Vue Function-based API RFC

基于函数的 Vue API 设计

周炼赤

2019-10-29

目录

01 概要

02 设计动机

03 设计细节

04 潜在问题

05 升级策略

概要

1

一、概要

1、Vue Function-based API RFC，什么是 RFC？

Vue RFCs

What is an RFC?

The "RFC" (request for comments) process is intended to provide a consistent and controlled path for new features to enter the framework.

Many changes, including bug fixes and documentation improvements can be implemented and reviewed via the normal GitHub pull request workflow.

Some changes though are "substantial", and we ask that these be put through a bit of a design process and produce a consensus among the Vue core team and the community.

一、概要

- Vue Function-based API RFC 是 Vue 3.0 最重要的 RFC。
- 将 Vue 2.x 中与组件逻辑相关的选项以 API 函数的形式重新设计。

一、概要

```
import { value, computed, watch, onMounted } from 'vue'

const App = {
  template: `
    <div>
      <span>count is {{ count }}</span>
      <span>plusOne is {{ plusOne }}</span>
      <button @click="increment">count++</button>
    </div>
  `,
  setup() {
    const count = value(0)                    // data
    const plusOne = computed(() => count.value + 1) // computed
    const increment = () => { count.value++ }       // method
    watch(() => count.value * 2, val => {           // watch
      console.log(`count * 2 is ${val}`)
    })
    onMounted(() => { console.log(`mounted`) })     // 生命周期钩子
    return { count, plusOne, increment }
  }
}
```

设计动机

2

二、设计动机 ——（1）逻辑组合和复用

组件 API 设计所面对的核心问题之一就是如何组织逻辑，以及如何在多个组件之间抽取和复用逻辑。

基于 Vue 2.x 的 API 有一些常见的逻辑复用模式（Mixins、HOC、Renderless Components 等），但都或多或少存在一些问题：

- **模版中的数据来源不清晰。**例如，当一个组件中使用了多个 mixin 的时候，光看模版会很难分清一个属性到底是来自哪一个 mixin。HOC 同理。
- **命名空间冲突。**由不同开发者开发的 mixin 无法保证不会正好用到一样的属性或是方法名。HOC 在注入的 props 中也存在类似问题。
- **性能。**HOC 和 Renderless Components 都需要额外的组件实例嵌套来封装逻辑，导致无谓的性能开销。

二、设计动机 ——（1）逻辑组合和复用

Function-based API 受 React Hooks 的启发，提供了一个全新的逻辑复用方案，且不存在上述问题。

使用基于函数的 API，我们可以将相关联的代码抽取到一个「composition function（组合函数）」中 —— 该函数封装了相关联的逻辑，并将需要暴露给组件的状态以响应式的数据源的方式返回出来。

二、设计动机——（1）逻辑组合和复用

一个用组合函数来封装鼠标位置侦听逻辑的例子：

```
function useMouse() {
  const x = value(0)
  const y = value(0)
  const update = e => {
    x.value = e.pageX
    y.value = e.pageY
  }
  onMounted(() => {
    window.addEventListener('mousemove', update)
  })
  onUnmounted(() => {
    window.removeEventListener('mousemove', update)
  })
  return { x, y }
}

// 在组件中使用该函数
const Component = {
  setup() {
    const { x, y } = useMouse()
    // 与其它函数配合使用
    const { z } = useOtherLogic()
    return { x, y, z }
  },
  template: `<div>{{ x }} {{ y }} {{ z }}</div>`
}
```

二、设计动机 ——（2）类型推导

Vue 3.0 的一个主要设计目标是增强对 TypeScript 的支持。

Vue 开发团队：

> 原本我们期望通过 Class API 来达成这个目标，但是经过讨论和原型开发，我们认为 Class 并不是解决这个问题的正确路线，基于 Class 的 API 依然存在类型问题。

- 基于函数的 API 天然对类型推导很友好，因为 TS 对函数的参数、返回值和泛型的支持已经非常完备。
- 更值得一提的是基于函数的 API 在使用 TS 或是原生 JS 时写出来的代码几乎是完全一样的。

二、设计动机 ——（3）打包尺寸

- 基于函数的 API 每一个函数都可以作为 named ES export 被单独引入，这使得它们对 tree-shaking 非常友好。没有被使用的 API 的相关代码可以在最终打包时被移除。

- 同时，基于函数 API 所写的代码也有更好的压缩效率，因为所有的函数名和 setup 函数体内部的变量名都可以被压缩，但对象和 class 的属性/方法名却不可以。

设计细节

3

三、设计细节 ——（1）setup() 函数

顾名思义，这个函数将会是我们 setup 我们组件逻辑的地方，它会在一个组件实例被创建时，初始化了 props 之后调用。setup() 会接收到初始的 props 作为参数：

```
const MyComponent = {
  props: {
    name: String
  },
  setup(props) {
    console.log(props.name)
  }
}
```

