

Vue3快速上手

1. Vue3简介

1.1. 【性能的提升】

1.2. 【源码的升级】

1.3. 【拥抱TypeScript】

1.4. 【新的特性】

2. 创建Vue3工程

2.1. 【基于 vue-cli 创建】

2.2. 【基于 vite 创建】(推荐)

2.3. 【一个简单的效果】

3. Vue3核心语法

3.1. 【OptionsAPI 与 CompositionAPI】

Options API 的弊端

Composition API 的优势

3.2. 【拉开序幕的 setup】

setup 概述

setup 的返回值

setup 与 Options API 的关系

setup 语法糖

3.3. 【ref 创建：基本类型的响应式数据】

3.4. 【reactive 创建：对象类型的响应式数据】

3.5. 【ref 创建：对象类型的响应式数据】

3.6. 【ref 对比 reactive】

3.7. 【toRefs 与 toRef】

3.8. 【computed】

3.9. 【watch】

* 情况一

* 情况二

* 情况三

* 情况四

* 情况五

3.10. 【watchEffect】

3.11. 【标签的 ref 属性】

3.12. 【props】

3.13. 【生命周期】

3.14. 【自定义hook】

4. 路由

4.1. 【对路由的理解】

4.2. 【基本切换效果】

4.3. 【两个注意点】

4.4. 【路由器工作模式】

4.5. 【to的两种写法】

4.6. 【命名路由】

4.7. 【嵌套路由】

4.8. 【路由传参】

query参数

params参数

4.9. 【路由的props配置】

4.10. 【 replace属性】

4.11. 【编程式导航】

4.12. 【重定向】

5. pinia

5.1 【准备一个效果】

5.2 【搭建 pinia 环境】

5.3 【存储+读取数据】

5.4. 【修改数据】(三种方式)

5.5. 【storeToRefs】

5.6. 【getters】

5.7. 【\$subscribe】

5.8. 【store组合式写法】

6. 组件通信

- 6.1. 【props】
- 6.2. 【自定义事件】
- 6.3. 【mitt】
- 6.4. 【v-model】
- 6.5. 【\$attrs】
- 6.6. 【\$refs、\$parent】
- 6.7. 【provide、inject】

2. 具名插槽

3. 作用域插槽

7. 其它 API

7.1. 【shallowRef 与 shallowReactive】

shallowRef

shallowReactive

总结

7.2. 【readonly 与 shallowReadonly】

readonly

shallowReadonly

7.3. 【toRaw 与 markRaw】

toRaw

markRaw

7.4. 【customRef】

8. Vue3新组件

8.1. 【Teleport】

8.2. 【Suspense】

8.3. 【全局API转移到应用对象】

8.4. 【其他】

1. Vue3简介

- 2020年9月18日，`Vue.js` 发布版 `3.0` 版本，代号：`One Piece` (n
- 经历了：4800+次提交、40+个RFC、600+次PR、300+贡献者
- 官方发版地址：[Release v3.0.0 One Piece · vuejs/core](#)

- 截止2023年10月，最新的公开版本为： 3.3.4



1.1. 【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%。

1.2. 【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`。

1.3. 【拥抱TypeScript】

- `Vue3` 可以更好的支持 `TypeScript`。

1.4. 【新的特性】

1. `Composition API` (组合 `API`) :

- `setup`
- `ref` 与 `reactive`
- `computed` 与 `watch`

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.....

2. 创建Vue3工程

2.1. 【基于 vue-cli 创建】

点击查看[官方文档](#)

备注：目前 `vue-cli` 已处于维护模式，官方推荐基于 `Vite` 创建项目。

```
1 ## 查看@vue/cli版本，确保@vue/cli版本在4.5.0以上
2 vue --version
3
4 ## 安装或者升级你的@vue/cli
5 npm install -g @vue/cli
6
7 ## 执行创建命令
8 vue create vue_test
9
10 ## 随后选择3.x
11 ## Choose a version of Vue.js that you want to start the project with (Use arrow keys)
12 ## > 3.x
13 ##   2.x
14
15 ## 启动
16 cd vue_test
17 npm run serve
```

2.2. 【基于 vite 创建】(推荐)

`vite` 是新一代前端构建工具，官网地址：<https://vitejs.cn>，`vite` 的优势如下：

- 轻量快速的热重载（`HMR`），能实现极速的服务启动。
- 对 `TypeScript`、`JSX`、`CSS` 等支持开箱即用。

- 真正的按需编译，不再等待整个应用编译完成。

- `webpack` 构建与 `vite` 构建对比图如下：



- 具体操作如下（点击查看[官方文档](#)）

```
1 ## 1. 创建命令
2 npm create vue@latest
3
4 ## 2. 具体配置
5 ## 配置项目名称
6 ✓ Project name: vue3_test
7 ## 是否添加TypeScript支持
8 ✓ Add TypeScript? Yes
9 ## 是否添加JSX支持
10 ✓ Add JSX Support? No
11 ## 是否添加路由环境
12 ✓ Add Vue Router for Single Page Application development? No
13 ## 是否添加pinia环境
14 ✓ Add Pinia for state management? No
15 ## 是否添加单元测试
16 ✓ Add Vitest for Unit Testing? No
17 ## 是否添加端到端测试方案
18 ✓ Add an End-to-End Testing Solution? » No
19 ## 是否添加ESLint语法检查
20 ✓ Add ESLint for code quality? Yes
21 ## 是否添加Prettier代码格式化
22 ✓ Add Prettier for code formatting? No
```

自己动手编写一个App组件

```
1 <template>
2   <div class="app">
3     <h1>你好啊! </h1>
4
5   </div>
6
7 </template>
8
9 <script lang="ts">
10  export default {
11    name: 'App' //组件名
12  }
13 </script>
14
15 <style>
16  .app {
17    background-color: #ddd;
18    box-shadow: 0 0 10px;
19    border-radius: 10px;
20    padding: 20px;
21  }
22 </style>
23
```

安装官方推荐的 `vscode` 插件：



总结：

- `Vite` 项目中，`index.html` 是项目的入口文件，在项目最外层。
- 加载 `index.html` 后，`Vite` 解析 `<script type="module" src="xxx">` 指向的 `JavaScript`。
- `Vue3` 中是通过 `createApp` 函数创建一个应用实例。

2.3. 【一个简单的效果】

`Vue3` 向下兼容 `Vue2` 语法，且 `Vue3` 中的模板中可以没有根标签

```
1  <template>
2    <div class="person">
3      <h2>姓名: {{name}}</h2>
4
5      <h2>年龄: {{age}}</h2>
6
7      <button @click="changeName">修改名字</button>
8
9      <button @click="changeAge">年龄+1</button>
10
11     <button @click="showTel">点我查看联系方式</button>
12
13   </div>
14
15 </template>
16
17 <script lang="ts">
18   export default {
19     name:'App',
20     data() {
21       return {
22         name:'张三',
23         age:18,
24         tel:'13888888888'
25       }
26     },
27     methods:{
28       changeName(){
29         this.name = 'zhang-san'
30       },
31       changeAge(){
32         this.age += 1
33       },
34       showTel(){
35         alert(this.tel)
36       }
37     },
38   }
39 </script>
40
```

3. Vue3核心语法

3.1. 【OptionsAPI 与 CompositionAPI】

- Vue2 的 API 设计是 Options (配置) 风格的。
- Vue3 的 API 设计是 Composition (组合) 风格的。

Options API 的弊端

Options 类型的 API，数据、方法、计算属性等，是分散在： data 、 methods 、 computed 中的，若想新增或者修改一个需求，就需要分别修改： data 、 methods 、 computed ，不便于维护和复用。



Composition API 的优势

可以用函数的方式，更加优雅的组织代码，让相关功能的代码更加有序的组织在一起。



说明：以上四张动图原创作者：大帅老猿

3.2. 【拉开序幕的 setup】

setup 概述

setup 是 Vue3 中一个新的配置项，值是一个函数，它是 Composition API “表演的舞台”，组件中所用到的：数据、方法、计算属性、监视……等等，均配置在 setup 中。

