

vite 构件脚手架

npm init vue@latest

cd firstVue

npm install

npm run dev

1.webpack配置

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：

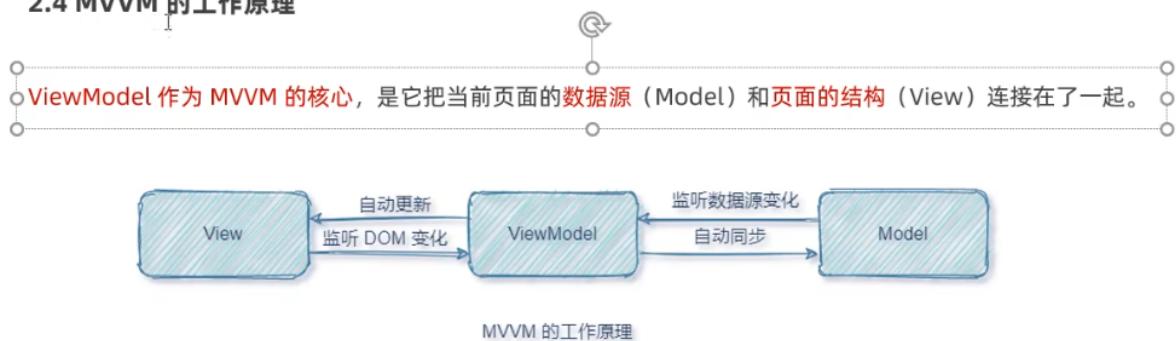
```
1 const path = require('path') // 导入 node.js 中专门操作路径的模块
2
3 module.exports = {
4   entry: path.join(__dirname, './src/index.js'), // 打包入口文件的路径
5   output: {
6     path: path.join(__dirname, './dist'), // 输出文件的存放路径
7     filename: 'bundle.js' // 输出文件的名称
8   }
9 }
```

2.webpack改默认打包

3.webpack-dev-server插件的配置

mvvm原理

2.4 MVVM 的工作原理



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

ts基础

ts直接引用的话会报错,因为用到了ts语法

如果ts的函数形参如果用来某个修饰,编译之后没有

```
// 总结:ts的文件中如果直接书写js语法的代码,那么在html文件中直接引入ts文件,在谷歌的浏览器中是可以直接使用的
```

```
// 如果ts文件中有了ts的语法代码,那么就需要把这个ts文件编译成js文件,在html文件中引入js的文件来使用
```

```
// ts文件中的函数中的形参,如果使用了某个类型进行修饰,那么最终在编译的js文件中是没有这个类型的
```

```
// ts文件中的变量使用的是let进行修饰,编译的js文件中的修饰符就变成了var了
```

接口

```
// 定义人的接口
interface IPerson {
    id: number
    name: string
    age: number
    sex: string
}

const person1: IPerson = {
    id: 1,
    name: 'tom',
    age: 20,
    sex: '男'
}
```

在vue3中,组件就是一个普通的对象

```
1 const root={}
```

template模板实例

两种写法

1.

```
const root = {
  data() { //data是一个函数需要有返回值
    return {
      message: "vue好棒棒" //data方法返回对象，其中的属性会自动添加到组件实例中
    }
  },
  // 在组件模板中可以直接访问组件实例属性 {{属性名}}
  template: "<h1>我爱vue,{{message}}</h1>", //组件模板
}
```

2.

```
<div id="app">
  <!--
    如果直接将模板定义到网页中，此时模板必须符合html的规范
    My-Button -> my-button
  -->
  <button @click="count++>点我一下</button> - 点了{{count}}次
</div>
```

index.js/min.js 入口文件

1. App.vue 根组件（引入子组件）

一般只创建一个应用实例

```
1 const vm = createApp(App)  
2 mount("#app"); //挂载到dom上，里面写dom选择器，一旦挂载就会清空所有原dom内容  
3
```

