



下载APP



28 | 响应式：万能的面试题，怎么手写响应式系统

2021-12-24 大圣

《玩转Vue 3全家桶》

课程介绍 >

**讲述：大圣**

时长 09:03 大小 8.30M



你好，我是大圣。

经过前面课程的学习，相信你对 Vue3 的实战和组件有了新的认识，也掌握了很多实战秘籍，从今天开始，我将带你进入 Vue 框架的内部世界，探究一下 Vue 框架的原理，让你能知其然，也知其所以然。

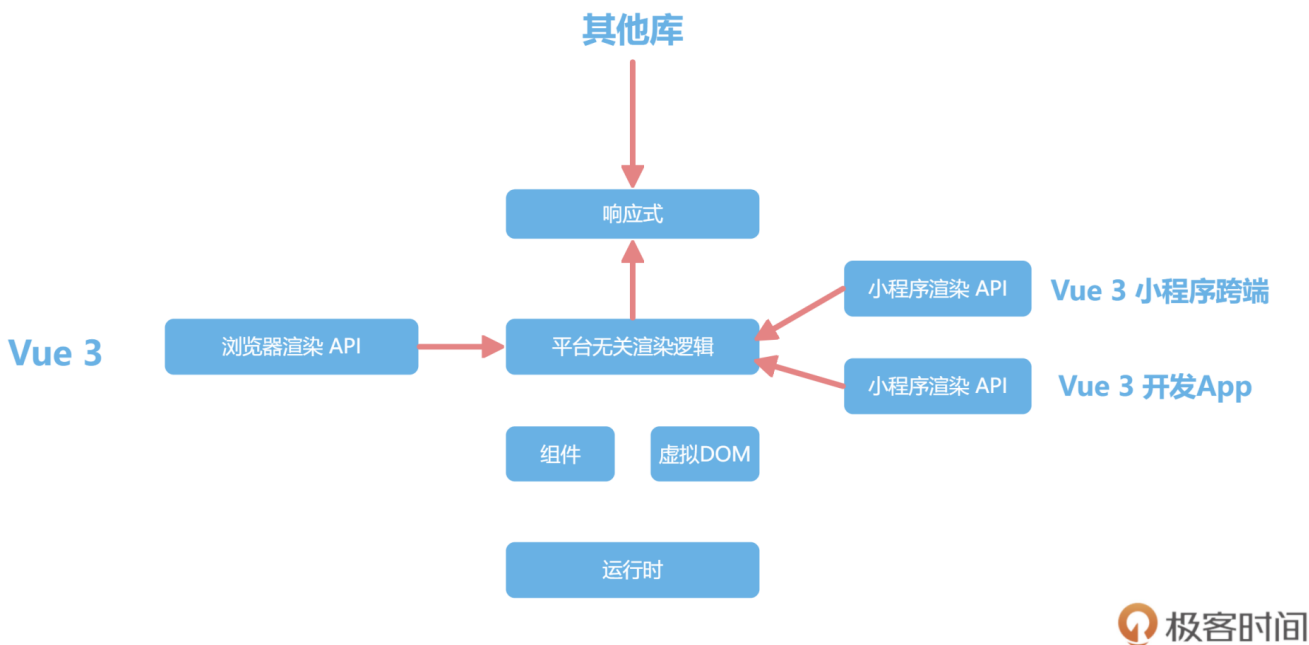
我们将手写一个迷你的 Vue 框架，实现 Vue3 的主要渲染和更新逻辑，项目就叫 weiyoui，你可以在 [GitHub](#) 上看到所有的核心代码。



响应式

在第三讲的 Vue3 新特性中，我们剖析了 Vue3 的功能结构，就是下图所示的 Vue 核心模块，可以看到，Vue3 的组件之间是通过响应式机制来通知的，响应式机制可以自动收集系统中数据的依赖，并且在修改数据之后自动执行更新，极大提高开发的效率。