- 需要留意的是，这里传进来的 props 对象是响应式的 —— 它可以被当作数据源去观测，当后续 props 发生变动时它也会被框架内部同步更新。但对于用户代码来说，它是不可修改的（会导致警告）。
- 在 setup 内部可以使用 this，但你大部分时候不会需要它。

三、设计细节 ——（2）组件状态 State

类似 data()，setup() 可以返回一个对象 —— 其属性将会被暴露给模版的渲染上下文：

```
const MyComponent = {
  props: {
    name: String
  },
  setup(props) {
    return {
      msg: `hello ${props.name}!`
    }
  },
  template: `<div>{{ msg }}</div>`
}
```

- 上面这个例子跟 data() 一模一样：msg 可以在模版中被直接使用。
- 但如果我们想要创建一个可以在 setup() 内部被管理的值，可以使用 value 函数：

三、设计细节 ——（2）组件状态 State

```
import { value } from 'vue'

const MyComponent = {
  setup(props) {
    const msg = value('hello')
    const appendName = () => {
      msg.value = `hello ${props.name}`
    }
    return {
      msg,
      appendName
    }
  },
  template: `<div @click="appendName">{{ msg }}</div>`
}
```

- value() 返回的是一个 value wrapper （包装对象）。
- 一个包装对象只有一个属性： **.value** ，该属性指向内部被包装的值。
- 在上面的例子中，msg 包装的是一个字符串。

三、设计细节 ——（2）组件状态 State

配合手写 Render 函数使用

如果你的组件不使用模版，你也可以选择在 setup() 中直接返回一个渲染函数：

```
import { value, createElement as h } from 'vue'

const MyComponent = {
  setup(initialProps) {
    const count = value(0)
    const increment = () => { count.value++ }

    return (props, slots, attrs, vnode) => (
      h('button', {
        onClick: increment
      }, count.value)
    )
  }
}
```

三、设计细节 ——（2）组件状态 State

Computed Value（计算值）

除了直接包装一个可变的值，我们也可以包装通过计算产生的值：

```
import { value, computed } from 'vue'

const count = value(0)
const countPlusOne = computed(() => count.value + 1)

console.log(countPlusOne.value) // 1

count.value++
console.log(countPlusOne.value) // 2
```

- computed() 返回的是一个只读的包装对象。
- 默认情况下，如果用户试图去修改一个只读包装对象，会触发警告。
- 双向计算值可以通过传给 computed 第二个参数作为 setter 来创建。

三、设计细节 —— （3）Watchers

watch() API 提供了基于观察状态的变化来执行副作用的能力。

watch() 接收的第一个参数被称作 “数据源” ，它可以是：

- 一个返回任意值的函数
- 一个包装对象
- 一个包含上述两种数据源的数组

第二个参数是回调函数。回调函数只有当数据源发生变动时才会被触发。

三、设计细节 —— （3）Watchers

```
watch(
  // getter
  () => count.value + 1,
  // callback
  (value, oldValue) => {
    console.log('count + 1 is: ', value)
  }
)
// -> count + 1 is: 1

count.value++
// -> count + 1 is: 2
```

- 和 2.x 的 $watch 不同的是，watch() 的回调会在创建时就执行一次。这有点类似 2.x watcher 的 immediate: true 选项，但有一个重要的不同：watch()的回调在触发时，DOM 总是会在一个已经被更新过的状态下。 这个行为是可以通过选项来定制的。
- 在 2.x 的代码中，我们经常会遇到同一份逻辑需要在 mounted 和一个 watcher 的回调中执行，3.0 的 watch() 默认行为可以直接表达这样的需求。

三、设计细节 —— （3）Watchers

观察 props

上面提到了 setup() 接收到的 props 对象是一个可观测的响应式对象：

```
const MyComponent = {
  props: {
    id: Number
  },
  setup(props) {
    const data = value(null)
    watch(() => props.id, async (id) => {
      data.value = await fetchData(id)
    })
    return {
      data
    }
  }
}
```

三、设计细节 ——（3）Watchers

观察包装对象

watch()可以直接观察一个包装对象：

```
// double 是一个计算包装对象
const double = computed(() => count.value * 2)

watch(double, value => {
  console.log('double the count is: ', value)
}) // -> double the count is: 0

count.value++ // -> double the count is: 2
```

三、设计细节 ——（3）Watchers

观察多个数据源

watch() 也可以观察一个包含多个数据源的数组 – 这种情况下，任意一个数据源的变化都会触发回调，同时回调会接收到包含对应值的数组作为参数：

```
watch(
  [valueA, () => valueB.value],
  ([a, b], [prevA, prevB]) => {
    console.log(`a is: ${a}`)
    console.log(`b is: ${b}`)
  }
)
```