特点如下：

- setup 函数返回的对象中的内容，可直接在模板中使用。
- setup 中访问 this 是 undefined 。
- setup 函数会在 beforeCreate 之前调用，它是“领先”所有钩子执行的。

```
1  <template>
2    <div class="person">
3      <h2>姓名: {{name}}</h2>
4
5      <h2>年龄: {{age}}</h2>
6
7      <button @click="changeName">修改名字</button>
8
9      <button @click="changeAge">年龄+1</button>
10
11     <button @click="showTel">点我查看联系方式</button>
12
13   </div>
14
15 </template>
16
17 <script lang="ts">
18   export default {
19     name: 'Person',
20     setup() {
21       // 数据, 原来写在data中 (注意: 此时的name、age、tel数据都不是响应式数据)
22       let name = '张三'
23       let age = 18
24       let tel = '13888888888'
25
26       // 方法, 原来写在methods中
27       function changeName() {
28         name = 'zhang-san' // 注意: 此时这么修改name页面是不变的
29         console.log(name)
30       }
31       function changeAge() {
32         age += 1 // 注意: 此时这么修改age页面是不变的
33         console.log(age)
34       }
35       function showTel() {
36         alert(tel)
37       }
38
39       // 返回一个对象, 对象中的内容, 模板中可以直接使用
40       return {name, age, tel, changeName, changeAge, showTel}
41     }
42   }
43 </script>
44
```

setup 的返回值

- 若返回一个对象：则对象中的：属性、方法等，在模板中均可以直接使用**（重点关注）。**
- 若返回一个函数：则可以自定义渲染内容，代码如下：

```
1  setup(){
2      return ()=> '你好啊！'
3 }
```

setup 与 Options API 的关系

- Vue2 的配置（`data`、`methos`）中可以访问到 `setup` 中的属性、方法。
- 但在 `setup` 中不能访问到 Vue2 的配置（`data`、`methos`）。
- 如果与 Vue2 冲突，则 `setup` 优先。

setup 语法糖

`setup` 函数有一个语法糖，这个语法糖，可以让我们把 `setup` 独立出去，代码如下：

```
1  <template>
2    <div class="person">
3      <h2>姓名: {{name}}</h2>
4
5      <h2>年龄: {{age}}</h2>
6
7      <button @click="changName">修改名字</button>
8
9      <button @click="changAge">年龄+1</button>
10
11     <button @click="showTel">点我查看联系方式</button>
12
13   </div>
14
15 </template>
16
17 <script lang="ts">
18   export default {
19     name: 'Person',
20   }
21 </script>
22
23 <!-- 下面的写法是setup语法糖 -->
24 <script setup lang="ts">
25   console.log(this) //undefined
26
27   // 数据 (注意: 此时的name、age、tel都不是响应式数据)
28   let name = '张三'
29   let age = 18
30   let tel = '13888888888'
31
32   // 方法
33   function changName(){
34     name = '李四' //注意: 此时这么修改name页面是不变化的
35   }
36   function changAge(){
37     console.log(age)
38     age += 1 //注意: 此时这么修改age页面是不变化的
39   }
40   function showTel(){
41     alert(tel)
42   }
43 </script>
44
```

扩展：上述代码，还需要编写一个不写 `setup` 的 `script` 标签，去指定组件名字，比较麻烦，我们可以借助 `vite` 中的插件简化

1. 第一步：`npm i vite-plugin-vue-setup-extend -D`

2. 第二步：`vite.config.ts`

```
1 import { defineConfig } from 'vite'
2 import VueSetupExtend from 'vite-plugin-vue-setup-extend'
3
4 export default defineConfig({
5   plugins: [ VueSetupExtend() ]
6 })
```

3. 第三步：`<script setup lang="ts" name="Person">`

3.3. 【ref 创建：基本类型的响应式数据】

- **作用：**定义响应式变量。
- 语法：`let xxx = ref(初始值)`。
- **返回值：**一个 `RefImpl` 的实例对象，简称 `ref对象` 或 `ref`，`ref` 对象的 `value` 属性是响应式的。
- 注意点：
 - JS 中操作数据需要：`xxx.value`，但模板中不需要 `.value`，直接使用即可。
 - 对于 `let name = ref('张三')` 来说，`name` 不是响应式的，`name.value` 是响应式的。

```
1  <template>
2    <div class="person">
3      <h2>姓名: {{name}}</h2>
4
5      <h2>年龄: {{age}}</h2>
6
7      <button @click="changeName">修改名字</button>
8
9      <button @click="changeAge">年龄+1</button>
10
11     <button @click="showTel">点我查看联系方式</button>
12
13   </div>
14
15 </template>
16
17 <script setup lang="ts" name="Person">
18   import {ref} from 'vue'
19   // name和age是一个RefImpl的实例对象，简称ref对象，它们的value属性是响应式的。
20   let name = ref('张三')
21   let age = ref(18)
22   // tel就是一个普通的字符串，不是响应式的
23   let tel = '13888888888'
24
25   function changeName(){
26     // JS中操作ref对象时候需要.value
27     name.value = '李四'
28     console.log(name.value)
29
30     // 注意：name不是响应式的，name.value是响应式的，所以下面代码并不会引起页面的更新。
31     // name = ref('zhang-san')
32   }
33   function changeAge(){
34     // JS中操作ref对象时候需要.value
35     age.value += 1
36     console.log(age.value)
37   }
38   function showTel(){
39     alert(tel)
40   }
41 </script>
42
```

3.4. 【reactive 创建：对象类型的响应式数据】

- 作用：定义一个响应式对象（基本类型不要用它，要用 `ref`，否则报错）
- 语法：`let 响应式对象 = reactive(源对象)`。
- **返回值：**一个 `Proxy` 的实例对象，简称：响应式对象。
- 注意点：`reactive` 定义的响应式数据是“深层次”的。

```

1  <template>
2    <div class="person">
3      <h2>汽车信息: 一台{{ car.brand }}汽车, 价值{{ car.price }}万</h2>
4
5      <h2>游戏列表: </h2>
6
7      <ul>
8        <li v-for="g in games" :key="g.id">{{ g.name }}</li>
9
10     </ul>
11
12    <h2>测试: {{obj.a.b.c.d}}</h2>
13
14    <button @click="changeCarPrice">修改汽车价格</button>
15
16    <button @click="changeFirstGame">修改第一游戏</button>
17
18    <button @click="test">测试</button>
19
20  </div>
21
22 </template>
23
24 <script lang="ts" setup name="Person">
25   import { reactive } from 'vue'
26
27   // 数据
28   let car = reactive({ brand: '奔驰', price: 100 })
29   let games = reactive([
30     { id: 'ahsgdyfa01', name: '英雄联盟' },
31     { id: 'ahsgdyfa02', name: '王者荣耀' },
32     { id: 'ahsgdyfa03', name: '原神' }
33   ])
34   let obj = reactive({
35     a: {
36       b: {
37         c: {
38           d: 666
39         }
40       }
41     }
42   })
43
44   function changeCarPrice() {
45     car.price += 10
46   }
47   function changeFirstGame() {

```

```
48     games[0].name = '流星蝴蝶剑'
49   }
50   function test(){
51     obj.a.b.c.d = 999
52   }
53 </script>
54
```

3.5. 【ref 创建：对象类型的响应式数据】

- 其实 `ref` 接收的数据可以是：基本类型、对象类型。
- 若 `ref` 接收的是对象类型，内部其实也是调用了 `reactive` 函数。

```
1  <template>
2    <div class="person">
3      <h2>汽车信息: 一台{{ car.brand }}汽车, 价值{{ car.price }}万</h2>
4
5      <h2>游戏列表: </h2>
6
7      <ul>
8        <li v-for="g in games" :key="g.id">{{ g.name }}</li>
9
10     </ul>
11
12    <h2>测试: {{obj.a.b.c.d}}</h2>
13
14    <button @click="changeCarPrice">修改汽车价格</button>
15
16    <button @click="changeFirstGame">修改第一游戏</button>
17
18    <button @click="test">测试</button>
19
20  </div>
21
22 </template>
23
24 <script lang="ts" setup name="Person">
25   import { ref } from 'vue'
26
27   // 数据
28   let car = ref({ brand: '奔驰', price: 100 })
29   let games = ref([
30     { id: 'ahsgdyfa01', name: '英雄联盟' },
31     { id: 'ahsgdyfa02', name: '王者荣耀' },
32     { id: 'ahsgdyfa03', name: '原神' }
33   ])
34   let obj = ref({
35     a: {
36       b: {
37         c: {
38           d: 666
39         }
40       }
41     }
42   })
43
44   console.log(car)
45
46   function changeCarPrice() {
47     car.value.price += 10
```

```
48  }
49  function changeFirstGame() {
50  games.value[0].name = '流星蝴蝶剑'
51 }
52 function test(){
53   obj.value.a.b.c.d = 999
54 }
55 </script>
56
```

3.6. 【ref 对比 reactive】

宏观角度看：

- 1. `ref` 用来定义：基本类型数据、对象类型数据；
- 2. `reactive` 用来定义：对象类型数据。

- 区别：

1. `ref` 创建的变量必须使用 `.value` (可以使用 `volar` 插件自动添加 `.value`) 。



2. `reactive` 重新分配一个新对象，会失去响应式 (可以使用 `Object.assign` 去整体替换) 。

- 使用原则：

1. 若需要一个基本类型的响应式数据，必须使用 `ref` 。
2. 若需要一个响应式对象，层级不深，`ref` 、`reactive` 都可以。
3. 若需要一个响应式对象，且层级较深，推荐使用 `reactive` 。