我们今天就要自己做一个迷你的响应式原型，希望你能通过自己手写，搞清楚响应式的实现原理。



根据响应式组件通知效果可以知道，**响应式机制的主要功能就是，可以把普通的 JavaScript 对象封装成为响应式对象，拦截数据的获取和修改操作，实现依赖数据的自动化更新。**

所以，一个最简单的响应式模型，我们可以通过 reactive 或者 ref 函数，把数据包裹成响应式对象，并且通过 effect 函数注册回调函数，然后在数据修改之后，响应式地通知 effect 去执行回调函数即可。

整个流程这么概括地说，你估计不太理解，我们先通过一个简单的小例子直观感受一下响应式的效果。

Vue 的响应式是可以独立在其他平台使用的。比如你可以新建 test.js，使用下面的代码在 node 环境中使用 Vue 响应。以 reactive 为例，我们使用 reactive 包裹 JavaScript 对象之后，每一次对响应式对象 counter 的修改，都会执行 effect 内部注册的函数：

```

1  const {effect, reactive} = require('@vue/reactivity')
2
3  let dummy
4  const counter = reactive({ num1: 1, num2: 2 })
5  effect(() => {
6    dummy = counter.num1 + counter.num2
7    console.log(dummy) // 每次counter.num1修改都会打印日志
8  })
9  setInterval(()=>{
10   counter.num1++
11 },1000)

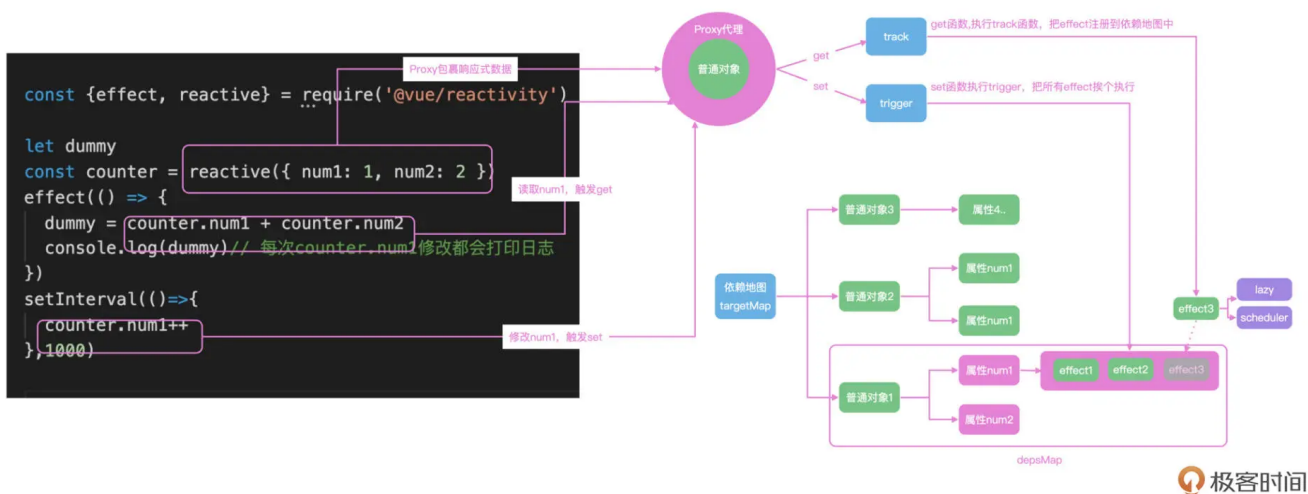
```

执行 node test.js 之后，你就可以看到 effect 内部的函数会一直调用，每次 count.value 修改之后都会执行。

看到这个 API 估计你有点疑惑，effect 内部的函数式如何知道 count 已经变化了呢？

我们先来看一下响应式整体的流程图，上面的代码中我们使用 reactive 把普通的 JavaScript 对象包裹成响应式数据了。

所以，在 effect 中获取 counter.num1 和 counter.num2 的时候，就会触发 counter 的 get 拦截函数；**get 函数，会把当前的 effect 函数注册到一个全局的依赖地图中去。**这样 counter.num1 在修改的时候，**就会触发 set 拦截函数，去依赖地图中找到注册的 effect 函数，然后执行。**



