> 本文由 简悦 SimpRead 转码， 原文地址 kaiwu.lagou.com

上节课，我们讲到了在 Vue.js 3.0 中引入 reactive API，它可以把对象数据变成响应式，所以我们着重分析 reactive API 的实现原理，并学习了收集依赖的 get 函数， 这节课我们继续来分析 reactive API 中需要关注的另一个内容——派发通知的过程。

reactive API

派发通知：set 函数

派发通知发生在数据更新的阶段，由于我们用 Proxy API 劫持了数据对象，所以当这个响应式对象属性更新的时候就会执行 set 函数。我们来看一下 set 函数的实现，它是执行 createSetter 函数的返回值：

```
function createSetter() {

return function set(target, key, value, receiver) {

const oldValue = target[key]

    value = toRaw(value)

const hadKey = hasOwn(target, key)

const result = Reflect.set(target, key, value, receiver)

if (target === toRaw(receiver)) {

if (!hadKey) {

        trigger(target, "add" , key, value)

      }

else if (hasChanged(value, oldValue)) {

        trigger(target, "set" , key, value, oldValue)

      }

    }
```

```
return result

  }

}
```

结合上述代码来看，set 函数的实现逻辑很简单，主要就做两件事情， **首先通过 Reflect.set 求值** ， **然后通过 trigger 函数派发通知** ，并依据 key 是否存在于 target 上来确定通知类型，即新增还是修改。

整个 set 函数最核心的部分就是 **执行 trigger 函数派发通知** ，下面我们将重点分析这个过程。

我们先来看一下 trigger 函数的实现，为了分析主要流程，这里省略了 trigger 函数中的一些分支逻辑：

```
const targetMap = new WeakMap()

function trigger(target, type, key, newValue) {

const depsMap = targetMap.get(target)

if (!depsMap) {

return

  }

const effects = new Set()

const add = (effectsToAdd) => {

if (effectsToAdd) {

      effectsToAdd.forEach(effect => {

        effects.add(effect)

      })

    }

  }
```

```
if (key !== void 0) {

    add(depsMap.get(key))

  }

const run = (effect) => {

if (effect.options.scheduler) {

      effect.options.scheduler(effect)

    }

else {

      effect()

    }

  }

  effects.forEach(run)

}
```

trigger 函数的实现也很简单，主要做了四件事情：

1. 通过 targetMap 拿到 target 对应的依赖集合 depsMap；
2. 创建运行的 effects 集合；
3. 根据 key 从 depsMap 中找到对应的 effects 添加到 effects 集合；
4. 遍历 effects 执行相关的副作用函数。

所以每次 trigger 函数就是根据 target 和 key ，从 targetMap 中找到相关的所有副作用函数遍历执行一遍。

在描述依赖收集和派发通知的过程中，我们都提到了一个词：副作用函数，依赖收集过程中我们把 activeEffect（当前激活副作用函数）作为依赖收集，它又是什么？接下来我们来看一下副作用函数的庐山真面目。

副作用函数

介绍副作用函数前，我们先回顾一下响应式的原始需求，即我们修改了数据就能自动执行某个函数，举个简单的例子：

```
import { reactive } from 'vue'

const counter = reactive({

  num: 0

})

function logCount() {

  console.log(counter.num)

}

function count() {

  counter.num++

}

logCount()

count()
```

可以看到，这里我们定义了响应式对象 counter，然后我们在 logCount 中访问了 counter.num，我们希望通过执行 count 函数修改 counter.num 值的时候，能自动执行 logCount 函数。

按我们之前对依赖收集过程的分析，如果这个 logCount 就是 activeEffect 的话，那么就可以实现需求，但显然是做不到的，因为代码在执行到 `console.log(counter.num)` 这一行 的时候，它对自己在 logCount 函数中的运行是一无所知的。

那么该怎么办呢？其实只要我们运行 logCount 函数前，把 logCount 赋值给 activeEffect 就好了，如下：

```
activeEffect = logCount

logCount()
```