3.7. 【toRefs 与 toRef】

- 作用：将一个响应式对象中的每一个属性，转换为 `ref` 对象。
- 备注：`toRefs` 与 `toRef` 功能一致，但 `toRefs` 可以批量转换。
- 语法如下：

```

1  <template>
2    <div class="person">
3      <h2>姓名: {{person.name}}</h2>
4
5      <h2>年龄: {{person.age}}</h2>
6
7      <h2>性别: {{person.gender}}</h2>
8
9      <button @click="changeName">修改名字</button>
10
11     <button @click="changeAge">修改年龄</button>
12
13     <button @click="changeGender">修改性别</button>
14
15   </div>
16
17 </template>
18
19 <script lang="ts" setup name="Person">
20   import {ref, reactive, toRefs, toRef} from 'vue'
21
22   // 数据
23   let person = reactive({name:'张三', age:18, gender:'男'})
24
25   // 通过toRefs将person对象中的n个属性批量取出，且依然保持响应式的能力
26   let {name,gender} = toRefs(person)
27
28   // 通过toRef将person对象中的gender属性取出，且依然保持响应式的能力
29   let age = toRef(person, 'age')
30
31   // 方法
32   function changeName(){
33     name.value += '~'
34   }
35   function changeAge(){
36     age.value += 1
37   }
38   function changeGender(){
39     gender.value = '女'
40   }
41 </script>
42

```

3.8. [computed]

作用：根据已有数据计算出新数据（和 Vue2 中的 `computed` 作用一致）。



```

1  <template>
2    <div class="person">
3      姓: <input type="text" v-model="firstName"> <br>
4      名: <input type="text" v-model="lastName"> <br>
5      全名: <span>{{fullName}}</span> <br>
6      <button @click="changeFullName">全名改为: li-si</button>
7
8    </div>
9
10   </template>
11
12  <script setup lang="ts" name="App">
13    import {ref, computed} from 'vue'
14
15    let firstName = ref('zhang')
16    let lastName = ref('san')
17
18    // 计算属性—只读取, 不修改
19    /* let fullName = computed(()=>{
20      return firstName.value + '-' + lastName.value
21    }) */
22
23
24    // 计算属性—既读取又修改
25    let fullName = computed({
26      // 读取
27      get(){
28        return firstName.value + '-' + lastName.value
29      },
30      // 修改
31      set(val){
32        console.log('有人修改了fullName',val)
33        firstName.value = val.split('-')[0]
34        lastName.value = val.split('-')[1]
35      }
36    })
37
38    function changeFullName(){
39      fullName.value = 'li-si'
40    }
41  </script>
42

```

3.9. 【watch】

- 作用：监视数据的变化（和 Vue2 中的 watch 作用一致）

- 特点：Vue3 中的 watch 只能监视以下四种数据：

- ref 定义的数据。
- reactive 定义的数据。
- 函数返回一个值（getter 函数）。
- 一个包含上述内容的数组。

我们在 Vue3 中使用 watch 的时候，通常会遇到以下几种情况：

* 情况一

监视 ref 定义的【基本类型】数据：直接写数据名即可，监视的是其 value 值的改变。

```
1  <template>
2    <div class="person">
3      <h1>情况一：监视【ref】定义的【基本类型】数据</h1>
4
5      <h2>当前求和为：{{sum}}</h2>
6
7      <button @click="changeSum">点我sum+1</button>
8
9    </div>
10
11  </template>
12
13 <script lang="ts" setup name="Person">
14   import {ref,watch} from 'vue'
15   // 数据
16   let sum = ref(0)
17   // 方法
18   function changeSum(){
19     sum.value += 1
20   }
21   // 监视，情况一：监视【ref】定义的【基本类型】数据
22   const stopWatch = watch(sum,(newValue,oldValue)=>{
23     console.log('sum变化了',newValue,oldValue)
24     if(newValue >= 10){
25       stopWatch()
26     }
27   })
28 </script>
29
```

* 情况二

监视 `ref` 定义的【对象类型】数据：直接写数据名，监视的是对象的【地址值】，若想监视对象内部的数据，要手动开启深度监视。

注意：

- 若修改的是 `ref` 定义的对象中的属性，`newValue` 和 `oldValue` 都是新值，因为它们是同一个对象。
- 若修改整个 `ref` 定义的对象，`newValue` 是新值，`oldValue` 是旧值，因为不是同一个对象了。

```

1  <template>
2    <div class="person">
3      <h1>情况二：监视【ref】定义的【对象类型】数据</h1>
4
5      <h2>姓名: {{ person.name }}</h2>
6
7      <h2>年龄: {{ person.age }}</h2>
8
9      <button @click="changeName">修改名字</button>
10
11     <button @click="changeAge">修改年龄</button>
12
13     <button @click="changePerson">修改整个人</button>
14
15   </div>
16
17 </template>
18
19 <script lang="ts" setup name="Person">
20   import {ref,watch} from 'vue'
21   // 数据
22   let person = ref({
23     name:'张三',
24     age:18
25   })
26   // 方法
27   function changeName(){
28     person.value.name += '~'
29   }
30   function changeAge(){
31     person.value.age += 1
32   }
33   function changePerson(){
34     person.value = {name:'李四',age:90}
35   }
36   /*
37     监视，情况一：监视【ref】定义的【对象类型】数据，监视的是对象的地址值，若想监视对象
38     内部属性的变化，需要手动开启深度监视
39     watch的第一个参数是：被监视的数据
40     watch的第二个参数是：监视的回调
41     watch的第三个参数是：配置对象（deep、immediate等等.....）
42   */
43   watch(person,(newValue,oldValue)=>{
44     console.log('person变化了',newValue,oldValue)
45   },{deep:true})
46 </script>

```

* 情况三

监视 `reactive` 定义的【对象类型】数据，且默认开启了深度监视。

```
1  <template>
2    <div class="person">
3      <h1>情况三：监视【reactive】定义的【对象类型】数据</h1>
4
5    <h2>姓名: {{ person.name }}</h2>
6
7    <h2>年龄: {{ person.age }}</h2>
8
9    <button @click="changeName">修改名字</button>
10
11   <button @click="changeAge">修改年龄</button>
12
13   <button @click="changePerson">修改整个人</button>
14
15   <hr>
16   <h2>测试: {{obj.a.b.c}}</h2>
17
18   <button @click="test">修改obj.a.b.c</button>
19
20 </div>
21
22 </template>
23
24 <script lang="ts" setup name="Person">
25   import {reactive,watch} from 'vue'
26   // 数据
27   let person = reactive({
28     name:'张三',
29     age:18
30   })
31   let obj = reactive({
32     a:{
33       b:{
34         c:666
35       }
36     }
37   })
38   // 方法
39   function changeName(){
40     person.name += '~'
41   }
42   function changeAge(){
43     person.age += 1
44   }
45   function changePerson(){
46     Object.assign(person,{name:'李四',age:80})
47   }
```

```
48     function test(){
49         obj.a.b.c = 888
50     }
51
52     // 监视，情况三：监视【reactive】定义的【对象类型】数据，且默认是开启深度监视的
53     watch(person,(newValue,oldValue)=>{
54         console.log('person变化了',newValue,oldValue)
55     })
56     watch(obj,(newValue,oldValue)=>{
57         console.log('Obj变化了',newValue,oldValue)
58     })
59 
```

```
</script>
```

* 情况四

监视 `ref` 或 `reactive` 定义的【对象类型】数据中的某个属性，注意点如下：

1. 若该属性值不是【对象类型】，需要写成函数形式。
2. 若该属性值是依然是【对象类型】，可直接编，也可写成函数，建议写成函数。

结论：监视的要是对象里的属性，那么最好写函数式，注意点：若是对象监视的是地址值，需要关注对象内部，需要手动开启深度监视。

```
1  <template>
2    <div class="person">
3      <h1>情况四：监视【ref】或【reactive】定义的【对象类型】数据中的某个属性</h1>
4
5    <h2>姓名: {{ person.name }}</h2>
6
7    <h2>年龄: {{ person.age }}</h2>
8
9    <h2>汽车: {{ person.car.c1 }}、{{ person.car.c2 }}</h2>
10
11   <button @click="changeName">修改名字</button>
12
13   <button @click="changeAge">修改年龄</button>
14
15   <button @click="changeC1">修改第一台车</button>
16
17   <button @click="changeC2">修改第二台车</button>
18
19   <button @click="changeCar">修改整个车</button>
20
21   </div>
22
23 </template>
24
25 <script lang="ts" setup name="Person">
26   import {reactive,watch} from 'vue'
27
28   // 数据
29   let person = reactive({
30     name:'张三',
31     age:18,
32   car:{
33     c1:'奔驰',
34     c2:'宝马'
35   }
36 })
37   // 方法
38   function changeName(){
39     person.name += '~'
40   }
41   function changeAge(){
42     person.age += 1
43   }
44   function changeC1(){
45     person.car.c1 = '奥迪'
46   }
47   function changeC2(){
```

```
48     person.car.c2 = '大众'
49 }
50 function changeCar(){
51     person.car = {c1:'雅迪',c2:'爱玛'}
52 }
53
54 // 监视，情况四：监视响应式对象中的某个属性，且该属性是基本类型的，要写成函数式
55 /* watch(()=> person.name,(newValue,oldValue)=>{
56     console.log('person.name变化了',newValue,oldValue)
57 }) */
58
59 // 监视，情况四：监视响应式对象中的某个属性，且该属性是对象类型的，可以直接写，也能写
60 // 函数，更推荐写函数
61 watch(()=>person.car,(newValue,oldValue)=>{
62     console.log('person.car变化了',newValue,oldValue)
63 },{deep:true})
64 </script>
```