具体是怎么实现的呢？我们从第一步把数据包裹成响应式对象开始。先看 reactive 的实现。

reactive

我们进入到 src/reactivity 目录中，新建 reactive.spec.js，使用下面代码测试 reactive 的功能，能够在响应式数据 ret 更新之后，执行 effect 中注册的函数：

[复制代码](#)

```
1 import { effect } from '../effect'
2 import { reactive } from '../reactive'
3
4 describe('测试响应式', () => {
5   test('reactive基本使用', () => {
6     const ret = reactive({ num: 0 })
7     let val
8     effect(() => {
9       val = ret.num
10    })
11    expect(val).toBe(0)
12    ret.num++
13    expect(val).toBe(1)
14    ret.num = 10
15    expect(val).toBe(10)
16  })
17 })
```

之前讲过在 Vue3 中，reactive 是通过 ES6 中的 Proxy 特性实现的属性拦截，所以，在 reactive 函数中我们直接返回 newProxy 即可：

[复制代码](#)

```
1 export function reactive(target) {
2   if (typeof target !== 'object') {
3     console.warn(`reactive ${target} 必须是一个对象`);
4     return target
5   }
6
7   return new Proxy(target, mutableHandlers);
8 }
```

可以看到，下一步我们需要实现的就是 Proxy 中的处理方法 mutableHandles。

这里会把 Proxy 的代理配置抽离出来单独维护，是因为，其实 Vue3 中除了 reactive 还有很多别的函数需要实现，比如只读的响应式数据、浅层代理的响应式数据等，并且

reactive 中针对 ES6 的代理也需要单独的处理。

这里我们只处理 js 中对象的代理设置：

[复制代码](#)

```
1  const proxy = new Proxy(target, mutableHandlers)
```

mutableHandles

好，看回来，我们剖析 mutableHandles。它要做的事就是配置 Proxy 的拦截函数，这里我们只拦截 get 和 set 操作，进入到 baseHandlers.js 文件中。

我们使用 createGetter 和 createSetters 来创建 set 和 get 函数，mutableHandles 就是配置了 set 和 get 的对象返回。

get 中直接返回读取的数据，这里的 Reflect.get 和 target[key]实现的结果是一致的；并且返回值是对象的话，还会嵌套执行 reactive，并且调用 track 函数收集依赖。

