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.reation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reative

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

```vue
<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>
```


```vue
  </div>
</template>

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

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

  let {name,age} = toRefs(person)
  let nl = toRef(person,'age')
  console.log(nl.value)

  // 方法
  function changeName(){
    name.value += '~'
    console.log(name.value,person.name)
  }
```

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）

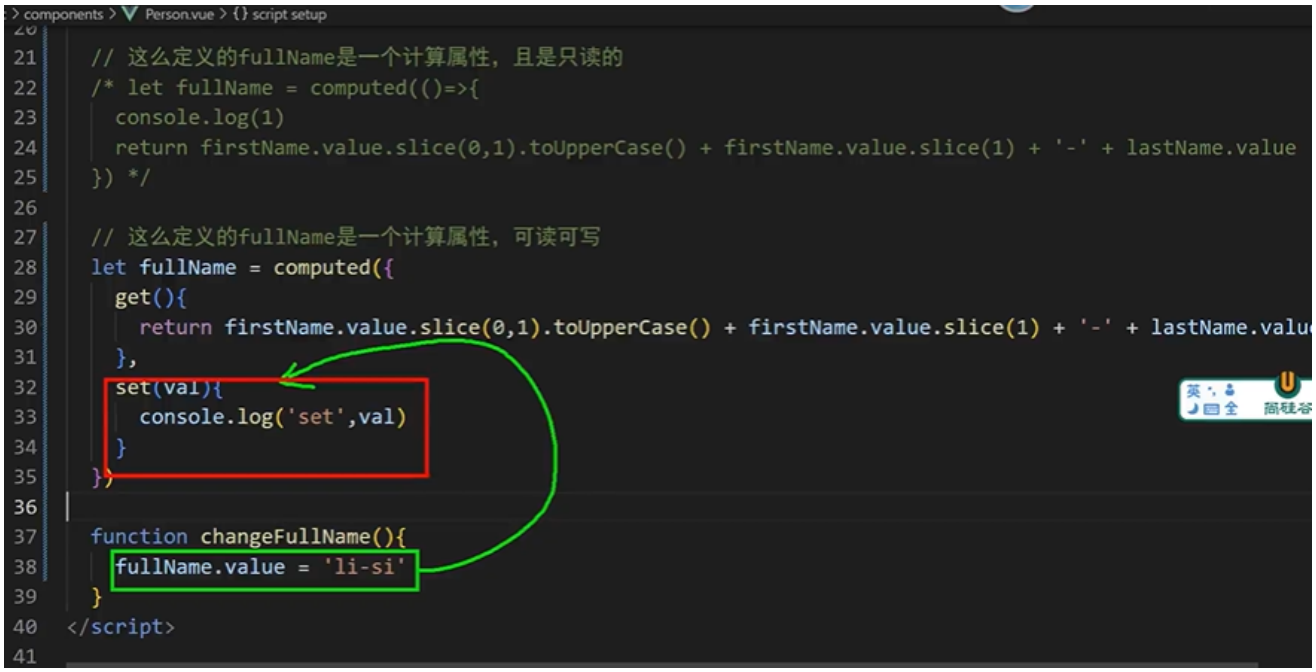
toRef 和 toRefs 的核心价值是保留响应式对象属性的响应性，尤其在解构或传递属性时非常有用，避免因普通解构导致响应性丢失。

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
  }
})
```



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**

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

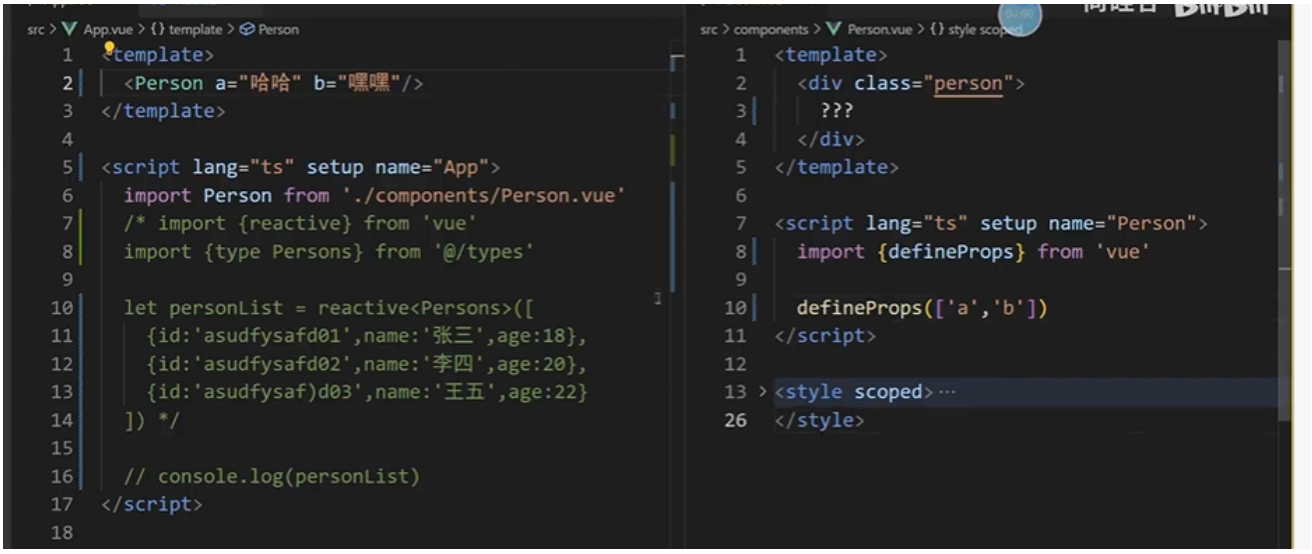
```ts
//定义一个接口,用于显示person对象的具体属性
export interface PersonInter{
    id: string;
    name: string;
    age: number;
}
//自定义类型
//export type Persons = Array<PersonInter>;
export type Persons = PersonInter[]
```

