vue3 与 hook (一)

组合式函数是 vue3 带来的重大特性，是组织组件的新方式，它使得**复用状态逻辑**轻而易举。

主要分享编写组合函数的技巧或模式。

什么是组合式 API?

vue 官网说，组合式 API 是一系列 API 的集合。它包含以下 API:

- 响应式 API: ref、reactive 等；
- 生命周期钩子：onMounted 等；
- 依赖注入：provide & inject。

使用了以上 API 的函数，叫组合函数，通常使用`use`前缀命名。

状态 vs 状态逻辑

状态即组件状态：影响组件 UI 层的数据，可理解成 props 和 state，往往是变量，是数据，不含函数逻辑。为何说往往？有时候函数也可以作为 prop 传入组件，较少。

状态逻辑：**操作**组件状态的**函数**，希望能在组件之间**复用**。

操作的动作通常包括：订阅（获取）状态、修改状态、监听状态的变化等。

stackoverflow 上更多讨论

vue2 复用状态逻辑的方式

1. mixin

2. extends

> mixin 在插件中用得比较多，比如 vuex 全局混入状态。
>
> extends 可复用逻辑、状态和模板。

这两种方式都不好，滥用会导致命名冲突，代码难以理解，数据来源难以追踪。

3. renderProp

4. renderLess

> 复用逻辑、状态和模板。

比 1、2 好点，使用得当，可让组件易扩展、易使用，好理解。

5. provide inject

这在常规的项目开发中，很少用到，使用多了，会让组件变得强耦合，数据来源难以追踪。

> 复用状态、逻辑，在 vue 插件使用得多。

6. 将函数或者属性绑定到 vue 原型上

没有严重的缺点，主要复用逻辑，比如挂载 http 请求函数。

vue2 复用状态逻辑的问题

- 状态和 this 绑定了，导致复用困难

3 和 4 的方式，可使得状态脱离 this，非常强大，我非常喜欢这两种方式，但根据个人经验，vue2 的用户中，熟悉这两种的较少。

- 不能单独测试，需要依赖组件

在 vue3 中，1、2、6 的方式基本可以不用了。

- 状态来源难以追踪
- 命名冲突
- 类型支持弱

> 总之，问题比较多。

如何理解 setup?

vue3 的组合式函数解决 vue2 难以复用状态逻辑的问题。

组合式函数的优点

1. 状态不再和 this 绑定，独立于组件，可单独测试
2. 状态之间的依赖关系更加清晰，容易调试
3. 都是函数，可灵活组合，可从是否为纯函数的角度考虑
4. 类型支持好

参考 react hook 的写法，使用 use 作为组合函数的前缀。

在编写组合函数时，相同的功能，使用 react hook 实现一遍，加深理解两者的区别。

状态、状态逻辑和组件的模板是如何连接的?

setup 函数在组件创建时**只执行一次**，在 setup 中组合函数建立状态（数据）和逻辑（函数）、状态和模板之间的连接，即组合函数在 setup 钩入组件。

hooks.ts

```ts
export function useAdd(a: MaybeRef<number>,
 b: MaybeRef<number>) {
  console.log('useAdd') // NOTE 这个会执行几次?
  return computed(() => unref(a) + unref(b))
}
```

> 点击按钮，修改 b 时， useAdd 会再次执行?
> console.log('useAdd') 执行吗?

c:11

b:10

修改b

UseAdd.vue

```vue
<script>
import { useAdd } from './hooks'
import { ref } from 'vue'
export default {
  setup() {
    const a = 1
    const b = ref(10)
    const c = useAdd(a, b)
    return { b, c }
  },
}
</script>

<template>
  <p>c:{{ c }}</p>
  <p>b:{{ b }}</p>
  <button type="button" @click="++b">修改b</button>
</template>
```

useAdd 是如何和组件模板产生联系的?

setup 函数在组件创建时执行一次，useAdd 也执行了一次，a、b、c 之间就建立了关系，当修改 b 时，c 也会变化，*但 useAdd 不会再执行*。

react 版本的 useAdd:

```
export function useAdd(a: number, b: number) {
  console.log('useAdd')
  return useMemo(() => {
    console.log('useMemo')
    return a + b
  }, [a, b])
}
```

```
import { useAdd } from './hooks'
export default function UseAddDemo() {
  const a = 1
  const [b, setB] = useState(10)
  const c = useAdd(a, b)
  return (
    <div>
      <p>c:{c}</p>
      <button type='button' onClick={() => { console.log('onClick') setB(oldValue => ++oldValue)}}>
        修改b
      </button>
    </div>
  )
}
```