set 中调用 trigger 函数，执行 track 收集的依赖。

[复制代码](#)

```
1  const get = createGetter();
2  const set = createSetter();
3
4  function createGetter(shallow = false) {
5    return function get(target, key, receiver) {
6      const res = Reflect.get(target, key, receiver)
7      track(target, "get", key)
8      if (isObject(res)) {
9        // 值也是对象的话，需要嵌套调用reactive
10       // res就是target[key]
11       // 浅层代理，不需要嵌套
12       return shallow ? res : reactive(res)
13     }
14     return res
15   }
16 }
17
18 function createSetter() {
19   return function set(target, key, value, receiver) {
20     const result = Reflect.set(target, key, value, receiver)
21     // 在触发 set 的时候进行触发依赖
```



```

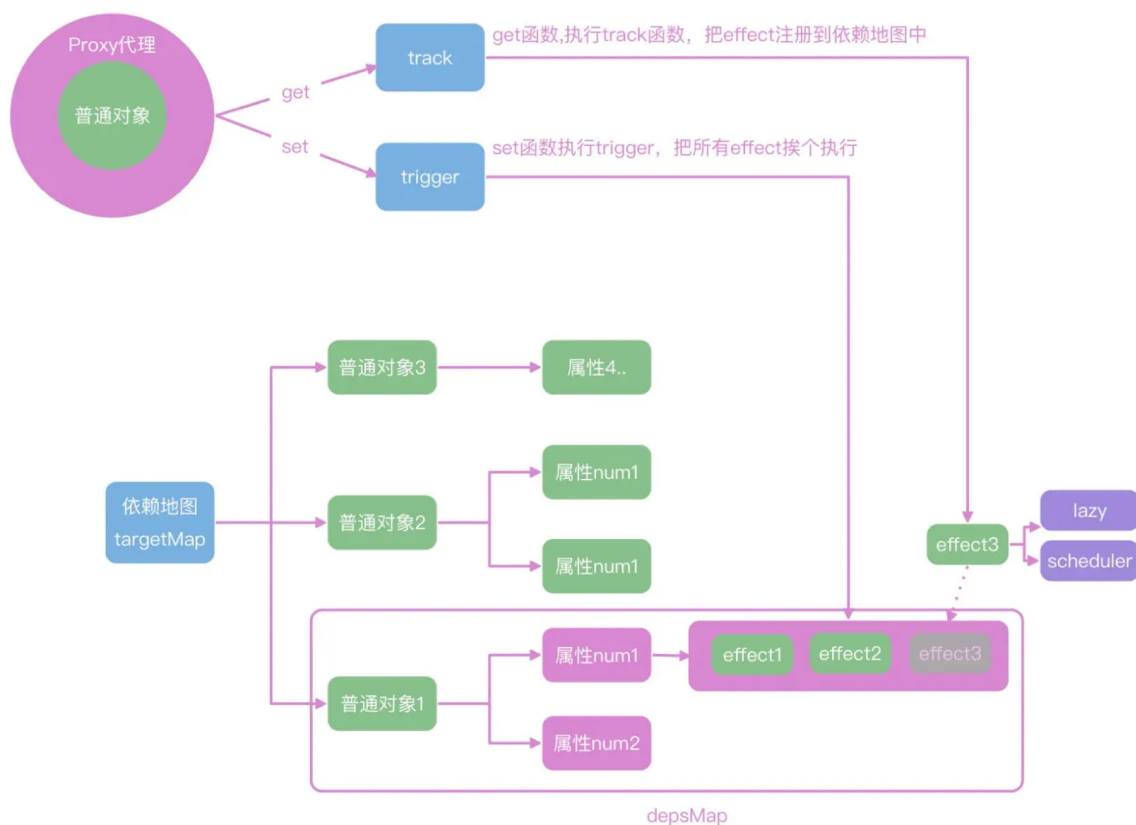
22     trigger(target, "set", key)
23     return result
24 }
25 }
26 export const mutableHandles = {
27   get,
28   set,
29 };

```

我们先看 get 的关键部分，track 函数是怎么完成依赖收集的。

track

具体写代码之前，把依赖收集和执行的原理我们梳理清楚，看下面的示意图：



在 track 函数中，我们可以使用一个巨大的 targetMap 去存储依赖关系。map 的 key 是我们要代理的 target 对象，值还是一个 depsMap，存储这每一个 key 依赖的函数，每一个 key 都可以依赖多个 effect。上面的代码执行完成，depsMap 中就有了 num1 和 num2 两个依赖。

而依赖地图的格式，用代码描述如下：

 复制代码

```
1 targetMap = {
2   target: {
3     key1: [回调函数1, 回调函数2],
4     key2: [回调函数3, 回调函数4],
5   },
6   target1: {
7     key3: [回调函数5]
8   }
9
10 }
```

好，有了大的设计思路，我们来进行具体的实现，在 reactive 下新建 effect.js。

由于 target 是对象，所以必须得用 map 才可以把 target 作为 key 来管理数据，每次操作之前需要做非空的判断。最终把 activeEffect 存储在集合之中：