三、设计细节 ——（3）Watchers

停止观察

watch() 返回一个停止观察的函数：

```
const stop = watch(...)
// stop watching
stop()
```

如果 watch() 是在一个组件的 setup() 或是生命周期函数中被调用的，那么该 watcher 会在当前组件被销毁时也一同被自动停止：

```
export default {
  setup() {
    // 组件销毁时也会被自动停止
    watch(/* ... */)
  }
}
```

三、设计细节 ——（4）生命周期函数

所有现有的生命周期钩子都会有对应的 onXXX 函数（只能在 setup() 中使用）：

```
import { onMounted, onUpdated, onUnmounted } from 'vue'

const MyComponent = {
  setup() {
    onMounted(() => {
      console.log('mounted!')
    })
    onUpdated(() => {
      console.log('updated!')
    })
    // destroyed 调整为 unmounted
    onUnmounted(() => {
      console.log('unmounted!')
    })
  }
}
```

三、设计细节 ——（5）依赖注入

```
import { provide, inject } from 'vue'

const CountSymbol = Symbol()

const Ancestor = {
  setup() {
    // providing a value can make it reactive
    const count = value(0)
    provide({
      [CountSymbol]: count
    })
  }
}
const Descendent = {
  setup() {
    const count = inject(CountSymbol)
    return {
      count
    }
  }
}
```

如果注入的是一个包装对象，则该注入绑定会是响应式的（也就是说，如果 Ancestor 修改了 count，会触发 Descendent 的更新）。

潜在问题

4

四、潜在问题

Q：新的 API 使得动态地修改一个组件的选项变得更困难（原来是一个对象，现在是一段无法被检视的函数体）。

A：这可能是一件好事，因为通常在用户代码中动态地修改组件是一类比较危险的操作，对于运行时也增加了许多潜在的边缘情况（特别是组件继承和使用 mixin 的情况下）。新 API 的灵活性应该在绝大部分情况下都可以用更显式的代码达成同样的结果。

四、潜在问题

Q： 缺乏经验的用户可能会写出「面条代码」，因为新 API 不像旧 API 那样强制将组件代码基于选项切分开来。

A： 能写出「面条代码」的人在哪都会写出「面条代码」。

基于选项的 API 只是看上去更整洁。例如从服务端请求数据，可能需要用到 props, data(), mounted 和 watch。极端情况下，如果一个应用中的所有的逻辑任务都在一个组件里，这个组件必然会变得庞大且难以维护。

对比之下，基于函数的 API 可以把每个逻辑任务的代码都整理到一个对应的函数中。当逻辑变得复杂时，可以对 setup() 函数进行切分。而如果是基于选项，则无法做到这样的切分，因为用 mixin 只会让事情变得更糟糕。

从这个角度看，基于选项 vs. 基于函数就好像基于 HTML/CSS/JS 组织代码 vs. 基于单文件组件来组织代码。

升级策略

5

五、升级策略

新的 API 和 2.x 的 API 理论上完全兼容（只是多了一个 setup() 选项）。
但是，新 API 的引入实际上会让相当一部分的旧选项长远来说变得没有必要。
如果能够去掉对这些旧选项的支持，可以获得相当的代码尺寸和性能提升。

因此，Vue 3.0 计划提供两个不同的版本：

- **兼容版本**：同时支持新 API 和 2.x 的所有选项；
- **标准版本**：只支持新 API 和部分 2.x 选项。

五、升级策略

1、保留的选项

以下选项行为和 2.x 保持一致，并在兼容和标准版本中都会支持。标有 * 的选项可能会有进一步的调整。

- name
- props
- template
- render
- components
- directives
- Filters *
- Delimiters *
- comments *

五、升级策略

2、由于本 RFC 提案而不再必须的选项

以下选项将会在标准版本中被移除，只在兼容版本中支持。

- data（由 setup() + value() + state() 取代）
- computed（由 computed() 取代）
- methods（ 由 setup() 中声明的函数取代）
- watch （由 watch() 取代）
- provide/inject（由 provide() 和 inject() 取代）
- mixins （由组合函数取代）
- extends （由组合函数取代）
- 所有的生命周期选项 （由 onXXX 函数取代）

五、升级策略

3、被其它 RFC 提案废弃的选项

以下选项将会在标准版本中被移除，只在兼容版本中支持。

- el（应用将不再由 new Vue() 来创建，而是通过新的 createApp 来创建）
- propsData（给 root component 的 props 通过新的 createApp API 创建的应用实例来提供）
- functional(3.0 函数式组件直接用函数来声明)
- model(v-model 指令参数使得该选项不再必要)
- inhertiAttrs (非 props 属性的继承行为改动使得该选项不再必要)

THANKS