```ts
<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)

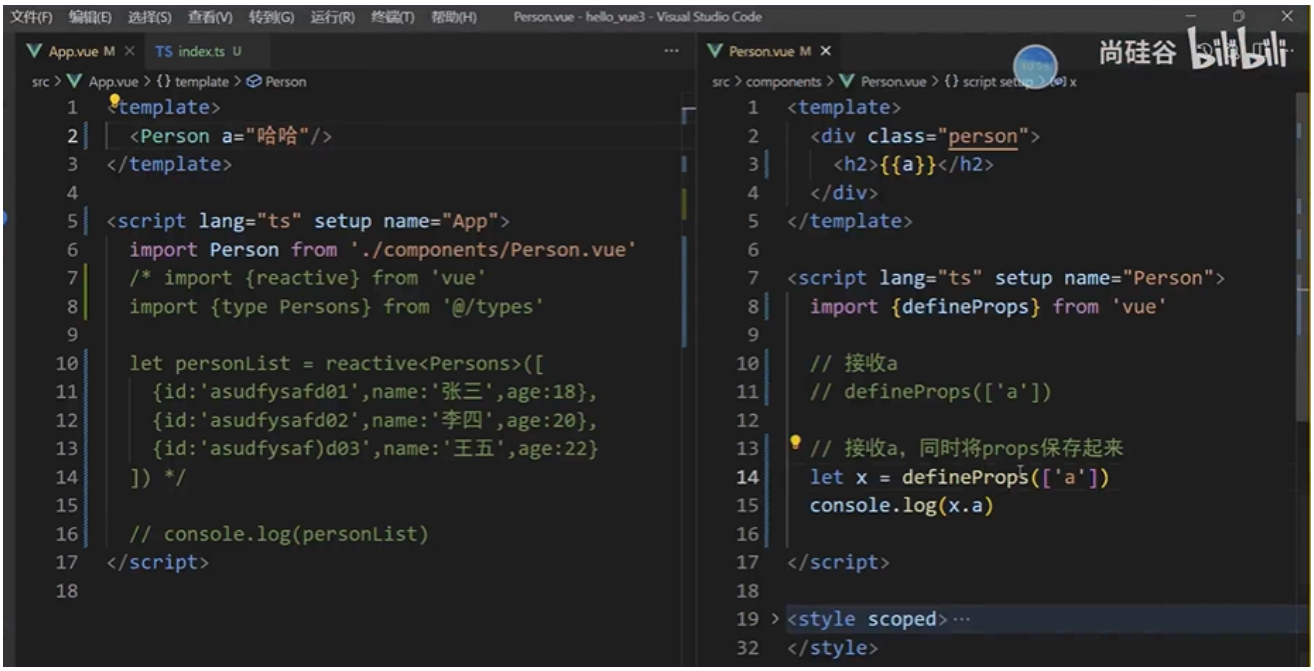
推荐reative直接用泛型

```ts
<script lang="ts" setup name="App">
  import Person from './components/Person.vue'
  import {reactive} from 'vue'
  import {type Persons} from '@/types'

  let personList = reactive<Persons>([
    {id:'asudfysafd01',name:'张三',age:18},
    {id:'asudfysafd02',name:'李四',age:20},
    {id:'asudfysaf)d03',name:'王五',age:22}
  ])

  console.log(personList)
</script>
```


src > App.vue > {} template > Person
<template>
<Person a="哈哈" b="嘿嘿"/>
</template>
<script lang="ts" setup name="App">
import Person from './components/Person.vue'
/* import {reactive} from 'vue'
import {type Persons} from '@/types'
let personList = reactive<Persons>([
{id:'asudfysafd01',name:'张三',age:18},
{id:'asudfysafd02',name:'李四',age:20},
{id:'asudfysaf)d03',name:'王五',age:22}
]) */
// console.log(personList)
</script>
src > components > Person.vue > {} style scoped
<template>

???

</template>
<script lang="ts" setup name="Person">
import {defineProps} from 'vue'
defineProps(['a','b'])
</script>
<style scoped>…
</style>

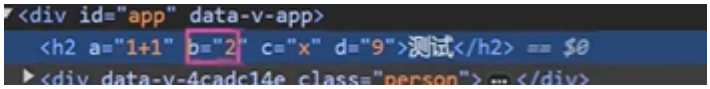

文件(F) 编辑(E) 选择(S) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H) Person.vue - hello_vue3 - Visual Studio Code
App.vue M
TS index.ts U
src > App.vue > {} template > Person
<template>
<Person a="哈哈"/>
</template>
<script lang="ts" setup name="App">
import Person from './components/Person.vue'
/* import {reactive} from 'vue'
import {type Persons} from '@/types'
let personList = reactive<Persons>([
{id:'asudfysafd01',name:'张三',age:18},
{id:'asudfysafd02',name:'李四',age:20},
{id:'asudfysaf)d03',name:'王五',age:22}
]) */
// console.log(personList)
</script>
Person.vue M
src > components > Person.vue > {} script setup > x
<template>

<h2>{{a}}</h2>

</template>
<script lang="ts" setup name="Person">
import {defineProps} from 'vue'
// 接收a
// defineProps(['a'])
// 接收a，同时将props保存起来
let x = defineProps(['a'])
console.log(x.a)
</script>
<style scoped>…
</style>

