# Vue3与vue2的一些区别

## main.js

### vue3

import { createApp } from 'vue'

import App from './App.vue'

import router from './router'

import store from './store'

// 简写形式

createApp(App).use(store).use(router).mount('#app')

// 完整写法

const app = createApp(App);

app.use(store);

app.use(router);

app.mount("#app");

### Vue2

import Vue from "vue";

import App from "./App.vue";

import router from "./router"; // 引入路由

import store from "./store"; // 引入vuex

new Vue({

  render: (h) => h(App),

  router,

  store

}).$mount("#app");

## 响应式原理的区别

### Vue2

#### 实现原理：

* + 对象类型：通过Object.defineProperty()对属性的读取、修改进行拦截（数据劫持）。
  + 数组类型：通过重写更新数组的一系列方法来实现拦截。（对数组的变更方法进行了包裹）。

Object.defineProperty(data, 'count', {  
   get () {},   
   set () {}  
})

#### 存在问题：

* + 新增属性、删除属性, 界面不会更新。
  + 直接通过下标修改数组, 界面不会自动更新。

#### 解决方案：

this.$set()或 Vue.set()

this.$delete() 或 Vue.delete()

#### 基本代码

let obj = {

      name: "张三",

      age: 12,

    }

    const ob = new Observer(obj);

    function Observer(obj) {

      let res = {}; // 创建一个对象

      const keys = Object.keys(obj);

      keys.forEach((key) => {

        // 将每一个obj的key值添加到新创建的res对象上

        Object.defineProperty(res, key, {

          configurable: true,

          get() {

            return obj[key];

          },

          set(val) {

            console.log("数据劫持");

            obj[key] = val;

          },

        });

      });

      return res; // 返回对象

    }

### Vue3

#### 实现原理:

- 通过Proxy（代理）: 拦截对象中任意属性的变化, 包括：属性值的读写、属性的添加、属性的删除等。

- 通过Reflect（反射）: 对源对象的属性进行操作。

#### 基本代码

let obj = {

    name: "张三",

    age: 12,

};

const res = new Proxy(obj, {

      // 获取

      get(target, propName) {

        console.log(`获取了obj下的${propName}`);

        return Reflect.set(target, propName);

      },

      // 增加与修改

      set(target, propName, value) {

        console.log(`操作了obj下的${propName}`);

        Reflect.set(target, propName, value);

      },

      // 删除

      deleteProperty(target, propName) {

        console.log(`删除了obj下的${propName}`);

        return Reflect.deleteProperty(target, propName);

      },

    });

# 异步引入方式

异步引入：组件是先后出现的

先后出现就会有可能导致某个组件过晚出现。

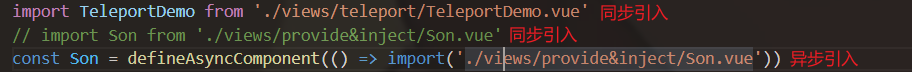
此时可以使用Suspense定义一些组件没出现前的加载提示

[Suspense](#_Suspense)

同步引入：组件是一起出现的

import {defineAsyncComponent} from “vue”

const Child = defineAsyncComponent(()=>import(“./components/Child”)) // 异步引入





# Composition API

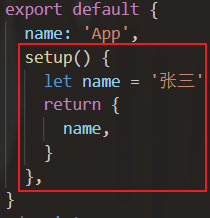
## 用到哪个接口就要引入该接口

import {ref} from 'vue'

## setup入口函数

所有的组合API函数都在此使用, 只在初始化时执行一次

### 基本使用



### setup返回值

若返回一个对象，则对象中的属性、方法, 在模板中均可以直接使用。（重点关注！）

若返回一个渲染函数：则可以自定义渲染内容。（了解）

会覆盖你模板书写好的结构

import {h} from “vue”

setup(){

return ()=> h(“h1”,”xxx”)

}

### 注意点：

#### 1、尽量不要与Vue2.x配置混用

Vue2.x配置（data、methos、computed...）中**可以访问到**setup中的属性、方法。

但在setup中**不能访问到**Vue2.x配置（data、methos、computed...）。

如果有重名, setup优先。

#### 2、setup不能是一个async函数

因为返回值不再是return的对象, 而是promise, 模板看不到return对象中的属性。

（[***后期也可以返回一个Promise实例，但需要Suspense和异步组件的配合***](#_Suspense)）

## setup细节

### setup执行的时机

在beforeCreate之前执行(一次), 此时组件对象还没有创建

this是undefined, 不能通过this来访问data/computed/methods / props

其实所有的composition API相关回调函数中也都不可以

### setup的返回值

一般都返回一个对象: 为模板提供数据, 也就是模板中可以直接使用此对象中的所有属性/方法

返回对象中的属性会与data函数返回对象的属性合并成为组件对象的属性

返回对象中的方法会与methods中的方法合并成为组件对象的方法

如果有重名, setup优先

### setup的参数

setup(props, context) / setup(props, {attrs, slots, emit})