> 修改 b 后，useAdd 再次执行， 导致 console.log('useAdd') 再执行，注意这和 vue 的组合函数的重要区别。

vue 的组合函数在单独的 js 文件中使用，也会建立这样的依赖关系。

`testUseAdd.js`:

```
import { useAdd } from './hooks'
let a = 10
const b = ref(20)
const c = useAdd(a, b)
console.log(c.value) // 30
setTimeout(() => {
  a = 100
  b.value = 1000 // 修改 b console.log('useAdd') 会执行吗?
  console.log('setTimeout')
  console.log(c.value) // c 变成 1010, 而不是 1100 why? ?
}, 4000)
```

把`testUseAdd.js`引入组件，进行测试：

```
<script>
  import './testUseAdd'
</script>
```

> useAdd 的行为和在组件中的一致。console.log('useAdd') 不会再执行。

总结：

- 组合函数在 setup 执行时建立状态和模板的连接
- 组合函数可有自己的状态、计算属性、监听器、生命周期
- 注意在组合函数中只会执行一次的函数
- 对状态之间的**关系**需要有清晰的认识，否则无法提取组合函数

理解这四点是写好组合函数的关键。

理解状态之间的关系是提取组合函数的关键

识别出组件状态之间的关系对提取组合函数极为重要，否则就无法提取组合函数。

组件内部状态和状态逻辑

```
const a = ref(0)
const b = ref('')
const c = ref(true)
const d = reactive({})
const actionA = () => {
  a.value++
}
const actionC = () => {
  c.value = !c.value
}
const actionB = () => {
  b.value += 'test'
}
const actionD = async () => {
  const res = await http(`url`)
  d.a = res.a
  d.b = res.b
  d.c = res.c
}
const resetD = () => {
  Object.keys(d).forEach(key => delete d[key])
}
```

关系复杂，难以阅读

不能明显看出状态之间的关系，比 option API 更加难以理解。理清楚状态之间的关系后，提取组合函数

```
export const useHookA = () => {
  const a = ref(0)
  function actionA(){
    a.value++
  }
  return { a,actionA }
}
export const useHookD = () => {
  const d = reactive({})
  async function actionD(){
    const res = await http(`url`)
    d.a = res.a
    d.b = res.b
    d.c = res.c
  }
  function resetD (){
    Object.keys(d).forEach(key => delete d[key])
  }
  return { d,actionD,resetD }
}
```

组合 hook

```
// 从 hooks/index.js 导出 hooks
import { useHookA, useHookD } from './hooks'
const { a, actionA } = useHookA()
const { d, restD } = useHookD()
```

理解状态之间的关系，对提取组合函数时尤其重要

要求开发者有良好的**代码设计意识**和对业务有比较全面的理解，否则极可能写出难以阅读和维护的组件。

那么，如何设计组合函数呢?

编写组合函数的常见模式或技巧

hook 是特殊的函数，从**返回值**、**参数**和函数**内部操作**考虑。

返回响应式状态

使用组合函数封装一个响应式的 storage。

useStorage

```
export function useStorage(key, type = 'session') {
  let storage = null
  switch (type) {
    case 'session':
      storage = sessionStorage
      break
    case 'local':
      storage = localStorage
    default:
      break
  }
  const value = shallowRef(getItem(key, storage))
  function setItem(storage) {
    return newValue => {
      value.value = newValue
      storage.setItem(key, JSON.stringify(newValue))
    }
  }
  // NOTE 返回数组，可像 react 中的 useState 一样解构
  return [value, setItem(storage)]
}
```

用法

```
const [person, setItem] = useStorage('jack')
setItem({ name: 'reactive session storage' })
setTimeout(() => {
  setItem({ name: 'session storage' })
}, 4000)
watch(person, value => {
  console.log(value)
})
```