- 会返回一个根组件的实例，组件的实例通常可以命名为vm I
- 组件实例是一个Proxy对象（代理对象）

```
// 在data中，this就是当前的组件实例vm  
// 如果使用箭头函数，则无法通过this来访问组件实例  
// 使用vue时，减少使用箭头函数  
// data: vm => {  
//   console.log("data", vm)  
  
//   return {  
//     msg: "我爱Vue"  
//   }  
// }  
  
data() {  
  console.log("data", this) I  
  return {  
    msg: "我爱Vue"  
  }  
}
```

.this是根组件的返回值 proxy

```
▼ Proxy {...} ⓘ  
  ► [[Handler]]: Object  
  ► [[Target]]: Object  
  ► [[IsRevoked]]: false
```

， 在根组件中定义的属性是响应式的，（被vue代理了）

而this.name="八戒"，这样赋值添加的不是响应式的（没有被vue代理了）

```
// vm.$data 是实际的代理对象，通过vm可以直接访问到$data中的属性  
// vm.$data.msg 等价于 vm.msg  
// 可以通过vm.$data动态的向组件中添加响应式数据
```

```
this.$data.name = "孙悟空"
```

```
created(){
    // 会在组件创建完毕后调用
    // 可以通过vm.$data动态的向组件中添加响应式数据，但是不建议这么做
    this.$data.name = "孙悟空"
}
```

2. components/文件夹内存放子组件

```
import MyButton from './components/MyButton'
import { shallowReactive } from "vue"
```

```
// 有些场景下，可以通过shallowReactive()来创建一个浅层的响应式对象
return shallowReactive({
    msg: "大闸蟹今天没去玩游戏！",
    stu: {
        name: "孙悟空",
        age: 18,
        gender: "男",
        friend: {
            name: "猪八戒"
        }
    }
})
```

缺点:

只能 返回对象的响应式代理

```
/*
  computed 用来指定计算属性
  {
    属性名:getter
  }
  - 计算属性，只在其依赖的数据发生变化时才会重新执行
*/
computed: {
  info() {
    console.log("---> , info调用了!")
    return this.stu.age >= 18
      ? "你是一个成年人!!!!"
      : "你是一个未成年人!!!!"
  }
}
```

computed是计算属性,调用时 info ,会对数据进行缓存

而methods里面是函数,调用时 info();

仅用来做get不修改数据,纯读

```
computed: {
  info() {
    // 在计算属性的getter中，尽量只做读取相关的逻辑
    // 不要执行那些会产生（副）作用的代码
    console.log("---> , info调用了!")
  }
}
```

计算属性设置set后双向绑定(代理对象)

```
// 计算属性的简写（只有getter时）
// name(){
//   return this.lastName + this.firstName
// }
name: {
  get() {
    return this.lastName + this.firstName
  },
  set(value){
    // set在计算属性被修改时调用
    this.lastName = value[0]
    this.firstName = value.slice(1)
  }
}
```

```
export default {
  setup() {
    // 定义变量
    let msg = "今天天气真不错！"
    let count = 0

    // 在setup()中可以通过返回值来指定那些内容要暴露给外部 I
    // 在
    return {
      msg
    }
  }
}
</script>
<template>
  <h1>演示组合式API</h1>
  <h2>{{ msg }}</h2>
```

steup钩子函数,(暴露)

```
1 setup() {
2   let msg = "哈哈哈vue组合式api" //这里定义的变量就是普通的变量,不是响应式的
3   let count = 0
4   const stu=reactive({ //现在就是响应式变量
5     name:"八戒",
6     age:18
7   })
8   return {
9     msg, count
10    }
11 }
```

```
//可以通过 reactive() 来创建一个响应式的对象
const stu = reactive({
    name: "孙悟空",
    age: 18,
    gender: "男"
})
```

- 全局变量

```
<template>
<!--
- 在模板中，可以直接访问到组件中声明的变量
- 除了组件中的变量外，vue也为我们提供了一些全局对象可以访问：
    比如：Date、Math、RegExp ...
    除此之外，也可以通过app对象来向vue中添加一些全局变量
    app.config.globalProperties
-->
```