第一个

```vue
<template>
  <!-- 务必看懂下面这一行代码 -->
  <!-- <h2 a="1+1" :b="1+1" c="x" :d="x" ref="qwe"> -->

  <Person a="哈哈" :list="personList"/>
</template>

<script lang="ts" setup name="App">
  import Person from './components/Person.vue'
  import {reactive} from 'vue'
  import {type Persons} from '@/types'

  let x = 9

  let personList = reactive<Persons>([
    {id:'asudfysafd01',name:'张三',age:18},
    {id:'asudfysafd02',name:'李四',age:20},
    {id:'asudfysaf)d03',name:'王五',age:22}
  ])

</script>
```



`<Person>` 标签的本质是**子组件** `Person.vue` **的“占位符”**，作用

- 在父组件中渲染子组件的 UI；
- 作为父子组件数据传递的桥梁；
- 实现组件的复用与页面模块化。

```vue
      <li v-for="p in list" :key="p.id">
        {{p.name}} -- {{p.age}}
      </li>
    </ul>
  </div>
</template>

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

  // 接收a和list
  defineProps(['a','list'])

  // 接收a，同时将props保存起来
  /* let x = defineProps(['a'])
  console.log(x.a) */
```

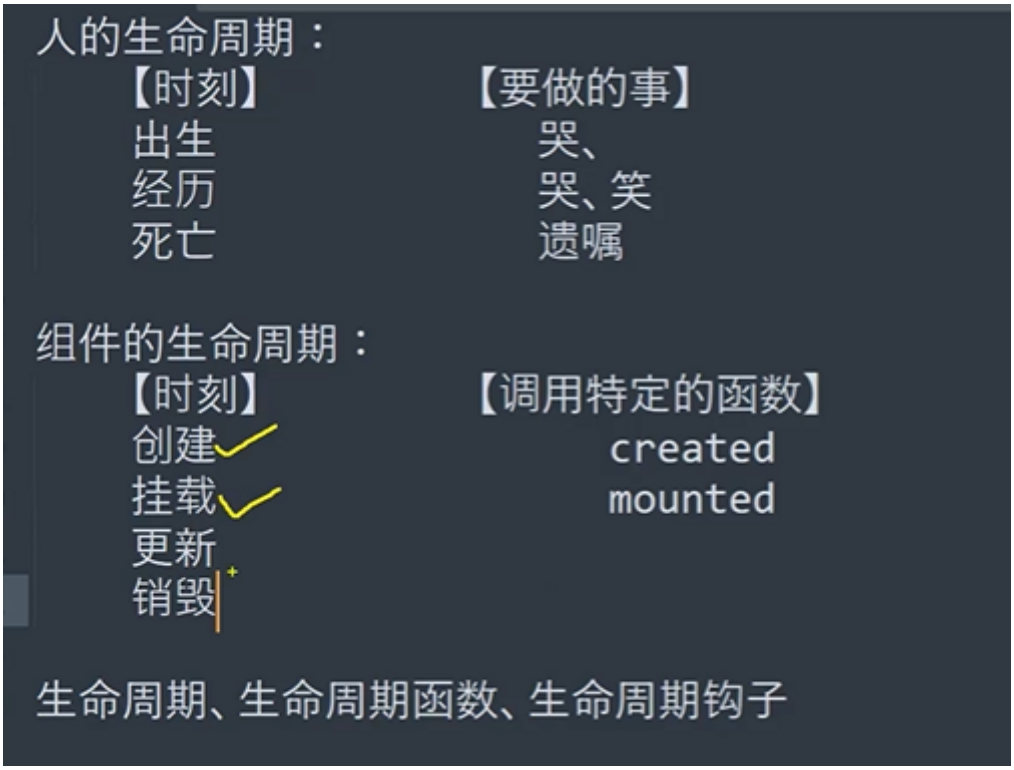
```vue
  // 只接收list
  defineProps(['list'])


  // 接收list，同时将props保存起来
  /* let x = defineProps(['list'])
  console.log(x.list) */
```

define都是宏函数可以不用引入

```vue
  import {defineProps,withDefaults} from 'vue'
  import {type Persons} from '@/types'

  // 只接收list
  // defineProps(['list'])

  // 接收list + 限制类型
  // defineProps<{list:Persons}>()

  //  接收list + 限制类型 + 限制必要性 + 指定默认值
  /* withDefaults(defineProps<{list?:Persons}>(),{
    list:()=> [{id:'ausydgyu01',name:'康师傅·王麻子·特仑苏',age:19}]
  }) */


  // 接收list，同时将props保存起来
  /* let x = defineProps(['list'])
  console.log(x.list) */

</script>
```

