

一.Vue是什么

二.Vue核心语法

1.setup

1.setup概述

2.setup返回值

3.setup与OptionsAPI

4.setup的语法糖

5.ref创建_基本类型的响应式数据

6.reaction创建_对象类型的数据 (只能定义对象类型的响应式数据)

7.ref创建_对象类型的响应式数据

8.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】

修改整个对象的方法,第一张图片不可以,第二张图片可以

```
// 数据
let car = reactive({brand:'奔驰',price:100})
let sum = ref(0)

// 方法
function changeBrand(){
  car.brand = '宝马'
}
function changePrice(){
  car.price += 10
}
function changeCar(){
  car = {brand:'奥拓',price:1}
}
function changeSum(){
  sum.value += 1
}

/script>
```

```
function changeCar(){
  // car = reactive({brand:'奥拓',price:1})
  Object.assign(car,{brand:'奥拓',price:1})
}
```

```

// 数据
let car = ref({brand:'奔驰',price:100})
let sum = ref(0)

// 方法
function changeBrand(){
  car.value.brand = '宝马'
}
function changePrice(){
  car.value.price += 10
}
function changeCar(){
  * // car = {brand:'奥拓',price:1} //这么写页面不更新的
  // car = reactive({brand:'奥拓',price:1}) //这么写页面不更新的

  // 下面这个写法页面可以更新
  // Object.assign(car,{brand:'奥拓',price:1})
  car.value = {brand:'奥拓',price:1}
}

```

9.toRefs和toRef

```

<script lang="ts" setup name="Person">
  import {reactive} from 'vue'

  // 数据
  let person = reactive({
    name: '张三',
    age: 18
  })

  let {name, age} = person

  // 方法
  function changeName(){
    name += '~'
  }
  function changeAge(){
    age += 1
  }

```

person.name = 666
 person.age = 777

666
 let name = person.name 666
 let age = person.age

英、全 商晓睿

```

<script lang="ts" setup name="Person">
  import {reactive,toRefs} from 'vue'

  // 数据
  let person = reactive({
    name:'张三',
    age:18
  })

  let {name,age} = toRefs(person)

  // 方法
  function changeName(){
    name += '~'
    console.log(name,person.name)
  }
  function changeAge(){
    age += 1
  }
</script>

```

Diagram illustrating the use of `toRefs` and `ref`:

- A yellow arrow points from the `name` property in the `toRefs` destructuring assignment to the word `ref`.
- A green arrow points from the `age` property in the `toRefs` destructuring assignment to the word `ref`.

```

7   </div>
8   </template>
9
10  <script lang="ts" setup name="Person">
11    import {reactive,toRefs,toRef} from 'vue'
12
13    // 数据
14    let person = reactive({
15      name:'张三',
16      age:18
17    })
18
19    let {name,age} = toRefs(person)
20    let nl = toRef(person,'age')
21    console.log(nl.value)
22
23    // 方法
24    function changeName(){
25      name.value += '~'
26      console.log(name.value,person.name)
27    }

```

Diagram illustrating the use of `toRef` and `ref`:

- A yellow arrow points from the `age` property in the `toRef` function call to the word `ref`.

核心区别

场景	toRef	toRefs
处理范围	单个属性	所有属性
返回值	单个 ref 对象	包含多个 ref 的普通对象
用途	单独提取某个响应式属性	解构整个响应式对象（避免丢失响应性）

