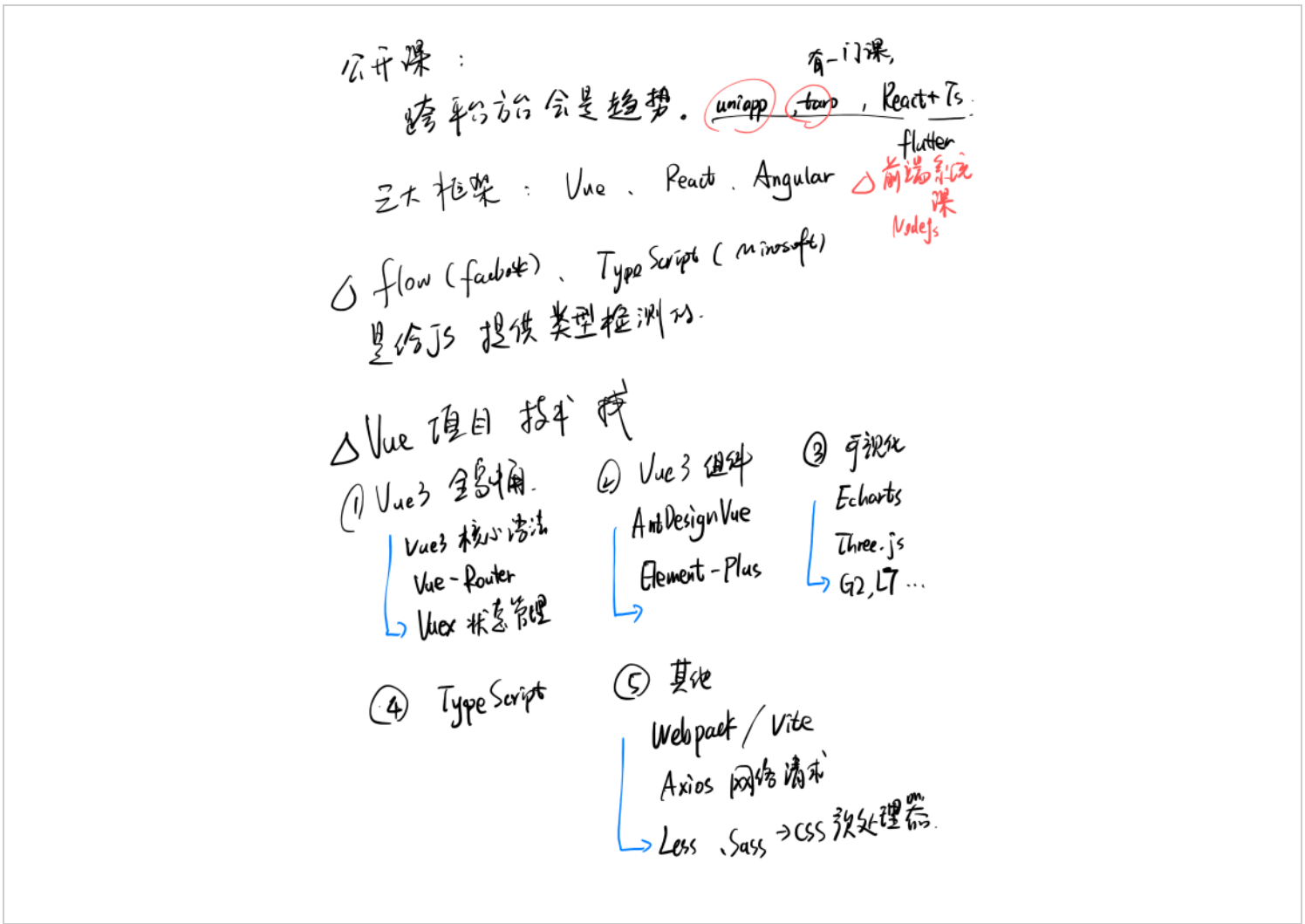
Vue3基础语法

2021年11月5日 星期五　上午 4:27

目录:

公开课



一、遍历 Vue3

渐进式框架 Vue.js → JS库

项目里一点点用，不需全部

① CDN 引用 <...src="https://...com/vue@next"...>

② 下载 Vue.js 文件，手动 src="../js/vue.js"

③ 用 npm 包管理

④ 用 Vue CLI，脚手架

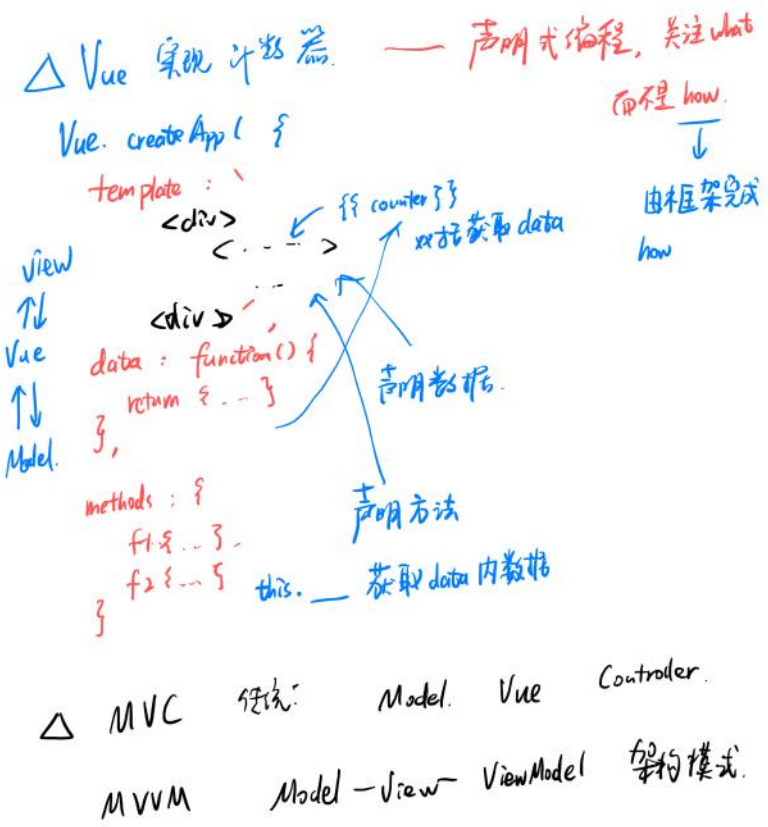
开发 → webpack 打包 → AJax... 发布到 服务器 ← 用户

△原生 JS 实现计数器 — 命令式编程，具体过程

1. 获取元素，通过 class、id、querySelector("...")