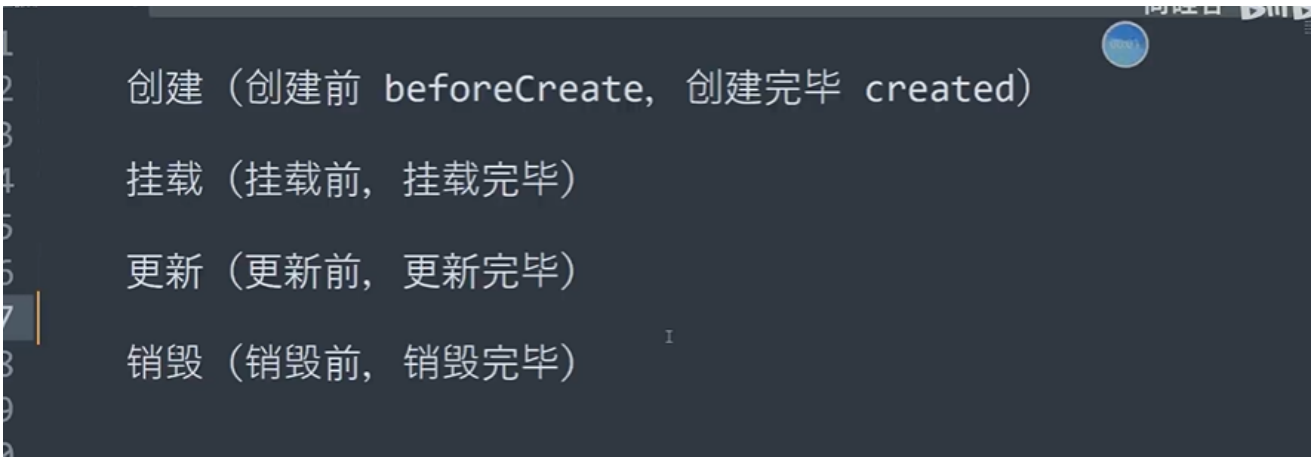
7.生命周期(组件的一生)

创建-挂载-更新-销毁



1.Vue2的生命周期

创建有4个阶段,钩子有8个



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 `<template>`	是	否

2.Vue3的生命周期

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

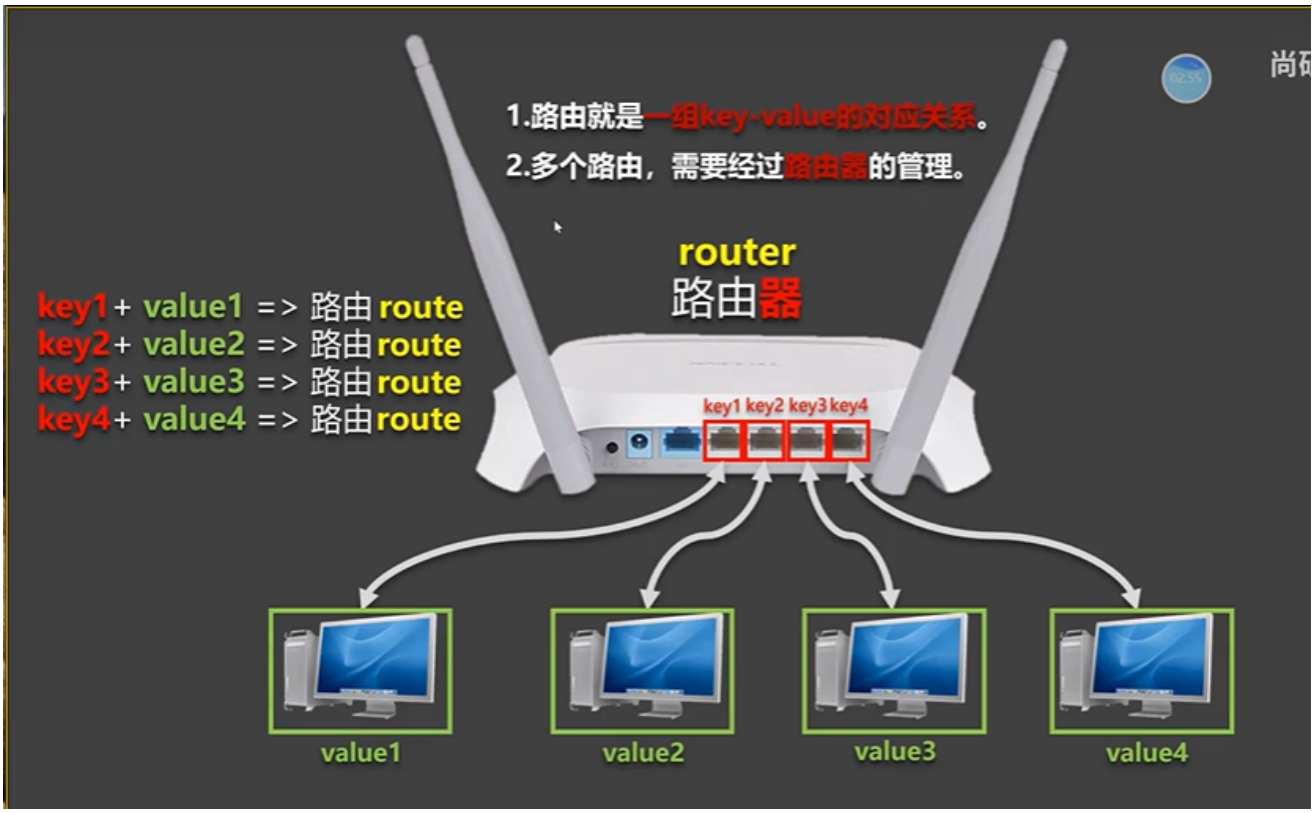
```
}
// 创建
console.log('创建')

// 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')
})
```