props: 包含props配置声明且传入了的所有属性的对象

attrs: 包含没有在props配置中声明的属性的对象, 相当于 this.$attrs

slots: 包含所有传入的插槽内容的对象, 相当于 this.$slots

emit: 用来分发自定义事件的函数, 相当于 this.$emit

### 注意

#### 一般不要混合使用:

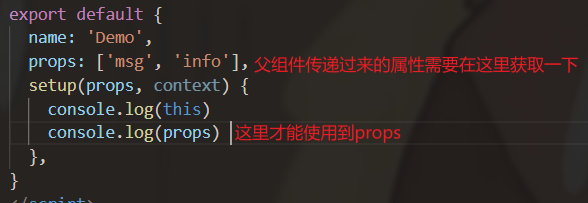
methods中可以访问setup提供的属性和方法, 但在setup方法中不能访问data和methods

#### setup不能是一个async函数:

因为返回值不再是return的对象, 而是promise, 模板看不到return对象中的属性数据

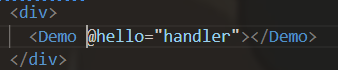
#### props

父组件传递给子组件的属性,子组件需要使用props接收，才能在setup(props)的参数中访问到。如果没有接收默认都跑到***context.attrs***上了

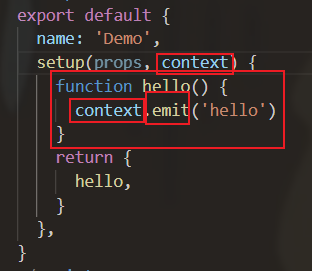


#### 触发自定义事件context.emit

父组件

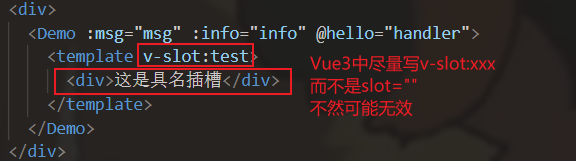


子组件



#### 插槽context.slot

父组件



子组件



## ref

### 作用

定义一个数据的响应式

### 语法

const xxx = ref(initValue) // 这里使用了const，就是因为ref()返回的是一个对象,

在const声明的对象中修改属性是允许的

* 创建一个包含响应式数据的**引用对象（reference对象，简称ref对象）**。
* JS中操作数据： xxx.value
* 模板中读取数据: 不需要.value，直接：<div>{{xxx}}</div>

一般用来定义一个基本类型的响应式数据

* 接收的数据可以是：基本类型、也可以是对象类型。
* 基本类型的数据：响应式依然是靠Object.defineProperty()的get与set完成的。
* 对象类型的数据：内部 ***“ 求助 ”*** 了Vue3.0中的一个新函数—— reactive函数。

### 基本使用

<!-- 模板自动添加了.value不需要我们手动写上 -->

    <div>{{name}}</div>

    <button @click="changeName">切换</button>

setup() {

   const name = ref('张三') // 传入基本数据类型

const obj = ref({ // 传入引用数据类型

name: '李四',

age: 12

})

    function changeName() {

      name.value = '李四'