toRef 和 toRefs 的核心价值是保留响应式对象属性的响应性，尤其在解构或传递属性时非常有用，避免因普通解构导致响应性丢失。

10.computed计算属性

```
let firstName = ref('zhang');
let lastName = ref('san');
let fullName = computed(() => {
  return firstName.value.slice(0,1).toUpperCase() + firstName.value.slice(1) + '-' + la
});

function changeFullName(){
  firstName.value = 'li';
  lastName.value = 'si';
}
```

```
// 这么定义的fullName是一个计算属性，且是只读的
/* let fullName = computed(()=>{
  console.log(1)
  return firstName.value.slice(0,1).toUpperCase() + firstName.value.slice(1) + '-' + lastName.value
}) */
```

```
// 这么定义的fullName是一个计算属性，可读可写
let fullName = computed({
  get(){
    return firstName.value.slice(0,1).toUpperCase() + firstName.value.slice(1) + '-' + la
  },
  set(val){
    const [str1,str2] = val.split('-')
    firstName.value = str1
    lastName.value = str2
  }
})
```

```

> components > Person.vue > {} script setup
21 // 这么定义的fullName是一个计算属性，且是只读的
22 /* let fullName = computed(()=>{
23   console.log(1)
24   return firstName.value.slice(0,1).toUpperCase() + firstName.value.slice(1) + '-' + lastName.value
25 }) */
26
27 // 这么定义的fullName是一个计算属性，可读可写
28 let fullName = computed({
29   get(){
30     return firstName.value.slice(0,1).toUpperCase() + firstName.value.slice(1) + '-' + lastName.value
31   },
32   set(val){
33     console.log('set',val)
34   }
35 })
36
37 function changeFullName(){
38   fullName.value = 'li-si'
39 }
40 </script>
41

```

2.watch

3.watchEffect

watch必须明确指出监视的谁,watchEffect直接用,不用告诉他监视的谁

```

> components > Person.vue > {} template > div.person > h2
15 let temp = ref(10)
16 let height = ref(0)
17
18 // 方法
19 function changeTemp(){
20   temp.value += 10
21 }
22 function changeHeight(){
23   height.value += 10
24 }
25
26 // 监视
27 watch([temp,height],(value)=>{
28   // 从value中获取最新的水温(newTemp)、最新的水位(newHeight)
29   let [newTemp,newHeight] = value
30   // 逻辑
31   if(newTemp >= 60 || newHeight >= 80){
32     console.log('给服务器发请求')
33   }
34 })
35

```

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：

```
<template>
  <div class="person">
    <h1>需求：水温达到50℃，或水位达到20cm，则联系服务器</h1>
    <h2 id="demo">水温：{{temp}}</h2>
    <h2>水位：{{height}}</h2>
    <button @click="changePrice">水温+1</button>
    <button @click="changeSum">水位+10</button>
  </div>
</template>

<script lang="ts" setup name="Person">
  import {ref,watch,watchEffect} from 'vue'
```

4.标签的ref属性

5.回顾TS中的接口,泛型,自定义类型

1.export的使用

2.接口-对一个值

3.对数组

1.第一种方式

```
let personList:Array<PersonInter> = [
  {id:'asyud7asfd01',name:'张三',age:60},
  {id:'asyud7asfd02',name:'李四',age:18},
  {id:'asyud7asfd03',name:'王五',age:5}
]
```

2.第二种方式

```

1 //定义一个接口,用于显示person对象的具体属性
2 export interface PersonInter{
3     id: string;
4     name: string;
5     age: number;
6 }
7 //自定义类型
8 //export type Persons = Array<PersonInter>;
9 export type Persons = PersonInter[]
10

```

```

<script lang="ts" setup name="app">
import { ref ,reactive ,toRefs,computed,watch,watchEffect} from 'vue';
import { type PersonInter,type Persons} from '../types'

let person: PersonInter = {id: '1', name: '张三', age: 25};
let personList: Persons = [
    {id: '1001', name: '张三', age: 25},
    {id: '1002', name: '李四', age: 30}
]

```

6.props的使用(父子组件通信的核心机制)

7.生命周期(组件的一生)