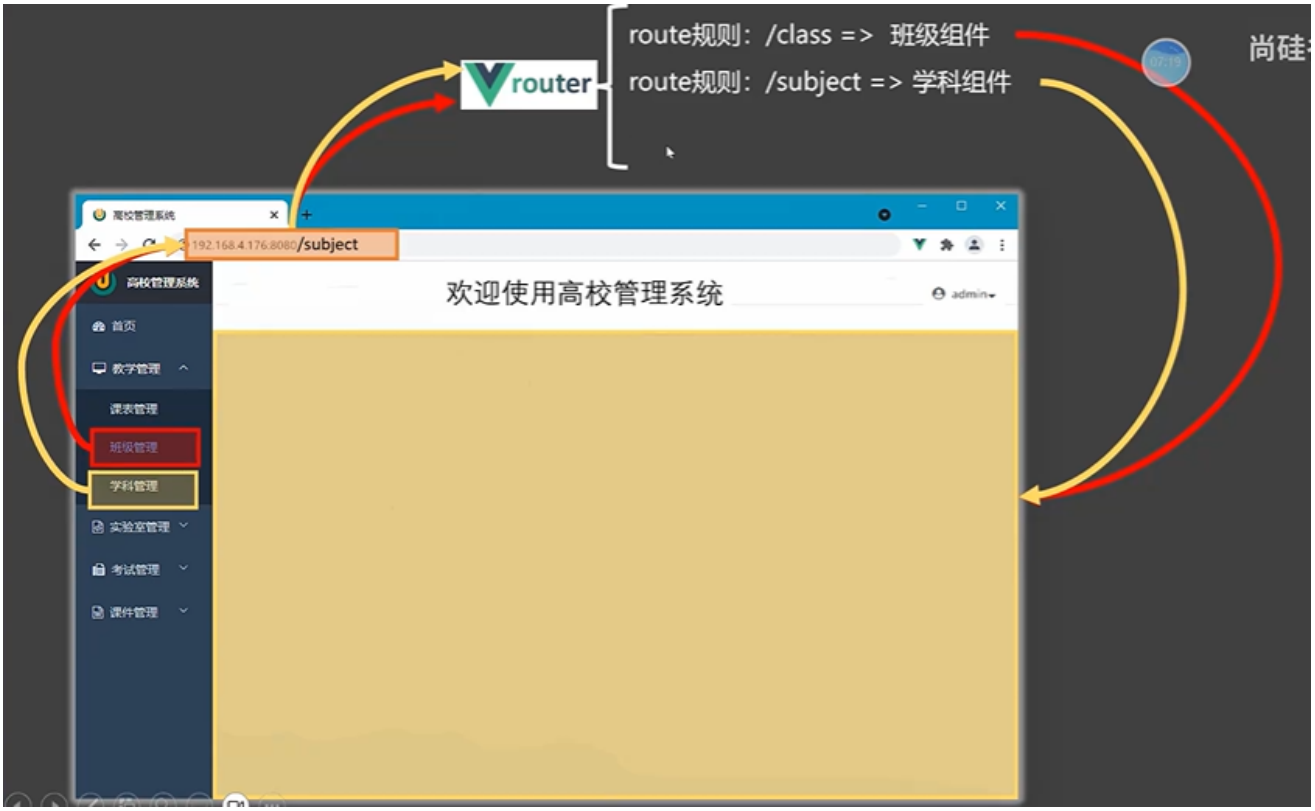
```
  console.log('挂载完毕')
})
// 更新前
onBeforeUpdate(()=>{
  console.log('更新前')
})
// 更新完毕
onUpdated(()=>{
  console.log('更新完毕')
})
// 卸载前
onBeforeUnmount(()=>{
  console.log('卸载前')
})
// 卸载完毕
onUnmounted(()=>{
  console.log('卸载完毕')
})
</script>
```

父组件先开始挂载，但子组件先完成挂载

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

```vue
<template>
    <!-- html -->
     <div class="app">
        <h2>Vue路由测试</h2>
        <!-- 导航区 -->
         <div class="navigate">
            <RouterLink to="/home" active-class="active">首页</RouterLink>
            <RouterLink to="/news" active-class="active">新闻</RouterLink>
            <RouterLink to="/about" active-class="active">关于</RouterLink>
         </div>
         <!-- 展示区 -->
          <div class="main-content">
            <RouterView></RouterView>
          </div>
     </div>
</template>

<script setup name="app">
  import { RouterView,RouterLink} from 'vue-router'
</script>


<style scoped>
    /* css */
    /* 基础样式重置与全局设置 */
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
}

/* 容器样式 */
.app {
    max-width: 1200px;
    margin: 0 auto;
    padding: 2rem;
    min-height: 100vh;
```

```
    background-color: #f5f7fa;
}

/* 标题样式 */
h2 {
    color: #2c3e50;
    text-align: center;
    margin-bottom: 2rem;
    font-size: 2rem;
    font-weight: 600;
    letter-spacing: 0.5px;
}

/* 导航区样式 */
.navigate {
    display: flex;
    justify-content: center;
    gap: 0.5rem;
    margin-bottom: 2rem;
    padding: 1rem;
    background-color: white;
    border-radius: 8px;
    box-shadow: 0 2px 10px rgba(0, 0, 0, 0.05);
}

/* 导航链接样式 */
.navigate a {
    text-decoration: none;
    color: #34495e;
    padding: 0.8rem 1.5rem;
    border-radius: 6px;
    font-weight: 500;
    transition: all 0.3s ease;
    position: relative;
    overflow: hidden;
}

/* 导航链接 hover 效果 */
.navigate a:hover {
    color: #3498db;
    background-color: #f8f9fa;
```

```
}

/* 导航链接点击/激活状态（后续可配合路由激活类使用） */
.navigate a:active {
    color: #2980b9;
}

.navigate a::after {
    content: '';
    position: absolute;
    bottom: 0;
    left: 0;
    width: 0;
    height: 2px;
    background-color: #3498db;
    transition: width 0.3s ease;
}

.navigate a:hover::after {
    width: 100%;
}

/* 主内容区样式 */
.main-content {
    background-color: white;
    min-height: 400px;
    border-radius: 8px;
    padding: 2rem;
    box-shadow: 0 2px 15px rgba(0, 0, 0, 0.07);
    color: #4a6fa5;
    font-size: 1.1rem;
    display: flex;
    align-items: center;
    justify-content: center;
    text-align: center;
    border: 1px dashed #e1e4e8;
    transition: all 0.3s ease;
}

.main-content:hover {
    box-shadow: 0 5px 20px rgba(0, 0, 0, 0.1);
```

```
    border-color: #d1d5da;
}

/* 响应式设计 - 适配手机屏幕 */
@media (max-width: 768px) {
    .app {
        padding: 1rem;
    }

    h2 {
        font-size: 1.5rem;
        margin-bottom: 1.5rem;
    }

    .navigate {
        flex-direction: column;
        align-items: stretch;
        gap: 0.3rem;
    }

    .navigate a {
        text-align: center;
        padding: 0.7rem;
    }

    .main-content {
        min-height: 300px;
        padding: 1.5rem;
        font-size: 1rem;
    }
}
</style>
```