* 情况五

监视上述的多个数据

```
1  <template>
2    <div class="person">
3      <h1>情况五：监视上述的多个数据</h1>
4
5      <h2>姓名: {{ person.name }}</h2>
6
7      <h2>年龄: {{ person.age }}</h2>
8
9      <h2>汽车: {{ person.car.c1 }}、{{ person.car.c2 }}</h2>
10
11     <button @click="changeName">修改名字</button>
12
13     <button @click="changeAge">修改年龄</button>
14
15     <button @click="changeC1">修改第一台车</button>
16
17     <button @click="changeC2">修改第二台车</button>
18
19     <button @click="changeCar">修改整个车</button>
20
21   </div>
22
23 </template>
24
25 <script lang="ts" setup name="Person">
26   import {reactive,watch} from 'vue'
27
28   // 数据
29   let person = reactive({
30     name:'张三',
31     age:18,
32     car:{
33       c1:'奔驰',
34       c2:'宝马'
35     }
36   })
37   // 方法
38   function changeName(){
39     person.name += '~'
40   }
41   function changeAge(){
42     person.age += 1
43   }
44   function changeC1(){
45     person.car.c1 = '奥迪'
46   }
47   function changeC2(){
```

```
48     person.car.c2 = '大众'
49 }
50 function changeCar(){
51   person.car = {c1:'雅迪',c2:'爱玛'}
52 }
53
54 // 监视，情况五：监视上述的多个数据
55 watch([()=>person.name,person.car],(newValue,oldValue)=>{
56   console.log('person.car变化了',newValue,oldValue)
57 },{deep:true})
58
59 </script>
60
```

3.10. 【watchEffect】

- 官网：立即运行一个函数，同时响应式地追踪其依赖，并在依赖更改时重新执行该函数。
- `watch` 对比 `watchEffect`
 1. 都能监听响应式数据的变化，不同的是监听数据变化的方式不同
 2. `watch`：要明确指出监视的数据
 3. `watchEffect`：不用明确指出监视的数据（函数中用到哪些属性，那就监视哪些属性）。
- 示例代码：

```

1  <template>
2    <div class="person">
3      <h1>需求：水温达到50℃，或水位达到20cm，则联系服务器</h1>
4
5      <h2 id="demo">水温：{{temp}}</h2>
6
7      <h2>水位：{{height}}</h2>
8
9      <button @click="changePrice">水温+1</button>
10
11     <button @click="changeSum">水位+10</button>
12
13   </div>
14
15 </template>
16
17
18 <script lang="ts" setup name="Person">
19   import {ref,watch,watchEffect} from 'vue'
20   // 数据
21   let temp = ref(0)
22   let height = ref(0)
23
24   // 方法
25   function changePrice(){
26     temp.value += 10
27   }
28   function changeSum(){
29     height.value += 1
30   }
31
32   // 用watch实现，需要明确的指出要监视：temp、height
33   watch([temp,height],(value)=>{
34     // 从value中获取最新的temp值、height值
35     const [newTemp,newHeight] = value
36     // 室温达到50℃，或水位达到20cm，立刻联系服务器
37     if(newTemp >= 50 || newHeight >= 20){
38       console.log('联系服务器')
39     }
40   })
41
42   // 用watchEffect实现，不用
43   const stopWatch = watchEffect(()=>{
44     // 室温达到50℃，或水位达到20cm，立刻联系服务器
45     if(temp.value >= 50 || height.value >= 20){
46       console.log(document.getElementById('demo')?.innerText)
47       console.log('联系服务器')

```

```
48      }
49      // 水温达到100，或水位达到50，取消监视
50      if(temp.value === 100 || height.value === 50){
51          console.log('清理了')
52          stopWtach()
53      }
54  })
55 </script>
56
```

3.11. 【标签的 ref 属性】

作用：用于注册模板引用。

- 用在普通 DOM 标签上，获取的是 DOM 节点。
- 用在组件标签上，获取的是组件实例对象。

用在普通 DOM 标签上：

```
1  <template>
2    <div class="person">
3      <h1 ref="title1">尚硅谷</h1>
4
5      <h2 ref="title2">前端</h2>
6
7      <h3 ref="title3">Vue</h3>
8
9    <input type="text" ref="inpt"> <br><br>
10   <button @click="showLog">点我打印内容</button>
11
12  </div>
13
14 </template>
15
16 <script lang="ts" setup name="Person">
17   import {ref} from 'vue'
18
19   let title1 = ref()
20   let title2 = ref()
21   let title3 = ref()
22
23   function showLog(){
24     // 通过id获取元素
25     const t1 = document.getElementById('title1')
26     // 打印内容
27     console.log((t1 as HTMLElement).innerText)
28     console.log((<HTMLElement>t1).innerText)
29     console.log(t1?.innerText)
30
31     /*****/
32
33     // 通过ref获取元素
34     console.log(title1.value)
35     console.log(title2.value)
36     console.log(title3.value)
37   }
38 </script>
39
```

用在组件标签上：

```
1  <!-- 父组件App.vue -->
2  <template>
3      <Person ref="ren"/>
4      <button @click="test">测试</button>
5
6  </template>
7
8  <script lang="ts" setup name="App">
9      import Person from './components/Person.vue'
10     import {ref} from 'vue'
11
12     let ren = ref()
13
14     function test(){
15         console.log(ren.value.name)
16         console.log(ren.value.age)
17     }
18 </script>
19
20
21  <!-- 子组件Person.vue中要使用defineExpose暴露内容 -->
22  <script lang="ts" setup name="Person">
23      import {ref,defineExpose} from 'vue'
24      // 数据
25      let name = ref('张三')
26      let age = ref(18)
27      /*****
28      *****/
29      // 使用defineExpose将组件中的数据交给外部
30      defineExpose({name,age})
31 </script>
32
```

3.12. 【props】

App.vue 中代码：

```
1 // 定义一个接口，限制每个Person对象的格式
2 export interface PersonInter {
3     id:string,
4     name:string,
5     age:number
6 }
7
8 // 定义一个自定义类型Persons
9 export type Persons = Array<PersonInter>
```

```
1 <template>
2     <Person :list="persons"/>
3 </template>
```

```
1
2 `Person.vue`中代码:
3
4 ``Vue
5 <template>
6 <div class="person">
7     <ul>
8         <li v-for="item in list" :key="item.id">
9             {{item.name}}--{{item.age}}
10        </li>
```

```
1 </ul>
```

```
1
```

3.13. 【生命周期】

- 概念：Vue 组件实例在创建时要经历一系列的初始化步骤，在此过程中 Vue 会在合适的时机，调用特定的函数，从而让开发者有机会在特定阶段运行自己的代码，这些特定的函数统称为：生命周期钩子
- 规律：

生命周期整体分为四个阶段，分别是：创建、挂载、更新、销毁，每个阶段都有两个钩子，一前一后。

- Vue2 的生命周期

创建阶段：beforeCreate、created

挂载阶段: `beforeMount`、`mounted`

更新阶段: `beforeUpdate`、`updated`

销毁阶段: `beforeDestroy`、`destroyed`

- `Vue3` 的生命周期

创建阶段: `setup`

挂载阶段: `onBeforeMount`、`onMounted`

更新阶段: `onBeforeUpdate`、`onUpdated`

卸载阶段: `onBeforeUnmount`、`onUnmounted`

- 常用的钩子: `onMounted` (挂载完毕)、`onUpdated` (更新完毕)、`onBeforeUnmount` (卸载之前)
- 示例代码:

```
1  <template>
2    <div class="person">
3      <h2>当前求和为: {{ sum }}</h2>
4
5      <button @click="changeSum">点我sum+1</button>
6
7    </div>
8
9  </template>
10
11
12  <!-- vue3写法 -->
13  <script lang="ts" setup name="Person">
14    import {
15      ref,
16      onBeforeMount,
17      onMounted,
18      onBeforeUpdate,
19      onUpdated,
20      onBeforeUnmount,
21      onUnmounted
22    } from 'vue'
23
24    // 数据
25    let sum = ref(0)
26    // 方法
27    function changeSum() {
28      sum.value += 1
29    }
30    console.log('setup')
31    // 生命周期钩子
32    onBeforeMount(()=>{
33      console.log('挂载之前')
34    })
35    onMounted(()=>{
36      console.log('挂载完毕')
37    })
38    onBeforeUpdate(()=>{
39      console.log('更新之前')
40    })
41    onUpdated(()=>{
42      console.log('更新完毕')
43    })
44    onBeforeUnmount(()=>{
45      console.log('卸载之前')
46    })
47    onUnmounted(()=>{
```

```
48     console.log('卸载完毕')
49   })
50 </script>
51
```