创建-挂载-更新-销毁

人的生命周期：

【时刻】	【要做的事】
出生	哭、
经历	哭、笑
死亡	遗嘱

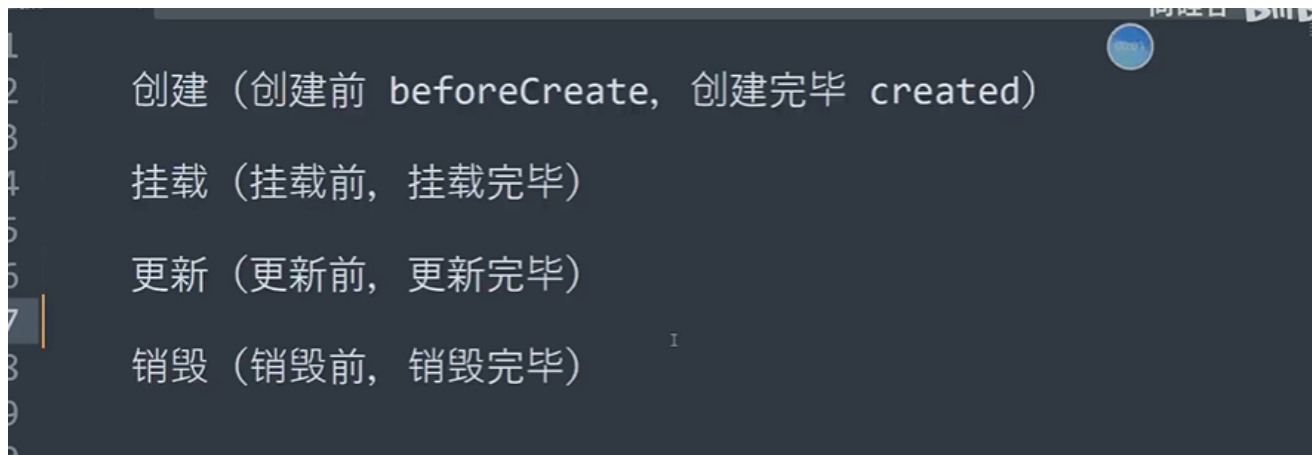
组件的生命周期：

【时刻】	【调用特定的函数】
创建✓	created
挂载✓	mounted
更新	
销毁 ⁺	

生命周期、生命周期函数、生命周期钩子

1.Vue2的生命周期

创建有4个阶段,钩子有8个



```
},
// 创建前的钩子
beforeCreate(){
  console.log('创建前')
},
// 创建完毕的钩子
created(){
  console.log('创建完毕')
}
```

debugger可以暂停运行

```
},
// 挂载前
beforeMount(){
  console.log('挂载前')
  debugger;
},
// 挂载完毕
mounted(){
  console.log('挂载完毕')
}
```

```
},
// 挂载完毕
mounted(){
  console.log('挂载完毕')
},
// 更新前
beforeUpdate(){
  console.log('更新前')
},
// 更新完毕
updated(){
  console.log('更新完毕')
},
// 销毁前
beforeDestroy(){
  console.log('销毁前')
},
// 销毁完毕
destroyed(){
  console.log('销毁完毕')
}
}
```

行 56, 列 24 空格: 2 UTF-8 CRLF vue

所以v-if触发销毁,而v-show不会

特性	v-if	v-show
DOM 存在性	条件 false 时移除, true 时创建	始终存在, 仅通过 CSS 隐藏
切换成本	高 (DOM 操作)	低 (仅修改 CSS)
初始渲染成本	低 (条件 false 时不渲染)	高 (无论条件如何都渲染)
适用场景	条件很少变化 (如权限判断)	频繁切换 (如弹窗、标签页)
支持 <code><template></code>	是	否

2.Vue3的生命周期

```
beforeMount, onMounted, onBeforeUpdate, onUpdated, onBeforeUnmount, onUnmounted }
```

```

}
// 创建
console.log('创建')

// 挂载前
onBeforeMount(()=>{
  console.log('挂载前')
})
// 挂载完毕
onMounted(()=>{
  console.log('挂载完毕')
})

// 更新前
onBeforeUpdate(()=>{
  console.log('更新前')
})
// 更新完毕
onUpdated(()=>{
  console.log('更新完毕')
})

```

```

  console.log('挂载完毕')
})
// 更新前
onBeforeUpdate(()=>{
  console.log('更新前')
})
// 更新完毕
onUpdated(()=>{
  console.log('更新完毕')
})
// 卸载前
onBeforeUnmount(()=>{
  console.log('卸载前')
})
// 卸载完毕
onUnmounted(()=>{
  console.log('卸载完毕')
})
</script>

```

父组件先开始挂载，但子组件先完成挂载

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`

销毁阶段: `onBeforeUnmount`、`onUnmounted`

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)