main.js

```
import './assets/main.css'
//引入createApp用于创建应用
import { createApp } from 'vue'
import { createPinia } from 'pinia'

```

```
//引入App根组件
import App from './App.vue'
//引入路由
import router from './router'

//创建应用
const app = createApp(App)

app.use(createPinia())
//使用路由
app.use(router)

//挂载应用
app.mount('#app')

```

index.js

```
import { createRouter, createWebHistory } from 'vue-router'
//引入一个一个可能要呈现的组件
import Home from '@/components/Home.vue'
import News from '@/components/News.vue'
import About from '@/components/About.vue'

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),//路由器的工作模式
  //创建路由
  routes: [
    {
      path: '/',
      component: Home
    },
    {
      path: '/home',
      component: Home
    },
    {
      path: '/news',
      component: News
    },
```

```
    {
      path: '/about',
      component: About
    }

  ]
})
//暴露出去router
export default router
```

2.两个注意点

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

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

```
import { createRouter, createWebHistory } from 'vue-router'
//引入一个一个可能要呈现的组件
import Home from '@/components/Home.vue'
import News from '@/components/News.vue'
import About from '@/components/About.vue'
import Detail from '@/components/Detail.vue'

const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),//路由器的工作模式
  //创建路由
  routes: [
    {
      path: '/',
      component: Home
    },
    {
      path: '/home',
      component: Home
    },
    {
      name: 'news',
      path: '/news',
      component: News,
```

```
      children: [
        {
          path: 'detail',
          component: Detail
        }]
    },
    {
      path: '/about',
      component: About
    }

  ]
})
//暴露出去router
export default router
```

7.路由_query参数

2. 接收参数:

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

Detail.vue

```
<template>
    <ul class="news-list">
        <li>编号:{{ query.id}}</li>
        <li>标题:{{ query.title}}</li>
        <li>内容:{{ query.content}}</li>
    </ul>``
</template>
<script setup >
    import {useRoute} from 'vue-router'
    import {toRefs} from 'vue'
    let route = useRoute()
    let {query} = toRefs(route)
</script>
<style scoped>
</style>
```

```
<template>
    <div class="news">
        <!-- 导航区 -->
        <ul>
            <li v-for="news in newsList" :key="news.id">
                <!-- 第一种写法 -->
                <!-- <RouterLink :to="`/news/detail?id=${news.id}&title=${news.title}&content=${news.content}`">{{news.title}}</RouterLink> -->
                 <!-- 第二种写法 -->
                <!-- <RouterLink :to="{path:'/news/detail',query:{id:news.id,title:news.title,content:news.content}}">{{news.title}}</RouterLink> -->
                <RouterLink :to="{name:'detail1',query:{id:news.id,title:news.title,content:news.content}}">{{news.title}}</RouterLink>
```

```
            </li>
        </ul>
        <!-- 展示区 -->
         <div class="news-content">
            <RouterView></RouterView>
         </div>
    </div>
</template>
<script setup name="news">
import {reactive} from 'vue'

const newsList = reactive([
    {id:'1001',title:'很好的抗癌食物',content:'西兰花'},
    {id:'1002',title:'如何一夜暴富',content:'学IT'},
    {id:'1003',title:'震惊,万万没想到',content:'明天是周一'},
    {id:'1004',title:'好消息,好消息',content:'快过年了'},
])
</script>
<style scoped>

</style>
```

8.路由_params参数

9.路由_props配置

10.replace属性

在导航区加入replace替换,就不可以查看历史记录,看完回不去

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：

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

```
<Header/>
<!-- 导航区 -->
<div class="navigate">
  <RouterLink replace to="/home" active-class="active">首页</RouterLink>
  <RouterLink replace :to="{name:'xinwen'}" active-class="active">新闻</Router
  <RouterLink replace :to="{path:'/about'}" active-class="active">关于</Router
</div>
```

11.路由_编程式路由导航

12.路由_重定向

重定向（Redirect） 指的是：当访问某个路径时，自动跳转到另一个指定的路径。它的作用是简化路由访问、统一入口或处理旧路径的兼容问题。

```
},
{
  path:'/',
  redirect:'/home'
}
]
```