顺着这个思路，我们可以利用高阶函数的思想，对 logCount 做一层封装，如下：

```
function wrapper(fn) {
```

```
const wrapped = function(...args) {

    activeEffect = fn

fn(...args)

  }

  return wrapped

}

const wrappedLog = wrapper(logCount)

wrappedLog()
```

这里，wrapper 本身也是一个函数，它接受 fn 作为参数，返回一个新的函数 wrapped，然后维护一个全局的 activeEffect，当 wrapped 执行的时候，把 activeEffect 设置为 fn，然后执行 fn 即可。

这样当我们执行 wrappedLog 后，再去修改 counter.num，就会自动执行 logCount 函数了。

实际上 Vue.js 3.0 就是采用类似的做法，在它内部就有一个 effect 副作用函数，我们来看一下它的实现：

```
const effectStack = []

let activeEffect

function effect(fn, options = EMPTY_OBJ) {

if (isEffect(fn)) {

    fn = fn.raw

  }

const effect = createReactiveEffect(fn, options)

if (!options.lazy) {
```

```
    effect()

  }

return effect

}

function createReactiveEffect(fn, options) {

const effect = function reactiveEffect(...args) {

if (!effect.active) {

return options.scheduler ? undefined : fn(...args)

    }

if (!effectStack.includes(effect)) {

      cleanup(effect)

try {

        enableTracking()

        effectStack.push(effect)

        activeEffect = effect

return fn(...args)

      }

finally {

        effectStack.pop()

        resetTracking()
```

```
        activeEffect = effectStack[effectStack.length - 1]
      }
    }
  }
  effect.id = uid++
  effect._isEffect = true
  effect.active = true
  effect.raw = fn
  effect.deps = []
  effect.options = options
return effect
}
```

结合上述代码来看，effect 内部通过执行 createReactiveEffect 函数去创建一个新的 effect 函数，为了和外部的 effect 函数区分，我们把它称作 reactiveEffect 函数，并且还给它添加了一些额外属性（我在注释中都有标明）。另外，effect 函数还支持传入一个配置参数以支持更多的 feature，我们这里就不展开了，在后续的章节会详细分析。

接着说，这个 reactiveEffect 函数就是响应式的副作用函数，当执行 trigger 过程派发通知的时候，执行的 effect 就是它。

按我们之前的分析，这个 reactiveEffect 函数只需要做两件事情：**把全局的 activeEffect 指向它**，**然后执行被包装的原始函数 fn 即可**。

但实际上它的实现要更复杂一些，首先它会判断 effect 的状态是否是 active，这其实是一种控制手段，允许在非 active 状态且非调度执行情况，则直接执行原始函数 fn 并返回，在后续学习完侦听器后你会对它的理解更加深刻。

接着判断 effectStack 中是否包含 effect，如果没有就把 effect 压入栈内。之前我们提到，只要设置 activeEffect = effect 即可，那么这里为什么要设计一个栈的结构呢？

其实是考虑到以下这样一个嵌套 effect 的场景：

```
import { reactive} from 'vue'

import { effect } from '@vue/reactivity'

const counter = reactive({

num: 0,

num2: 0

})

function logCount() {

effect(logCount2)

console.log('num:', counter.num)

}

function count() {

counter.num++

}

function logCount2() {

console.log('num2:', counter.num2)

}

effect(logCount)

count()
```

我们每次执行 effect 函数时，如果仅仅把 reactiveEffect 函数赋值给 activeEffect，那么针对这种嵌套场景，执行完 effect(logCount2) 后，activeEffect 还是 effect(logCount2) 返回的 reactiveEffect 函数，这样后续访问 counter.num 的时候，依赖收集对应的 activeEffect 就不对了，此时我们外部执行 count 函数修改 counter.num 后执行的便不是 logCount 函数，而是 logCount2 函数，最终输出的结果如下：

