouter from ‘vue-router’ Vue.use（VueRouter） import router from './router'
      2. Vue实例化时添加选项router：router
    1. 使用：
       1. <router-link to=“/xx”> </router-link> 本质是a标签，跳转到指定路径
          1. active-class=“xx” 仅router-link标签可用，标签被选中时自动添加class
          2. :replace=“true” 或 replace 开启历史记录的replace模式，仅该标签有效
       2. <router-view></router-view> 指定路由组件在该组件中的位置，无需引入路由组件
    2. 路由组件的实例属性：
    3. $router：全局的路由实例，是router构造方法的实例
    4. $route：当前路由信息对象。表示当前激活的路由的状态信息，其属性有：
       1. path【当前路由的路径】、params【一个对象，包含 动态片段 和 全匹配片段】、query【对象，表示URL查询参数】、hash【当前路由的hash值（不带#）】、fullPath【完成解析后的URL，包含query和hash】、matched【数组，包含当前匹配的路径中所包含的所有片段所对应的配置参数对象】、name【路径名称】、meta【路由元信息】
    5. 嵌套（多级）路由：
       1. router/index.js中：routes中的一级路由中添加children：[｛二级路由配置项｝]
          1. 二级及后面的路由的路径不需要以/开头，系统会自动解析出来
       2. template中：router-link中的to特性需要写 “/一级路径/xx”
    6. 路由传参：
       1. query方式：
          1. to的字符串写法：template中：router-link中 :to=“ `/一级/二级/三级？参数1=$｛xx｝&参数2=$｛xx｝` ”
          2. to的对象写法：template中：router-link中 :to=“｛path：‘/一级/二级/三级’，query：｛参数1：xx，参数2：xx｝｝”
          3. 路由组件中：通过 this.$route.query.参数1 访问
       2. params方式：该方式中的to的对象形式中只能用name不能用path
          1. to的字符串写法：:to= “` /一级/二级/三级/$｛值1｝/$｛值2｝`”
          2. to的对象写法：:to= “｛name：‘三级路由的name’，params：｛参数1：值1，参数2：值2｝｝”
          3. routes中的三级路由的path：“三级/：属性1/：属性2”
          4. 三级路由组件中：通过 this.$route.params.参数1 访问
       3. props：用于在路由组件中接收简写的属性
       4. 在**对应路由**（routes）中添加props属性：
          1. 对象写法：props：｛属性：值｝
          2. 布尔写法：props：true 若为真，把路由组件收到的所有params参数，传给路由组件
          3. 函数写法：props（$route）｛return｛属性：值｝｝
          4. 接收参数：路由组件通过 props：[‘属性’] 接收，参考组件实例的props使用
    7. 编程式路由导航：不借助router-link的路由导航
       1. this.$router.push（｛同to的对象写法｝） 以push的方式访问指定路径
       2. this.$router.replace（｛同to的对象写法｝） 以replace的方式访问指定路径
       3. this.$router.back（）/ forward（）/ go（整数n） 回退到上一个历史记录 / 前进到下一个历史记录 / -n表示后退n步
    8. 缓存路由组件：使组件在切换到其他路由时不被销毁
       1. <keep-alive include=“组件名”><router-view></router-view></keep-alive > 仅指定组件不销毁，不写include则全部组件都不销毁
    9. Vue实例的两个生命周期钩子：activated（）｛｝路由组件激活时触发 deactivated（）｛｝路由组件失活时触发
    10. 路由守卫：用于权限判断，接收实例对象router = new VueRouter（。。）
        1. 全局前置路由守卫：router.beforeEach（fn（to，from，next）｛｝） 每次路由切换之前和初始化时被调用
           1. to为对象，包含目标路由的信息；from为对象，包含原路由的信息；调用next（）进行跳转
           2. 若在路由配置对象中，添加meta【路由元信息】：｛isAuth：true｝，可以通过if（to.meta.isAuth）判断是否需要判断权限
           3. meta中可以存放任意信息
        2. 全局后置路由守卫：router.afterEach（fn（to，from）｛｝） 每次路由切换之后和初始化时被调用
        3. 独享路由守卫：在路由配置对象中添加beforeEnter：fn（to，from，next）｛｝ 没有后置路由守卫
        4. 组件内路由守卫：组件实例化时传入选项beforeRouteEnter（to，from，next）｛｝beforeRouteLeave（to，from，next）｛｝
        5. 执行顺序为：beforeEach、beforeEnter、beforeRouterEnter
  1. 路由懒加载：仅在需要时加载路由
     1. router/index.js中，不在开头引入路由组件，而是指定路由对象的component属性为（）=>import（‘url’）