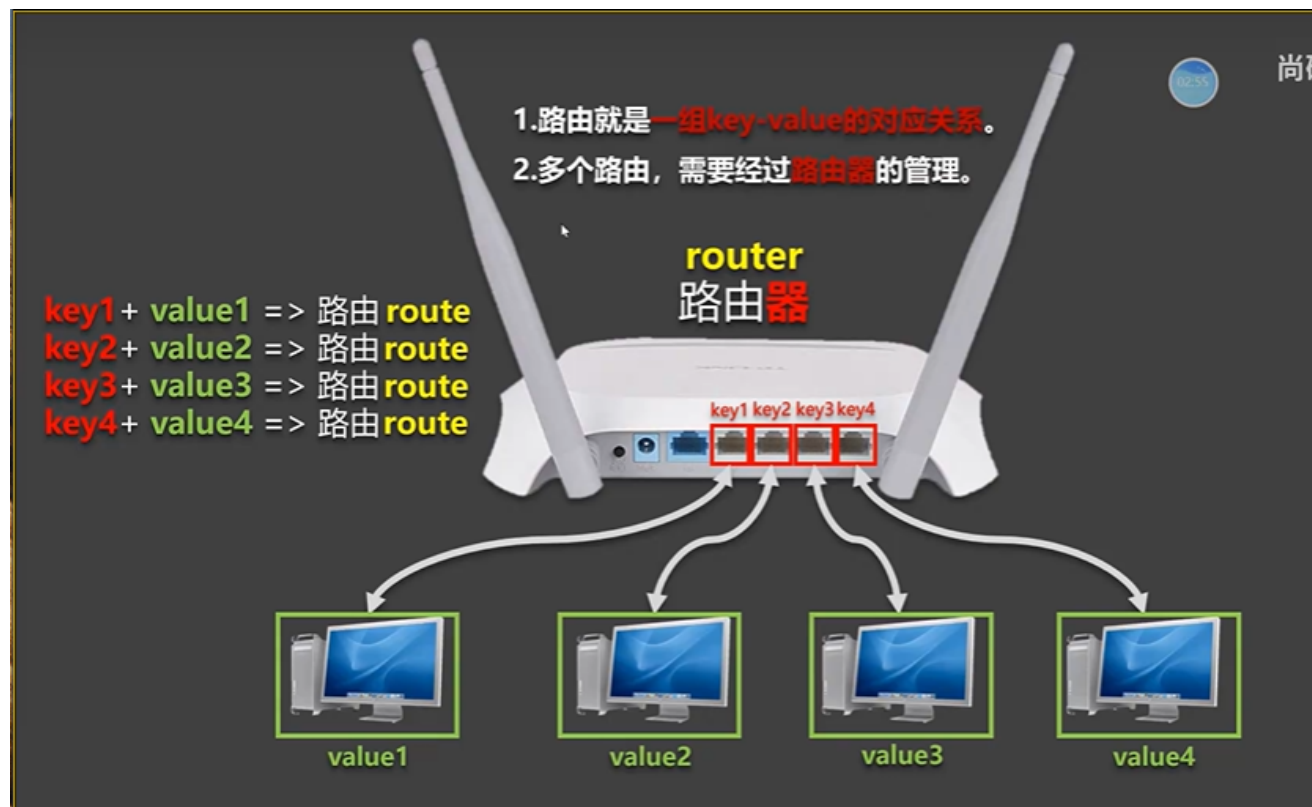
- 示例代码:

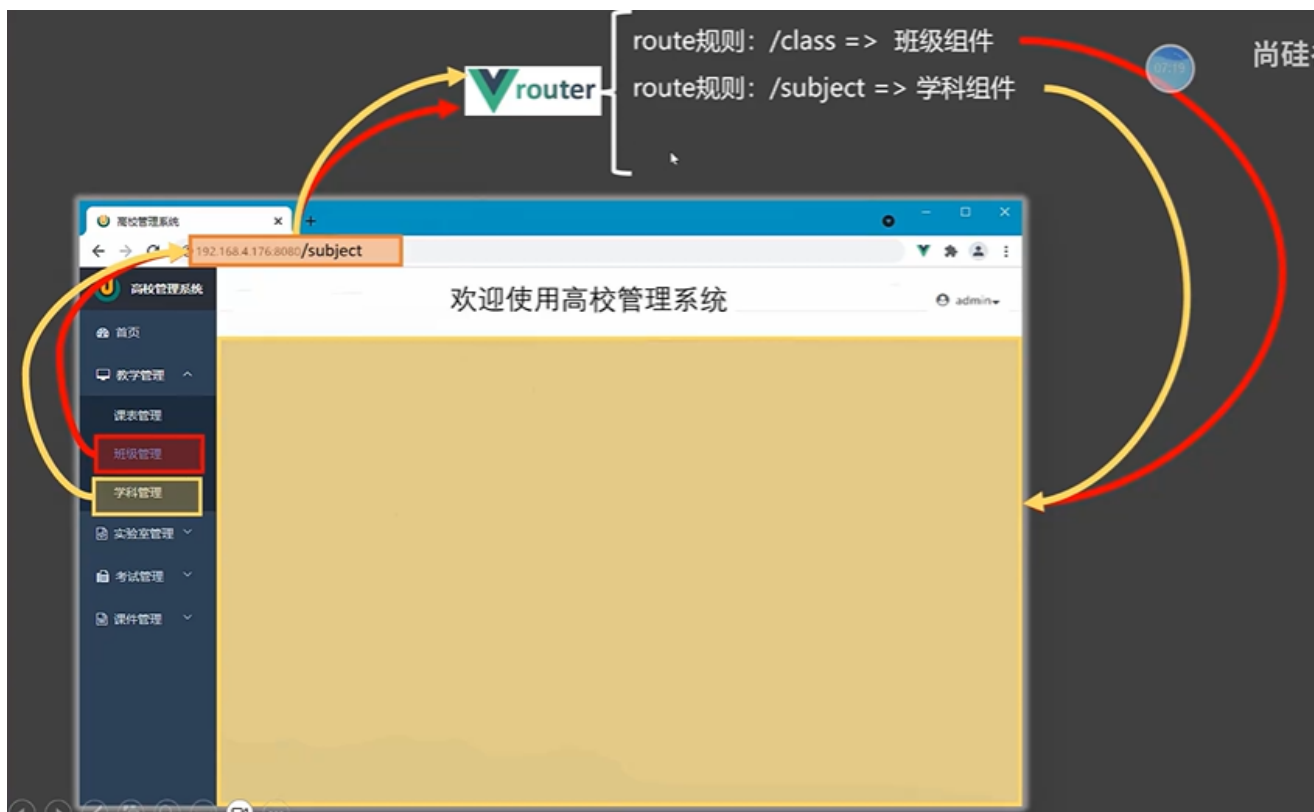
```
<template>
  <div class="person">
    <h2>当前求和为: {{ sum }}</h2>
    <button @click="changeSum">点我sum+1</button>
  </div>
</template>
```

<!-- vue3 写法 -->

8.自定义Hooks

9.路由





1.基本切换效果

- 1.导航区、展示区
- 2.请来路由器
- 3.制定路由的具体规则（什么路径，对应着什么组件）
- 4.形成 一个一个的 【???.vue】 Class.vue Subject.vue

App.vue

```

1 <template>
2   <!-- html -->
3   <div class="app">
4     <h2>Vue路由测试</h2>
5     <!-- 导航区 -->
6     <div class="navigate">
7       <RouterLink to="/home" active-class="active">首页</RouterLink>
8       <RouterLink to="/news" active-class="active">新闻</RouterLink>
9       <RouterLink to="/about" active-class="active">关于</RouterLink>
10    </div>
11    <!-- 展示区 -->
12    <div class="main-content">
13      <RouterView></RouterView>
14    </div>
15  </div>