可以在所有的模块中调用

1.添加全局变量

```
5
6 app.config.globalProperties.hello = "你好，我是全局的属性"
7 app.config.globalProperties.alert = alert.bind(this)
8
```

2.模板语法只能写表达式（有返回值的语句）

例如 if(){} 不可以，但可以写 **三元表达式**

3.插值语法实质就是修改标签的 textContent

4.指令

- **v-text**

v-text="ABC"将表达式的值作为元素的textContent插入，作用同{{}}同 {{ABC }} ,都是去找ABC 变量

- **v-html**

将表达式的值作为元素的innerHTML插入，有xss攻击的风险 innerText innerHTML

- **v-bind**

(简写 ":"),将模块变量写入html标签的属性中，如果传入对象，可以把对象中的所有元素传入。当我们为一个布尔值设置属性时，如果值为true，则元素上有该属性 (转换后为true，也算true)如果值为false，则元素没有该属性

```

```

小案例，这样就相当于在html中使用变量

```
1 <style scoped>
2 .box{
3     background-color: antiquewhite;
4 }
5 h1{
6     font-size: 20px;
7 }
8 </style>
```

局部样式原理

```
.box[data-v-37ddea6c]{
    background-color: antiquewhite;
}
h1[data-v-37ddea6c]{
    font-size: 20px;
} == $0

<h1 data-v-37ddea6c>ssssssss</h1>
<div class="box" data-v-37ddea6c>1111</div>
```

vue会对所有的标签添加 data-v-任意值 作为属性, 而且会修改css属性选择器,从而实现局部样式.

属性选择器

某某[属性] 选择到某标签且有 XX属性 的标签

例如a[vue], 选择a标签 带有vue属性的标签

适用

只适用于

```
1 <template>
2     <div>
3         <h1>ssssssss</h1>
4         <div class="box">1111</div>
5     </div>
6 </template>
```

这种单根组件,内有一个div

注意: scope还好对子组件的根元素添加 父组件的自生成的属性

深度选择器 :deep()

- 在父组件中(根组件),对引入的子组件直接进行样式选择,子组件并不会生效,这种设计原理是正确的,尽量减少各组件的耦合.

:deep()

例如

```
1 .btn:deep(h2) {  
2     color: antiquewhite;  
3 }
```

这样不管子组件标签的位置,都会生效,

- 见解: 如果写上.btn,就说明是子标签,如果是子组件的根标签,就不用写,因为子组件的根标签会带上vue设置的随机属性

全局选择器 :global()

- <style scoped></style>

<style></style>

用两个style标签写 一个是局部 一个是全局

- :global()

例如:

```
1 <style>  
2 :global(div) {  
3     border: 1px red solid;  
4 }  
5 </style>
```

1. css模块化-自动的对模块中的类名进行hash化来确保类名的唯一性
2. 在模板中可以通过 v-bind:class="\$style.类名" 来使用
3. 例子

```
▼ <div class="_box_b2kob_3" =  
  <h1>111</h1>  
  <h2>111</h2>  
</div>
```

```
1 <template>  
2   <div v-bind:class="$style.box">  
3     </div>  
4 </template>  
5 <!--module="classes 可以把 $style 改成 classes -->  
6 <style module>  
7 .box {  
8   background-color: antiquewhite;  
9   width: 50px;  
10  height: 50px;  
11 }  
12 </style>
```

类和内联样式,让html可以使用变量

```
<script setup>
const arr = ["box1", "box2", "box3"]
const arr2 = [{ box1: true }, { box2: false }, { box3: true }]
const style = {
  color: "red",
  backgroundColor: "#bfa"
}
You, 24小时前 • 模板语法 ...
</script>
<template>
  <h1 class="header">我爱Vue</h1>
  <div :class="arr2" :style="style">我是div</div>
</template>
```