# **Vue3**

## 综述

* 1. 注意：
     1. 具名插槽的父组件template标签内中指定名字需要用 v-slot：xx 的形式
     2. setup可以声明为async函数，但需要异步组件和Suspense配合
  2. 相对Vue2的改变：
     1. 全局API：大部分通过Vue.xx实现的API改为app.xx ； Vue.prototype改为app.config.globalProperties
     2. 模板结构可以没有根标签；Vue.config.productionTip被移除；data选项应始终被声明为一个函数
     3. .v-enter改为.v-enter-from；移除keyCode作为v-on的修饰符，也不再支持config.keyCodes
     4. 移除v-on.native修饰符：父组件通过v-on：close=‘xx’ 绑定事件，子组件vc中通过emits：[‘xx’] 声明事件，没声明的视为原生事件
     5. 移除过滤器filter，建议用方法调用或计算属性实现
  3. 搭建工程：
     1. 使用vue-cli创建（同vue2，见脚手架）
     2. 使用vite创建： npm init vite-app 项目名 cd 项目名 npm install npm run dev
  4. 引入：main.js中：
     1. import ｛createApp｝ from ‘vue’
     2. import App from ‘./App.vue’
     3. createApp（App）.mount（‘#app’） 创建应用实例对象app，类似于轻量化Vue实例对象
  5. 生命周期钩子：beforeDestroy改为beforeUnmount、Destroyed改为Unmounted，其他不变。可以通过配置项或组合式api的形式使用
  6. app的方法：
     1. app.mount / unmount（‘选择器’） 挂载/卸载指定元素到app中

## 组合式API

* 1. 组合式（Composition）API需写在setup中，这些api需要通过import引入
  2. 组合式API相对传统API的优势：将相关功能的代码有序地组织在一起，更好维护
  3. setup（props，context）｛｝ 是组合式api的入口，在beforeCreate之前执行，this值为undefined
     1. props为该Vue实例对象props属性的Proxy代理对象，context为上下文对象：
        1. .attrs 同Vue2中的$attrs .slots 同Vue2中的$slot
        2. .emit（‘自定义事件名’，参数） 可以触发该组件的自定义事件，vc上需要有emits属性
  4. 数据与方法：
     1. 非响应式数据：将原data、methods中的内容以let x = xx、function xx（）｛｝的形式写入其中
        1. 通过直接赋值修改值；模板中直接通过x获取值
     2. 响应式数据：
        1. 通过**ref（）**函数：let a =ref（xx），a的值为一个引用对象（RefImpl）
           1. 通过 x.value=xxx 修改值（对象类型：x.value.属性=xxx）
           2. xx若为基本类型数据，响应式仍是靠defineProperty的getter与setter完成的
           3. xx若为对象类型数据，内部使用reactive函数实现，a仍是一个RefImpl对象，但其value的值为Proxy对象
        2. 通过**reactive（）**函数：let x【代理对象】 = reactive（对象/数组类型数据【源对象】），x的值为Proxy对象
           1. 通过 x.属性=xxx / x[0]=xxx 修改属性值；通过代理对象对源对象内部数据进行操作
        3. **toRef**（对象，‘属性名’） 将对象的指定属性包装为Ref对象并返回，改变Ref对象的值时对象内的原属性也会改变
        4. **toRefs**（对象） 功能同上，对对象内的所有属性有效，返回一个对象｛属性名：对应Ref对象。。。｝
     3. 计算属性：let x = **computed**（｛get（）｛｝，set（value）｛｝｝）
        1. 若不需要setter，可用简写形式let x = computed（fn（）｛return xxx｝）
     4. 监视属性：
        1. **watch**函数： watch（属性，fn（newValue，oldValue）｛｝，｛配置选项｝） 属性需要是代理对象
           1. 监视多个：watch（[属性]，fn（newValue，oldValue）｛｝，｛...｝） newValue为数组，含函数绑定的所有属性的值
           2. 监视ref定义的对象类型数据时：默认不开启深度监视，可通过 属性.value 把Proxy对象给watch；或开深度监视
           3. 监视reactive定义的响应式数据：无法正确获取oldValue，可把指定数据写在对象外；强制开启深度监视（不能关）
           4. 监视reactive定义的响应式数据内的属性时：原属性处需要一个函数或函数数组，这些函数返回值为要监视的属性

