



可组合的 Vue

Composable Vue, 编写可组合可复用的 Vue 函数的最佳实践与技巧

ANTHONY FU

Hangzhou, China 2021

Anthony Fu

Vue 核心成员 / Vite 团队成员

VueUse, Slidev, Type Challenges 等项目创作者

全职开源



🐱 antfu

🐦 antfu7

知乎 Anthony Fu

gatsby antfu.me

Gold Sponsors



Leniolabs_ Nuxt Vue Mastery Evan You

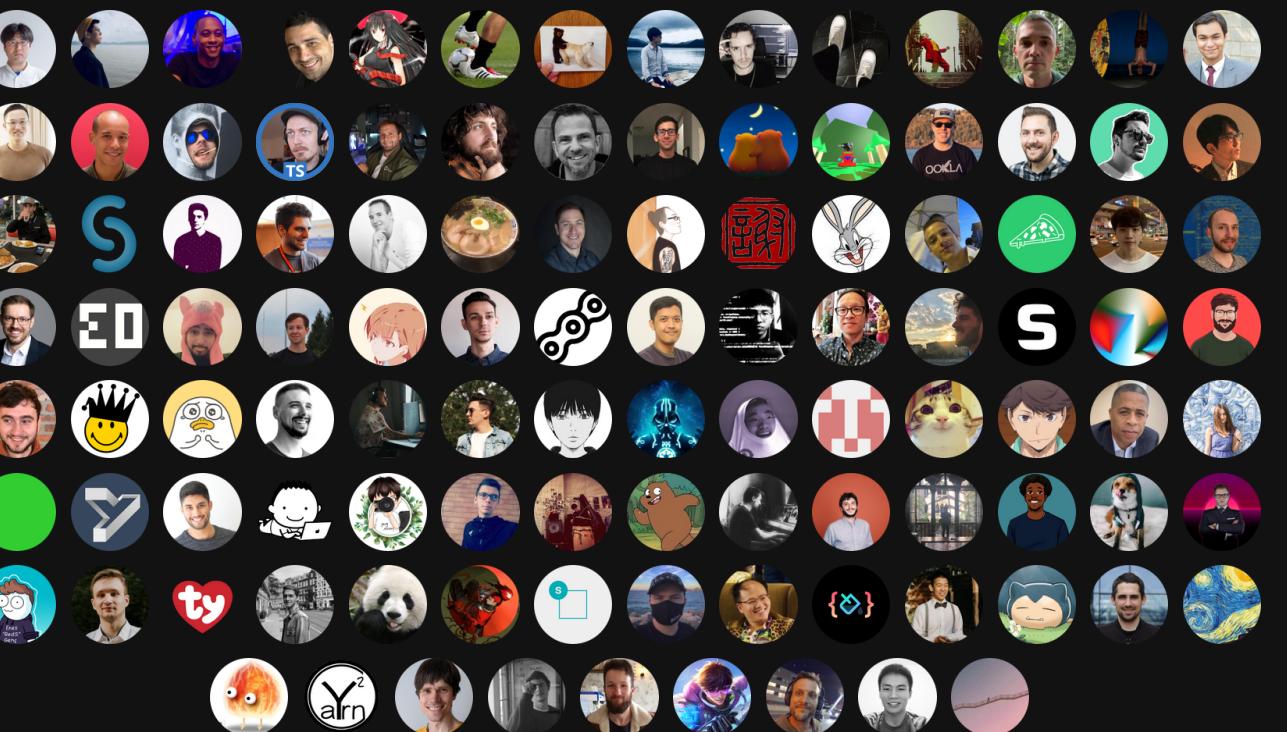
Sponsors



IU hiroki osame Hunter Liu Ben Hong PENG Rui 瑰致远 Jan-Henrik Fall Deadpool

Johann

Backers



在 GitHub 上赞助我

Vue Composition API

组合式 API

什么是组合式 API?