    }

    return {

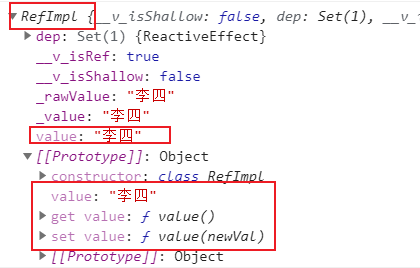
      name,

      changeName,

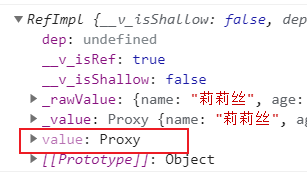
    }

  },

#### ref(“xxx”)的返回值就是一个RefImpl构造函数创建的实例对象



#### 如果const obj = ref({ name: ”莉莉丝”, age: 22 })里面传入的是一个对象, obj.value是一个Proxy代理对象

## reactive

### 作用

定义一个对象类型的响应式数据（基本类型不要用它，要用```ref```函数）

### 语法

```const 代理对象= reactive(源对象)```接收一个对象（或数组），

返回一个代理对象（Proxy的实例对象，简称proxy对象

reactive定义的响应式数据是“深层次的”。

内部基于 ES6 的 Proxy 实现，通过代理对象操作源对象内部数据进行操作。

### 基本使用

import {reactive} from 'vue'

export default {

  name: 'App',

  setup() {

    //obj是代理对象

    const obj = reactive({

      name: '张三',

      age: 12,

})

    function changeName() {

      console.log(obj)

      obj.name = '李四'

    }

    return {

      obj,

      changeName,

    }

  },

}

## ref和reactive对比

* 从定义数据角度对比：
  + ref***一般***用来定义：**基本类型数据**。
  + reactive用来定义：**对象（或数组）类型数据**。定义基本数据类型就不是响应式的
  + 备注：ref也可以用来定义**对象（或数组）类型数据**, 它内部会自动通过reactive转为**代理对象**。
* 从原理角度对比：
  + ref通过Object.defineProperty()的get与set来实现响应式（数据劫持）。
  + reactive通过使用**Proxy**来实现响应式（数据劫持）, 并通过**Reflect**操作**源对象**内部的数据。
* 从使用角度对比：
  + ref定义的数据：操作数据**需要**.value，读取数据时模板中直接读取**不需要**.value。
  + reactive定义的数据：操作数据与读取数据：**均不需要**.value。

## 计算属性computed

与vue2差不多

import {computed} from 'vue'

setup(){

...

//计算属性——简写

let fullName = computed(()=>{

return person.firstName + '-' + person.lastName

})

//计算属性——完整

let fullName = computed({

get(){

return person.firstName + '-' + person.lastName

},

set(value){

const nameArr = value.split('-')

person.firstName = nameArr[0]

person.lastName = nameArr[1]

}

})

}

## 监听属性watch

### // 函数形式-单个属性监听-ref所定义的响应式数据

    watch(num, (newVal, oldVal) => {

      console.log(`num数据发生了改变,${newVal},${oldVal}`)

    })

### // 函数形式-多个属性监听-ref所定义的响应式数据

    watch([num, text], (newVal, oldVal) => {

      console.log(`num数据发生了改变,${newVal},${oldVal}`)

})

### // 对象形式监听-单个属性监听-ref所定义的响应式数据

    watch(text,(newVal, oldVal) => {

        console.log(`text数据发生了改变,${newVal},${oldVal}`)

      },

      {immediate: true, deep: true}

)

### // 监听reactive声明的响应式数据

// 监视reactive定义的响应式数据时：oldValue无法正确获取(获取的值和newValue一样)、强制开启了深度监视（deep配置失效）。

    watch(person, (newVal, oldVal) => {

      console.log('人物数据发生改变:' + newVal + '  ' + oldVal)

})

### // 监视reactive定义的响应式数据中【某个属性】时：deep配置有效。

// oldValue依然无法获取正确值