若监视reactive定义的响应式数据内的属性为对象时，默认不开启深度监视，也无法正确获取oldValue

* + - 1. **watchEffect**函数：watchEffect（fn（）｛｝） 在函数体中使用的属性被改变时，调用回调函数
  1. 生命周期钩子：beforeCreate和Created没有组合式API 组合式API声明的钩子优先级高于配置项声明的钩子
     1. 其他生命周期钩子：**onBeforeMonted、onMonted、onBeforeUpdate、onBeforeUpdated、onBeforeUnmount、onUnmounted**
     2. 使用：onBeforeMounted（fn（）｛｝）
  2. setup的返回值：
     1. 若返回一个对象return ｛属性1，方法名1｝，对象中的属性可以在模板中直接使用；
     2. 若返回一个渲染函数return （）=> h（‘标签名’，‘内容’），此时模板会失效，只看渲染函数，需要import ｛h｝ from ‘vue’
  3. 其他组合式API：
     1. shallowReactive（） 用法同reactive（），但只考虑对象类型的最外层属性的响应式
     2. shallowRef（） 用法同ref（），但不处理对象内部属性的响应式
     3. readonly / shallowReadonly （响应式数据） 返回值类似传入值，但它只读，shallowReadonly只考虑对象最外层属性
     4. toRaw（响应式对象） 将对象转换为普通对象并返回，仅对reactive生成的数据有效
     5. markRaw（响应式对象） 将对象转换为普通对象并返回，并且该对象不会再被转换为响应式
        1. 有些值不应该被设计为响应式（复杂第三方库），这些值添加到响应式对象时，一般会自动转换为响应式，用此方法阻止
     6. 自定义Ref：fn 函数名（value）｛return customRef（fn（track,trigger）｛return｛get（）｛return value｝，set（newValue）｛｝｝｝）｝
        1. 使用时调用 函数名（数据），get中需在return前调用track（）来追踪数据变化，set中需调用trigger（）来重新解析模板
     7. provide（‘属性名’，值） / inject（‘属性名’） 在组件中调provide，在其后代组件中调inject，inject会返回数据的值
     8. isRef / isReactive / isReadonly / isProxy（） 判断类型，前三者检测是否由指定函数生成

## 配置项

* 1. Vue实例化时可添加如下属性：
     1. data（）｛｝ / methods：｛｝ / computed：｛｝ / watch：｛｝ 可选，同Vue2
        1. 尽量不与setup混用，data。。。可访问到setup中的数据，setup不能访问data。。。中的数据；如有重名，以setup优先
     2. props：[] 同Vue2
     3. emits：[‘自定义事件名’] 声明该组件上的自定义事件，不写可以跑但是有警告
  2. 响应式原理：
     1. ref通过Object.defineProperty（）的get与set实现响应式（数据劫持）
     2. reactive通过Proxy实现响应式（数据劫持），并通过Reflect操作源对象内部数据

## 自定义hook函数

* 1. 是setup中使用的组合式API代码段的封装
  2. 创建：在src/hooks中创建usexxx.js： export default function （）｛代码段｝
  3. 使用：引入usexxx.js到组件，在setup中调用

## 新组件

* 1. Fragment：组件模板没有根标签时，会自动把多个标签套在虚拟元素Fragment中
  2. Teleport：<teleport to=“选择器”>xxx</teleport> 将内部结构传到指定标签中，可跨组件传送，可用于实现遮罩层
  3. Suspense：<Suspense v-slot：default>异步组件</ Suspense><Suspense v-slot：fallback>加载中</ Suspense>
     1. 异步组件在加载时，显示fallback的内容，加载完成后显示default中的内容

## 异步组件

* 1. import ｛defineAsyncComponent｝from ‘vue’
  2. const Child = defineAsyncComponent（fn（）｛import（‘组件地址’）｝） 异步引入组件
  3. 普通引入为静态引入，静态引入所有组件加载完成后才显示界面；异步引入的组件不会阻塞其他组件的加载