在 Vue 3 中引入的一种新的编写 Vue 组件的方式。

```
<script>
export default {
  data() {
    return {
      dark: false
    },
  },
  computed: {
    light() {
      return !this.dark
    }
  },
  methods: {
    toggleDark() {
      this.dark = !this.dark
    }
  }
}
</script>
```

```
<script>
import { ref, computed } from 'vue'

export default {
  setup() {
    const dark = ref(false)
    const light = computed(() => !dark.value)

    return {
      dark,
      light,
      toggleDark() {
        dark.value = !dark.value
      }
    }
  }
}
</script>
```

为什么引入组合式 API？

对象式 API 存在的问题

- 不利于复用
- 潜在命名冲突
- 上下文丢失
- 有限的类型支持
- 按 API 类型组织

组合式 API 提供的能力

- 极易复用 (原生 JS 函数)
- 可灵活组合 (生命周期钩子可多次使用)
- 提供更好的上下文支持
- 更好的 TypeScript 类型支持
- 按功能/逻辑组织
- 可独立于 Vue 组件使用

什么是可组合的函数

可复用逻辑的集合，专注点分离

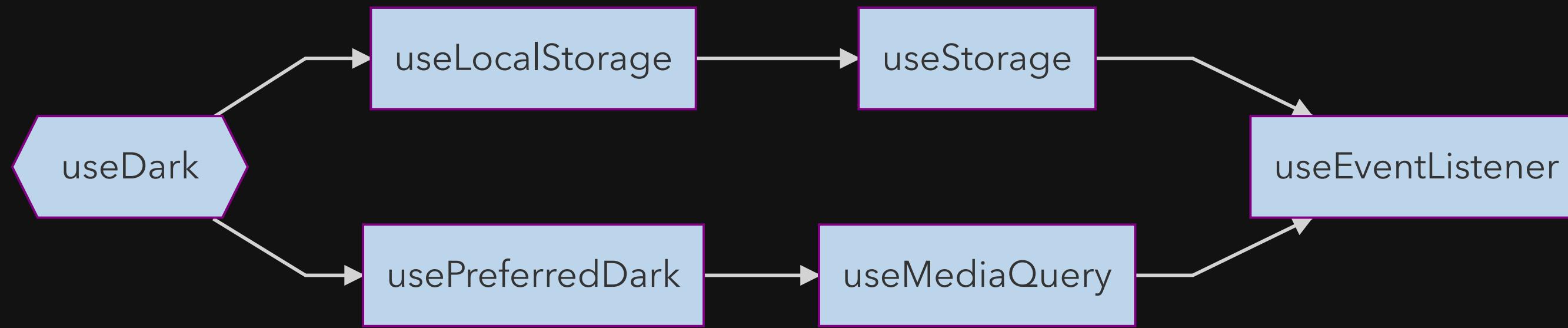
```
export function useDark(options: UseDarkOptions = {}) {
  const preferredDark = usePreferredDark() // <--
  const store = useLocalStorage('vueuse-dark', 'auto') // <--

  return computed<boolean>({
    get() {
      return store.value === 'auto'
        ? preferredDark.value
        : store.value === 'dark'
    },
    set(v) {
      store.value = v === preferredDark.value
        ? 'auto' : v ? 'dark' : 'light'
    },
  })
}
```