3.14. 【自定义hook】

- 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。

示例代码：

- `useSum.ts` 中内容如下：

```
1 import {ref, onMounted} from 'vue'
2
3 export default function(){
4   let sum = ref(0)
5
6   const increment = ()=>{
7     sum.value += 1
8   }
9   const decrement = ()=>{
10    sum.value -= 1
11  }
12  onMounted(()=>{
13    increment()
14  })
15
16 //向外部暴露数据
17 return {sum, increment, decrement}
18 }
```

- `useDog.ts` 中内容如下：

```
1 import {reactive, onMounted} from 'vue'
2 import axios,{AxiosError} from 'axios'
3
4 export default function(){
5     let dogList = reactive<string[]>([])
6
7     // 方法
8     async function getDog(){
9         try {
10             // 发请求
11             let {data} = await axios.get('https://dog.ceo/api/breed/pembroke/images/random')
12             // 维护数据
13             dogList.push(data.message)
14         } catch (error) {
15             // 处理错误
16             const err = <AxiosError>error
17             console.log(err.message)
18         }
19     }
20
21     // 挂载钩子
22     onMounted(()=>{
23         getDog()
24     })
25
26     // 向外部暴露数据
27     return {dogList, getDog}
28 }
```

- 组件中具体使用：

```
1  <template>
2    <h2>当前求和为: {{sum}}</h2>
3
4    <button @click="increment">点我+1</button>
5
6    <button @click="decrement">点我-1</button>
7
8    <hr>
9    
10   <span v-show="dogList.isLoading">加载中.....</span><br>
11   <button @click="getDog">再来一只狗</button>
12
13 </template>
14
15
16 <script lang="ts">
17   import {defineComponent} from 'vue'
18
19   export default defineComponent({
20     name: 'App',
21   })
22 </script>
23
24
25 <script setup lang="ts">
26   import useSum from './hooks/useSum'
27   import useDog from './hooks/useDog'
28
29   let {sum, increment, decrement} = useSum()
30   let {dogList, getDog} = useDog()
31 </script>
32
```

4. 路由

4.1. 【对路由的理解】

 图片加载失败

4.2. 【基本切换效果】

- Vue3 中要使用 vue-router 的最新版本，目前是 4 版本。
- 路由配置文件代码如下：

```
1 import {createRouter, createWebHistory} from 'vue-router'
2 import Home from '@/pages/Home.vue'
3 import News from '@/pages/News.vue'
4 import About from '@/pages/About.vue'
5
6 const router = createRouter({
7   history:createWebHistory(),
8   routes:[
9     {
10       path:'/home',
11       component:Home
12     },
13     {
14       path:'/about',
15       component:About
16     }
17   ]
18 })
19 export default router
```

- main.ts 代码如下：

```
1 import router from './router/index'
2 app.use(router)
3
4 app.mount('#app')
```

- App.vue 代码如下

```

1  <template>
2    <div class="app">
3      <h2 class="title">Vue路由测试</h2>
4
5      <!-- 导航区 -->
6      <div class="navigate">
7        <RouterLink to="/home" active-class="active">首页</RouterLink>
8
9        <RouterLink to="/news" active-class="active">新闻</RouterLink>
10
11       <RouterLink to="/about" active-class="active">关于</RouterLink>
12
13     </div>
14
15     <!-- 展示区 -->
16     <div class="main-content">
17       <RouterView></RouterView>
18
19     </div>
20
21   </div>
22
23 </template>
24
25
26 <script lang="ts" setup name="App">
27   import {RouterLink, RouterView} from 'vue-router'
28 </script>
29

```

4.3. 【两个注意点】

1. 路由组件通常存放在 `pages` 或 `views` 文件夹，一般组件通常存放在 `components` 文件夹。
2. 通过点击导航，视觉效果上“消失”了的路由组件，默认是被卸载掉的，需要的时候再去挂载。

4.4. 【路由器工作模式】

1. `history` 模式

优点： `URL` 更加美观，不带有 `#`，更接近传统的网站 `URL`。

缺点：后期项目上线，需要服务端配合处理路径问题，否则刷新会有 `404` 错误。

```
1 ▼ const router = createRouter({  
2     history:createWebHistory(), //history模式  
3     /***/  
4 })
```

2. hash 模式

优点：兼容性更好，因为不需要服务器端处理路径。

缺点： URL 带有 # 不太美观，且在 SEO 优化方面相对较差。

```
1 ▼ const router = createRouter({  
2     history:createWebHashHistory(), //hash模式  
3     /***/  
4 })
```

4.5. 【to的两种写法】

```
1 <!-- 第一种：to的字符串写法 -->  
2 <router-link active-class="active" to="/home">主页</router-link>  
3  
4 <!-- 第二种：to的对象写法 -->  
5 ▼ <router-link active-class="active" :to="{path: '/home'}">Home</router-link>  
6
```

4.6. 【命名路由】

作用：可以简化路由跳转及传参（后面就讲）。

给路由规则命名：

```
1 routes: [
2   {
3     name:'zhuye',
4     path:'/home',
5     component:Home
6   },
7   {
8     name:'xinwen',
9     path:'/news',
10    component:News,
11  },
12  {
13    name:'guanyu',
14    path:'/about',
15    component:About
16  }
17]
```

跳转路由：

```
1 <!--简化前：需要写完整的路径（to的字符串写法）-->
2 <router-link to="/news/detail">跳转</router-link>
3
4 <!--简化后：直接通过名字跳转（to的对象写法配合name属性）-->
5 <router-link :to="{name: 'guanyu'}">跳转</router-link>
6
```

4.7. 【嵌套路由】

1. 编写 `News` 的子路由：`Detail.vue`
2. 配置路由规则，使用 `children` 配置项：

```
1 const router = createRouter({
2   history:createWebHistory(),
3   routes:[
4     {
5       name:'zhuye',
6       path:'/home',
7       component:Home
8     },
9     {
10       name:'xinwen',
11       path:'/news',
12       component:News,
13       children:[
14         {
15           name:'xiang',
16           path:'detail',
17           component:Detail
18         }
19       ]
20     },
21     {
22       name:'guanyu',
23       path:'/about',
24       component:About
25     }
26   ]
27 })
28 export default router
```

3. 跳转路由（记得要加完整路径）：

```
1 <router-link to="/news/detail">xxxx</router-link>
2
3 <!-- 或 -->
4 <router-link :to="{path: '/news/detail'}">xxxx</router-link>
5
```

4. 记得去 Home 组件中预留一个 <router-view>

```
1 <template>
2   <div class="news">
3     <nav class="news-list">
4       <RouterLink v-for="news in newsList" :key="news.id" :to="{path: '/news/detail'}">
5         {{news.name}}
6       </RouterLink>
7
8     </nav>
9
10    <div class="news-detail">
11      <RouterView/>
12    </div>
13
14  </div>
15
16 </template>
17
```

4.8. 【路由传参】

query参数

1. 传递参数

```
1  <!-- 跳转并携带query参数 (to的字符串写法) -->
2  <router-link to="/news/detail?a=1&b=2&content=欢迎你">
3      跳转
4  </router-link>
5
6
7  <!-- 跳转并携带query参数 (to的对象写法) -->
8  <RouterLink
9  :to="{
10      //name:'xiang', //用name也可以跳转
11      path:'/news/detail',
12      query:{
13          id:news.id,
14          title:news.title,
15          content:news.content
16      }
17  }"
18  >
19  {{news.title}}
20 </RouterLink>
21
```

2. 接收参数:

```
1 import {useRoute} from 'vue-router'
2 const route = useRoute()
3 // 打印query参数
4 console.log(route.query)
```

params参数

1. 传递参数

```
1  <!-- 跳转并携带params参数 (to的字符串写法) -->
2  <RouterLink :to="/news/detail/001/新闻001/内容001">{{news.title}}</Router
Link>
3
4
5  <!-- 跳转并携带params参数 (to的对象写法) -->
6  <RouterLink
7    :to={
8      name:'xiang', //用name跳转
9      params:{
10        id:news.id,
11        title:news.title,
12        content:news.title
13      }
14    }"
15  >
16  {{news.title}}
17 </RouterLink>
18
```

2. 接收参数:

```
1 import {useRoute} from 'vue-router'
2 const route = useRoute()
3 // 打印params参数
4 console.log(route.params)
```

备注1: 传递 `params` 参数时, 若使用 `to` 的对象写法, 必须使用 `name` 配置项, 不能用 `path`。

备注2: 传递 `params` 参数时, 需要在规则中占位。

4.9. 【路由的props配置】

作用: 让路由组件更方便的收到参数 (可以将路由参数作为 `props` 传给组件)

```
1  {
2      name:'xiang',
3      path:'detail/:id/:title/:content',
4      component:Detail,
5
6      // props的对象写法，作用：把对象中的每一组key-value作为props传给Detail组件
7      // props:{a:1,b:2,c:3},
8
9      // props的布尔值写法，作用：把收到了每一组params参数，作为props传给Detail组件
10     // props:true
11
12     // props的函数写法，作用：把返回的对象中每一组key-value作为props传给Detail组件
13     props(route){
14         return route.query
15     }
16 }
```