```

```
16 </template>
17
18 <script setup name="app">
19   import { RouterView, RouterLink } from 'vue-router'
20 </script>
21
22
23 <style scoped>
24   /* css */
25   /* 基础样式重置与全局设置 */
26   * {
27     margin: 0;
28     padding: 0;
29     box-sizing: border-box;
30     font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
31   }
32
33   /* 容器样式 */
34   .app {
35     max-width: 1200px;
36     margin: 0 auto;
37     padding: 2rem;
38     min-height: 100vh;
39     background-color: #f5f7fa;
40   }
41
42   /* 标题样式 */
43   h2 {
44     color: #2c3e50;
45     text-align: center;
46     margin-bottom: 2rem;
47     font-size: 2rem;
48     font-weight: 600;
49     letter-spacing: 0.5px;
50   }
51
52   /* 导航区样式 */
53   .navigate {
54     display: flex;
55     justify-content: center;
56     gap: 0.5rem;
```

```
57     margin-bottom: 2rem;
58     padding: 1rem;
59     background-color: white;
60     border-radius: 8px;
61     box-shadow: 0 2px 10px rgba(0, 0, 0, 0.05);
62 }
63
64 /* 导航链接样式 */
65 .navigate a {
66     text-decoration: none;
67     color: #34495e;
68     padding: 0.8rem 1.5rem;
69     border-radius: 6px;
70     font-weight: 500;
71     transition: all 0.3s ease;
72     position: relative;
73     overflow: hidden;
74 }
75
76 /* 导航链接 hover 效果 */
77 .navigate a:hover {
78     color: #3498db;
79     background-color: #f8f9fa;
80 }
81
82 /* 导航链接点击/激活状态（后续可配合路由激活类使用） */
83 .navigate a:active {
84     color: #2980b9;
85 }
86
87 .navigate a::after {
88     content: '';
89     position: absolute;
90     bottom: 0;
91     left: 0;
92     width: 0;
93     height: 2px;
94     background-color: #3498db;
95     transition: width 0.3s ease;
96 }
97
```

```
98 .navigate a:hover::after {
99     width: 100%;
100 }
101
102 /* 主内容区样式 */
103 .main-content {
104     background-color: white;
105     min-height: 400px;
106     border-radius: 8px;
107     padding: 2rem;
108     box-shadow: 0 2px 15px rgba(0, 0, 0, 0.07);
109     color: #4a6fa5;
110     font-size: 1.1rem;
111     display: flex;
112     align-items: center;
113     justify-content: center;
114     text-align: center;
115     border: 1px dashed #e1e4e8;
116     transition: all 0.3s ease;
117 }
118
119 .main-content:hover {
120     box-shadow: 0 5px 20px rgba(0, 0, 0, 0.1);
121     border-color: #d1d5da;
122 }
123
124 /* 响应式设计 - 适配手机屏幕 */
125 @media (max-width: 768px) {
126     .app {
127         padding: 1rem;
128     }
129
130     h2 {
131         font-size: 1.5rem;
132         margin-bottom: 1.5rem;
133     }
134
135     .navigate {
136         flex-direction: column;
137         align-items: stretch;
138         gap: 0.3rem;
```

```
139     }  
140  
141     .navigate a {  
142         text-align: center;  
143         padding: 0.7rem;  
144     }  
145  
146     .main-content {  
147         min-height: 300px;  
148         padding: 1.5rem;  
149         font-size: 1rem;  
150     }  
151 }  
152 </style>
```

main.js

```
1 import './assets/main.css'  
2 //引入createApp用于创建应用  
3 import { createApp } from 'vue'  
4 import { createPinia } from 'pinia'  
5  
6 //引入App根组件  
7 import App from './App.vue'  
8 //引入路由  
9 import router from './router'  
10  
11 //创建应用  
12 const app = createApp(App)  
13  
14 app.use(createPinia())  
15 //使用路由  
16 app.use(router)  
17  
18 //挂载应用  
19 app.mount('#app')  
20
```

index.js

```
1 import { createRouter, createWebHistory } from 'vue-router'
2 //引入一个一个可能要呈现的组件
3 import Home from '@/components/Home.vue'
4 import News from '@/components/News.vue'
5 import About from '@/components/About.vue'
6
7 const router = createRouter({
8   history: createWebHistory(import.meta.env.BASE_URL), //路由器的工作模式
9   //创建路由
10  routes: [
11    {
12      path: '/',
13      component: Home
14    },
15    {
16      path: '/home',
17      component: Home
18    },
19    {
20      path: '/news',
21      component: News
22    },
23    {
24      path: '/about',
25      component: About
26    }
27  ]
28 })
29 //暴露出去router
30 export default router
```