U 在 VueUse 中可用: `usePreferredDark` `useLocalStorage` `useDark`

组合关系



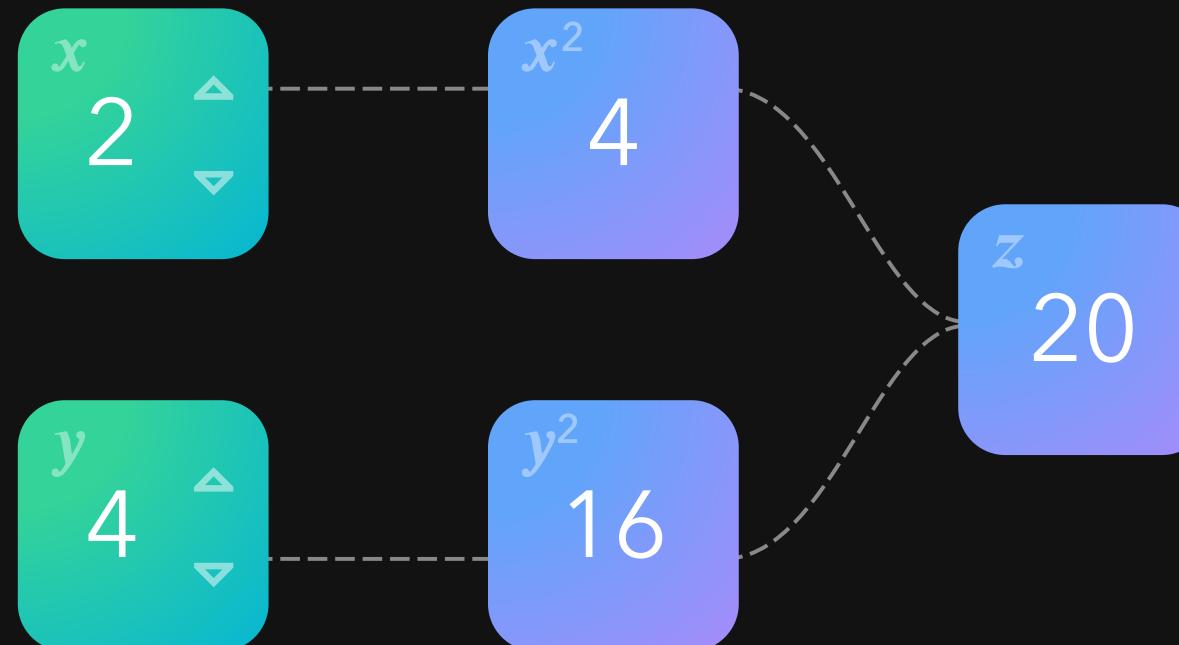
- 其中每一个函数都可以独立使用
- 专注点分离

建立"连结" 模式

不同于 React, Vue 的 `setup()` 只会在组件建立时执行一次，并建立数据与逻辑之间的连结。

- 建立 输入 → 输出 的连结
- 输出会自动根据输入的改变而改变

$$z = x^2 + y^2 = 2 \times 2 + 4 \times 4 = 20$$



EXCEL 中的公式

A screenshot of an Excel spreadsheet titled 'SUM'. The formula bar shows '=C2/B2'. The spreadsheet has four columns: Student, Total Marks, Achieved Marks, and Percentage. Row 1 contains column headers: Student, Total Marks, Achieved Marks, and Percentage. Rows 2 through 8 contain student data: Ramu (Total Marks 600, Achieved Marks 490, Percentage =C2/B2), Rajitha (600, 483, blank), Komala (600, 448, blank), Patil (600, 530, blank), Pursi (600, 542, blank), and Gayathri (600, 578, blank). The formula in the Percentage column is highlighted with a red border.

	Student	Total Marks	Achieved Marks	Percentage
1				
2	Ramu	600	490	=C2/B2
3	Rajitha	600	483	
4	Komala	600	448	
5	Patil	600	530	
6	Pursi	600	542	
7	Gayathri	600	578	
8				

模式和技巧

编写可复用，可组合的逻辑



VueUse

Vue 组合式 API 工具包

v4.11.0

85k/month

docs & demos

114 functions



Stars



4k

同时兼容 Vue 2 和 Vue 3

Tree-shakeable ESM

TypeScript

CDN 兼容

核心包含 110+ 组合式函数

丰富的生态系统 8+ 扩展包

Ref

```
import { ref } from 'vue'

let foo = 0
let bar = ref(0)

foo = 1
bar = 1 // ts-error
```