    watch(() => person.wife,

      (newValue, oldValue) => {

        console.log('发生改变：' + newValue.name + '，' + oldValue.name)

      },

      {deep: true}

)

### // 监听reactive定义的响应式数据中的【多个属性】

    watch([() => person.age, () => person.income], (newVal, oldVal) => {

      console.log('多个属性发生改变：' + newVal + ',' + oldVal)

})

### // 特殊情况

[152\_尚硅谷Vue3技术\_watch监视reactive定义的数据\_哔哩哔哩\_bilibili](https://www.bilibili.com/video/BV1Zy4y1K7SH?p=152&spm_id_from=pageDriver&vd_source=a67e813436aadc4608992466bdaea6bc)

记住对象内的第二层属性内的属性发生改变，要想检测到，要加deep

    watch(

      () => person.wife,

      (newVal, oldVal) => {

        console.log('第二层属性发生改变：' + newVal + ',' + oldVal)

      },

      {deep: true}

    )

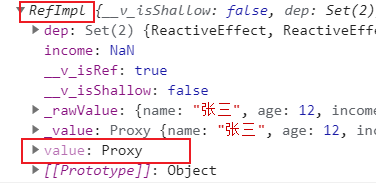
### // 监听ref声明的对象响应式数据

第一种方式：需要加上一个.value

第二种方式：加上深度监听 deep:true

因为通过ref声明的数据结构是下图所示。value里面是一个Proxy类型的对象,我们如果直接监听refImpl对象，那么value里面的整个对象没有发生改变，就不会被监听到。

此时就可以加上深度监听或者直接监听value所指的那个proxy对象



ref对传入的对象进行响应式处理就是通过reactive进行处理的

const person = ref() // 根据ref的特性，此时的person.value才是我们需要的响应式对象

[如果const obj = ref({ name: ”莉莉丝”, age: 22 })里面传入的是一个对象, obj.value是一个Proxy代理对象](#_如果const_obj_=)

const person = ref({

      name: '张三',

      age: 12,

      income: 10,

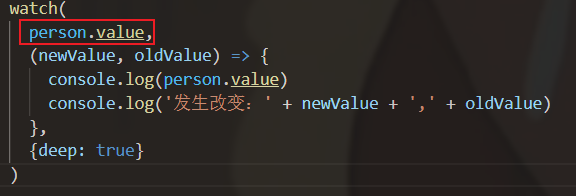
      wife: {

        name: 'ziqi',

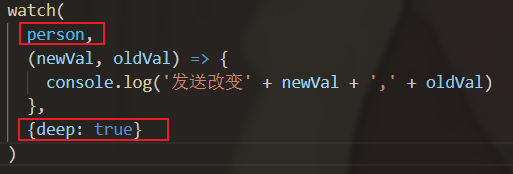
      },

    })

第一种



第二种



### 两个小坑

* 监视reactive定义的响应式数据时：oldValue无法正确获取、强制开启了深度监视（deep配置失效）。
* 监视reactive定义的响应式数据中某个属性时：deep配置有效。

### 总结

#### 监视ref声明的响应式数据

ref一般都是对基本数据类型进行监听，正常使用即可

ref虽然可以传入对象作为参数，但是对对象的响应式处理依然是通过reactive进行

#### 监视reactive声明的响应式数据

如果对整个对象进行监视有以下两个特点：

·oldValue无法正确获取

·默认开始deep:true 无法取消

·如果对象中的属性是基本数据类型的，那么对该数据进行监听，oldValue是正常的

可以理解为对【对象】进行监听oldValue无效,对基本数据类型则不会

对对象中的属性进行监视(此时的deep已正常,oldValue依然不正常)

·如果对象的属性是一个【对象】，那么需要【开启深度监视】

·如果对象的属性是【基本数据类型】，那么【不需要开启】

## watchEffect

### 基本使用

 watchEffect(() => {

      console.log('新资变化了', person.income)

    })

//watchEffect所指定的回调中用到的数据只要发生变化，则直接重新执行回调。

watchEffect(()=>{

const x1 = sum.value

const x2 = person.age

console.log('watchEffect配置的回调执行了')

})

### 特点

watch的套路是：

既要指明监视的属性，也要指明监视的回调。

**watchEffect的套路是**：

不用指明监视哪个属性，监视的回调中用到哪个属性，那就监视哪个属性。

watchEffect有点像computed：

但computed注重的计算出来的值（回调函数的返回值），所以必须要写返回值。

而watchEffect更注重的是过程（回调函数的函数体），所以不用写返回值。

## toRef

### - 作用：

创建一个 ref 对象，其value值指向另一个对象中的某个属性。

### - 语法：