返回的响应式状态，可直接绑定到模板上、可监听、可用于计算属性，就和在 setup 里声明的具有同等效果。

setItem 用于修改状态，状态被修改了，会响应到模板上。

react 版本的 useStorage 用法

```
import { useState } from 'react'
function useStorage(key, type = 'session') {
  let storage = null
  switch (type) {
    case 'session':
      storage = sessionStorage
      break
    case 'local':
      storage = localStorage
    default:
      break
  }

  const [value, setValue] = useState(getItem(key, storage))
  function setItem(storage) {
    return newValue => {
      setValue(newValue)
      storage.setItem(key, JSON.stringify(newValue))
    }
  }
  // NOTE 返回数组，可像 react 中的 useState 一样解构
  return [value, setItem(storage)]
}
export default useStorage
```

```
import { useEffect } from 'react'

import { useStorage } from '../../hooks'

export default function WindowResize() {
  const [jack, setJack] = useStorage('jack')
  // NOTE 不能这样调用
  // setJack({ name: 'JackChou', age: 24 })
  useEffect(() => {
    setJack({ name: 'JackChou', age: 24 })
  }, [])
  return (
    <div>
      <p>
        jack's name {jack.name}, age {jack.age}
      </p>
      <button onClick={() =>
          setJack({ name: 'JACK', age: 10 })
       }>
        修改jack
      </button>
    </div>
  )
}
```

> 技巧：返回 toRefs 的数据，可使用解构且变量保持响应性。

有一跟踪鼠标位置的`useMouse`的 hook:

```
import { useOn } from './useOn' // 稍后有定义

function useMouse() {
  const x = ref(0)
  const y = ref(0)

  function update(event: MouseEvent) {
    x.value = event.pageX
    y.value = event.pageY
  }

  useOn('mousemove', update, window)

  return { x, y }
}
```

使用：

```
const { x, y } = useMouse()
```

返回 ref 组成的对象，解构后的变量是响应性的。

改写 useMouse，返回 reactive 对象。

```
function useMouse() {
  const position = reactive({ x: 0, y: 0 })

  function update(event: MouseEvent) {
    position.x = event.pageX
    position.y = event.pageY
  }
  useOn('mousemove', update, window)

  return position
}
```

解构后属性失去响应性。

```
const { x, y } = useMouse() // 怎么办
```

返回`toRefs`可解决：

```
return toRefs(position)
```

> 技巧：返回响应式状态和在组件内声明的响应式状态一样：可监听，可用于生成计算属性。

比如：

```
import { useFetch } from '@vueuse/core'

const { data } = useFetch('https://API.github.com/users/jackchoumine').json()
const avatar = computed(() => data.value?.avatar_url)
```

页面需要展示用户头像，可通过计算属性拿到，等接口返回后，再计算出用户头像。

react 版本的 useMouse

```
import { useState } from 'react'

import useOn from './useOn'

function useMouse() {
  const [position, setPosition] = useState({ x: 0, y: 0 })
  function update(event: MouseEvent) {
    const { pageX, pageY } = event
    setPosition({ x: pageX, y: pageY })
  }
  useOn('mousemove', update, window)

  return position
}
export default useMouse
```

返回响应式状态及其修改函数

> 为何要返回一个修改状态的函数?

返回修改函数，使得状态可变化，就可把修改状态的操作封装在 hook 内部。

`useCounter`:

```
export default function useCounter(initCount: number = 0) {
  const count = ref(initCount)
  function add(step = 1) {
    count.value += step
  }
  function reduce(step = 1) {
    count.value -= step
  }
  return {
    count,
    add,
    reduce,
  }
}
```

`SimpleCounter.vue`

```
<script setup lang="ts">
  import {useCounter} from './hooks'

  const { count, add, reduce } = useCounter(10)
</script>

<template>
  <div class="counter">
    <button @click="() => reduce()">-</button>
    {{ count }}
    <button @click="() => add()">+</button>
  </div>
</template>
```

这样就得到一个简单的 Counter:

- 10 +

react 版本的 useCounter:

```
import { useState } from 'react'

function useCounter(initCount = 1) {
  const [count, setCount] = useState(initCount)
  function add(step = 1) {
    setCount(count + step)
  }
  function reduce(step = 1) {
    setCount(count - step)
  }
  return { count, add, reduce }
}

export default useCounter
```