PROS

- 显式调用，类型检查
- 相比 Reactive 局限更少

CONS

- `value`

Reactive

```
import { reactive } from 'vue'

const foo = { prop: 0 }
const bar = reactive({ prop: 0 })

foo.prop = 1
bar.prop = 1
```

PROS

- 自动 Unwrap (即不需要 `value`)

CONS

- 在类型上和一般对象没有区别
- 使用 ES6 解构会使响应性丢失
- 需要使用箭头函数包装才能使用 `watch`

Ref 自动解包

核心

在众多情况下，我们可以减少 ` `.value` 的使用

- `watch` 直接接受 Ref 作为监听对象，并在回调函数中返回解包后的值
- Ref 在模板中自动解包
- 使用 Reactive 解包嵌套的 Ref

```
const counter = ref(0)

watch(counter, count => {
  console.log(count) // same as `counter.value`
})
```

```
<template>
  <button @click="counter += 1">
    Counter is {{ counter }}
  </button>
</template>
```

```
import { ref, reactive } from 'vue'
const foo = ref('bar')
const data = reactive({ foo, id: 10 })
data.foo // 'bar'
```

'unref' - Ref 的反操作

核心

- 如果传入一个 Ref, 返回其值
- 否则原样返回

实现

```
function unref<T>(r: Ref<T> | T): T {  
  return isRef(r) ? r.value : r  
}
```

使用

```
import { unref, ref } from 'vue'  
  
const foo = ref('foo')  
unref(foo) // 'foo'  
  
const bar = 'bar'  
unref(bar) // 'bar'
```

接受 Ref 作为函数参数

模式

纯函数

接受 Ref 作为参数，
返回一个响应式的结果

同时接受传入值和 Ref

实现

```
function add(a: number, b: number) {  
    return a + b  
}
```

```
function add(a: Ref<number>, b: Ref<number>) {  
    return computed(() => a.value + b.value)  
}
```

```
function add(  
    a: Ref<number> | number,  
    b: Ref<number> | number  
) {  
    return computed(() => unref(a) + unref(b))  
}
```

用例

```
let a = 1  
let b = 2  
  
let c = add(a, b) // 3
```

```
const a = ref(1)  
const b = ref(2)  
  
const c = add(a, b)  
c.value // 3
```

```
const a = ref(1)  
  
const c = add(a, 5)  
c.value // 6
```

MaybeRef 类型工具

技巧

```
type MaybeRef<T> = Ref<T> | T
```

在 VueUse 中我们大量地使用 `MaybeRef` 来支持可选择性的响应式参数

```
export function useTimeAgo(
  time: Date | number | string | Ref<Date | number | string>,
) {
  return computed(() => someFormatting(unref(time)))
}
```

```
import { computed, unref, Ref } from 'vue'

type MaybeRef<T> = Ref<T> | T

export function useTimeAgo(
  time: MaybeRef<Date | number | string>,
) {
  return computed(() => someFormatting(unref(time)))
}
```

让你的函数变得更灵活

模式

就像乐高，让你的函数可以适应不同的使用场景。

构造一个 "特殊的" REF

```
import { useTitle } from '@vueuse/core'  
  
const title = useTitle()  
  
title.value = 'Hello World'  
// 网页的标题随 Ref 改变
```

绑定上一个现有的 REF

```
import { ref, computed } from 'vue'  
import { useTitle } from '@vueuse/core'  
  
const name = ref('Hello')  
const title = computed(() => {  
  return `${name.value} - World`  
})  
  
useTitle(title) // Hello - World  
  
name.value = 'Hi' // Hi - World
```

U 在 VueUse 中可用: useTitle

用例

'useTitle'