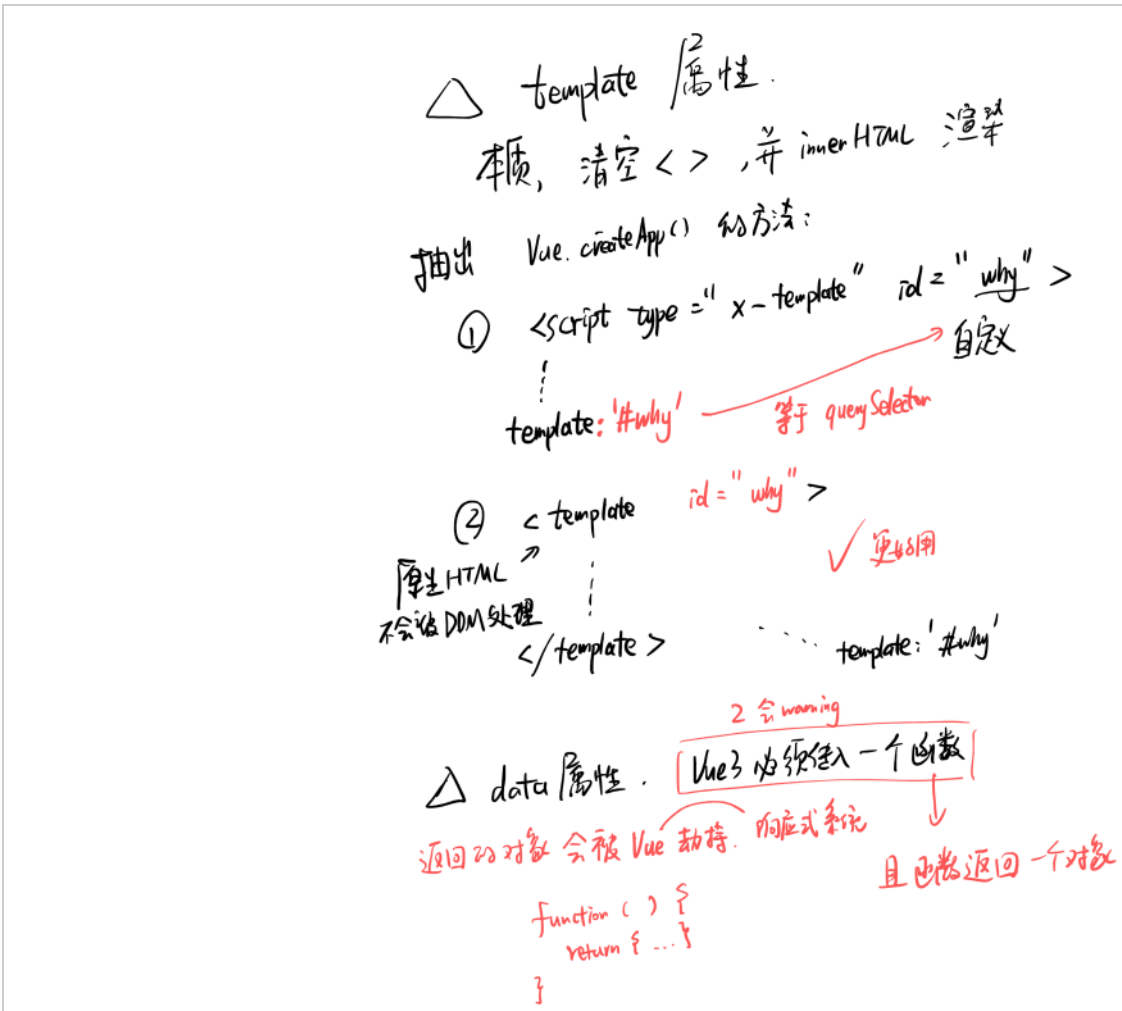
2. 定义变量

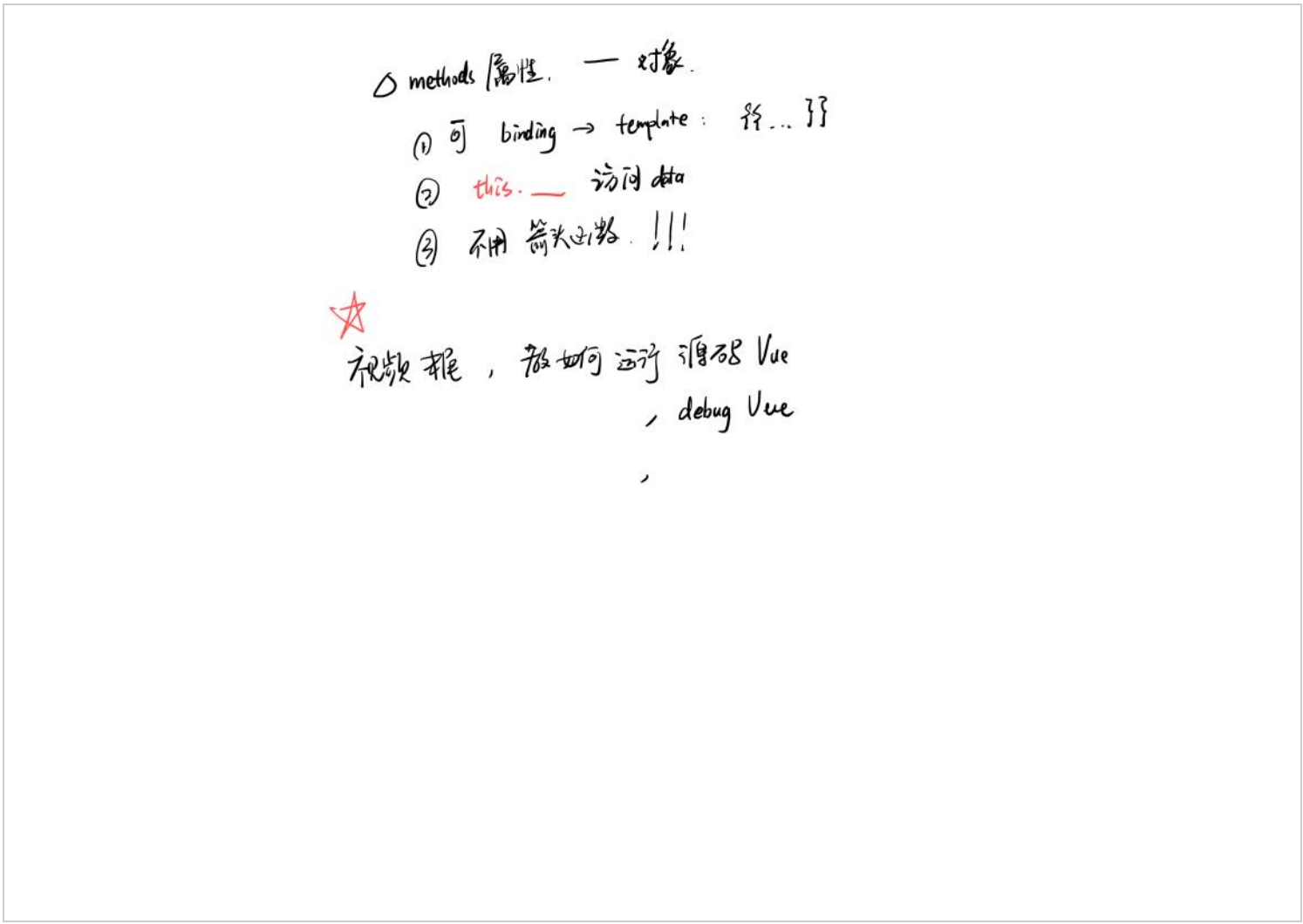
3. 监听按钮的点击，并响应事件



△ MVC 传统： Model Vue Controller

MVVM Model-View-ViewModel 架构模式


△ template 属性.
本质，清空 < > ，并 innerHTML 渲染
抽出 Vue.createApp() 的方法：
① <script type="x-template" id="why">
template: '#why'
自定义
等于 querySelector
② <template id="why">
原生HTML
不会被DOM处理
</template>
template: '#why'
✓ 更好用
△ data 属性.
2 会warning
Vue3 必须注入一个函数
且函数返回一个对象
返回的对象 会被 Vue 劫持. 响应式系统
function ( ) {
return { ... }
}


methods 属性. — 对象.
① 可 binding → template: 好...了
② this.__ 访问 data
③ 不用 箭头函数!!!
视频推荐，教如何调试源码 Vue
，debug Vue
，

2. Vue 3 基础语法 (一)

Mustache + v指令

△ this 看文章去

用 Vue 得明白 methods 里 this.

源码中: ctx[key] = methodHandler.bind(publicThis)

methodHandler: 某一个 method

publicThis = instance.proxy! 代理对象

data: 对象

所以 methods 中 this 指向 createApp.data

△ React 开发模式: JSX.

```
function () {
  return <div></div>
}
```

Vue: 也有jsx, 但大多是 HTML 模板

底层 Vue → 虚拟 DOM

template: {{...}} → 获取底层 data

① Mustache 语法 {{..}}

1. 基本: {{ message }}

2. 表达式: {{ counter * 10 }}

: {{ message.split(" ").join ... }}

3. 可以调用函数 {{ ... [methods] () }}

且可以使用 computed（计算属性）

4. 三元运算符：{{ isShow ? 啥... : "_" }}

5. 可以多个一起使用

Tips：1. 不可以是赋值语句 → 须是表达式

2. 不可以是 if 语句.

原理：data 返回的对 会添加到 Vue的响应式系统

data中数据 → 对应的内容也会改变.

② v-once 指令 —— template内元素标签内的指令

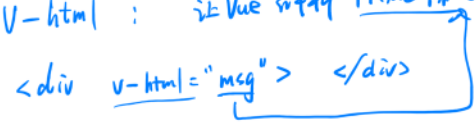
↳ 指定内容只渲染一次，后续不再发生改变

渲后，v-once内容视为静态内容

③ v-text 指令 ：更新 textContent

和 mustache 类似

④ v-html ： 让 Vue 解析 HTML 代码



⑤ v-pre ：展示原始 Mustache 标签 "{{msg}}"

⑥ v-cloak：和 css 的 [v-cloak] {display:none} 一起用

为了隐藏 未加载完全的 Mustache 标签

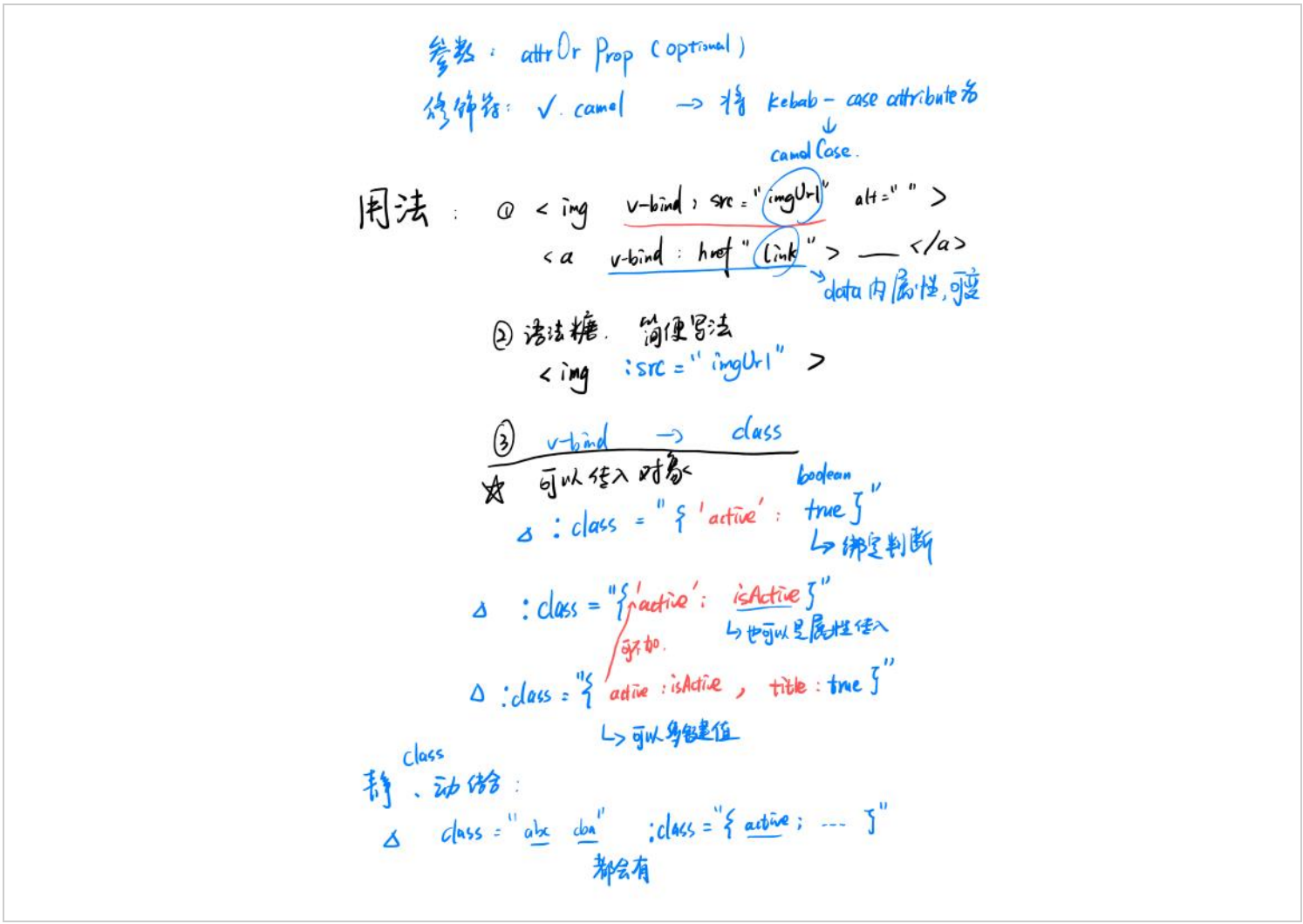
△ Vue3 ：Composition API

Vue2 ：Option API

⑦ v-bind 动态绑定为元素的属性 / prop 组件

缩写 → :

预期 ：any (with argument) | Object (without argument)


参数：attrOr Prop (optional)
修饰符：√. camel → 将 kebab-case attribute 为 camelCase.
用法：① <img v-bind:src="imgUrl" alt="" >
<a v-bind:href="link" > ——
data 内属性，可变
② 语法糖. 简便写法
<img :src="imgUrl" >
③ v-bind → class
可以传入对象
△ :class="{'active': true}"
boolean
绑定判断
△ :class="{'active': isActive}"
也可以是属性传入
△ :class="{active:isActive, title:true}"
可以多包建值
静、动结合：
class
△ class="abx cba" :class="{active: ---}"
都会有

△ : class = " classObj " → 也可以传入对象属性 { ... : true, ... : false }

△ : class = " getClassObj() ", 可以是返回 obj 的 methods 或 computed

☆ 可以传入数组

△ <div :class="['abc', title, 三元运算符 数组内可以嵌套 表达式/对象 , {active : isActive}]

④ v-bind → style 样式

☆ 对象语法

<div :style="{'color': finalColor或'red'
, 'font-size': '30px'
, fontSize : '30px' 或 fontSize +'px'}">

不一样!! 属性 拼接

也可以 :style = " finalStyleObj " = " getStyleObj() " → 对象

☆ 数组 bind style

:style = " [styleObj1, styleObj2] 就是合并对象

⑤ bind — 未确定属性.