2.两个注意点

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，默认是被销毁掉的，需要的时候再去挂载。

4.4. 【to的两种写法】

```
<!-- 第一种: to的字符串写法 -->
<router-link active-class="active" to="/home">主页</router-link>

<!-- 第二种: to的对象写法 -->
<router-link active-class="active" :to="{path: '/home'}">Home</router-link>
```

4.5. 【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。

```
const router = createRouter({
```

Demo.vue

路由组件: 靠路由的规则渲染出来的

```
routes:[
  {path: '/demo', component: Demo}
]
```

一般组件: 亲手写标签出来的

3.路由器工作模式

在 Vue Router 中,“路由器的工作模式”指的是路由在浏览器中处理 URL 路径和历史记录的方式, 主要影响 URL 的表现形式和页面跳转时的行为。Vue Router 提供了两种核心工作模式, 分别是 hash 模式和 history 模式, 通过 `createWebHashHistory` 和 `createWebHistory` 函数创建。

前端多的喜欢用history模式,后端多的喜欢用hash模式

4.5. 【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。

```
const router = createRouter({
  history: createWebHistory(), //history模式
  //*****//
})
```

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。

```
const router = createRouter({
  history: createWebHashHistory(), //hash模式
  //*****//
})
```



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。

```
const router = createRouter({
  history: createWebHashHistory(), //hash模式
  //*****//
})
```

4.to的两种写法

5.命名路由

6.嵌套路由

7.路由_query参数

query参数

1. 传递参数

```
<!-- 跳转并携带query参数 (to的字符串写法) -->
<router-link to="/news/detail?a=1&b=2&content=欢迎你">
  跳转
</router-link>
```

```
<!-- 跳转并携带query参数 (to的对象写法) -->
```

```
<RouterLink
  :to="{
    //name:'xiang', //用name也可以跳转
    path:'/news/detail',
    query:{
      id:news.id,
      title:news.title,
      content:news.content
    }
  }"
>
  {{news.title}}
</RouterLink>
```

2. 接收参数:

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

Detail.vue

```
1 <template>
2   <ul class="news-list">
3     <li>编号:{{ query.id}}</li>
4     <li>标题:{{ query.title}}</li>
5     <li>内容:{{ query.content}}</li>
6   </ul>`
7 </template>
8 <script setup >
9   import {useRoute} from 'vue-router'
10  import {toRefs} from 'vue'
11  let route = useRoute()
12  let {query} = toRefs(route)
13 </script>
14 <style scoped>
15 </style>
```

```
1 <template>
2   <div class="news">
3     <!-- 导航区 -->
4     <ul>
5       <li v-for="news in newsList" :key="news.id">
6         <!-- 第一种写法 -->
7         <!-- <RouterLink :to="`/news/detail?
id=${news.id}&title=${news.title}&content=${news.content}`">{{news.title}}</RouterLink> -
->
8         <!-- 第二种写法 -->
9         <!-- <RouterLink :to="{path: '/news/detail', query:
{id:news.id,title:news.title,content:news.content}}">{{news.title}}</RouterLink> -->
10        <RouterLink :to="{name: 'detail1', query:
{id:news.id,title:news.title,content:news.content}}">{{news.title}}</RouterLink>
11      </li>
12    </ul>
13    <!-- 展示区 -->
14    <div class="news-content">
15      <RouterView></RouterView>
16    </div>
17  </div>
18 </template>
19 <script setup name="news">
20   import {reactive} from 'vue'
21
22   const newsList = reactive([
23     {id: '1001', title: '很好的抗癌食物', content: '西兰花'},
24     {id: '1002', title: '如何一夜暴富', content: '学IT'},
25     {id: '1003', title: '震惊,万万没想到', content: '明天是周一'},
26     {id: '1004', title: '好消息,好消息', content: '快过年了'},
27   ])
28 </script>
29 <style scoped>
30
31 </style>
```

8.路由_params参数

9.路由_props配置

10.replace属性

H3