13.Pinia

14.组件通信

1._props

2.自定义事件

ref定义的事件可以在模板里不写.value

1. 概述：自定义事件常用于：**子 => 父。**

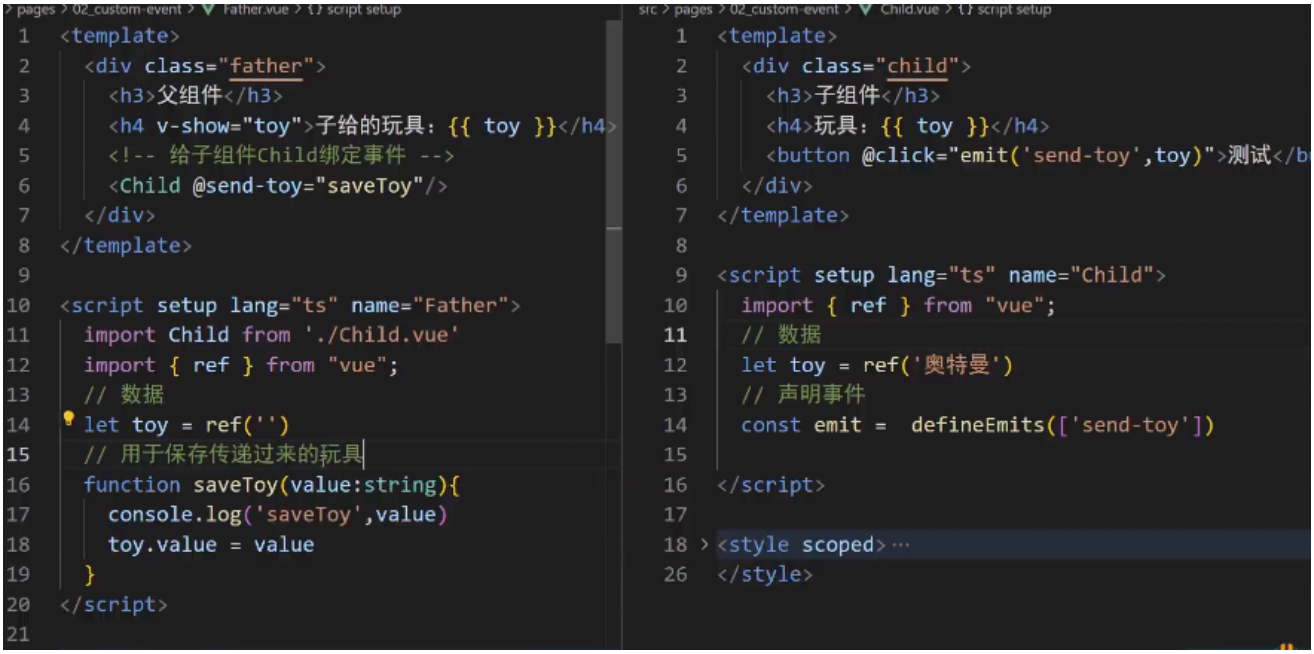
2. 注意区分好：原生事件、自定义事件。

- 原生事件：
  - 事件名是特定的（`click`、`mosueenter`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！！！

1. 示例：

```
<!--在父组件中，给子组件绑定自定义事件：-->
<Child @send-toy="toy = $event"/>

<!--注意区分原生事件与自定义事件中的$event-->
<button @click="toy = $event">测试</button>

```

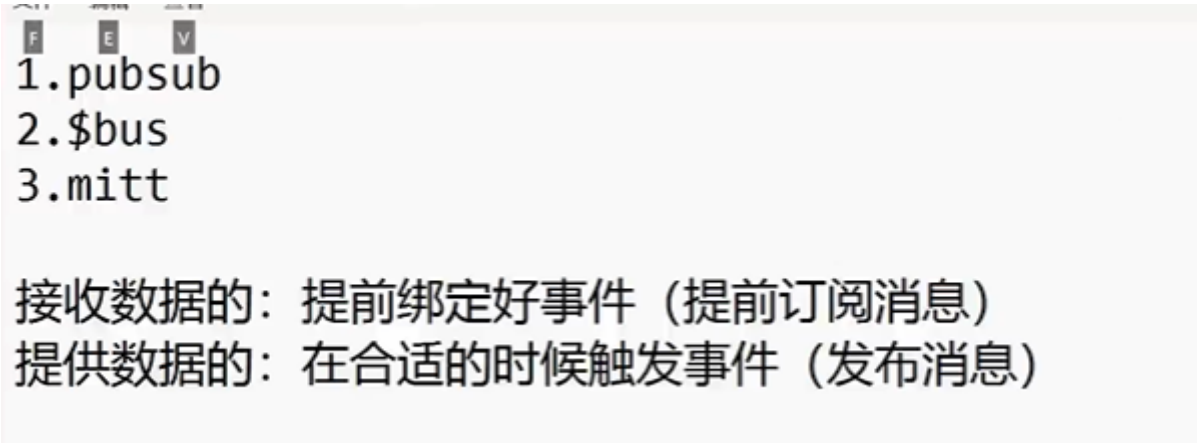
```
//子组件中，触发事件：
this.$emit('send-toy', 具体数据)

```



```
<template>
  <div class="father">
    <h3>父组件</h3>
    <h4 v-show="toy">子给的玩具：{{ toy }}</h4>
    <!-- 给子组件Child绑定事件 -->
    <Child @send-toy="saveToy"/>
  </div>
</template>

<script setup lang="ts" name="Father">
  import Child from './Child.vue'
  import { ref } from "vue";
  // 数据
  let toy = ref('')
  // 用于保存传递过来的玩具
  function saveToy(value:string){
    console.log('saveToy',value)
    toy.value = value
  }
</script>
```

```
<template>
  <div class="child">
    <h3>子组件</h3>
    <h4>玩具：{{ toy }}</h4>
    <button @click="emit('send-toy',toy)">测试</b
  </div>
</template>

<script setup lang="ts" name="Child">
  import { ref } from "vue";
  // 数据
  let toy = ref('奥特曼')
  // 声明事件
  const emit =  defineEmits(['send-toy'])

</script>

<style scoped>···
</style>
```