```const name = toRef(person,'name')```

### - 应用:

要将响应式对象中的某个属性单独提供给外部使用时。

### 基本使用

<div>

    <div>{{person}}</div>

    <div>姓名：{{name}}</div>

    <div>年龄：{{age}}</div>

    <div>收入：{{income}}</div>

    <button @click="name+='!'">修改姓名</button>

    <button @click="age+=1">修改年龄</button>

    <button @click="income+=100">修改收入</button>

  </div>

#### 错误写法1

下面的标红代码，无法对拆分出来的数据进行响应式处理

setup() {

    const person = reactive({

      name: 'Zs',

      age: 12,

      income: 3000,

    })

    return {

      person,

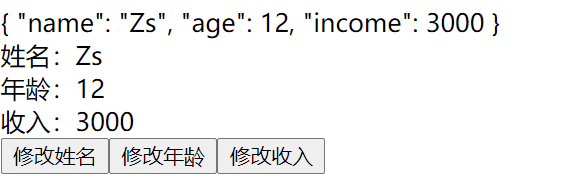
    name: person.name, // 等同于将zs赋值给了name新变量

      age: person.age, // 等同于将12赋值给了age这个新变量

      income: person.income, // 等同于将3000赋值给了income这个新变量

    }

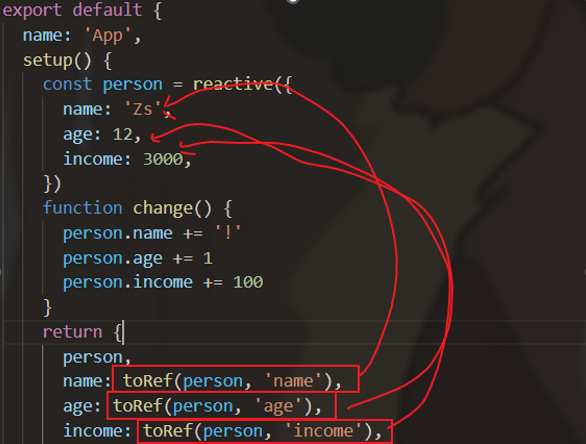
  },

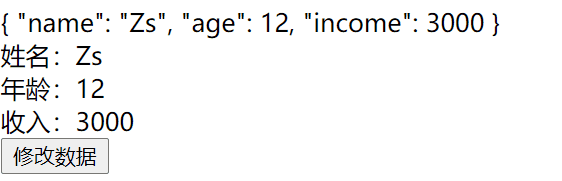


#### 正确的使用姿势

toRef可以将拆分出来的数据响应式处理

原理其实就是引用person这个Proxy对象上的属性





#### 错误写法2

setup() {

    const person = reactive({

      name: 'Zs',

      age: 12,

      income: 3000,

    })

    return {

      person,

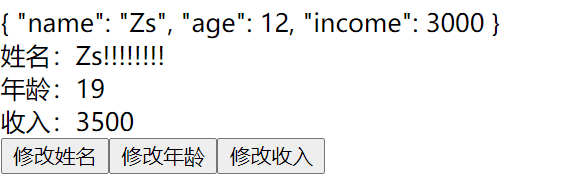
      name: ref(person.name), // name是一个新的响应式数据，和person没有关系