4.10. 【replace属性】

1. 作用：控制路由跳转时操作浏览器历史记录的模式。
2. 浏览器的历史记录有两种写入方式：分别为 `push` 和 `replace`：
 - `push` 是追加历史记录（默认值）。
 - `replace` 是替换当前记录。
3. 开启 `replace` 模式：

```
1 <RouterLink replace .....>News</RouterLink>
2
```

4.11. 【编程式导航】

路由组件的两个重要的属性：`$route` 和 `$router` 变成了两个 `hooks`

```
1 import {useRoute,useRouter} from 'vue-router'  
2  
3 const route = useRoute()  
4 const router = useRouter()  
5  
6 console.log(route.query)  
7 console.log(route.parmas)  
8 console.log(router.push)  
9 console.log(router.replace)
```

4.12. 【重定向】

1. 作用：将特定的路径，重新定向到已有路由。
2. 具体编码：

```
1 {  
2   path: '/',
3   redirect: '/about'  
4 }
```

5. pinia

5.1 【准备一个效果】



5.2 【搭建 pinia 环境】

第一步： `npm install pinia`

第二步：操作 `src/main.ts`

```

1 import { createApp } from 'vue'
2 import App from './App.vue'
3
4 /* 引入createPinia, 用于创建pinia */
5 import { createPinia } from 'pinia'
6
7 /* 创建pinia */
8 const pinia = createPinia()
9 const app = createApp(App)
10
11 /* 使用插件 */
12 app.use(pinia)
13 app.mount('#app')

```

此时开发者工具中已经有了 `pinia` 选项



5.3 【存储+读取数据】

- `Store` 是一个保存：状态、业务逻辑 的实体，每个组件都可以读取、写入它。
- 它有三个概念：`state`、`getter`、`action`，相当于组件中的：`data`、`computed` 和 `methods`。
- 具体编码：`src/store/count.ts`

```
1 // 引入defineStore用于创建store
2 import {defineStore} from 'pinia'
3
4 // 定义并暴露一个store
5 export const useCountStore = defineStore('count',{
6   // 动作
7   actions:{},
8   // 状态
9   state(){
10  return {
11    sum:6
12  }
13 },
14 // 计算
15 getters:{}
16 })
```

4. 具体编码: `src/store/talk.ts`

```
1 // 引入defineStore用于创建store
2 import {defineStore} from 'pinia'
3
4 // 定义并暴露一个store
5 export const useTalkStore = defineStore('talk',{
6   // 动作
7   actions:{},
8   // 状态
9   state(){
10  return {
11    talkList:[
12      {id:'yuysada01',content:'你今天有点怪，哪里怪？怪好看的！'},
13      {id:'yuysada02',content:'草莓、蓝莓、蔓越莓，你想我了没？'},
14      {id:'yuysada03',content:'心里给你留了一块地，我的死心塌地'}
15    ]
16  }
17 },
18 // 计算
19 getters:{}
20 })
```

5. 组件中使用 `state` 中的数据

```

1  <template>
2    <h2>当前求和为: {{ sumStore.sum }}</h2>
3
4  </template>
5
6
7  <script setup lang="ts" name="Count">
8    // 引入对应的useXXXXStore
9  import {useSumStore} from '@/store/sum'
10
11   // 调用useXXXXStore得到对应的store
12   const sumStore = useSumStore()
13 </script>
14

```

```

1  <template>
2    <ul>
3      <li v-for="talk in talkStore.talkList" :key="talk.id">
4        {{ talk.content }}
5      </li>
6
7    </ul>
8
9  </template>
10
11
12 <script setup lang="ts" name="Count">
13   import axios from 'axios'
14  import {useTalkStore} from '@/store/talk'
15
16  const talkStore = useTalkStore()
17 </script>
18

```

5.4. 【修改数据】(三种方式)

1. 第一种修改方式，直接修改

```
1  countStore.sum = 666
```

2. 第二种修改方式：批量修改

```
1 ▶ countStore.$patch({
2     sum:999,
3     school:'atguigu'
4 })
```

3. 第三种修改方式：借助 `action` 修改（`action` 中可以编写一些业务逻辑）

```
1 import { defineStore } from 'pinia'
2
3 ▶ export const useCountStore = defineStore('count', {
4     /*****
5     actions: {
6         //加
7         increment(value:number) {
8             if (this.sum < 10) {
9                 //操作countStore中的sum
10                this.sum += value
11            }
12        },
13        //减
14        decrement(value:number){
15            if(this.sum > 1){
16                this.sum -= value
17            }
18        }
19    },
20    *****/
21 })
```

4. 组件中调用 `action` 即可

```
1 // 使用countStore
2 const countStore = useCountStore()
3
4 // 调用对应action
5 countStore.incrementOdd(n.value)
```

5.5. 【storeToRefs】

- 借助 `storeToRefs` 将 `store` 中的数据转为 `ref` 对象，方便在模板中使用。
- 注意：`pinia` 提供的 `storeToRefs` 只会将数据做转换，而 `Vue` 的 `toRefs` 会转

换 `store` 中数据。

```
1  <template>
2    <div class="count">
3      <h2>当前求和为: {{sum}}</h2>
4
5    </div>
6
7  </template>
8
9  <script setup lang="ts" name="Count">
10 import { useCountStore } from '@/store/count'
11 /* 引入storeToRefs */
12 import { storeToRefs } from 'pinia'
13
14 /* 得到countStore */
15 const countStore = useCountStore()
16 /* 使用storeToRefs转换countStore, 随后解构 */
17 const {sum} = storeToRefs(countStore)
18 </script>
19
```

5.6. 【getters】

1. 概念：当 `state` 中的数据，需要经过处理后再使用时，可以使用 `getters` 配置。
2. 追加 ```getters``` 配置。

```
1 // 引入defineStore用于创建store
2 import {defineStore} from 'pinia'
3
4 // 定义并暴露一个store
5 export const useCountStore = defineStore('count',{
6   // 动作
7   actions:{
8     /*****/
9   },
10  // 状态
11  state(){
12    return {
13      sum:1,
14      school:'atguigu'
15    }
16  },
17  // 计算
18  getters:{
19    bigSum:(state):number => state.sum *10,
20    upperSchool():string{
21      return this.school.toUpperCase()
22    }
23  }
24 })
```

3. 组件中读取数据：

```
1 const {increment,decrement} = countStore
2 let {sum,school,bigSum,upperSchool} = storeToRefs(countStore)
```

5.7. 【\$subscribe】

通过 store 的 `$subscribe()` 方法侦听 `state` 及其变化

```
1 talkStore.$subscribe((mutate,state)=>{
2   console.log('LoveTalk',mutate,state)
3   localStorage.setItem('talk',JSON.stringify(talkList.value))
4 })
```

5.8. 【store组合式写法】

```
1 import {defineStore} from 'pinia'
2 import axios from 'axios'
3 import {nanoid} from 'nanoid'
4 import {reactive} from 'vue'
5
6 export const useTalkStore = defineStore('talk', ()=>{
7     // talkList就是state
8     const talkList = reactive(
9         JSON.parse(localStorage.getItem('talkList') as string) || []
10    )
11
12    // getATalk函数相当于action
13    async function getATalk(){
14        // 发请求，下面这行的写法是：连续解构赋值+重命名
15        let {data:{content:title}} = await axios.get('https://api.uomg.com/api/rand.qinghua?format=json')
16        // 把请求回来的字符串，包装成一个对象
17        let obj = {id:nanoid(),title}
18        // 放到数组中
19        talkList.unshift(obj)
20    }
21    return {talkList,getATalk}
22 })
```

6. 组件通信

Vue3 组件通信和 Vue2 的区别：

- 移出事件总线，使用 `mitt` 代替。
- `vuex` 换成了 `pinia`。
- 把 `.sync` 优化到了 `v-model` 里面了。
- 把 `$listeners` 所有的东西，合并到 `$attrs` 中了。
- `$children` 被砍掉了。

常见搭配形式：



6.1. 【props】

概述： `props` 是使用频率最高的一种通信方式，常用与：父 ↔ 子。

- 若 父传子：属性值是`非函数`。
- 若 子传父：属性值是`函数`。

父组件：

```
1  <template>
2    <div class="father">
3      <h3>父组件, </h3>
4
5      <h4>我的车: {{ car }}</h4>
6
7      <h4>儿子给的玩具: {{ toy }}</h4>
8
9      <Child :car="car" :getToy="getToy"/>
10   </div>
11
12 </template>
13
14 <script setup lang="ts" name="Father">
15   import Child from './Child.vue'
16   import { ref } from "vue";
17   // 数据
18   const car = ref('奔驰')
19   const toy = ref()
20   // 方法
21   function getToy(value:string){
22     toy.value = value
23   }
24 </script>
25
```

子组件

```

1  <template>
2    <div class="child">
3      <h3>子组件</h3>
4
5      <h4>我的玩具: {{ toy }}</h4>
6
7      <h4>父给我的车: {{ car }}</h4>
8
9      <button @click="getToy(toy)">玩具给父亲</button>
10
11    </div>
12
13  </template>
14
15  <script setup lang="ts" name="Child">
16    import { ref } from "vue";
17    const toy = ref('奥特曼')
18
19    defineProps(['car', 'getToy'])
20  </script>
21