 复制代码

```
1 const targetMap = new WeakMap()
2
3 export function track(target, type, key) {
4
5   // console.log(`触发 track -> target: ${target} type:${type} key:${key}`)
6
7   // 1. 先基于 target 找到对应的 dep
8   // 如果是第一次的话，那么就需要初始化
9   // {
10    //   target1: { //depsmap
11    //     key:[effect1,effect2]
12    //   }
13    // }
14   let depsMap = targetMap.get(target)
15   if (!depsMap) {
16     // 初始化 depsMap 的逻辑
17     // depsMap = new Map()
18     // targetMap.set(target, depsMap)
19     // 上面两行可以简写成下面的
20     targetMap.set(target, (depsMap = new Map()))
21   }
22   let deps = depsMap.get(key)
23   if (!deps) {
24     deps = new Set()
25   }
```

```
26   if (!deps.has(activeEffect) && activeEffect) {
27     // 防止重复注册
28     deps.add(activeEffect)
29   }
30   depsMap.set(key, deps)
31 }
```

get 中关键的收集依赖的 track 函数我们已经讲完了，继续看 set 中关键的 trigger 函数。

trigger

有了上面 targetMap 的实现机制，trigger 函数实现的思路就是从 targetMap 中，根据 target 和 key 找到对应的依赖函数集合 deps，然后遍历 deps 执行依赖函数。

看实现的代码：

 复制代码

```
1 export function trigger(target, type, key) {
2   // console.log(`触发 trigger -> target:  type:${type} key:${key}`)
3   // 从targetMap中找到触发的函数，执行他
4   const depsMap = targetMap.get(target)
5   if (!depsMap) {
6     // 没找到依赖
7     return
8   }
9   const deps = depsMap.get(key)
10  if (!deps) {
11    return
12  }
13  deps.forEach((effectFn) => {
14
15    if (effectFn.scheduler) {
16      effectFn.scheduler()
17    } else {
18      effectFn()
19    }
20  })
21
22 }
```

可以看到执行的是 effect 的 scheduler 或者 run 函数，这是因为我们需要在 effect 函数中把依赖函数进行包装，并对依赖函数的执行时机进行控制，这是一个小的设计点。

effect

然后我们来实现 effect 函数。

下面的代码中，我们把传递进来的 fn 函数通过 effectFn 函数包裹执行，在 effectFn 函数内部，把函数赋值给全局变量 activeEffect；然后执行 fn() 的时候，就会触发响应式对象的 get 函数，get 函数内部就会把 activeEffect 存储到依赖地图中，完成依赖的收集：

[复制代码](#)

```
1 export function effect(fn, options = {}) {
2   // effect嵌套，通过队列管理
3   const effectFn = () => {
4     try {
5       activeEffect = effectFn
6       //fn执行的时候，内部读取响应式数据的时候，就能在get配置里读取到activeEffect
7       return fn()
8     } finally {
9       activeEffect = null
10    }
11  }
12  if (!options.lazy) {
13    //没有配置lazy 直接执行
14    effectFn()
15  }
16  effectFn.scheduler = options.scheduler // 调度时机 watchEffect回用到
17  return effectFn
18
19 }
```

effect 传递的函数，比如可以通过传递 lazy 和 scheduler 来控制函数执行的时机，默认是同步执行。

scheduler 存在的意义就是我们可以手动控制函数执行的时机，方便应对一些性能优化的场景，比如数据在一次交互中可能会被修改很多次，我们不想每次修改都重新执行依次 effect 函数，而是合并最终的状态之后，最后统一修改一次。

scheduler 怎么用你可以看下面的代码，我们使用数组管理传递的执行任务，最后使用 Promise.resolve 只执行最后一次，这也是 Vue 中 watchEffect 函数的大致原理。

[复制代码](#)

```
1 const obj = reactive({ count: 1 })
```

```
2 effect(() => {
3   console.log(obj.count)
4 }, {
5   // 指定调度器为 queueJob
6   scheduler: queueJob
7 })
8 // 调度器实现
9 const queue: Function[] = []
10 let isFlushing = false
11 function queueJob(job: () => void) {
12   if (!isFlushing) {
13     isFlushing = true
14     Promise.resolve().then(() => {
15       let fn
16       while(fn = queue.shift()) {
17         fn()
18       }
19     })
20   }
21 }
```

好了，绕了这么一大圈终于执行完了函数，估计你也看出来了封装了很多层。


之所以封装这么多层就是因为，Vue 的响应式本身有很多的横向扩展，除了响应式的封装，还有只读的拦截、浅层数据的拦截等等，这样，响应式系统本身也变得更加灵活和易于扩展，我们自己在设计公用函数的时候也可以借鉴类似的思路。