      age: ref(person.age), // age是一个新的响应式数据，和person没有关系

      income: ref(person.income), // income是一个新的响应式数据，和person没有关系

    }

  },

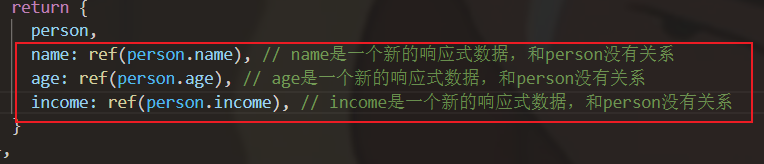


## toRefs

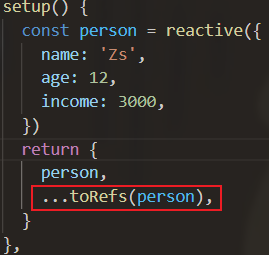
```toRefs``` 与```toRef```功能一致，但可以批量创建多个 ref 对象，

语法：```toRefs(person)```

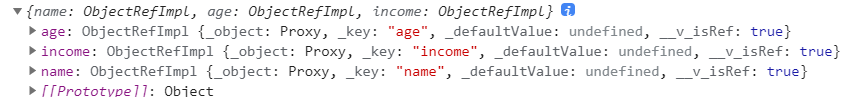
如果一个对象有多个属性，通过toRef写法就不是那么简便了



此时可以通过toRefs处理



console.log(toRefs(person))⬇



## shallowReactive 与 shallowRef

- shallowReactive：只处理对象最外层属性的响应式（浅响应式）。

- shallowRef：只处理基本数据类型的响应式, 不进行对象的响应式处理。

- 什么时候使用?

- 如果有一个对象数据，结构比较深, 但变化时只是外层属性变化 ===> shallowReactive。

shallowReactive({

xxx:Xxx,

xxx:{xxx:Xxx}

})

- 如果有一个对象数据，后续功能不会修改该对象中的属性，而是生新的对象来替换 ===> shallowRef。

shallowRef({xx:xxx,xxx:Xxx})

## readonly 与 shallowReadonly

readonly: 让一个响应式数据变为只读的（深只读）。

shallowReadonly：让一个响应式数据变为只读的（浅只读）。

### 应用场景:

不希望数据被修改时。

### 基本使用

shallowReadonly：只有第一层是只读，里面的层级都是可以读写的

setup() {

    let person = reactive({

      name: 'Zs',

      age: 12,

      income: 3000,

      wife: {

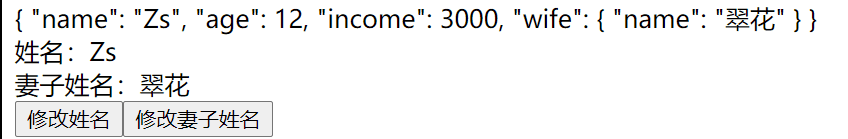
        name: '翠花',

      },

    })

person = shallowReadonly(person)

}



readonly：整个对象都是只读的

setup() {

    let person = reactive({

      name: 'Zs',

      age: 12,

      income: 3000,

      wife: {

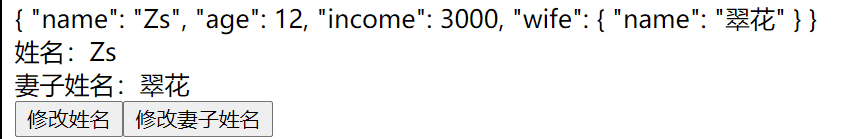
        name: '翠花',

      },

    })

person = readonly(person)

}



我写了个响应式的数据，结果又转为了只读的。只读的肯定就不会触发页面更新了

那么我还用啥响应式，一开始就别响应不就好了？

有可能这个数据是其他地方传递过来的。那么这个数据是不希望被其他组件所修改。

所以可以让接收该数据的组件，使用readonly设置为只读

## toRaw 与 markRaw

### toRaw：

#### 作用：

返回由 reactive 或 readonly 方法转换成响应式代理的普通对象

#### 使用场景：

用于读取响应式对象对应的普通对象，对这个普通对象的所有操作，不会引起页面更新。

#### 基本使用

function toOrigin() {

      let p = toRaw(person) // 转换普通对象

      console.log(p) // 普通对象

      console.log(person) // Proxy代理对象

    }



### markRaw：

#### 作用：

标记一个对象，使其永远不会再成为响应式对象。

#### 应用场景:

1. 有些值不应被设置为响应式的，例如复杂的第三方类库等。

2. 当渲染具有不可变数据源的大列表时，跳过响应式转换可以提高性能。

#### 基本使用

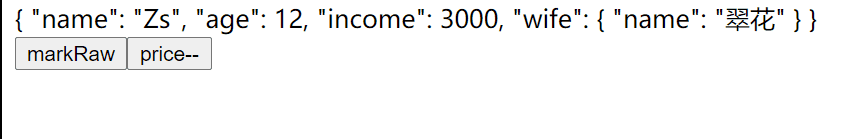
function toOrigin() {

      person.car = markRaw({brand: 'bezi', price: 12000})

// person.car这个对象不再是响应式的

      console.log(person.car) 

    }



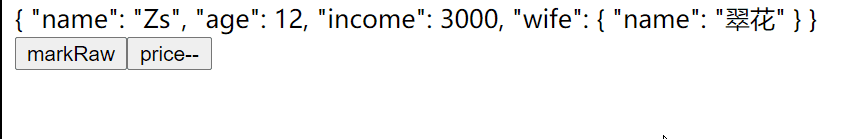
function toOrigin() {

      person.car = {brand: 'bezi', price: 12000}

// person.car是响应式对象

      console.log(person.car) 

    }



## customRef(难懂)

创建一个自定义的 ref，并对其依赖项跟踪和更新触发进行显式控制

[161\_尚硅谷Vue3技术\_customRef\_哔哩哔哩\_bilibili](https://www.bilibili.com/video/BV1Zy4y1K7SH?p=161&spm_id_from=pageDriver&vd_source=a67e813436aadc4608992466bdaea6bc)

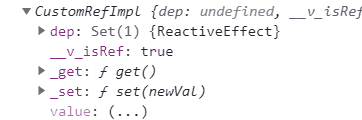
### 基本使用

<input type="text" v-model="keywords">

    <div>{{keywords}}</div>

setup() {

    const keywords = myRefs('')

    console.log(keywords) 

    // 自定义myRefs函数