返回组件

希望实现一个跟随鼠标移动的组件

```ts
export function unravel<T>(value: MaybeLazyRef<T>): T {
  if (typeof value === 'function') {
    return (value as () => T)()
  }
  return unref(value)
}

type LazyOrRef<T> = Ref<T> | (() => T)

export function useMouseFollower(position: LazyOrRef<{ x: number; y: number }>) {
  const style = computed(() => {
    const { x, y } = unravel(position)
    return {
      position: 'fixed',
      top: 0,
      left: 0,
      // NOTE添加一定的偏移，否则鼠标被遮挡，无法聚焦其他元素
      transform: `translate3d(${x + 15}px, ${y + 15}px, 0)`,
    }
  })

  const Follower = defineComponent(
    (props, { slots }) => () => h('div', { ...props, style: style.value }, slots)
  )
  return Follower
}
```

```vue
<template>
    <Follower>
      <div class="follower-content">I follow your mouse</div>
    </Follower>
    <pre>x: {{ x }}, y: {{ y }}</pre>
</template>

<script lang="ts" setup>
  import { useMouse, useMouseFollower } from '../../hooks'

  const { x, y } = useMouse()
  const Follower = useMouseFollower(() => ({ x: x.value, y: y.value }))
</script>

<style scoped lang="css">
  .follower-content {
    background-color: antiquewhite;
    padding: 10px 12px;
    border: 1px solid lightblue;
    border-radius: 6px;
    font-size: 14px;
    color: black;
  }
</style>
```

react 版本的 useMouseFollower.tsx

```tsx
import type { CSSProperties, PropsWithChildren } from 'react'

type Props = {}

function useMouseFollower(position: Record<'x' | 'y', number>) {
  const style: CSSProperties = {
    position: 'fixed',
    top: 0,
    left: 0,
    transform: `translate3d(${position.x + 15}px, ${position.y + 15}px, 0)`,
  }
  const Follower = (props: PropsWithChildren<Props>) => {
    return <div style={style}>{props.children}</div>
  }
  return Follower
}

export default useMouseFollower
```

输入响应式状态，再返回响应式状态

vue 是副作用驱动的，很多场景下，某些状态变化时（可理解为副作用的依赖），需要执行副作用，比如发送网络请求，此时可提取 hook, 把依赖作为 hook 的参数。

请输入关键字

- vue
- react
- solidjs
- angular
- svelte
- preact

组件的基本功能：拉取后台数据，且用户输入时，再调用接口拉取数据，非常普遍的功能。

```
<template>
  <input type="text" v-model="input" style="background-color:azure;" placeholder="请求输入关键字" />
  <ul>
    <li v-for="(item, index) in list" :key="index">{{ item.name }}</li>
  </ul>
</template>

<script setup>
import { http } from './utils'
import { ref, watch } from 'vue'
const input = ref('')
const list = ref([])

httpGet()
watch(input, value => {
  httpGet(value)
})

function httpGet(key='') {
  http(key).then(res => {
    list.value = res
  })
}
</script>
```

如何使用 hook 写出相同的功能?

关键点：如何处理 httpGet 的依赖？将用户输入作为参数。

`hooks.ts`

```
import type { Ref } from 'vue'

export function useHttpGet(key: Ref<string>) {
  const list = ref([])
  watch(
    key,
    newKey => {
      http(newKey)
    },
    { immediate: true }
  )

  return list
}

function httpGet(key ='') {
  http(key).then(res => {
    list.value = res
  })
}
```

使用方式：

```
<script setup>
  import { useHttpGet } from './hooks'

  const input = ref('')
  const list = useHttpGet(input)
</script>
```

使用 hook 之后，代码简洁多了。

react 版本的 useHttpGet：

```
import { useEffect, useState } from 'react'

function useHttpGet(key = '') {
  const [list, setList] = useState([])
  useEffect(() => {
    http(key).then(res => {
      setList(res)
    })
  }, [key])

  return list
}
```

修改 hook 返回的响应式状态

hook 返回的响应式状态，可在组件里修改，然后触发 hook 内部的 watch、computed 执行。

```ts
export function useTitle(newTitle?: MaybeRef<string>) {
  const title = ref(newTitle)
  watchEffect(() => {
    const _title = title.value || document.title
    document.title = _title
  })
  return title
}
```