<div :[name] = "value" >

name是data中变量

☆ ⑥ bind 绑定一个对象 ⇒ 多个属性 class href ste...

用处：封装高级组件

配置整体传输 → UI组件

data中 → info: { name: "why",

age: 18,

class: ____ , style: ____ }

⑧ v-on : 绑定监听事件 → 简写@

△ @click ⇒ v-on : click 事件 = "bclick"

[methods] 方法.

△ 可绑表达式 : @click = "counter ++"

data

△ 绑定多个事件 : v-on = "{ click = ____, mousemove : ____ }"

@ = "

△ v-on 参数传递

默认传入 event：@click = "btclick()"

$event：@click = "btclick($event, '...', 18)"

event

传入多个参数

△ v-on 修饰符 — 很多 .prevent .capture

如 @click.stop → 停止冒泡 dom.stopPropagation()

@key.enter → 特定键 enter .up

tips：鼠标中间 (event) { event.target.value } 获取内容

3. Vue3 基础语法（二）

条件和循环

① 条件渲染

```
<h2 v-if="score > 90">优</h2>
<h2 v-else-if="score > 60">良</h2>
<h2 v-else>不及格</h2>
```

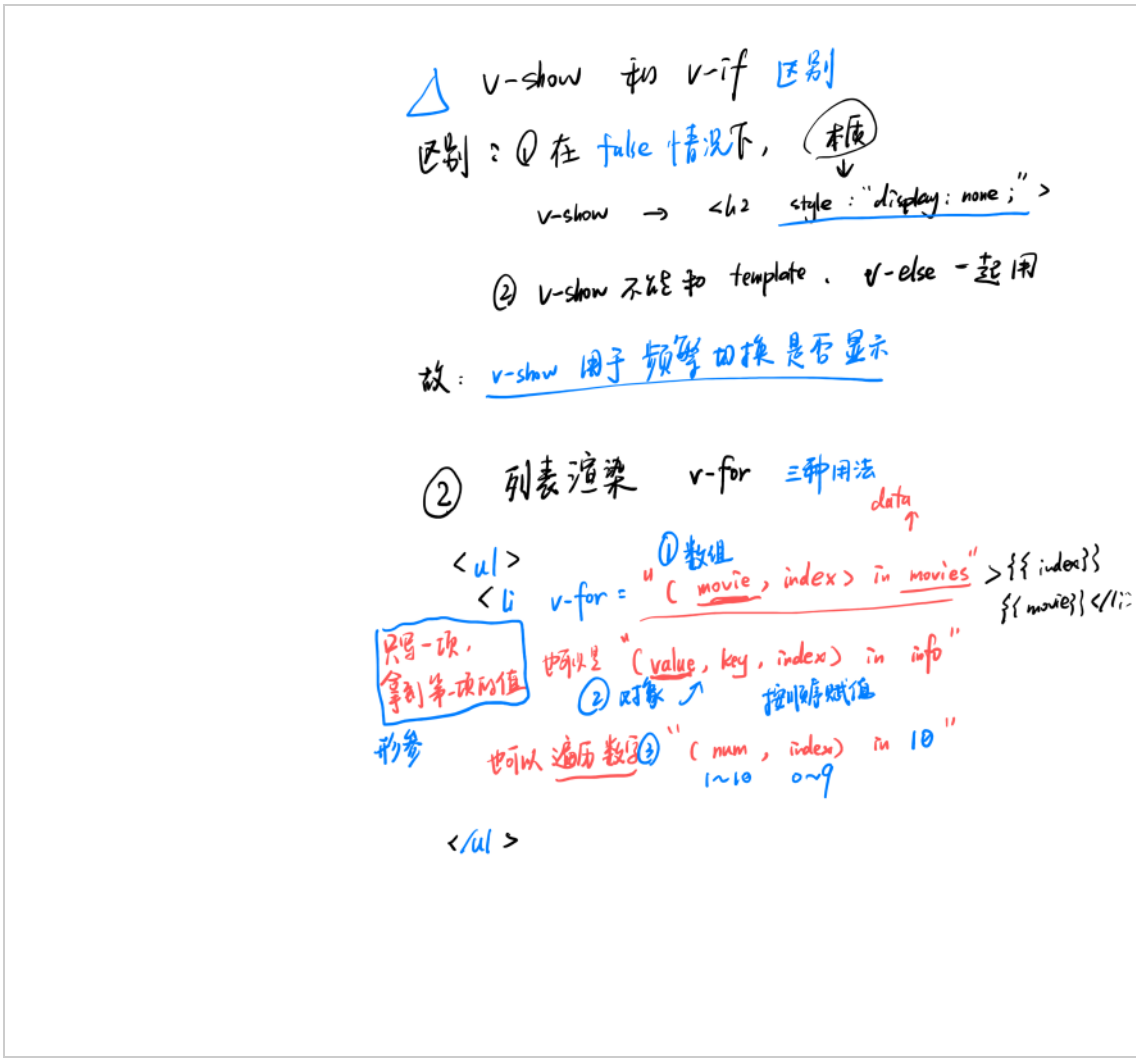
可以用输入：`<input type="text" v-model="score">`

原理：v-if 惰性，false 则标签销毁，不渲染不存在于 DOM 里

△ template 元素，包裹元素

在 template 上进行 v-if、else，会好很多。

```
<template id="app">
  <template v-if="...">
    <div></div>  ← template 不会被渲染出来
  </template>
</template>
```


v-show 和 v-if 区别
区别：① 在 false 情况下，
本质
v-show → <h2 style: "display: none;" >
② v-show 不能和 template 、v-else 一起用
故：v-show 用于频繁切换是否显示
② 列表渲染 v-for 三种用法
data

<li v-for = "① 数组 (movie, index) in movies" >{{ index }} {{ movie }}
只写一项，拿到第一项的值
形参
也可以 ② 对象 (value, key, index) in info
控制台赋值
也可以遍历数字 ③ (num, index) in 10
1~10 0~9

△ v-for + template，循环展示块

```
<template>
  <ul>
    <template v-for="(v, k) in info">
      <li>{{k}}</li>
      <li>{{v}}</li>
      <li class="_"></li>
    </template>
  </ul>
</template>
```

age
18
name
why

template 不会被渲染，不浪费资源

△ 数组更新检测

Array 方法 Vue 内，对数组方法侦听，会触发视图更新

pop push shift unshift splice，sort，reverse.

但是 filter()，slice()，concat() 生成新数组，则不会触发更新.

△ 数组更新用的挺多

③ v-for 中，key 是什么作用？

给节点绑定一个 key :key="_"

key 用处：key 属性用在 Vue 的虚拟 DOM 算法，新旧 Nodes 对比辨识 VNodes.

key:

VNode：Virtual Node 虚拟节点，本质是 JS Objects

先理解 HTML 中的：

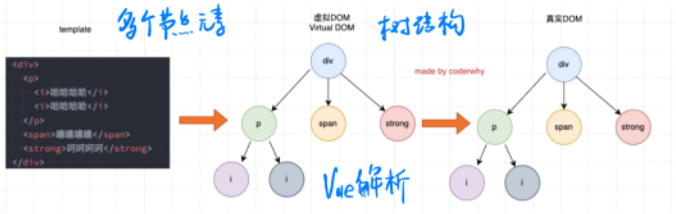
- Vue 中的元素、组件 先转成 VNode

转化后：

```
const vnode = {
  type: —
  props: —
  children: —
};
```

template → VNode → 真实DOM（浏览器）

用处 → 多平台渲染



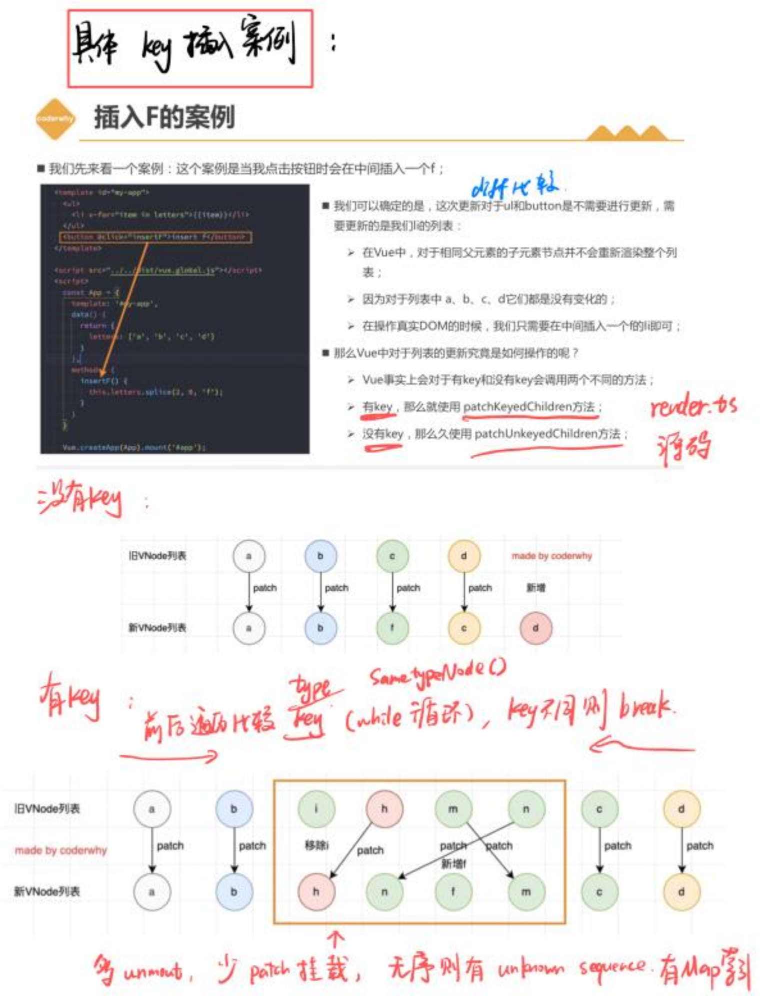
diff 算法：

新 VNodes（如数组更新）

旧 VNodes

diff 比较

比较哪里改变了


具体 key 插入案例：
插入F的案例
我们先来看一个案例：这个案例是当我点击按钮时会在中间插入一个f；
diff 比较
我们可以确定的是，这次更新对于ul和button是不需要进行更新，需要更新的是我们li的列表：
在Vue中，对于相同父元素的子元素节点并不会重新渲染整个列表；
因为对于列表中 a、b、c、d它们都是没有变化的；
在操作真实DOM的时候，我们只需要在中间插入一个f的li即可；
那么Vue中对于列表的更新究竟是如何操作的呢？
Vue事实上会对于有key和没有key会调用两个不同的方法；
有key，那么就使用 patchKeyedChildren方法；
没有key，那么久使用 patchUnkeyedChildren方法；
render.ts
源码
没有key：
旧VNode列表
新VNode列表
patch
新增
made by coderwhy
有key：
sameTypeNode()
移除i
新增f
unmount
patch
unknown sequence
Map