    function myRefs(value) {

      return customRef((track, trigger) => {

        // 返回一个对象

        return {

          get() {

            console.log(`有人读取了myRefs这个容器中的数据:${value}`)

            track() // 通知Vue追踪value的变化

            return value

          },

          set(newVal) {

            console.log(`有人将myRefs这个容器中的数据改为了:${newVal}`)

            value = newVal

            trigger() // 通知Vue重新解析模板

          },

        }

      })

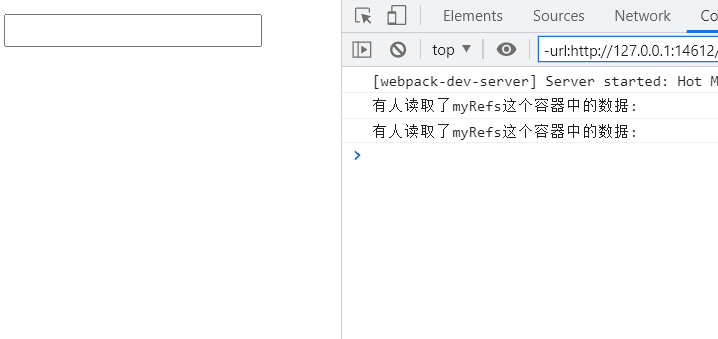
    }

    return {

      keywords,

    }

有两个Dom引用了keywords，所以会出现两句读取的信息



### 应用：防抖

setup() {

    const keywords = myRefs('', 500)

    // 自定义myRefs函数

    function myRefs(value, delay) {

      let timer = null // 定时器

      return customRef((track, trigger) => {

        // 返回一个对象

        return {

          get() {

            track() // 通知Vue追踪value的变化

            return value

          },

          set(newVal) {

            clearTimeout(timer)

            timer = setTimeout(() => {

              value = newVal

              trigger() // 通知Vue重新解析模板

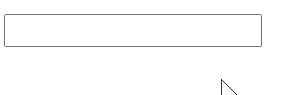
            }, delay)

          },

        }

      })

    }



### customRef工作原理

[161\_尚硅谷Vue3技术\_customRef\_哔哩哔哩\_bilibili](https://www.bilibili.com/video/BV1Zy4y1K7SH?p=161&spm_id_from=pageDriver&vd_source=a67e813436aadc4608992466bdaea6bc)

## provide和inject

### 传递非响应式数据

祖组件传递数据

setup() {

    // 非响应式数据

    const person = {

      name: '颤三',

      age: 12,

    }

    provide('person', person)

  },

后代组件接收数据

setup() {

    let person = inject('person')

    return {

      person,

    }

  },



### 传递响应式数据

祖组件传递数据

setup() {

    // 响应式数据

    const person = reactive({

      name: '颤三',

      age: 12,

    })

    provide('person', person)

  },

后代组件接收数据

setup() {

    let person = inject('person')

    return {

      person,

    }

  },



## 响应式数据的判断

isRef(): 检查一个值是否为一个 ref 对象

isReactive(): 检查一个对象是否是由 reactive 创建的响应式代理

isReadonly(): 检查一个对象是否是由 readonly 创建的只读代理

isProxy(): 检查一个对象是否是由 reactive 或者 readonly 方法创建的代理

# 新的组件

## Fragment

在Vue2中: 组件必须有一个根标签

在Vue3中: 组件可以没有根标签, 内部会将多个标签包含在一个Fragment虚拟元素中

好处: 减少标签层级, 减小内存占用

## 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。

<teleport to="移动位置">

<div v-if="isShow" class="mask">

<div class="dialog">

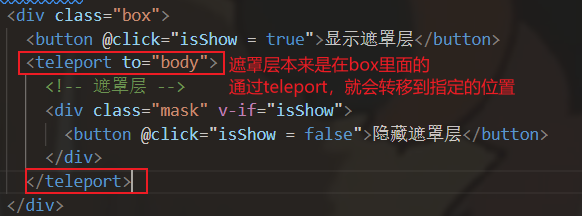
<h3>我是一个弹窗</h3>

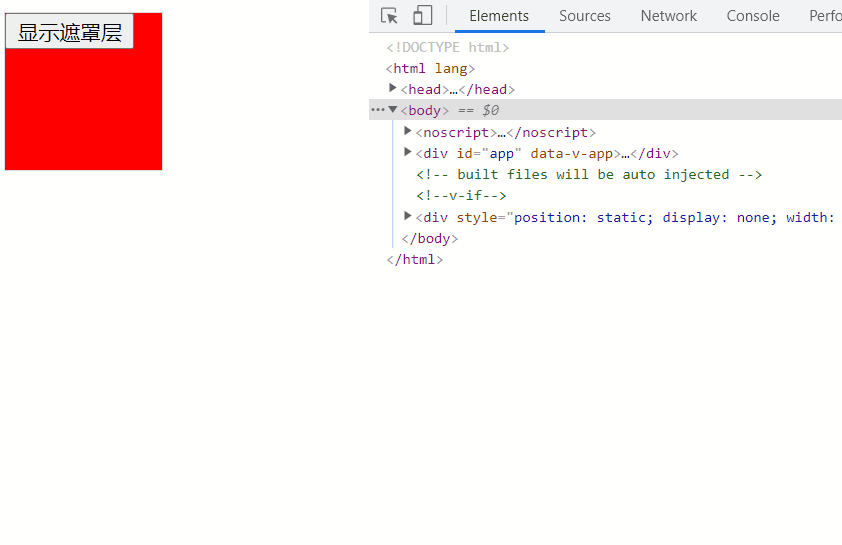
<button @click="isShow = false">关闭弹窗</button>

</div>

</div>

</teleport>





## 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

<!-- 同步引入的组件 -->

    <TeleportDemo></TeleportDemo>

    <Suspense>

      <!-- 显示组件 -->

      <template v-slot:default><!-- 插槽名固定，不能修改 -->

        <Son></Son><!-- 异步引入的组件 -->