props

```
1  /*
2  使用props
3  -现在子组件定义props
4  defineProps(["item"]),用变量接收,根组件传值, 使用驼峰命名法或者“-”(大写变小写加-)
5
6  props的值只读不可改,但有修改的方法:
7  1.在父组件给props重新传值,
8  2.在父组件传递一个对象{XXX:0},子组件通过对 props.item.XXX修改,
9  vue只对 props.item这个变量进行锁定,尽量不要在子组件中修改父组件,
10 的数据。
11
12 方法二:
13
14 defineProps({
15   title: String,
16   likes: Number,
17   propD: {
18     type: Number,//照样接收只是在控制台报错
19     default: 100 //默认值
20   },
21 }),限制类型
```

v-show和v-if

v-show切换速度快

v-if加载速度快

v-show 可以根据值来决定元素是否显示（通过display来切换元素的显示状态）

v-if 可以根据表达式的值来决定是否显示元素（会直接将元素删除）

v-show通过css来切换组件的显示与否，切换时不会涉及到组件的重新渲染
切换的性能比较高。

但是初始化时，需要对所有组件进行初始化（即使组件暂时不显示）

所以它的初始化的性能要差一些！

v-if通过删除添加元素的方式来切换元素的显示，切换时反复的渲染组件，
切换的性能比较差。

v-if只会初始化需要用到的组件，所以它的初始化性能比较好

v-if可以和 v-else-if 和 v-else结合使用

v-if可以配合template使用

动态组件

is属性,可以写html自带标签也可以写组件,同时还可以写三元表达式,这样就可以实现选项卡功能

```
1 <component is="isShow:A?B"></component>
```

这样只要控制isShow的true和false就可以实现组件的切换

v-for遍历

基本写法

```
1 <li v-for="name in players">{{ name }}</li>
```

其中name是遍历数组的值

输出: { "name": "内马尔", "img": "/public/images/neymar.png", "rate": 3, "hot": 100524 }

组件引用的写法:

```
1 <TabItem v-for="name in players" :item="name" :maxhot="maxhot"></TabItem>
```

等同于

```
1 <TabItem :item="players[0]" :maxhot="maxhot"></TabItem>
2 <TabItem :item="players[1]" :maxhot="maxhot"></TabItem>
3 <TabItem :item="players[2]" :maxhot="maxhot"></TabItem>
```

从而循环引入组件

减少重排

方法一

将多个css放到一个类里面,减少重排次数

方法二

```
box1.style.display = "none"  
box1.style.width = "300px"  
box1.style.height = "400px"  
box1.style.fontSize = "20px"  
div.style.display = "block"
```

这样只会重排两次

注意

在现代的前端框架中，这些东西都已经被框架优化过了!所以使用vue、react这些框架这些框架开发时，几乎不需要考虑这些问题，唯独需要注意的是，尽量减少在框架中直接操作DOM

slot 插槽

简介

子组件挖坑，父组件来填坑

内容(入口)

<MyButtom>哈哈哈，我是入口</MyButtom>

出口

<slot></slot>

例子

```
1 <script setup>
2 import MyButton from './components/MyButton.vue';
3 import A from './components/A.vue';
4 const name = "小明"
5 </script>
6
7 <template>
8   <MyButton>
9     <A :name="name"> </A>
10  </MyButton>
11 </template>
```

具名插槽(命名插槽)

为了让子组件有多个插槽位置

```
1 <template v-slot:s1>s1</template>
```

注意:

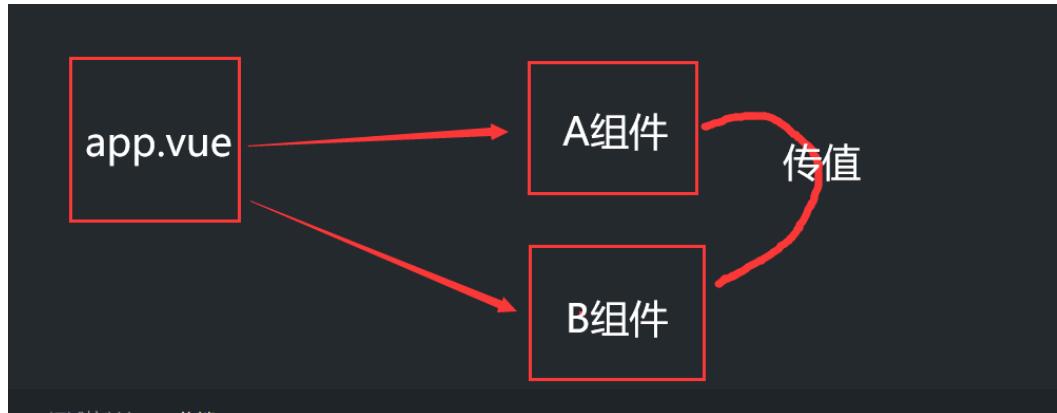
1. v-slot can only be used on components or <template> tags.
2. v-slot缩写#