使用：

```ts
const title = useTitle()
title.value = '修改hook的返回值' // 会触发 useTitle 内的监听器执行
```

> 在外部直接修改 hook 返回的状态，可能你不清楚内置执行了什么副作用，不太建议这样做。

react 版本的 useTitle

```tsx
import { useEffect, useState } from 'react'

function useTitle(initTitle = '') {
  const [title, setTitle] = useState<string>(initTitle ?? document.title)
  useEffect(() => {
    document.title = title
  }, [title])

  return { title, setTitle }
}

export default useTitle
```

> react 不能在外部修改 title，所以返回 setTitle

再看一个例子：

`MyInput.vue`

```
<script setup>
  defineProps({
    modelValue: {
      type: String,
    },
  })
  const emits = defineEmits(['update:modelValue'])
  function update(event) {
    emits('update:modelValue', event.target.value)
  }
</script>

<template>
  <input type="text" :value="modelValue" @input="update" />
</template>
```

创建一个返回计算属性的 hook，代替`value`和`input`事件。

```
export function useVModel(props, name) {
  const emit = getCurrentInstance().emit

  return computed({
    get() {
      return props[name]
    },
    set(v) {
      emit(`update:${name}`, v)
    },
  })
}
```

```
<script setup>
  import { useVModel } from './hooks'
  const props = defineProps({
    modelValue: {
      type: String,
    },
  })
  const value = useVModel(props, 'modelValue')
</script>
<template>
  <!-- NOTE 通过 v-model 修改 useVModel 的返回值 -->
  <input type="text" v-model="value" />
</template>
```

请输入

react 版本的 hook:

```
import { useState } from 'react'

function useInput(initialValue = '') {
  const [value, setState] = useState(initialValue)
  function onChange(event) {
    setState(event.target?.value)
  }
  return [value, onChange]
}

export default useInput
```

参数可能是 Ref

希望参数**可能是** Ref，在编写参数类型不确定的 hook 时很有用。

再对上面的`useHttpGet`改造：

```
import type { Ref } from 'vue'

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

export function useHttpGet(key: MaybeRef<string>) {
  const keyRef = ref(key)
  const list = ref([])
  watch(
    keyRef,
    newKey => {
      http(newKey).then(res => {
        // @ts-ignore
        list.value = res
      })
    },
    { immediate: true }
  )

  return  list
}
```

`ref`函数的参数是 ref，返回 ref，是普通变量，就将其包裹成 ref。

> 技巧

可让参数和现有的 ref 建立连接，修改现有 ref，触发 hook 内部的逻辑执行。

还是上面 useTitle，可这样使用：

```
const hello = ref('hello')
const title = computed(() => {
  return hello.value + Math.random() * 10
})
useTitle(title)
setTimeout(() => {
  hello.value = 'Hello'
}, 2000)
```

参数为函数

hook 的参数可为函数，这些函数往往是一些副作用，比如事件处理器、网络接口。

useOn

```ts
type Target = HTMLElement | Document | Window | BroadcastCh
export function useOn(
  eventName: string,
  handler: Handler,
  target: Target,
) {
  onMounted(() => {
    target.addEventListener(eventName, handler)
  })
  onUnmounted(() => {
    target.removeEventListener(eventName, handler)
  })
}
```

useMouse

```ts
function useMouse() {
  const x = ref(0)
  const y = ref(0)

  function update(event: MouseEvent) {
    x.value = event.pageX
    y.value = event.pageY
  }

  useOn('mousemove', update, window)

  return { x, y }
}
```

hook 内部可以有哪些操作?

- computed
- watch、watchEffect
- 事件监听

setup 函数能使用的，都可放在 hook 内部

> provide & inject 不行

```
import type { ComponentInternalInstance } from 'vue'
import { getCurrentInstance } from 'vue'

function useGlobalProps() {
  const { appContext } = getCurrentInstance() as ComponentInternalInstance
  const globalProps = appContext.config.globalProperties
  return { ...globalProps }
}

export default useGlobalProps
// main
app.config.globalProperties.globalFn = function testGlobal(name: string) {
  console.log(name)
}
```

下周分享预告

- 把第三库函数封装成 hook
- hook 共享状态
- 条件语句与 hook
- vue hook vs react hook
- hook vs 工具函数
- hook 与 headless component
- 其他

Q & A

谢谢