另一个选择 ref 函数

有了 track 和 trigger 的逻辑之后，我们用 ref 函数实现就变得非常简单了。

ref 的执行逻辑要比 reactive 要简单一些，不需要使用 Proxy 代理语法，直接使用对象语法的 getter 和 setter 配置，监听 value 属性即可。

看下面的实现，在 ref 函数返回的对象中，对象的 get value 方法，使用 track 函数去收集依赖，set value 方法中使用 trigger 函数去触发函数的执行。

 复制代码

```
1 export function ref(val) {
2   if (isRef(val)) {
3     return val
4   }
5 }
```

```
5   return new RefImpl(val)
6 }
7 export function isRef(val) {
8   return !! (val && val.__isRef)
9 }
10
11 // ref就是利用面向对象的getter和setters进行track和trigger
12 class RefImpl {
13   constructor(val) {
14     this.__isRef = true
15     this._val = convert(val)
16   }
17   get value() {
18     track(this, 'value')
19     return this._val
20   }
21
22   set value(val) {
23     if (val !== this._val) {
24       this._val = convert(val)
25       trigger(this, 'value')
26     }
27   }
28 }
29
30 // ref也可以支持复杂数据结构
31 function convert(val) {
32   return isObject(val) ? reactive(val) : val
33 }
```

你能很直观地看到，ref 函数实现的相对简单很多，只是利用面向对象的 getter 和 setter 拦截了 value 属性的读写，这也是为什么我们需要操作 ref 对象的 value 属性的原因。

值得一提的是，ref 也可以包裹复杂的数据结构，内部会直接调用 reactive 来实现，这也解决了大部分同学对 ref 和 reactive 使用时机的疑惑，现在你可以全部都用 ref 函数，ref 内部会帮你调用 reactive。

computed

Vue 中的 computed 计算属性也是一种特殊的 effect 函数，我们可以新建 computed.spec.js 来测试 computed 函数的功能，**computed 可以传递一个函数或者对象，实现计算属性的读取和修改。**比如说可以这么用：

```

1 import { ref } from '../ref'
2
3 import { reactive } from '../reactive'
4 import { computed } from '../computed'
5
6
7
8 describe('computed测试', () => {
9   it('computed基本使用', () => {
10     const ret = reactive({ count: 1 })
11     const num = ref(2)
12     const sum = computed(() => num.value + ret.count)
13     expect(sum.value).toBe(3)
14
15     ret.count++
16     expect(sum.value).toBe(4)
17     num.value = 10
18     expect(sum.value).toBe(12)
19   })
20   it('computed属性修改', () => {
21     const author = ref('大圣')
22     const course = ref('玩转Vue3')
23     const title = computed({
24       get() {
25         return author.value + ":" + course.value
26       },
27       set(val) {
28         [author.value, course.value] = val.split(':')
29       }
30     })
31     expect(title.value).toBe('大圣:玩转Vue3')
32
33     author.value = "winter"
34     course.value = "重学前端"
35     expect(title.value).toBe('winter:重学前端')
36     // 计算属性赋值
37     title.value = '王争:数据结构与算法之美'
38     expect(author.value).toBe('王争')
39     expect(course.value).toBe('数据结构与算法之美')
40
41   })
42 })

```

怎么实现呢？我们新建 computed 函数，看下面的代码，我们拦截 computed 的 value 属性，并且定制了 effect 的 lazy 和 scheduler 配置，computed 注册的函数就不会直接执行，而是要通过 scheduler 函数中对 _dirty 属性决定是否执行。