作用域插槽

解释：插槽里面传参数，子父组件之间相互通信

```
1 <template v-slot:s1="PropSlot">s1</template> //这里s1="PropSlot"相当于props的定义属性名  
来接收值
```

要实现子组件之间相互传值



利用 slot插槽的方式,将A组件的值传入app.vue,然后再从app.vue传值给B组件,这样App.vue起一个中间人的作用

```
<MyButton>  
  <template #s1="props">  
    {{ props.Sname }}  
    <A1></A1>  
  </template>  
  <!-- <template v-slot:s2>  
  </MyButton>
```

子组件中是

```
1 <slot name="s1" :Sname="Sname">  
2 </slot>
```

根组件是

```
1 <template #s1="props"> //这里不光可以写变量,也可以写一个对象进行解构{name,age} ,调用时直接  
写name  
2   {{ props.Sname }}  
3 </template>
```

这样子组件中的**slot标签**被命名成为 **具名插槽**,根组件对应的 **template标签**

#**s1**:= "props",其中**props**就相当于一个**变量**来接收子组件传递来的值.并存储为**props对象**,但是作用域只在根组件的<template>标签内.

事件

1. 事件绑定类型

@事件名

绑定事件的两种方式

a. 内联事件处理器 (自己调用函数)

- 事件触发时，直接执行js语句
- 内联事件处理器，回调函数的参数由我们自己传递

b. 方法事件处理器 (vue帮我们调用函数)

- 事件触发时，vue会对事件的函数进行调用
- 方法事件处理器，回调函数的参数由vue帮我们传
- 参数就是事件对象

c. vue如何区分两种处理器：

检查事件的值是否是合法的js标识符或属性访问路径

如果是，则表示它是方法事件事件处理器

否则，表示它是内联事件处理器

```
<button @click="clickHandler">方法事件处理器</button>
<hr />
<button @click="clickHandler2($event, 1, 2, 'hello')">
    |   内联事件处理器
</button>
```

2. 修饰符

```
<div class="box3" @click.stop="boxHandler2('box3')">box3</div>
@click.stop 阻止事件冒泡(点一个子元素会触发父元素事件)
```


3、Vue.js内置指令

Vue.js针对一些常用的页面功能提供了以指令封装的使用形式，以HTML元素属性的方式使用。

- **v-show:** 根据表达式值的真假，显示或隐藏HTML元素。
- **v-if:** 根据表达式值的真假来生成或删除一个元素。
- **v-else-if:** 与v-if指令一起使用，可以实现互斥的条件判断。
- **v-for:** 通过循环方式渲染一个列表，循环的对象可以是数组，也可以是一个JavaScript对象。
- **v-bind:** 用于响应更新HTML元素的属性，将一个/多个属性或一个组件的prop动态绑定到表达式。
- **v-model:** 用来在表单<input><textarea>和<select>元素上创建双向数据绑定，它会根据控件类型自动选取正确的方法来更新元素。
- **v-on:** 用于监听DOM事件，并在触发时运行一些JavaScript代码。