'useTitle' 的实现

```
import { ref, watch } from 'vue'
import { MaybeRef } from '@vueuse/core'

export function useTitle(
  newTitle: MaybeRef<string | null | undefined>
) {
  const title = ref(newTitle || document.title)

  watch(title, (t) => {
    if (t != null)
      document.title = t
  }, { immediate: true })

  return title
}
```

<-- 1. 重复使用用户提供的 Ref，或者建立一个新的

<-- 2. 将页面标题与 Ref 进行同步

重复使用已有 Ref

核心

如果将一个 `ref` 传递给 `ref()` 构造函数，它将会原样将其返回。

```
const foo = ref(1)    // Ref<1>
const bar = ref(foo) // Ref<1>

foo === bar // true
```

```
function useFoo(foo: Ref<string> | string) {
  // 不需要额外操作
  const bar = isRef(foo) ? foo : ref(foo)

  // 与上面的代码等效
  const bar = ref(foo)

  /* ... */
}
```

这个技巧在编写不确定参数类型的函数时十分有用。

\`ref\` / \`unref\`

技巧

- `MaybeRef<T>` 可以很好的配合 `ref` 和 `unref` 进行使用。
- 使用 `ref()` 当你想要将其标准化为 Ref
- 使用 `unref()` 当你想要获得其值

```
type MaybeRef<T> = Ref<T> | T

function useBalala<T>(arg: MaybeRef<T>) {
  const reference = ref(arg) // 得到 ref
  const value = unref(arg)   // 得到值
}
```

由 Ref 组成的对象

模式

以在使用可组合的函数式，同时获得 `ref` 和 `reactive` 的好处。

```
import { ref, reactive } from 'vue'

function useMouse() {
  return {
    x: ref(0),
    y: ref(0)
  }
}

const { x, y } = useMouse()
const mouse = reactive(useMouse())

mouse.x === x.value // true
```

- 可以直接使用 ES6 解构其中的 Ref 使用
- 根据使用方式，当想要自动解包的功能时，可以使用 `reactive` 将其转换为对象

将异步操作转换为“同步” 技巧

使用组合式 API, 我们甚至可以将异步请求转换为“同步”的

异步

```
const data = await fetch('https://api.github.com/').then(r => r.json())

// use data
```

组合式 API

```
const { data } = useFetch('https://api.github.com/').json()

const user_url = computed(() => data.value?.user_url)
```

先建立数据间的“连结”, 然后再等待异步请求返回将数据填充。概念和 React 中的 SWR (stale-while-revalidate) 类似。

'useFetch'

用例

```
export function useFetch<R>(url: MaybeRef<string>) {
  const data = shallowRef<T | undefined>()
  const error = shallowRef<Error | undefined>()

  fetch(unref(url))
    .then(r => r.json())
    .then(r => data.value = r)
    .catch(e => error.value = e)

  return {
    data,
    error
  }
}
```

U 在 VueUse 中可用: [useFetch](#)

副作用自动清除

模式

Vue 中原生的 `watch` 和 `computed` API 会在组件销毁时自动解除其内部的依赖监听。我们可以编写我们的函数时，遵循同样的模式。

```
import { onUnmounted } from 'vue'

export function useEventListener(target: EventTarget, name: string, fn: any) {
  target.addEventListener(name, fn)

  onUnmounted(() => {
    target.removeEventListener(name, fn) // <-- 
  })
}
```

U 在 VueUse 中可用: [useEventListener](#)

`effectScope` RFC

即将到来

一个新的 API 用于自动收集副作用，计划在 Vue 3.2 中引入

```
// 在函数在 Scope 内创建的 effect, computed, watch, watchEffect 等将会被自动收集

const scope = effectScope(() => {
  const doubled = computed(() => counter.value * 2)

  watch(doubled, () => console.log(double.value))

  watchEffect(() => console.log('Count: ', double.value))
})

// 清除 Scope 内的所有 Effect
stop(scope)
```