```

6.2. 【自定义事件】

1. 概述：自定义事件常用于：子 => 父。
2. 注意区分好：原生事件、自定义事件。
 - 原生事件：
 - 事件名是特定的（`click`、`mouseenter` 等等）
 - 事件对象 `$event`：是包含事件相关信息的对象（`pageX`、`pageY`、`target`、`keyCode`）
 - 自定义事件：
 - 事件名是任意名称
 - 事件对象 `$event`：是调用 `emit` 时所提供的数据，可以是任意类型！！！
3. 示例：

```
1 <!--在父组件中，给子组件绑定自定义事件：-->
2 <Child @send-toy="toy = $event"/>
3
4 <!--注意区分原生事件与自定义事件中的$event-->
5 <button @click="toy = $event">测试</button>
6
```

```
1 //子组件中，触发事件：
2 this.$emit('send-toy', 具体数据)
```

6.3. 【mitt】

概述：与消息订阅与发布（pubsub）功能类似，可以实现任意组件间通信。

安装 mitt

```
1 npm i mitt
```

新建文件：src\utils\emitter.ts

```

1 // 引入mitt
2 import mitt from "mitt";
3
4 // 创建emitter
5 const emitter = mitt()
6
7 /*
8     // 绑定事件
9     emitter.on('abc',(value)=>{
10         console.log('abc事件被触发',value)
11     })
12     emitter.on('xyz',(value)=>{
13         console.log('xyz事件被触发',value)
14     })
15
16     setInterval(() => {
17         // 触发事件
18         emitter.emit('abc',666)
19         emitter.emit('xyz',777)
20     }, 1000);
21
22     setTimeout(() => {
23         // 清理事件
24         emitter.all.clear()
25     }, 3000);
26 */
27
28 // 创建并暴露mitt
29 export default emitter

```

接收数据的组件中：绑定事件、同时在销毁前解绑事件：

```

1 import emitter from "@/utils/emitter";
2 import { onUnmounted } from "vue";
3
4 // 绑定事件
5 /*
6     emitter.on('send-toy',(value)=>{
7         console.log('send-toy事件被触发',value)
8     })
9 */
10    onUnmounted(()=>{
11        // 解绑事件
12        emitter.off('send-toy')
13    })

```

【第三步】：提供数据的组件，在合适的时候触发事件

```
1 import emitter from '@/utils/emitter';
2
3 ▼ function sendToy(){
4     // 触发事件
5     emitter.emit('send-toy', toy.value)
6 }
```

注意这个重要的内置关系，总线依赖着这个内置关系

6.4. 【v-model】

1. 概述：实现 父 ↔ 子 之间相互通信。
2. 前序知识 —— v-model 的本质

```
1 <!-- 使用v-model指令 -->
2 <input type="text" v-model="userName">
3
4 <!-- v-model的本质是下面这行代码 -->
5 <input
6   type="text"
7   :value="userName"
8   @input="userName = (<HTMLInputElement>$event.target).value"
9 >
```

3. 组件标签上的 v-model 的本质：:modelValue + update:modelValue 事件。

```
1 <!-- 组件标签上使用v-model指令 -->
2 ▼ <AtguiguInput v-model="userName"/>
3
4 <!-- 组件标签上v-model的本质 -->
5 <AtguiguInput :modelValue="userName" @update:model-value="userName = $even
t"/>
```

AtguiguInput 组件中：

```

1  <template>
2    <div class="box">
3      <!--将接收的value值赋给input元素的value属性，目的是：为了呈现数据 -->
4      <!--给input元素绑定原生input事件，触发input事件时，进而触发update:model-
value事件-->
5      <input
6        type="text"
7        :value="modelValue"
8        @input="emit('update:model-value', $event.target.value)"
9      >
10     </div>
11
12   </template>
13
14
15  <script setup lang="ts" name="AtguiguInput">
16    // 接收props
17    defineProps(['modelValue'])
18    // 声明事件
19    const emit = defineEmits(['update:model-value'])
20  </script>
21

```

4. 也可以更换 `value`，例如改成 `abc`

```

1  <!-- 也可以更换value，例如改成abc-->
2  <AtguiguInput v-model:abc="userName"/>
3
4  <!-- 上面代码的本质如下 -->
5  <AtguiguInput :abc="userName" @update:abc="userName = $event"/>

```

`AtguiguInput` 组件中：

```

1  <template>
2    <div class="box">
3      <input
4        type="text"
5        :value="abc"
6        @input="emit('update:abc', $event.target.value)"
7      >
8    </div>
9
10   </template>
11
12
13  <script setup lang="ts" name="AtguiguInput">
14    // 接收props
15    defineProps(['abc'])
16    // 声明事件
17    const emit = defineEmits(['update:abc'])
18  </script>
19

```

5. 如果 `value` 可以更换，那么就可以在组件标签上多次使用 `v-model`

```
1  <AtguiguInput v-model:abc="userName" v-model:xyz="password"/>
```

6.5. 【\$attrs】

1. 概述： `$attrs` 用于实现当前组件的父组件，向当前组件的子组件通信（祖→孙）。

2. 具体说明： `$attrs` 是一个对象，包含所有父组件传入的标签属性。

注意： `$attrs` 会自动排除 `props` 中声明的属性(可以认为声明过的 `props` 被子组件自己“消费”了)

父组件：

```

1  <template>
2    <div class="father">
3      <h3>父组件</h3>
4
5      <Child :a="a" :b="b" :c="c" :d="d" v-bind="{x:100,y:200}" :updateA
6        ="updateA"/>
7    </div>
8
9  </template>
10 <script setup lang="ts" name="Father">
11   import Child from './Child.vue'
12   import { ref } from "vue";
13   let a = ref(1)
14   let b = ref(2)
15   let c = ref(3)
16   let d = ref(4)
17
18   function updateA(value){
19     a.value = value
20   }
21 </script>
22

```

子组件:

```

1  <template>
2    <div class="child">
3      <h3>子组件</h3>
4
5      <GrandChild v-bind="$attrs"/>
6    </div>
7
8  </template>
9
10 <script setup lang="ts" name="Child">
11   import GrandChild from './GrandChild.vue'
12 </script>
13

```

孙组件:

```

1  <template>
2    <div class="grand-child">
3      <h3>孙组件</h3>
4
5    <h4>a: {{ a }}</h4>
6
7    <h4>b: {{ b }}</h4>
8
9    <h4>c: {{ c }}</h4>
10
11   <h4>d: {{ d }}</h4>
12
13   <h4>x: {{ x }}</h4>
14
15   <h4>y: {{ y }}</h4>
16
17   <button @click="updateA(666)">点我更新A</button>
18
19   </div>
20
21 </template>
22
23 <script setup lang="ts" name="GrandChild">
24   defineProps(['a','b','c','d','x','y','updateA'])
25 </script>
26

```

6.6. 【`refs`、`parent`】

1. 概述:

- `$refs` 用于 : 父→子。
- `$parent` 用于: 子→父。

2. 原理如下:

属性	说明
<code>\$refs</code>	值为对象，包含所有被 <code>ref</code> 属性标识的 <code>DOM</code> 元素或组件实例。
<code>\$parent</code>	值为对象，当前组件的父组件实例对象。

6.7. 【provide、inject】

1. 概述：实现祖孙组件直接通信

2. 具体使用：

- 在祖先组件中通过 `provide` 配置向后代组件提供数据
- 在后代组件中通过 `inject` 配置来声明接收数据

3. 具体编码：【第一步】父组件中，使用 `provide` 提供数据

```
1  <template>
2    <div class="father">
3      <h3>父组件</h3>
4
5      <h4>资产: {{ money }}</h4>
6
7      <h4>汽车: {{ car }}</h4>
8
9      <button @click="money += 1">资产+1</button>
10
11     <button @click="car.price += 1">汽车价格+1</button>
12
13     <Child/>
14   </div>
15
16 </template>
17
18
19 <script setup lang="ts" name="Father">
20   import Child from './Child.vue'
21   import { ref, reactive, provide } from "vue";
22   // 数据
23   let money = ref(100)
24   let car = reactive({
25     brand:'奔驰',
26     price:100
27   })
28   // 用于更新money的方法
29   function updateMoney(value:number){
30     money.value += value
31   }
32   // 提供数据
33   provide('moneyContext',{money,updateMoney})
34   provide('car',car)
35 </script>
36
```