key2：若不用key，尽量复用/修改相同类型元素的算法
patch UnkeyedChildren()

key3：使用key，则基于key变化重新排列元素，移/销key不存在的元素。

4. Vue 基础语法（三）

计算属性 computed.

① 复杂 data 的处理方式

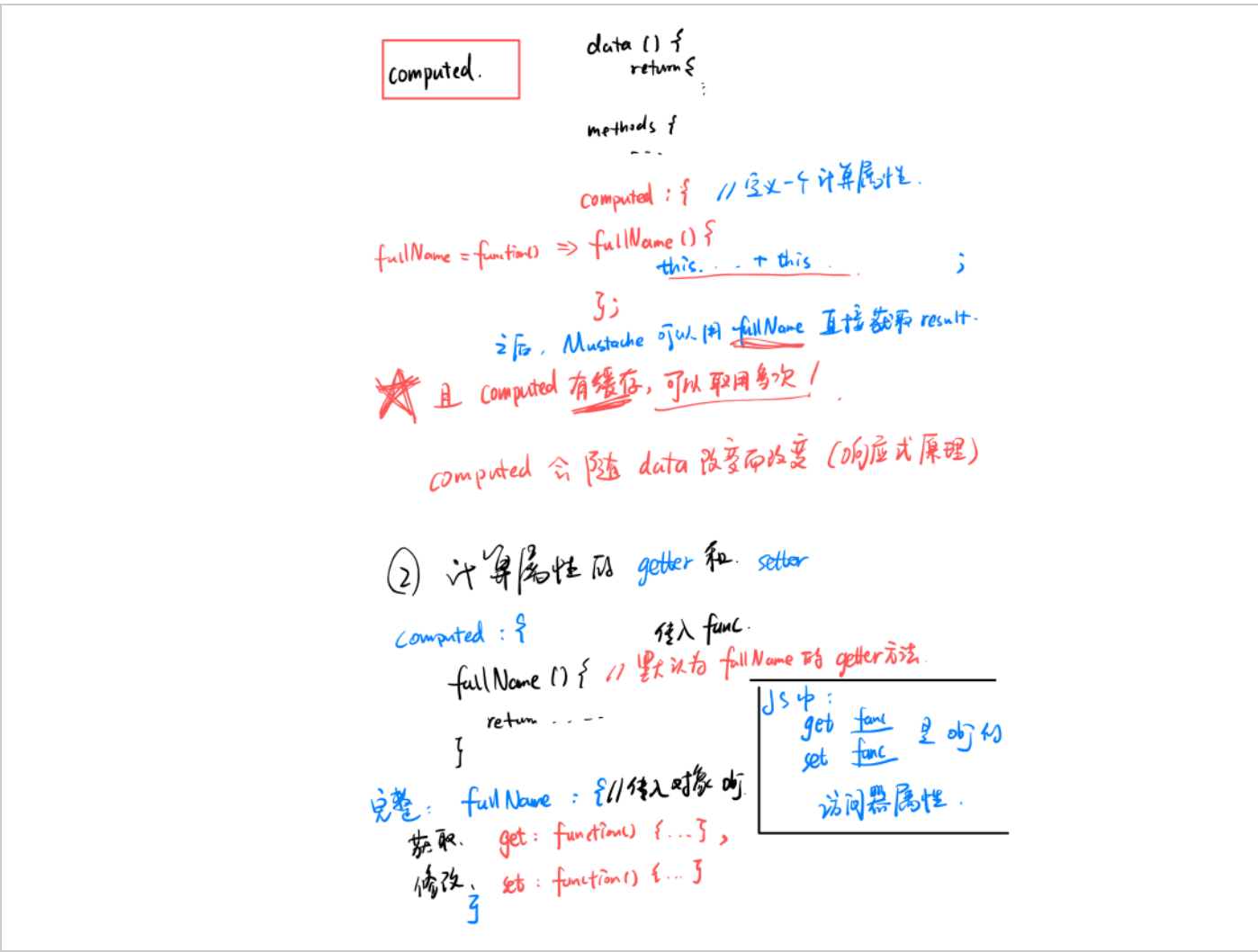
模板 template 中，经常引插值 Mustache

表达式：简单的运算

复杂计算 → methods：option 方法调用 不太好 重复不相等

→ 计算属性 computed ✓ 官方推荐

计算属性将被混入到组件实例中。所有 getter 和 setter 的 this 上下文自动地绑定为组件实例；


computed.
data () {
return {
}
methods {
computed : { // 定义一个计算属性.
fullName = function() => fullName () {
this. . . + this .
};
之后, Mustache 可以用 fullName 直接获取 result.
且 computed 有缓存, 可以取用多次!
computed 会随 data 改变而改变 (响应式原理)
② 计算属性的 getter 和 setter
computed : {
传入 func.
fullName () { // 默认为 fullName 的 getter方法.
return . . . .
}
JS中:
get func 是可行的
set func
访问器属性.
完整: fullName : {// 传入对象时
获取. get : function() {...},
修改. set : function() {...}
}

源码中，computed 为 function 会 bind → publicThis / proxy data.

↳ 判断传入 isFunction. | get 也会 bind

③ 侦听器 watch

侦听 data 的变化，不需要 click -- 事件辅助

进行 逻辑处理，如 JS、Ajax

选项：watch

类型：{ [key: string]: string | Function | Object | Array}

```
App = {//
  watch : {                          可简写
    Question : function (newValue, oldValue) {
      侦听的data属性
    }
  },
```

对象 →

新 旧值

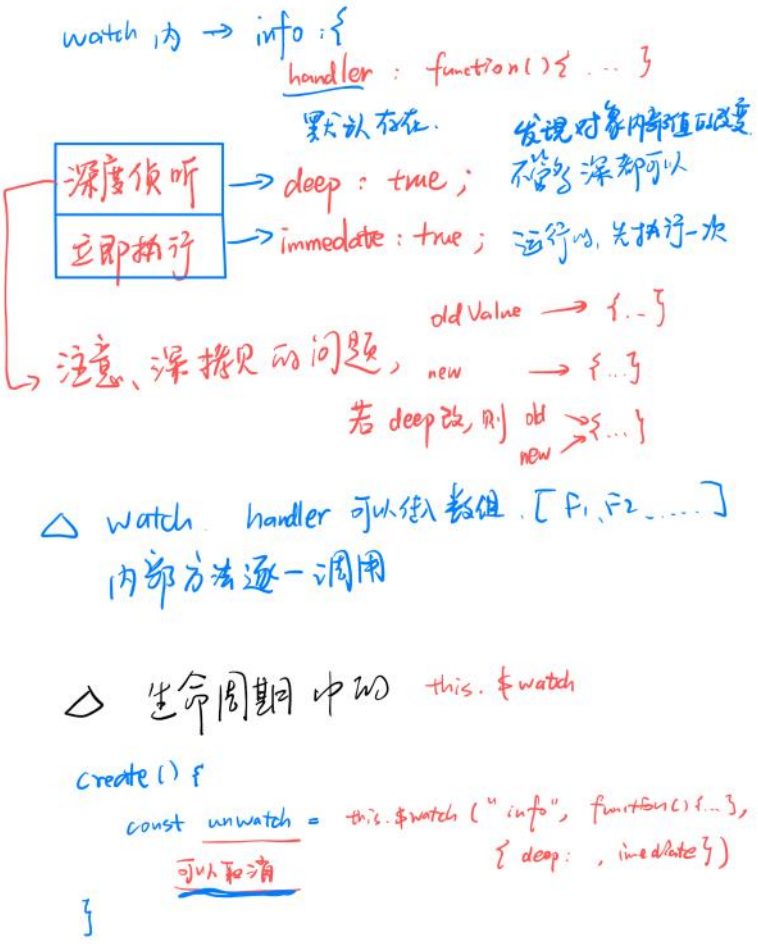
也可侦听对象属性.

"info.name" : func...

"friend[0].name" :

△ watch 配置.

默认下，侦听 对象 时，内部改变不可知. 只对 {} 整体有响应


watch 内 → info: {
handler: function(){...}
默认存在.
深度侦听
→ deep: true;
发现对象内部值的改变
不管深都可以
立即执行
→ immediate: true;
运行时, 先执行一次
注意、深拷贝的问题,
oldValue → {...}
new → {...}
若 deep 改, 则 old new → {...}
△ watch handler 可以传入数组, [f1, f2, ...]
内部方法逐一调用
△ 生命周期中的 this.$watch
create() {
const unwatch = this.$watch("info", function(){...},
{deep:, immediate})
可以取消
}

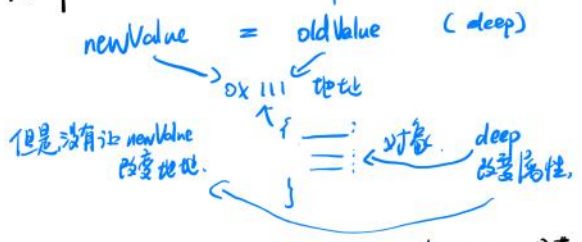
5. Vue 3 表单和开发模式

v-model + 认识、注册组件

+ 引入 webpack/vite 脚手架

△ 对之前的补充

- 查找官方文档，有更多案例。
- 侦听 watch，当使用 deep 时，



- 侦听数组 — 1. 直接 deep watch 数组，不推荐

2. 创建组件，v-for 遍历组件。

△ 深拷贝、浅拷贝

对象是引用类型.