而我们期望的结果应该如下：

```
num2: 0


num: 0


num2: 0


num: 1
```

因此针对嵌套 effect 的场景，我们不能简单地赋值 activeEffect，应该考虑到函数的执行本身就是一种入栈出栈操作，因此我们也可以设计一个 effectStack，这样每次进入 reactiveEffect 函数就先把它入栈，然后 activeEffect 指向这个 reactiveEffect 函数，接着在 fn 执行完毕后出栈，再把 activeEffect 指向 effectStack 最后一个元素，也就是外层 effect 函数对应的 reactiveEffect。

这里我们还注意到一个细节，**在入栈前会执行 cleanup 函数清空 reactiveEffect 函数对应的依赖**。在执行 track 函数的时候，除了收集当前激活的 effect 作为依赖，还通过 activeEffect.deps.push(dep) 把 dep 作为 activeEffect 的依赖，这样在 cleanup 的时候我们就可以找到 effect 对应的 dep 了，然后把 effect 从这些 dep 中删除。cleanup 函数的代码如下所示：

```
function cleanup(effect) {


const { deps } = effect


if (deps.length) {


for (let i = 0; i < deps.length; i++) {


      deps[i].delete(effect)


    }


    deps.length = 0


  }


}
```

为什么需要 cleanup 呢？如果遇到这种场景：

```
<template>

<div v-if="state.showMsg">

    {{ state.msg }}

</div>

<div v-else>

    {{ Math.random()}}

</div>

<button @click="toggle">Toggle Msg</button>

<button @click="switchView">Switch View</button>

</template>

<script>

import { reactive } from 'vue'

export default {

    setup() {

const state = reactive({

msg: 'Hello World',

showMsg: true

      })

function toggle() {
```

```
        state.msg = state.msg === 'Hello World' ? 'Hello Vue' : 'Hello World'

      }

function switchView() {

        state.showMsg = !state.showMsg

      }

return {

        toggle,

        switchView,

        state

      }

    }

  }

</script>
```

结合代码可以知道，这个组件的视图会根据 showMsg 变量的控制显示 msg 或者一个随机数，当我们点击 Switch View 的按钮时，就会修改这个变量值。

假设没有 cleanup，在第一次渲染模板的时候，activeEffect 是组件的副作用渲染函数，因为模板 render 的时候访问了 state.msg，所以会执行依赖收集，把副作用渲染函数作为 state.msg 的依赖，我们把它称作 render effect。然后我们点击 Switch View 按钮，视图切换为显示随机数，此时我们再点击 Toggle Msg 按钮，由于修改了 state.msg 就会派发通知，找到了 render effect 并执行，就又触发了组件的重新渲染。

但这个行为实际上并不符合预期，因为当我们点击 Switch View 按钮，视图切换为显示随机数的时候，也会触发组件的重新渲染，但这个时候视图并没有渲染 state.msg，所以对它的改动并不应该影响组件的重新渲染。

因此在组件的 render effect 执行之前，如果通过 cleanup 清理依赖，我们就可以删除之前 state.msg 收集的 render effect 依赖。这样当我们修改 state.msg 时，由于已经没有依赖了就不会触发组件的重新渲染，符合预期。

至此，我们从 reactive API 入手了解了整个响应式对象的实现原理。除了 reactive API，Vue.js 3.0 还提供了其他好用的响应式 API，接下来我们一起分析一些常用的。

readonly API

如果用 const 声明一个对象变量，虽然不能直接对这个变量赋值，但我们可以修改它的属。如果我们希望创建只读对象，不能修改它的属性，也不能给这个对象添加和删除属性，让它变成一个真正意义上的只读对象。

```
const original = {


  foo: 1


}


const wrapped = readonly(original)


wrapped.foo = 2
```