4、虚拟DOM

- Vue.js之所以执行性能好，一个很重要的原因就是它的DOM机制。
- 传统DOM模式中，对元素、文本的操作就是对DOM节点的操作，每一次对DOM的修改都会引起浏览器对网页的重新渲染，过程比较耗时。
- 虚拟DOM使用普通JavaScript对象描述DOM元素，访问JavaScript对象自然要比访问真实DOM要快得多，Vue更新真实DOM前，会比较更新前后虚拟DOM的差异部分，然后采用异步更新方式将差异更新到真实DOM中。

v-model 指令用来对

1. input 输入框
2. textarea 文字框
3. option 选项框

v-model 指令修饰符

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 v-model.number="age" />
.trim	自动过滤用户输入的首尾空白字符	<input v-model.trim="msg" />
.lazy	在 “change” 时而非 “input” 时更新	<input v-model.lazy="msg" />

示例用法如下：

单选框,多选框,下拉框案例

进行数据采集

```
<div>
  {{picked}}
  <input type="radio" value="One" v-model="picked"> One
  <input type="radio" value="Two" v-model="picked"> Two
</div>

<div>
  {{favorites}}
  <input type="checkbox" value="教育" v-model="favorites">教育
  <input type="checkbox" value="动漫" v-model="favorites">动漫
  <input type="checkbox" value="新闻" v-model="favorites">新闻
</div>

<div>
  <select>
    <option value="001">北京</option>
    <option value="001">北京</option>
    <option value="001">北京</option>
  </select>
</div>
```

```
<!-- (重要)
遇到复选框，v-model的变量值
非数组 - 关联的是复选框的checked属性
数组    - 关联的是复选框的value属性
-->
```

下拉菜单v-model写在哪里？

在select, value在option上

v-model用在复选框时, 需要注意什么?