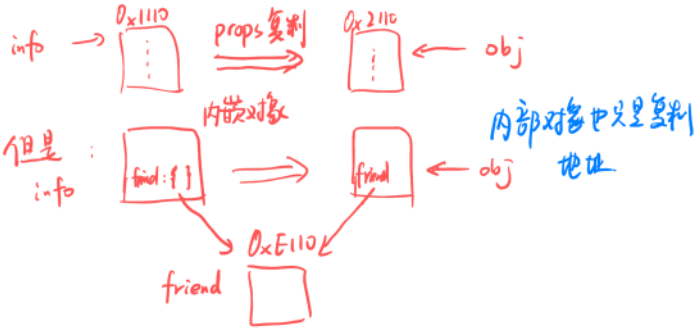
const info = { ___ , ___ } address 0x110

保存地址. → 指向. 指针. 引用

info = obj 地址拷贝、赋值

浅拷贝：

const obj = Object.assign({}, info);



深拷贝 全部新建 类似(malloc)地址，

① const obj = JSON.parse(JSON.stringify(info));

② lodash 库. 引入 src="CDN库.lodash"

下划线 _.clone(info) 浅拷贝

_.cloneDeep(info) 深拷贝

① v-model 表单提交，与用户交互的重要方法

登陆、注册、检索、创建、更新.

可在 <input>、<textarea>、<select> 元素上创建双向数据绑定。会自动匹配控件方式，本质：语法糖. 监听用户输入事件→更新数据

- v-bind绑定value属性的值；
- v-on绑定input事件监听到函数中，函数会获取最新的值赋值到绑定的属性中；

```html
<input v-model="searchText" />
```

等价于：

1. bind.
2. 监听+更新.

```html
<input :value="searchText" @input="searchText = $event.target.value" />
```

△ <input> 中 type checkbox、radio

<select> — <option>

需要通过 value 给 v-model 任意参数.

实际开发中，动态绑定 value :value = "服务器数据"

② v-model 修饰符

提交时

- v-model.lazy = "___" ，(敲回车)再返回

  替换：@change 事件

- v-model.number = "___"

  v-model 默认赋值 String，number 转为数字

- v-model.trim = "___"

  删除前后的空格

△ v-model 在组件上使用与 Vue2 不同

③ 组件认识、思想

思想：拆解复杂问题 → 小问题 + 合并

页面拆分为小功能块，独立功能，方便管理、维护

组件可以再细分！ 组件可以复用！

createApp(App).mount('#app')

组件、根组件

都会被抽象成一颗组件树；

④ 注册组件

```
const App = {
  ...
}

const app = Vue.createApp(App);
```

创建根组件

全局组件注册

可以 kebab-case，也可 PascalCase

```
app.component(组件名, 组件对象{包括 template、data、methods ...});
```

app 注册组件

局部组件：开发推荐局部注册组件.

```
const ComponentA = {
    template: "#___"
}

const App = {
    template: '#my-app',
    components: {
        ComponentA : ComponentA
        key,名称   value 组件对象
    },
    data ... methods ...
}

const app = Vue.createApp(App)
app.mount("#app");
```

局部.

局部注册.

☆不在这的组件都会被webpack打包

⑤ Vue 开发模式


.Vue 文件 → single-file components 单文件组件
可能 Webpack、Vite、rollup 工具处理 → JavaScript
服务器
client
包含 <template>
<script>
<style>
组件相关的所有
<template>
{{ greeting }} World!
</template>
<script>
module.exports = {
data: function() {
return {
greeting: "Hello"
}
}
}
</script>
<style scoped>
p {
font-size: 2em;
text-align: center;
}
</style>
→ ES6、CommonJS
→ 组件作用域的 CSS
→ 可以预处理 如 TypeScript、Babel、Less、Sass
如何支持 SFC：① Vue CLI 创建项目 最终工作方式
③ webpack 打包

Vue3组件化开发

2021年11月4日 星期四　下午 11:11　　这个文件夹里src有代码参考

MENU:

代码编写技巧

- 快捷键 vbase-css 生成代码
- Import 可以不需要写后缀，CLI基于webpack已经做好了
- 组件Header开头大写避免和html元素名重复，其他组件小写开头比较多
- 先import 并且添加组件到template，再写template会有组件名字的提示

1.组件的拆分

核心思想，对组件拆分成一个个小的组件

分成小组件，在根组件的components:{}注册好

2. 组件css

组件最好用div包裹，或者写好自己的class，不然vuecli会给样式添加根目录的id，就算style scoped，也会让根组件的css渲染到子组件。

开发最好不要直接写元素的样式，而是用class，

.title{

}

```
<div id="app" data-v-app>
  <h2 data-v-f23aade0>App</h2>
  <div data-v-b26b787e data-v-f23aade0> == $0
    <h2 data-v-b26b787e>Hello World</h2>
    <h2 data-v-b26b787e>Hello World</h2>
    <h2 data-v-b26b787e>Hello World</h2>
  </div>
</div>
```

3. 父子组件之间传递通信

①父传子 -> props 传递

父给子组件attributes传数据，子组件用props接收

attributes绑定数据的方式：

父：

```
<show-message title="呵呵呵" content="我是呵呵呵呵"></show-message>
<show-message :title="title" :content="content"></show-message>
<show-message :title="message.title" :content="message.content"></show-message>
<show-message v-bind="message"></show-message>    传递整个对象，会被拆解成各个属性
```

子组件接收数据的方式：

第一种：props数组

```
export default {
    props: ['title', 'content']
  }
```

props中，大小写不敏感。
因为DOM都会转为小写

第二种：props的对象用法

可以指定传入数据的类型 key: Type， 且有options

```
props: {
      title: [String, Number],
      content: {
        type: String,
        required: true,
        default: "123"
      },
      counter: {
        type: Number
      },
      info: {
```

对象的type可以是：

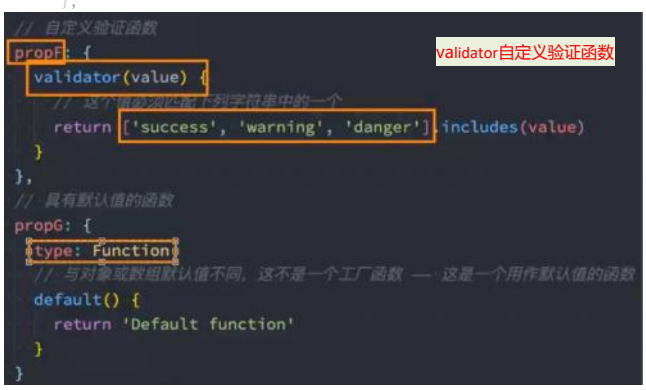
- String
- Number
- Boolean
- Array
- Object
- Date
- Function

对象(引用)类型必须是
一个返回对象的函数，
不然会造成多个组件共

```
    counter: {
      type: Number
    },
    info: {
      type: Object,
      default() {
        return {name: "why"}
      }
    },
```

对象(引用)类型必须是
一个返回对象的函数，
不然会造成多个组件共
用一个地址的对象。

- Object
- Date
- Function
- Symbol



Attribute的继承：

当组件有单个根节点时，非props的Attributes会被组件继承

例如class，id等会被直接继承

阻止继承：将attributes用于子组件自己的根元素之外的其他元素

```
inheritAttrs: false,
```

禁用后，通过$attr来获取非props的Attributes

```
<h2 v-bind="$attrs">{{title}}</h2>
<h2 :id="$attrs.id">MultiRootElement</h2>
```

②子传父 -> 事件触发

子：

<button @click = 'increment'> </button>

设置触发事件的名称

```
    emits: ["add", "sub", "addN"],
    // 对象写法的目的是为了进行参数的验证
    emits: {
      add: null,
      sub: null,
      addN: (num, name, age) => {
        console.log(num, name, age);
        if (num > 10) {
          return true
        }
        return false;
      }
```

设置触发事件对应的函数：

```
incrementN() {
        this.$emit('addN', this.num, "why", 18);
}
```

//$emit(名称，参数1 2 3)

父：

接受触发事件及其数据，接受到就执行v-on绑定的函数，函数会接收到传过来的数据：

```
<counter-operation @add="addOne"
                   @sub="subOne"
                   @addN="addNNum">
</counter-operation>
```

绑定的函数

```
addNNum(num, name, age) {
        console.log(name, age);
        this.counter += num;
      }
```

4.商品页面练习

根组件展示页面内容 子组件展示NavBar （其实页面内容也可以写一个组件）

根组件传递NavBar内容给子组件
子组件展示内容并且根据事件触发一些函数并传递一些数据给根组件。

5.非父子组件通信

一般来说工程里，用Vuex来更多，任意两个组件都能通信，无连接组件。

①Provide和Inject --- 简单数据：父传给子孙
无论层级多深，父组件都可以是子组件的依赖
Provide：{} 的This环境，绑定的是：<Script>的undefined
如果要provide引用data内的数据，则需要用provide函数

```
import { computed } from 'vue';

provide() {
      return {
        name: "why",
        age: 18,
        length: computed(() => this.names.length) // ref对象 .value
      }
    },
    data() {
      return {
        names: ["abc", "cba", "nba"]
      }
    },
```

但是需要注意的是data给provide的赋值是一次性的，并不是响应式。所以不是this.names.length,而是computed(() => this.names.length)
这是响应式语法，返回的是一个对象，所以要对length取value

②全局事件总线mitt库 vue3官方推荐

--创建一个emitter对象作为事件通道

```
import mitt from 'mitt';
const emitter = mitt();
// export const emitter1 = mitt();
// export const emitter2 = mitt();
// export const emitter3 = mitt();
export default emitter;
```

@click btnClick 在某组件触发事件

```
btnClick() {
        console.log('yes');
        emitter.emit("why", {name:"why", age:18});
      }
```

在某个组件通过emitter监听事件发生

```
created() {
  emitter.on("why", (info)=> {
    console.log(info)
  });
  //监听所有事件
  emitter.on("*", (type, e)=> {
    console.log(type,'+', e)
  });
},
```

mitt的事件取消

```
// 取消emitter中所有的监听
emitter.all.clear()

// 定义一个函数
function onFoo() {}
emitter.on('foo', onFoo)  // 监听
emitter.off('foo', onFoo)  // 取消监听
```