      </template>

      <!-- 组件未显示时的加载动画 -->

      <template v-slot:fallback>

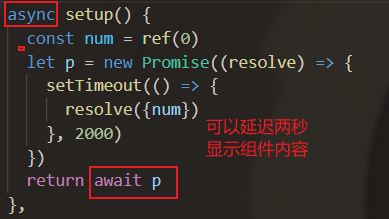
        <h1>加载中。。。</h1>

      </template>

    </Suspense>



和Suspense配合，setup可以是异步的函数(async setup(){})



# Composition API 的优势

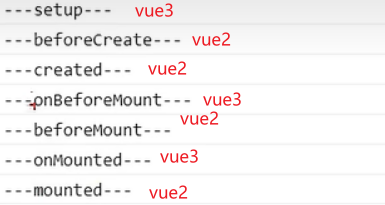
[optionsAPI与compositionAPI\_高级白嫖工程师的博客-CSDN博客](https://blog.csdn.net/weixin_45701125/article/details/122579236)

查百度去

# 生命周期

同时使用vue2和vue3的生命周期

vue2的生命周期优先级比vue3低



## Vue3.0中可以继续使用Vue2.x中的生命周期钩子，但有有两个被更名：

beforeDestroy改名为 beforeUnmount

destroyed改名为 unmounted

## Vue3.0也提供了 Composition API 形式的生命周期钩子，与Vue2.x中钩子对应关系如下：

beforeCreate===>setup()

created=======>setup()

beforeMount ===>onBeforeMount(()={})

mounted=======>onMounted(()={})

beforeUpdate===>onBeforeUpdate(()={})

updated =======>onUpdated(()={})

beforeUnmount ==>onBeforeUnmount(()={})

unmounted =====>onUnmounted(()={})

setup(){

onBeforeMount(() => {

      console.log('onBeforeMount')

    })

    onMounted(() => {

      console.log('onMounted')

    })

    onBeforeUpdate(() => {

      console.log('onBeforeUpdate')

    })

    onUpdated(() => {

      console.log('onUpdated')

    })

    onBeforeUnmount(() => {

      console.log('onBeforeUnmount')

    })

    onUnmounted(() => {

      console.log('onUnmounted')

})

}

# hook函数

## 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。

## 基本使用

### 一般都使用useXxxx做名字，导入的时候也是使用useXxxx

#### hook/usePoint.js

import { onMounted, onUnmounted, reactive } from "vue";

export default function () {

  const point = reactive({

    x: 0,

    y: 0,

  });

  function getPoint(e) {

    point.x = e.pageX;

    point.y = e.pageY;

  }

  onMounted(() => {

    window.addEventListener("click", getPoint);

  });

  onUnmounted(() => {

    window.removeEventListener("click", getPoint);

  });

  return point;

}

#### 组件

<script>

import usePoint from '@/hook/usePoint'

export default {

  name: 'UseHook',

  setup() {

    const point = usePoint()

    return {

      point,

    }

  },

}

</script>

# Vue3的调整

## 全局Api的转移

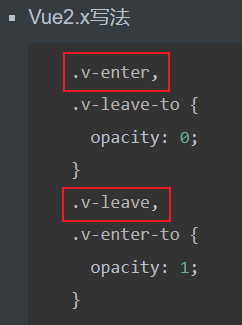
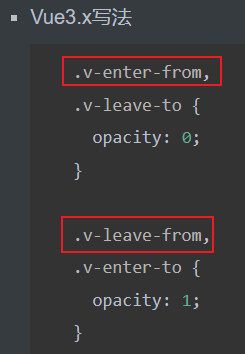




## 其他改变

### data选项应始终被声明为一个函数。

### 过度类名的更改

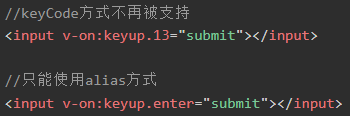
### 移除$on，$off and $once $children

上述三个方法被认为不应该由vue来提供，因此被移除了，可以使用其他第三方库来实现mitt实现中央事件总线(bus)

### $destory实例方法已删除

**用户不应再手动管理单个 Vue 组件的生命周期。**

### 移除keyCode作为 v-on 的修饰符，同时也不再支持config.keyCodes



### 移除```v-on.native```修饰符

#### .native修饰符

在子组件上绑定@click事件，会默认判定为自定义事件。

此时点击子组件是没有对应的响应的。

需要给click事件添加事件修饰符native，来告诉子组件，这是Vue原生的click事件

Vue3用法⬇



 emits没有声明的事件名，就默认是Vue原生的事件

### 移除过滤器（filter）

过滤器虽然这看起来很方便，但它需要一个自定义语法，打破大括号内表达式是 “只是 JavaScript” 的假设，这不仅有学习成本，而且有实现成本！建议用方法调用或计算属性去替换过滤器。

。。。。