v-model的vue变量是

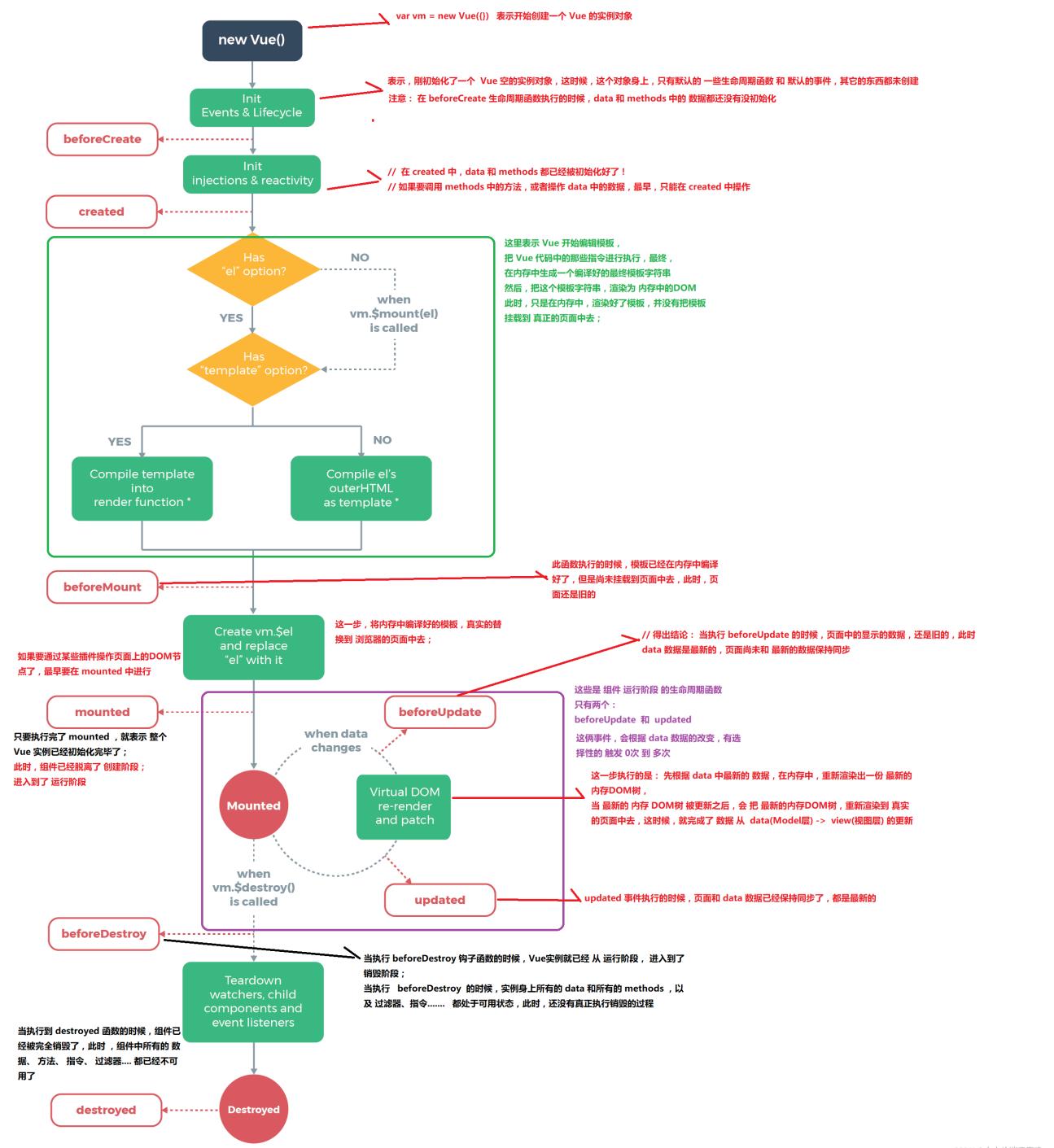
非数组 – 关联的是checked属性

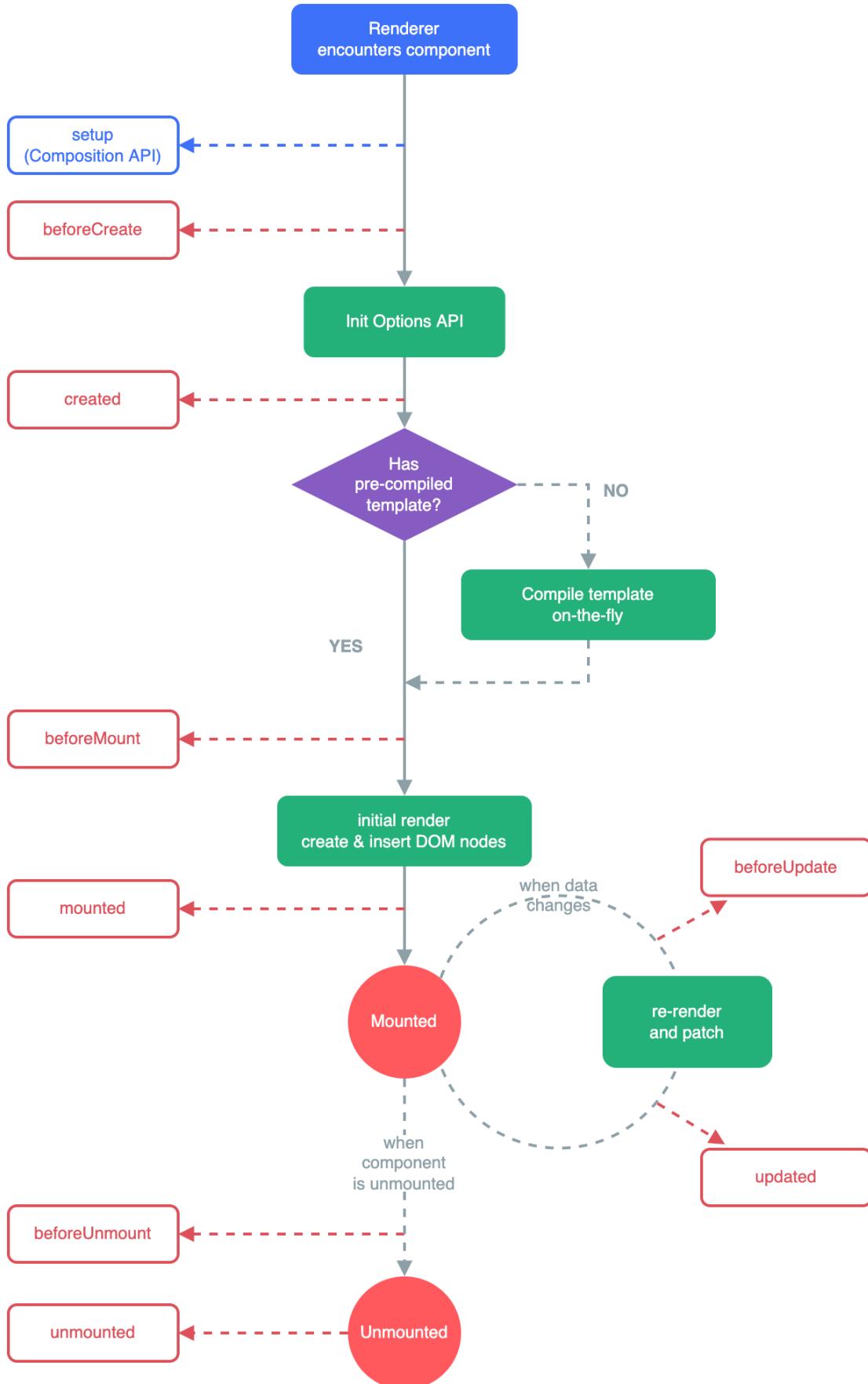
数组 – 关联的是value属性

vue变量初始值会不会影响表单的默认状态?

会影响, 因为双向数据绑定-互相影响

可以在页面执行的时候自动触发一些函数





```

    // title.value = 'world'
}

// watchEffect(() => {
//   console.log('count变化为: ' + count.value)
//   console.log(title.value)
// })

watch(
  count,
  () => {
    console.log(title.value)
  }
)

```

推荐第二种 watch

只要count(对象、函数)发生变化,

就会出发下面的函数,这样更精准

当然, count也可以是一个函数,但要有返回值,根据返回值的变化来执行下面的函数

复制笔记内容

```

1 <script setup>
2   import { ref, watch, onMounted, watchEffect } from 'vue'
3
4   const x = ref(0)
5   const y = ref(0)
6
7   // 单个 ref
8   watch(x, (newX) => {
9     console.log(`x is ${newX}`)
10  })
11
12   // getter 函数
13   watch(
14     () => x.value + y.value,
15     (sum) => {
16       console.log(`sum of x + y is: ${sum}`)
17     }
18   )

```

```
17     }
18 )
19
20 // 多个来源组成的数组
21 watch([x, () => y.value], ([newX, newY]) => {
22     console.log(`x is ${newX} and y is ${newY}`)
23 })
24