详见 <https://github.com/vuejs/rfcs/pull/212>

类型安全的 Provide / Inject

核心

使用 Vue 提供的 `InjectionKey<T>` 类型工具来在不同的上下文中共享类型。

```
// context.ts
import { InjectionKey } from 'vue'

export interface UserInfo {
  id: number
  name: string
}

export const injectKeyUser: InjectionKey<UserInfo> = Symbol()
```

类型安全的 Provide / Inject

核心

从同一个模组中为 `provide` 和 `inject` 引入相同的 Key

```
// parent.vue
import { provide } from 'vue'
import { injectKeyUser } from './context'

export default {
  setup() {
    provide(injectKeyUser, {
      id: '7', // 类型错误
      name: 'Anthony'
    })
  }
}
```

```
// child.vue
import { inject } from 'vue'
import { injectKeyUser } from './context'

export default {
  setup() {
    const user = inject(injectKeyUser)
    // UserInfo | undefined

    if (user)
      console.log(user.name) // Anthony
  }
}
```

状态共享

模式

由于组合式 API 天然提供的灵活性，状态可以独立于组件被创建并使用。

```
// shared.ts
import { reactive } from 'vue'

export const state = reactive({
  foo: 1,
  bar: 'Hello'
})
```

```
// A.vue
import { state } from './shared.ts'

state.foo += 1

// B.vue
import { state } from './shared.ts'

console.log(state.foo) // 2
```

⚠ 此方案不兼容 SSR!

兼容 SSR 的状态共享

模式

使用 `provide` 和 `inject` 来共享应用层面的状态。

```
export const myStateKey: InjectionKey<MyState> = Symbol()
export function createMyState() {
  const state = {
    /* ... */
  }

  return {
    install(app: App) {
      app.provide(myStateKey, state)
    }
  }
}

export function useMyState(): MyState {
  return inject(myStateKey)!
}
```

```
// main.ts
const App = createApp(App)
app.use(createMyState())
```

// A.vue

```
// 在任何组件中使用这个函数来获得状态对象
const state = useMyState()
```

- Vue Router v4 也使用的类似的方式
 - `createRouter()`
 - `useRouter()`

useVModel

技巧

一个让使用 props 和 emit 更加容易的工具

```
export function useVModel(props, name) {  
  const emit = getCurrentInstance().emit  
  
  return computed({  
    get() {  
      return props[name]  
    },  
    set(v) {  
      emit(`update:${name}`, v)  
    }  
  })  
}
```

```
export default defineComponent({  
  setup(props) {  
    const value = useVModel(props, 'value')  
  
    return { value }  
  }  
})
```

```
<template>  
  <input v-model="value" />  
</template>
```

U 在 VueUse 中可用: [useVModel](#)

以上所述，均可使用于 Vue 2 和 3

'@vue/composition-api'

库

为 Vue 2 提供组合式 API 的插件。

 [vuejs/composition-api](#)

```
import Vue from 'vue'  
import VueCompositionAPI from '@vue/composition-api'  
  
Vue.use(VueCompositionAPI)
```

```
import { ref, reactive } from '@vue/composition-api'
```

Vue 2.7

即将到来

Vue 2.7 计划

- 将 `@vue/composition-api` 整合进 Vue 2 的核心
- 在 SFC 中 `

Vue Demi

库

创建 Vue 2 和 3 兼容的插件/库

 [vueuse/vue-demi](https://github.com/vueuse/vue-demi)

```
// same syntax for both Vue 2 and 3
import { ref, reactive, defineComponent } from 'vue-demi'
```

23

VueDemi
Creates Universal Library for Vue 2 & 3

快速回顾

- 建立“连结”
- 接受 Ref 作为函数参数
- 返回由 Ref 组成的对象
- 使用 ref / unref 让函数变得更加灵活
- 将异步操作转换为“同步”
- 副作用自动清除
- 类型安全的 Provide / Inject
- 状态共享
- `useVModel`

谢谢！

幻灯片可以在我的网站 antfu.me 上下载