6.插槽Slot & v-slot语法

让组件具有更强的通用性，不固定内容跟的形式div span button

举个栗子：假如我们定制一个通用的导航组件 - NavBar

- 这个组件分成三块区域：左边-中间-右边，每块区域的内容是不固定；
- 左边区域可能显示一个菜单图标，也可能显示一个返回按钮，可能什么都不显示；
- 中间区域可能显示一个搜索框，也可能是一个列表，也可能是一个标题，等等；
- 右边可能是一个文字，也可能是一个图标，也可能什么都不显示；

Slot位置可以插入任何元素，包括组件也可以插入进去

抽取共性、预留不同子

```
<h2>组件开始</h2>
    <slot></slot>
```

```
<h2>组件结束</h2>
```

父

```
<my-slot-cpn>
   <button>ImButton</button>
</my-slot-cpn>
```

```
<template>
  <div class="nav-bar">
    <div class="left">
      <slot name="left"></slot>
    </div>
    <div class="center">
      <slot name="center"></slot>
    </div>
    <div class="right">
      <slot name="right"></slot>
    </div>
  </div>
</template>
```

name其实赋值也不是写死的

插槽的默认内容

子组件<slot>中可以预先放入元素，作为default

具名插槽：多个插槽的对应使用

```
    <nav-bar>
      <template v-slot:left>
        <button>left按钮</button>
      </template>
      <template v-slot:center>
        <h2>title</h2>
      </template>
      <template v-slot:right>
        <button>right按钮</button>
      </template>
    </nav-bar>
```

父组件可以决定插槽名

实现：父组件通过props给子组件传递name，而后slot通过vbind绑定name。

在父组件中，通过 v-slot:[name]来和具名插槽匹配。

语法：v-slot:[dynamicSlotName]

v-slot：的缩写 #　　#left　#right　#[attribute]

7. 渲染作用域 -- 作用域插槽

作用域的概念：

父组件:

```
<child-cpn :names="names">
  <template v-slot="slotProps">
    <button> {{slotProps.item}} </button>
  </template>
</child-cpn>
```

作用域插槽：

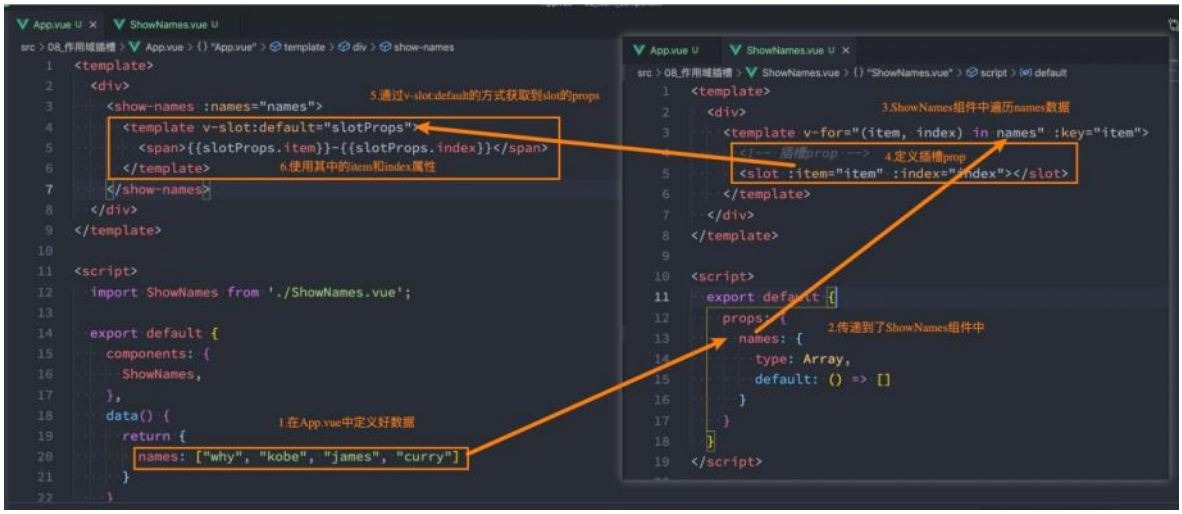
希望插槽可以访问或者指定子组件里的内容。

父组件给子组件传递数组，然后子组件遍历展示，父组件又需要通过item来完成操作

先是props传递，然后在子组件遍历出多个slot，然后给slot动态绑定对应的item，在父组件通过v-slot="slotProps" 来访问

子组件

```
<template v-for="item in names" :key="item">
      <slot :item="item"></slot>
</template>
```



后期可以用路由Router实现

所以v-slot最终语法： ***v-slot: 插槽名="自定义名字"*** <-这个包含插槽的attributes

8.动态组件 --切换组件 <component>+keep-alive

根组件按钮展示子组件的页面pages，用template+v-if else包裹对应的页面可以实现。
但是动态组件更好，<component> 特殊的attribute ***is***
is后面决定注册哪一个组件

```
<keep-alive include="home,about">
//默认组件切换的时候会销毁+新建，keep-alive可以保持在后台，提高性能
  <component :is="currentTab"
                //is决定注册哪个组件，后面的attribute是传给动态组件的参数
             name="coderwhy"
        别加空格//Ex/include使用时，pages组件最好加上name属性，方便匹配。
             :age="18"  //加冒号会传入确定传入类型，如果不加默认是string
             @pageClick="pageClick" >
  </component>
 </keep-alive>
```

keep-alive的属性

- **keep-alive有一些属性：**
  - include - string | RegExp | Array。只有名称匹配的组件会被缓存；
  - exclude - string | RegExp | Array。任何名称匹配的组件都不会被缓存；
  - max - number | string。最多可以缓存多少组件实例，一旦达到这个数字，那么缓存组件中最近没有被访问的实例会被销毁；
- **include 和 exclude prop 允许组件有条件地缓存：**
  - 二者都可以用逗号分隔字符串、正则表达式或一个数组来表示；
  - 匹配首先检查组件自身的 name 选项；

```
<!-- 逗号分隔字符串 -->
<keep-alive include="a,b">
  <component :is="view"></component>
</keep-alive>

<!-- regex (使用 `v-bind`) -->
<keep-alive :include="/a|b/">
  <component :is="view"></component>
</keep-alive>

<!-- Array (使用 `v-bind`) -->
<keep-alive :include="['a', 'b']">
  <component :is="view"></component>
</keep-alive>
```

9. Webpack的代码分包

自己的code文件都打包到*app.[HashNum].js*，第三方code文件是默认打包到chunk-vendors.js，如果工程太大不分包，会造成这个文件过大
为了优化首屏的渲染速度，最好是分包，需要的时候再下载。
解决方法：
用import函数导入所需模块，后续webpack对其打包就会进行分包操作。

```
@pageClick import("./组件使用/utils.math").then( (res) => {
  console.log(res.sum(20,30))
})//import 返回的是promise对象，result，可以解构，也可以直接使用，res.sum，sum就是其中的函数。
```

后续的异步组件就是基于webpack对import函数的操作

10. 异步组件 ——一般都是在路由里写一个异步组件

template内调用 <异步组件名> + components注册异步组件

第一种写法：传入一个函数 ——主要这种写法

```
import { defineAsyncComponent } from 'vue';

const AsyncCategory = defineAsyncComponent(() => import("./AsyncCategory.vue"))
//defineAsyncComponent函数要求传入一个函数，就用箭头函数来包裹import() --这是一个promise对象
//返回了一个异步组件
```

第二种写法：传入一个对象

```
const AsyncCategory = defineAsyncComponent({
    loader: () => import("./AsyncCategory.vue"),
    loadingComponent: Loading, //占位组件，上面的loader没加载出来时显示loading组件
    // errorComponent,
    // 在显示loadingComponent组件之前，等待多长时间
    delay: 2000,
    /**监听异步组件状况onError
     * err: 错误信息,
     * retry: 函数，调用retry尝试重新加载
      * fail: 指示加载程序结束退出
     * attempts: 记录尝试的次数
     */
    onError: function(err, retry, attempts) {
    }
  })
```

11. 异步组件和Suspense（实验性api，看官网文档）

defineAsyncComponent()方法接受返回 Promise 的工厂函数。

动态导入组件：

```
const AsyncComp = defineAsyncComponent(() =>
  import('./components/AsyncComponent.vue')
)
app.component('async-component', AsyncComp)
```

局部注册：

```
components: {
    ……
    AsyncComponent: defineAsyncComponent(() =>
      import('./components/AsyncComponent.vue')
    ),
    ……
  }
```

Suspense 是内置的全局组件，有两个插槽

default & fallback应急

用Suspense包裹异步组件，相当于异步组件传入函数中的loadingComponent属性。

```
    <suspense>
      <template #default> //默认加载这个
        <async-category></async-category>
      </template>
      <template #fallback>
        <loading></loading>
      </template>
    </suspense>
```

react比vue灵活，可以通过属性而不是插槽传入组件。

12. $refs的使用 --用的蛮多的

最好在vue开发过程中别用DOM操作，比如document.getElement这种，因为vue都封装好了。

给元素或组件绑定一个ref属性，就可以用this.$refs（对象）来访问对应的元素

--绑定一个元素

```
<h2 ref="title">哈哈哈</h2>
```

获取该DOM对象，可用innerHTML访问内容等。。。

```
this.$refs.title
```

--绑定一个组件

```
<nav-bar ref="navBar"></nav-bar>
```

组件以proxy函数被代理传入，可以访问data内的属性，methods内的函数

```
console.log(this.$refs.navBar.message);
```

```
this.$refs.navBar.sayHello();
        // $el
console.log(this.$refs.navBar.$el);
```