25 onMounted((() => {
26     x.value = 100
27     url.value = 'https://yesno.wtf/api'
28 }))
29
30
31 const url = ref('')
32 const data = ref(null)
33
34 async function fetchData() {
35     const response = await fetch(url.value)
36     data.value = await response.json()
37     console.log(data.value)
38 }
39
40 // 立即获取
41 // fetchData()
42 // ...再侦听 url 变化
43 watch(url, fetchData)
44
45 /* watchEffect(async () => {
46     const response = await fetch(url.value)
47     data.value = await response.json()
48     console.log(data.value)
49 }) */
50 </script>
51
52 <template>
53 <div>
54
55 </div>
56 </template>
```

```
57
58 <style lang="css">
59
60 </style>
```

`watch` 和 `watchEffect` 都能响应式地执行有副作用的回调。它们之间的主要区别是追踪响应式依赖的方式：

- `watch`

只追踪明确侦听的源。它不会追踪任何在回调中访问到的东西。另外，仅在响应源确实改变时才会触发回调。`watch` 会避免在发生副作用时追踪依赖，因此，我们能更加精确地控制回调函数的触发时机。

- `watchEffect`

，则会在副作用发生期间追踪依赖。它会在同步执行过程中，自动追踪所有能访问到的响应式 `property`。这更方便，而且代码往往更简洁，但其响应性依赖关系不那么明确。

在main.js引入import 全局组件注册

```
1 MyComponent from './views/11-component-registration/MyComponent.vue'
```