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`

```
npm i mitt
```

新建文件： `src\utils\emitter.ts`

```
// 引入mitt
import mitt from "mitt";

// 创建emitter
const emitter = mitt()

/*
  // 绑定事件
  emitter.on('abc',(value)=>{
    console.log('abc事件被触发',value)
  })
  emitter.on('xyz',(value)=>{
    console.log('xyz事件被触发',value)
  })

  setInterval(() => {
    // 触发事件
    emitter.emit('abc',666)
    emitter.emit('xyz',777)
  }, 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
30
```

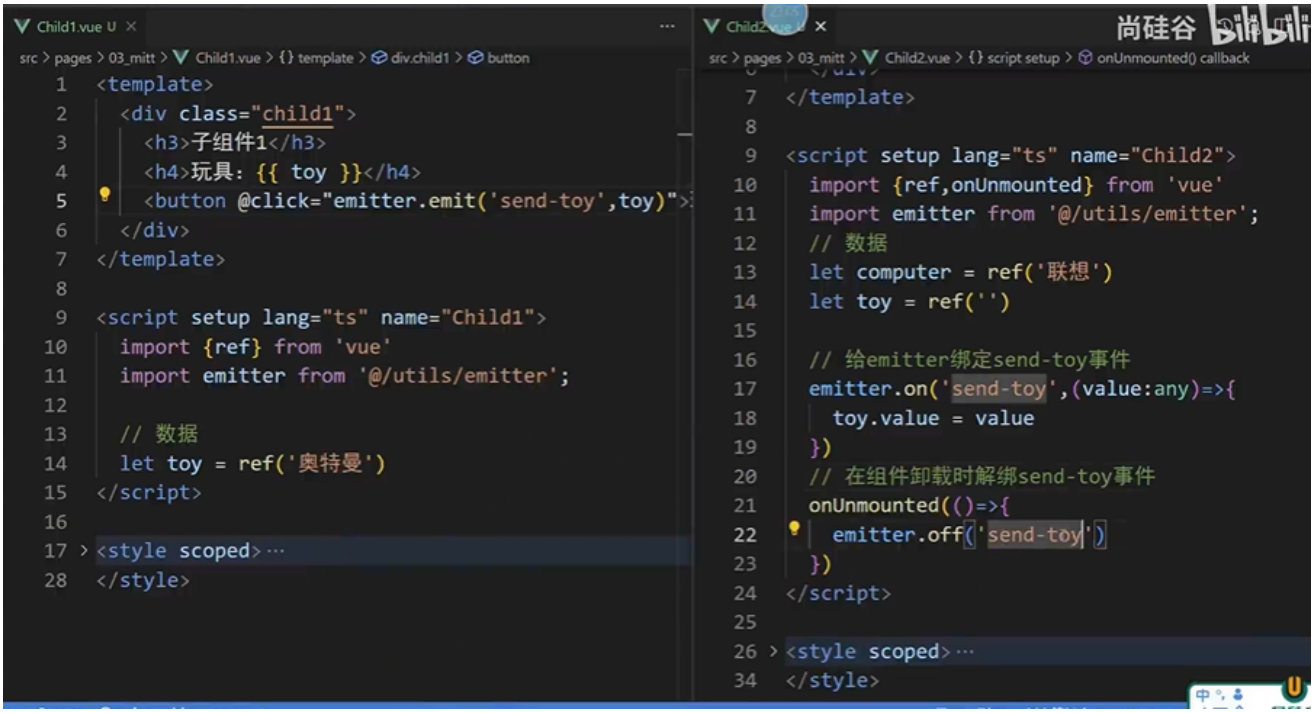
接收数据的组件中：绑定事件、同时在销毁前解绑事件：

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

注意这个重要的内置关系，总线依赖着这个内置关系



64.【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

```
<!-- 使用v-model指令 -->
<input type="text" v-model="userName">

<!-- v-model的本质是下面这行代码 -->
<input
  type="text"
  :value="userName"
  @input="userName =(<HTMLInputElement>$event.target).value"
>

```

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

```
<!-- 组件标签上使用v-model指令 -->
<AtguiguInput v-model="userName"/>

<!-- 组件标签上v-model的本质 -->
<AtguiguInput :modelValue="userName" @update:model-value="userName = $event"/>
```

```
6
```

`AtguiguInput` 组件中：

```
<template>
  <div class="box">
    <!--将接收的value值赋给input元素的value属性，目的是：为了呈现数据 -->
                <!--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-->
    <input
       type="text"
       :value="modelValue"
       @input="emit('update:model-value',$event.target.value)"
    >
  </div>
</template>

<script setup lang="ts" name="AtguiguInput">
  // 接收props
  defineProps(['modelValue'])
  // 声明事件
  const emit = defineEmits(['update:model-value'])
</script>

```

4. 也可以更换 `value` ，例如改成 `abc`

```
<!-- 也可以更换value，例如改成abc-->
<AtguiguInput v-model:abc="userName"/>

<!-- 上面代码的本质如下 -->
<AtguiguInput :abc="userName" @update:abc="userName = $event"/>

```

`AtguiguInput` 组件中：

```
<template>
```

```
  <div class="box">
    <input
       type="text"
       :value="abc"
       @input="emit('update:abc',$event.target.value)"
    >
  </div>
</template>

<script setup lang="ts" name="AtguiguInput">
  // 接收props
  defineProps(['abc'])
  // 声明事件
  const emit = defineEmits(['update:abc'])
</script>

```

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`

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

```

H2