13. $parent 和 $root组件 和 $el --用的少，耦合性太强

$parent拿到父元素

$root根组件

navbar.$el拿根元素——包裹着这个组件或者元素的根元素

14. 生命周期

组件周期：创建、挂载、更新、卸载等

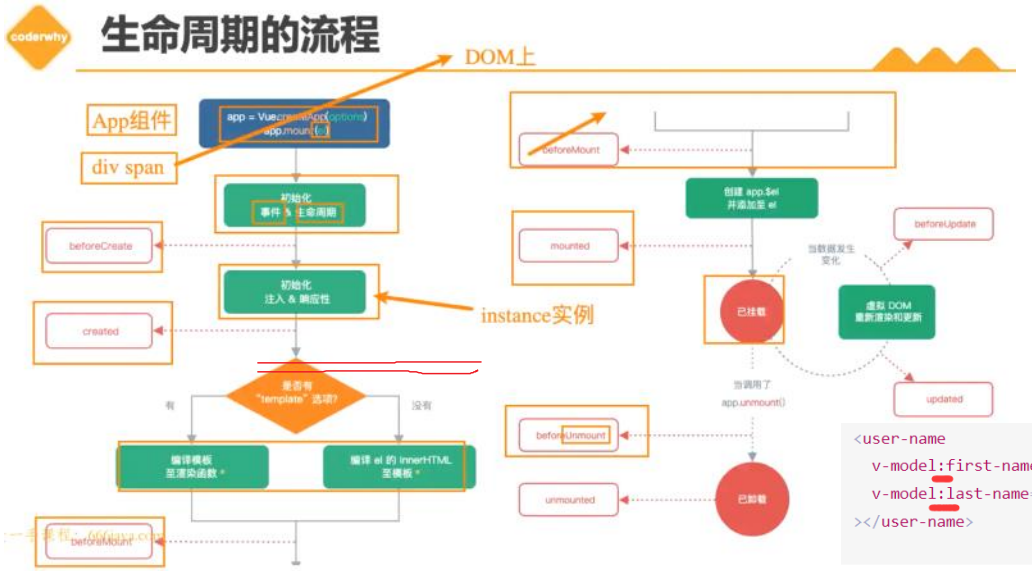
如果要在某一个阶段添加自己的代码逻辑，比如创建完请求服务器一些数据，需要生命周期函数。

生命周期函数是一些**钩子函数**，会被Vue源码内部回调，回调后就知道现在处于什么阶段，然后就可以编写该生命周期的逻辑代码了。

钩子函数：钩子函数是在一个事件触发的时候，在系统级捕获到了他，然后做一些操作。一段用以处理系统消息的程序。“钩子”就是在某个阶段给你一个做某些处理的机会。

1、钩子函数是个函数，在系统消息触发时被系统调用

2、不是用户自己触发的



每个阶段都是一个生命周期函数，到了阶段就触发

```
<user-name
  v-model:first-name="firstName"
  v-model:last-name="lastName"
></user-name>
```

```
app.component('user-name', {
  props: {
    firstName: String,
    lastName: String
  },
  emits: ['update:firstName', 'update:lastName'],
  template: `
    <input
      type="text"
      :value="firstName"
      @input="$emit('update:firstName', $event.target.value)">

    <input
      type="text"
      :value="lastName"
      @input="$emit('update:lastName', $event.target.value)">
  `
})
```

v-model.自定义修饰符

特定事件名: update: <props.xxx>
Default 为 update: modelValue
$emits:[包含此事件]，触发后，父组件会自动更新v-model 绑定的 <props.xxx>参数

15. v-model 绑定组件

```
<div id="app">
  <my-component v-model.capitalize="myText"></my-component>
  {{ myText }}
</div>
```

```
const app = Vue.createApp({
  data() {
    return {
      myText: ''
    }
  }
})

app.component('my-component', {
```

```html
<div id="app">
  <my-component v-model.capitalize="myText"></my-component>
  {{ myText }}
</div>
```

```
ht
      type="text"
      :value="lastName"
      @input="$emit('update:lastName', $event.target.value)">
  `
})
```

```
js
const app = Vue.createApp({
  data() {
    return {
      myText: ''
    }
  }
})

app.component('my-component', {
  props: {
    modelValue: String,
    modelModifiers: {
      default: () => ({})
    }
  },
  emits: ['update:modelValue'],
  methods: {
    emitValue(e) {
      let value = e.target.value
      if (this.modelModifiers.capitalize) {
        value = value.charAt(0).toUpperCase() + value.slice(1)
      }
      this.$emit('update:modelValue', value)
    }
  },
  template: `<input
    type="text"
    :value="modelValue"
    @input="emitValue">`
})

app.mount('#app')
```

可复用&组合 CompositionAPI

2021年11月29日 星期一　　上午 10:28

1．组合式 API

setup 选项是一个接收 props 和 context 的函数。

此外，我们将 setup 返回的所有内容都暴露给组件的其余部分（计算属性、方法、生命周期钩子等等）以及组件的模板。

setup不能用this ！！！

这是一个包含多个功能的setup，后面会对其进行拆分组合，就是组合式API的用法。

```
setup (props) {
  // 使用 `toRefs` 创建对prop的 `user` property 的响应式引用
  const { user } = toRefs(props)

  let repositories = ref([])
  const getUserRepositories = async () => {
    repositories = await fetchUserRepositories(props.user)
  }

  onMounted(getUserRepositories) // 在 `mounted` 时调用 `getUserRepositories`
  watch(user, getUserRepositories)

    const searchQuery = ref('')
  const repositoriesMatchingSearchQuery = computed(() => {
    return repositories.value.filter(
      repository => repository.name.includes(searchQuery.value)
    )
  })

  return {
    repositories,
    getUserRepositories // 返回的函数与方法的行为相同
       searchQuery,
       repositoriesMatchingSearchQuery
  }
}
```

每当我们调用 getUserRepositories 时，repositories 都将发生变化，视图也会更新以反映变化。

以上代码含有知识点：

- ref响应式变量 ： const counter = ref(0)　// counter = { value : 5 } ->Object in 响应式系统
- const {user} = toRefs(props) 确保侦听器能监听到props中的user变化，将其加入响应式
- 组合式 API 上的生命周期钩子与选项式 API 的名称相同，但前缀为 on：即 mounted 看起来会像 onMounted。
- watch(listener, callback) 侦听，和watch: { }功能一样
- computed 函数传递了第一个参数，它是一个类似 getter 的回调函数，输出的是一个只读的响应

式引用。

将上面的setup代码拆分为单独的功能模块，作为函数引入组件：

```
import { fetchUserRepositories } from '@/api/repositories'
import { ref, onMounted, watch } from 'vue'
export default function useUserRepositories(user) {
  const repositories = ref([])
  const getUserRepositories = async () => {
    repositories.value = await fetchUserRepositories(user.value)
  }

  onMounted(getUserRepositories)
  watch(user, getUserRepositories)

  return {
    repositories,
    getUserRepositories
  }
}

import { ref, computed } from 'vue'
export default function useRepositoryNameSearch(repositories) {
  const searchQuery = ref('')
  const repositoriesMatchingSearchQuery = computed(() => {
    return repositories.value.filter(repository => {
      return repository.name.includes(searchQuery.value)
    })
  })

  return {
    searchQuery,
    repositoriesMatchingSearchQuery
  }
}
setup (props) {
    const { user } = toRefs(props)

    const { repositories, getUserRepositories } = useUserRepositories(user)

    const {
      searchQuery,
      repositoriesMatchingSearchQuery
    } = useRepositoryNameSearch(repositories)      用对象接受该功能模块的return

    return {
      // 因为我们并不关心未经过滤的仓库
      // 我们可以在 `repositories` 名称下暴露过滤后的结果
      repositories: repositoriesMatchingSearchQuery,
      getUserRepositories,
      searchQuery,
    }
  },
```

2. setup 扩展

```
Setup( props, context) {  }
```

props解构，用toRefs()

或者

添加新的ref到props，const *newElement*= toRef(props, '*newElement*');

context 是一个普通 JavaScript 对象，暴露了其它可能在 setup 中有用的值

context可以直接解构

```
context = { attrs, slots, emit, expose }
```

执行 `setup` 时，组件实例尚未被创建。因此，你只能访问以下 property：

- `props`
- `attrs`
- `slots`
- `emit`

换句话说，你**将无法访问**以下组件选项：

- `data`
- `computed`
- `methods`
- `refs` (模板 ref)

setup 还可以返回一个渲染函数

```
expose({
      increment    //用expose来解决返回渲染函数时，还需要给外界传递attributes或func( )的情况
    })
return ( ) => h('div', count.value)
```

3. 生命周期钩子 lifecycle - hooks

8个阶段对应hooks( )

```
beforeCreate
Created
beforeMount    onBeforeMount
mounted        onMounted
beforeUpdate   onBeforeUpdate
updated        onUpdated
beforeUnmount  onBeforeUnmount
Unmounted
```

created和$watch实例方法一起用，创建后监听某个值的变化

```
 created() {
    // 顶层property 名
    this.$watch('a', (newVal, oldVal) => {
      // 做点什么 })
   this.$watch(
       () => this.a + this.b, //监听复杂表达式
       (newVal, oldVal) => {
        // 做点什么
    })
```

4. provide/inject

provide 函数：（name（<String> 类型），value）

inject 函数：（要 inject 的 property 的 name ，默认值（可选））

Tips：

- 为了增加 provide 值和 inject 值之间的响应性，我们可以在 provide 值时使用 ref 或 reactive。
- 尽可能将对响应式 property 的所有修改限制在定义 provide 的组件内部。
- 如果要确保通过 provide 传递的数据不会被 inject 的组件更改，建议对提供者的 property 使用 readonly。

5. Mixin

当组件和 mixin 对象含有同名选项时，这些选项将以恰当的方式进行“合并”。

```
const myMixin = {
  methods: {
    foo() {
      console.log('foo')
    },
    conflicting() {
      console.log('from mixin')
    }
  }
}

const app = Vue.createApp({
  mixins: [myMixin],
  methods: {
    bar() {
      console.log('bar')
    },
    conflicting() {
      console.log('from self')
    }
  }
})

const vm = app.mount('#mixins-basic')

vm.foo() // => "foo"
vm.bar() // => "bar"
vm.conflicting() // => "from self" // 两个对象键名冲突时，取组件对象的键值对。
```

全局Mixin：app.mixin({ })　一旦使用全局 mixin，它将影响每一个之后创建的组件（例如，每个子组件）。

6．自定义指令

和生命周期钩子一起建设

定义一个v-focus指令

```
directives: {
  focus: {
    // 指令的定义
    mounted(el , binding) {    //el是当前元素，binding是指令后的参数
      el.focus()
    }
  }
}
```