[复制代码](#)

```

1 export function computed(getterOrOptions) {

```

```
2 // getterOrOptions可以是函数，也可以是一个对象，支持get和set
3 // 还记得清单应用里的全选checkbox就是一个对象配置的computed
4 let getter, setter
5 if (typeof getterOrOptions === 'function') {
6   getter = getterOrOptions
7   setter = () => {
8     console.warn('计算属性不能修改')
9   }
10 } else {
11   getter = getterOrOptions.get
12   setter = getterOrOptions.set
13 }
14 return new ComputedRefImpl(getter, setter)
15 }
16 class ComputedRefImpl {
17   constructor(getter, setter) {
18     this._setter = setter
19     this._val = undefined
20     this._dirty = true
21     // computed就是一个特殊的effect，设置lazy和执行时机
22     this.effect = effect(getter, {
23       lazy: true,
24       scheduler: () => {
25         if (!this._dirty) {
26           this._dirty = true
27           trigger(this, 'value')
28         }
29       },
30     })
31   }
32   get value() {
33     track(this, 'value')
34     if (this._dirty) {
35       this._dirty = false
36       this._val = this.effect()
37     }
38     return this._val
39   }
40   set value(val) {
41     this._setter(val)
42   }
43 }
```

总结

最后我们来回顾一下今天学到的内容。通过手写迷你的响应式原型，我们学习了 Vue 中响应式的地位和架构。

响应式的主要功能就是可以把普通的 JavaScript 对象封装成为响应式对象，**在读取数据的时候通过 track 收集函数的依赖关系，把整个对象和 effect 注册函数的依赖关系全部存储在一个依赖图中。**

定义的 dependsMap 是一个巨大的 Map 数据，effect 函数内部读取的数据都会存储在 dependsMap 中，数据在修改的时候，通过查询 dependsMap，获得需要执行的函数，再去执行即可。

dependsMap 中存储的也不是直接存储 effect 中传递的函数，而是包装了一层对象对这个函数的执行实际进行管理，内部可以通过 active 管理执行状态，还可以通过全局变量 shouldTrack 控制监听状态，并且执行的方式也是判断 scheduler 和 run 方法，实现了对性能的提升。

我们在日常项目开发中也可以**借鉴响应式的处理思路，使用通知的机制，来调用具体数据的操作和更新逻辑**，灵活使用 effect、ref、reactive 等函数把常见的操作全部变成响应式数据处理，会极大的提高我们开发的体验和效率。

思考题

最后留一个思考题，Vue3.2 对响应式有一个性能的进一步提升，你都了解到有哪些呢？欢迎你在评论区分享自己的思考，我们下一讲再见。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 2  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 27 | 自定义渲染器：如何实现Vue的跨端渲染？

更多课程推荐

跟月影学可视化

系统掌握图形学与可视化核心原理

月影

奇虎 360 奇舞团团长

可视化 UI 框架 SpriteJS 核心开发者



新版升级：点击「👤 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

精选留言 (4)

写留言



润培

2021-12-24

刚刚提了一个PR

作者回复：给力给力



👍 2



润培

2021-12-24

相比于 Vue2 使用的 `Object.defineProperty`，Vue3 不需要提前递归收集依赖，初始化的速度更快；

Vue2 收集依赖的过程中会产生很多的 Dep 对象，Vue3 可以节省这部分的内存开销；

Vue2 无法监听数组、对象的动态添加、删除，需要通过 `$set`、`$delete`，增加学习成本；

Vue2 无法监听 Set、Map，只能处理普通对象。

展开 ∨



👍 1

**小胖**

2021-12-24

track函数哪里，依赖的添加，最后老师的判断是有问题的，那样写的话对于一个key就只能添加一次依赖。

// 测试用例改成下面这样，就过不了了

```
let val
```

```
let val2 ...
```

展开 ▾

作者回复: 已经支持这个测试案例了，把deps.add移到！dep的判断外层



👍 1

**Kim Yin**

2021-12-24

track 函数的 20~27 行好像有点问题，感觉应该把 22~26 行移到 if (!deps) 外面，不然 deps = new Set() 以后，if (!deps.has(activeEffect) && activeEffect) 这句判断就没意思了—— deps.has(activeEffect) 永远为 false

作者回复: 已经加上了这个test，deps.has判断移动到if外面了，灰常感谢提出bug



👍 1