注意：子组件中不用编写任何东西，是不受到任何打扰的

【第二步】孙组件中使用 `inject` 配置项接受数据。

```
1  <template>
2    <div class="grand-child">
3      <h3>我是孙组件</h3>
4
5      <h4>资产: {{ money }}</h4>
6
7      <h4>汽车: {{ car }}</h4>
8
9      <button @click="updateMoney(6)">点我</button>
10
11    </div>
12
13  </template>
14
15
16  <script setup lang="ts" name="GrandChild">
17    import { inject } from 'vue';
18    // 注入数据
19    let {money, updateMoney} = inject('moneyContext', {money:0, updateMoney:(x:number)=>{}})
20    let car = inject('car')
```

```
1
2
3 ## 6.8. 【pinia】
4
5 参考之前`pinia`部分的讲解
6
7 ## 6.9. 【slot】
8
9 ### 1. 默认插槽
10
11 ! [img] (http://49.232.112.44/images/default_slot.png)
12
13 `` `vue
14 父组件中:
15     <Category title="今日热门游戏">
16         <ul>
17             <li v-for="g in games" :key="g.id">{{ g.name }}</li>
18
19         </ul>
20
21     </Category>
22
23 子组件中:
24     <template>
25         <div class="item">
26             <h3>{{ title }}</h3>
27
28             <!-- 默认插槽 -->
29             <slot></slot>
30
31         </div>
32
33     </template>
34
```

2. 具名插槽

```

1 父组件中:
2 <Category title="今日热门游戏">
3   <template v-slot:s1>
4     <ul>
5       <li v-for="g in games" :key="g.id">{{ g.name }}</li>
6
7   </ul>
8
9   </template>
10
11 <template #s2>
12   <a href="">更多</a>
13
14 </template>
15
16 </Category>
17
18 子组件中:
19 <template>
20   <div class="item">
21     <h3>{{ title }}</h3>
22
23     <slot name="s1"></slot>
24
25     <slot name="s2"></slot>
26
27   </div>
28
29 </template>
30

```

3. 作用域插槽

1. 理解: **数据在组件的自身, 但根据数据生成的结构需要组件的使用者来决定。** (新闻数据在 `News` 组件中, 但使用数据所遍历出来的结构由 `App` 组件决定)
2. 具体编码:

```

1 父组件中:
2 <Game v-slot="params">
3   <!-- <Game v-slot:default="params"> -->
4   <!-- <Game #default="params"> -->
5   <ul>
6     <li v-for="g in params.games" :key="g.id">{{ g.name }}</li>
7
8   </ul>
9
10  </Game>
11
12
13 子组件中:
14  <template>
15    <div class="category">
16      <h2>今日游戏榜单</h2>
17
18      <slot :games="games" a="哈哈"></slot>
19
20    </div>
21
22  </template>
23
24
25  <script setup lang="ts" name="Category">
26  import {reactive} from 'vue'
27  let games = reactive([
28    {id:'asgdytsa01',name:'英雄联盟'},
29    {id:'asgdytsa02',name:'王者荣耀'},
30    {id:'asgdytsa03',name:'红色警戒'},
31    {id:'asgdytsa04',name:'斗罗大陆'}
32  ])
33  </script>
34

```

7. 其它 API

7.1. 【shallowRef 与 shallowReactive】

shallowRef

1. 作用：创建一个响应式数据，但只对顶层属性进行响应式处理。

2. 用法：

```
1 let myVar = shallowRef(initialValue);
```

3. 特点：只跟踪引用值的变化，不关心值内部的属性变化。

shallowReactive

1. 作用：创建一个浅层响应式对象，只会使对象的最顶层属性变成响应式的，对象内部的嵌套属性则不会变成响应式的

2. 用法：

```
1 const myObj = shallowReactive({ ... });
```

3. 特点：对象的顶层属性是响应式的，但嵌套对象的属性不是。

总结

通过使用 `shallowRef()` 和 `shallowReactive()` 来绕开深度响应。浅层式 API 创建的状态只在其顶层是响应式的，对所有深层的对象不会做任何处理，避免了对每一个内部属性做响应式所带来的性能成本，这使得属性的访问变得更快，可提升性能。

7.2. 【readonly 与 shallowReadonly】

readonly

1. 作用：用于创建一个对象的深只读副本。

2. 用法：

```
1 const original = reactive({ ... });
2 const readOnlyCopy = readonly(original);
```

3. 特点：

- 对象的所有嵌套属性都将变为只读。
- 任何尝试修改这个对象的操作都会被阻止（在开发模式下，还会在控制台中发出警告）。

4. 应用场景:

- 创建不可变的状态快照。
- 保护全局状态或配置不被修改。

shallowReadonly

1. 作用: 与 `readonly` 类似, 但只作用于对象的顶层属性。

2. 用法:

```
1 const original = reactive({ ... });
2 const shallowReadOnlyCopy = shallowReadonly(original);
```

3. 特点:

- 只将对象的顶层属性设置为只读, 对象内部的嵌套属性仍然是可变的。
- 适用于只需保护对象顶层属性的场景。

7.3. 【toRaw 与 markRaw】

toRaw

1. 作用: 用于获取一个响应式对象的原始对象, `toRaw` 返回的对象不再是响应式的, 不会触发视图更新。

官网描述: 这是一个可以用于临时读取而不引起代理访问/跟踪开销, 或是写入而不触发更改的特殊方法。不建议保存对原始对象的持久引用, 请谨慎使用。

何时使用? —— 在需要将响应式对象传递给非 `Vue` 的库或外部系统时, 使用 `toRaw` 可以确保它们收到的是普通对象

2. 具体编码:

```
1 import { reactive,toRaw,markRaw,isReactive } from "vue";
2
3 /* toRaw */
4 // 响应式对象
5 let person = reactive({name:'tony',age:18})
6 // 原始对象
7 let rawPerson = toRaw(person)
8
9
10 /* markRaw */
11 let citysd = markRaw([
12   {id:'asdda01',name:'北京'},
13   {id:'asdda02',name:'上海'},
14   {id:'asdda03',name:'天津'},
15   {id:'asdda04',name:'重庆'}
16 ])
17 // 根据原始对象citys去创建响应式对象citys2 — 创建失败，因为citys被markRaw标记了
18 let citys2 = reactive(citys)
19 console.log(isReactive(person))
20 console.log(isReactive(rawPerson))
21 console.log(isReactive(citys))
22 console.log(isReactive(citys2))
```

markRaw

1. 作用：标记一个对象，使其永远不会变成响应式的。

例如使用 `mockjs` 时，为了防止误把 `mockjs` 变为响应式对象，可以使用 `markRaw` 去标记 `mockjs`

2. 编码：

```
1 /* markRaw */
2 let citys = markRaw([
3   {id:'asdda01',name:'北京'},
4   {id:'asdda02',name:'上海'},
5   {id:'asdda03',name:'天津'},
6   {id:'asdda04',name:'重庆'}
7 ])
8 // 根据原始对象citys去创建响应式对象citys2 — 创建失败，因为citys被markRaw标记了
9 let citys2 = reactive(citys)
```

7.4. [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逻辑控制。

实现防抖效果（`useSumRef.ts`）：

```
1 import {customRef} from "vue";
2
3 export default function(initValue:string,delay:number){
4   let msg = customRef((track,trigger)=>{
5     let timer:number
6     return {
7       get(){
8         track() // 告诉Vue数据msg很重要，要对msg持续关注，一旦变化就更新
9         return initialValue
10      },
11      set(value){
12        clearTimeout(timer)
13        timer = setTimeout(() => {
14          initialValue = value
15          trigger() //通知Vue数据msg变化了
16        }, delay);
17      }
18    }
19  })
20  return {msg}
21 }
```

组件中使用：

8. Vue3新组件

8.1. 【Teleport】

- 什么是Teleport？—— Teleport 是一种能够将我们的组件html结构移动到指定位置的技术。

```
1 <teleport to='body' >
2   <div class="modal" v-show="isShow">
3     <h2>我是一个弹窗</h2>
4     <p>我是弹窗中的一些内容</p>
5   <button @click="isShow = false">关闭弹窗</button>
6 
7 </div>
8
9 </teleport>
10
11 </teleport>
12
```

8.2. 【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
 - 使用 `Suspense` 包裹组件，并配置好 `default` 与 `fallback`

```
1 import { defineAsyncComponent, Suspense } from "vue";
2 const Child = defineAsyncComponent(()=>import('./Child.vue'))
```

```

1  <template>
2    <div class="app">
3      <h3>我是App组件</h3>
4
5    <Suspense>
6      <template v-slot:default>
7        <Child/>
8      </template>
9
10     <template v-slot:fallback>
11       <h3>加载中.....</h3>
12
13     </template>
14
15   </Suspense>
16
17 </div>
18
19 </template>
20

```

8.3. 【全局API转移到应用对象】

- `app.component`
- `app.config`
- `app.directive`
- `app.mount`
- `app.unmount`
- `app.use`

8.4. 【其他】

- 过渡类名 `v-enter` 修改为 `v-enter-from`、过渡类名 `v-leave` 修改为 `v-leave-from`。
- `keyCode` 作为 `v-on` 修饰符的支持。
- `v-model` 指令在组件上的使用已经被重新设计，替换掉了 `v-bind.sync`。
- `v-if` 和 `v-for` 在同一个元素身上使用时的优先级发生了变化。

- 移除了 `$on`、`$off` 和 `$once` 实例方法。
- 移除了过滤器 `filter`。
- 移除了 `$children` 实例 `propert`。.....