7.Teleport

<teleport to="body">，告诉 Vue “Teleport 这个 HTML 到该 ‘body’ 标签”。

teleport标签内放置自己搭建好的元素，可以是html元素，也可以是组件标签，形成UI

8．渲染函数render

用处，快速生成template内重复的html节点内容，比如冗长的v-if else， v-for

```
<template v-slot:header>
   <h1>About Me</h1>
 </template>
...
...
const app = createApp({})

app.component('blog-post', {
  render() {
    return h('div', [
      h('header', this.$slots.header()),//this.$slot.元素--访问插槽分发的内容
      h('main', this.$slots.default()),
//也可以用Arrays.map()重复渲染子节点
      h('footer', this.$slots.footer())
    ])
  }
})
```

h() 函数是一个用于创建 VNode 的实用程序。也许可以更准确地将其命名为 createVNode()，但由于频繁使用和简洁，它被称为 h() 。

它接受三个参数，

h(元素
, { , , } -- props 或 attributes /可选
,[,] -- 包含其子节点的数组 /可选)

9. 插件 plugins

Install()
Use()

Vue3响应式原理

2021年11月29日 星期一　　下午 12:42

1．声明响应式状态reactive

```
const state = reactive({
  count: 0
})
```

reactive 相当于 Vue 2.x 中的 Vue.observable() API

该 API 返回一个响应式的对象状态。该响应式转换是“深度转换”—它会影响传递对象的所有嵌套 property。Vue 中响应式状态的基本用例是我们可以在渲染期间使用它。因为依赖跟踪的关系，当响应式状态改变时视图会自动更新。

data( ) { Return {
　　value:0;
　}}
data返回的对象在vue内部交由reactive()，使其成为响应式对象。

Tips: 当传入Arrays、Sets、Maps时，reactive会对其解包，访问其中的值需要.value
console.log(books[0].value)
console.log(map.get('count').value)

2．独立响应式值 refs

const count = ref(0)
也就是　reactive ({ ref() })　　reactive -> 对象　ref -> reactive()中的值，但是本质也是对象，警惕js对象引用的问题

如果将新的 ref 赋值给现有 ref 的 property，将会替换旧的 ref：

const otherCount = ref(2)

state.count = otherCount

```
//相当于state.count原来引用count = ref(1) 现在引用otherCount, 但是count本身没变。
console.log(state.count) // 2
console.log(count.value) // 1
```

3. 解构reactive()对象 —— toRefs(*)

toRefs将响应式对象转换为一组 ref。这些 ref 将保留与源对象的响应式关联。

4. 使用 readonly 防止更改响应式对象

本质： 创建一个只读的 proxy 对象

```
const original = reactive({ count: 0 })  //original 可改动

const copy = readonly(original)  //copy不可改动
```

Vue Router

2021年11月30日 星期二　　下午 8:00

1．安装

```
npm install vue-router@4
```

2．简单上手

用 Vue + Vue Router 创建单页应用非常简单：通过 Vue.js，我们已经用组件组成了我们的应用。

当加入 Vue Router 时，我们需要做的就是将我们的组件映射到路由上，让 Vue Router 知道在哪里渲染它们。

Vue.js + Vue Router示例

```
// 1. 定义路由组件.
// 也可以从其他文件导入
const Home = { template: '<div>Home</div>' }
const About = { template: '<div>About</div>' }

// 2. 定义一些路由
// 每个路由都需要映射到一个组件。
// 我们后面再讨论嵌套路由。
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
]

// 3. 创建路由实例并传递 `routes` 配置
// 你可以在这里输入更多的配置，但我们在这里暂时保持简单
const router = VueRouter.createRouter({
  // 4. 内部提供了 history 模式的实现。为了简单起见，我们在这里使用 hash 模式。
  history: VueRouter.createWebHashHistory(),
  routes, // `routes: routes` 的缩写
})

// 5. 创建并挂载根实例
const app = Vue.createApp({})
//确保 _use_ 路由实例使
//整个应用支持路由。
app.use(router)
```

```
app.mount('#app')
```

this.$router 访问路由

Setup() 中用 useRouter useRoute函数

3. 动态路由匹配

路径参数 用 : 表示

匹配模式	匹配路径	$route.params
/users/:username	/users/eduardo	{ username: 'eduardo' }
/users/:username/posts/:postId	/users/eduardo/posts/123	{ username: 'eduardo', postId: '123' }

Vuex 状态管理

2021年11月30日 星期二　下午 8:00

0. 简单原理

其实使用过 Vuex 的同学都知道，我们在页面或者组件中都是通过this.$store.xxx来调用的，那么其实，我们只要把你所创建的store对象赋值给页面或者组件中的$store变量即可.

Vuex的原理通俗讲就是：利用了全局混入Mixin，将你所创建的store对象，混入到每一个Vue实例中

```
const install = (v) => { // 参数v负责接收vue实例
    Vue = v;
    // 全局混入
    Vue.mixin({
        beforeCreate() {
            if (this.$options && this.$options.store) {
                // 根页面，直接将身上的store赋值给自己的$store，这也解释了为什么使用vuex要先把store放到入口文件main.js里的根Vue实例里
                this.$store = this.$options.store;
            } else {
                // 除了根页面以外，将上级的$store赋值给自己的$store
                this.$store = this.$parent && this.$parent.$store;
            }
        },
    })
}
```

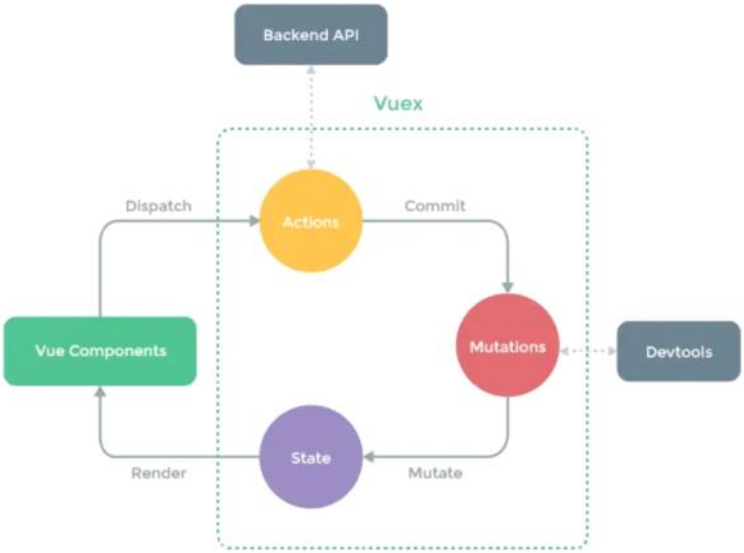
1. 简单入门

```
import { createApp } from 'vue'
import { createStore } from 'vuex'

// 创建一个新的 store 实例
const store = createStore({
  state () {
    return {
      count: 0
    }
  },
  mutations: {
    increment (state) {
      state.count++
    }
  }
})

const app = createApp({ /* 根组件 */ })

// 将 store 实例作为插件安装
app.use(store)
```



现在，可以通过 store.state 来获取状态对象，并通过 store.commit 方法触发状态变更:

```
store.commit('increment')
```

在 Vue 组件中， 可以通过 this.$store 访问store实例。现在我们可以从组件的方法提交一个变更:

```
methods: {
  increment() {
    this.$store.commit('increment')
    console.log(this.$store.state.count)
  }
}
```

2. 获取State

第一种：

```
computed: {
    count () {
      return this.$store.state.count
    }
  }
```

第二种：需要获取多个state时，用mapState函数，传入对象，或者字符串数组

```
computed: mapState({
    // 箭头函数可使代码更简练
    count: state => state.count,

    // 传字符串参数 'count' 等同于 `state => state.count`
    countAlias: 'count',

    // 为了能够使用 `this` 获取局部状态，必须使用常规函数
    countPlusLocalState (state) {
      return state.count + this.localCount
    }
  })
```

3.getters

```
  state: {
    todos: [
      { id: 1, text: '...', done: true },
      { id: 2, text: '...', done: false }
    ]
  },
  getters: {
    doneTodos: state => {
      return state.todos.filter(todo => todo.done)
    }
  }
```

注意，getter 在通过属性访问时是作为 Vue 的响应式系统的一部分缓存其中的。

Getters - state 类似computed - data的关系

需要获取多个getters时，用mapGetters(obj or [String]) 辅助函数

4. Mutation

mutation类似于事件，有组件申请mutation就会触发vuex中的mutation中对应的methods

CODE：

```
   state: {
    count: 1
  },
  mutations: {
    increment (state, n) { // n为参数，可设置，可不设置，此参数也称为“载荷”
      // 变更状态
      state.count++
    }
  }
```

使用：this.$store.commit('increment', 10)

辅助函数mapMutations( )

```
methods: {
    ...mapMutations([
      'increment', // 将 `this.increment()` 映射为 `this.$store.commit('increment')`

      // `mapMutations` 也支持载荷：
      'incrementBy' // 将 `this.incrementBy(amount)` 映射为 `this.$store.commit('incrementBy', amount)`
    ]),
    ...mapMutations({
      add: 'increment' // 将 `this.add()` 映射为 `this.$store.commit('increment')`
    })
  }
```

5. Action

Action 类似于 mutation，不同在于：

- Action 提交的是 mutation，而不是直接变更状态。
- Action 可以包含任意异步操作。

```
   actions: {
      // Action 函数接受一个与 store 实例具有相同方法和属性的 context 对
象
     incrementAsync (context , n) {
       setTimeout(() => {
         context.commit('increment') //提交mutation，且异步
       }, 1000)
      }
  }
```

使用：

dispatch方法，执行actions的'name'方法

```
    dispatch(name, payload) {
        this.actions[name](payload);
    }
```

也有辅助函数 mapAction

6.Module

由于使用单一状态树，应用的所有状态会集中到一个比较大的对象。当应用变得非常复杂时，store 对象就有可能变得相当臃肿。

为了解决以上问题，Vuex 允许我们将 store 分割成模块（module）。

每个模块预设好自己的 state、mutation、action、getter、甚至是嵌套子模块。

最后以modules:{ moduleA, moduleB, ….}的方式传入const store = Vue.Store(--);

```
const moduleA = {
  state: () => ({ ... }),
  mutations: { ... },
  actions: { ... },
  getters: { ... }
}

const moduleB = {
  state: () => ({ ... }),
  mutations: { ... },
  actions: { ... }
}

const store = new Vuex.Store({
  modules: {
    a: moduleA,
    b: moduleB
  }
})

store.state.a // -> moduleA 的状态
store.state.b // -> moduleB 的状态
```

对于模块内部的 mutation 和 getter，接收的第一个参数是模块的局部状态对象。

```
In moduleA
  mutations: {
    increment (state) {
      // 这里的 `state` 对象是模块的局部状态
      state.count++
    }
  },
```

同样，对于模块内部的 action，局部状态通过 context.state 暴露出来，

根节点状态则为 context.rootState：