显然，想实现上述需求就需要劫持对象，于是 Vue.js 3.0 在 reactive API 的基础上，设计并实现了 readonly API。

我们先来看一下 readonly 的实现：

```
function readonly(target) {


return createReactiveObject(target, true, readonlyHandlers,
readonlyCollectionHandlers)


}


function createReactiveObject(target, isReadonly, baseHandlers,
collectionHandlers) {


if (!isObject(target)) {


if ((process.env.NODE_ENV !== 'production')) {


      console.warn(`value cannot be made reactive: ${String(target)}`)


    }


return target
```

```
  }

if (target.__v_raw && !(isReadonly && target.__v_isReactive)) {

return target

  }

if (hasOwn(target, isReadonly ? "__v_readonly"  : "__v_reactive" )) {

return isReadonly ? target.__v_readonly : target.__v_reactive

  }

if (!canObserve(target)) {

return target

  }

const observed = new Proxy(target, collectionTypes.has(target.constructor) ?
collectionHandlers : baseHandlers)

  def(target, isReadonly ? "__v_readonly"  : "__v_reactive" , observed)

return observed

}
```

其实 readonly 和 reactive 函数的主要区别，就是执行 createReactiveObject 函数时的参数 isReadonly 不同。

我们来看这里的代码，首先 isReadonly 变量为 true，所以在创建过程中会给原始对象 target 打上一个 __v_readonly 的标识。另外还有一个特殊情况，如果 target 已经是一个 reactive 对象，就会把它继续变成一个 readonly 响应式对象。

其次就是 baseHandlers 的 collectionHandlers 的区别，我们这里仍然只关心基本数据类型的 Proxy 处理器对象，readonly 函数传入的 baseHandlers 值是 readonlyHandlers。

接下来，我们来看一下其中 readonlyHandlers 的实现：

```
const readonlyHandlers = {

  get: readonlyGet,

  has,

  ownKeys,

  set(target, key) {

if ((process.env.NODE_ENV !== 'production')) {

      console.warn(`Set operation on key "${String(key)}" failed: target is
readonly.`, target)

    }

return true

  },

  deleteProperty(target, key) {

if ((process.env.NODE_ENV !== 'production')) {

      console.warn(`Delete operation on key "${String(key)}" failed: target is
readonly.`, target)

    }

return true

  }

}
```

readonlyHandlers 和 mutableHandlers 的区别主要在 get、set 和 deleteProperty 三个函数上。很显然，作为一个只读的响应式对象，是不允许修改属性以及删除属性的，所以在非生产环境下 set 和 deleteProperty 函数的实现都会报警告，提示用户 target 是 readonly 的。

接下来我们来看一下其中 readonlyGet 的实现，它其实就是 createGetter(true) 的返回值：

```
function createGetter(isReadonly = false) {

return function get(target, key, receiver) {

    !isReadonly && track(target, "get" , key)

return isObject(res)

      ? isReadonly

        ?

        readonly(res)

        : reactive(res)

      : res

  }

}
```

可以看到，它和 reactive API 最大的区别就是不做依赖收集了，这一点也非常好理解，因为它的属性不会被修改，所以就不用跟踪它的变化了。

到这里，readonly API 就介绍完了，接下来我们分析一下另一个常用的响应式 API：ref。

ref API

通过前面的分析，我们知道 reactive API 对传入的 target 类型有限制，必须是对象或者数组类型，而对于一些基础类型（比如 String、Number、Boolean）是不支持的。

但是有时候从需求上来说，可能我只希望把一个字符串变成响应式，却不得不封装成一个对象，这样使用上多少有一些不方便，于是 Vue.js 3.0 设计并实现了 ref API。

```
const msg = ref('Hello World')


msg.value = 'Hello Vue'
```

我们先来看一下 ref 的实现：

```
function ref(value) {

return createRef(value)

}

const convert = (val) => isObject(val) ? reactive(val) : val

function createRef(rawValue) {

if (isRef(rawValue)) {

return rawValue

  }

  let value = convert(rawValue)

const r = {

    __v_isRef: true,

get value() {

      track(r, "get" , 'value')

return value

    },

set value(newVal) {

if (hasChanged(toRaw(newVal), rawValue)) {

        rawValue = newVal

        value = convert(newVal)

        trigger(r, "set" , 'value', void 0)
```

```
      }

    }

  }

return r

}
```

可以看到，函数首先处理了嵌套 ref 的情况，如果传入的 rawValue 也是 ref，那么直接返回。

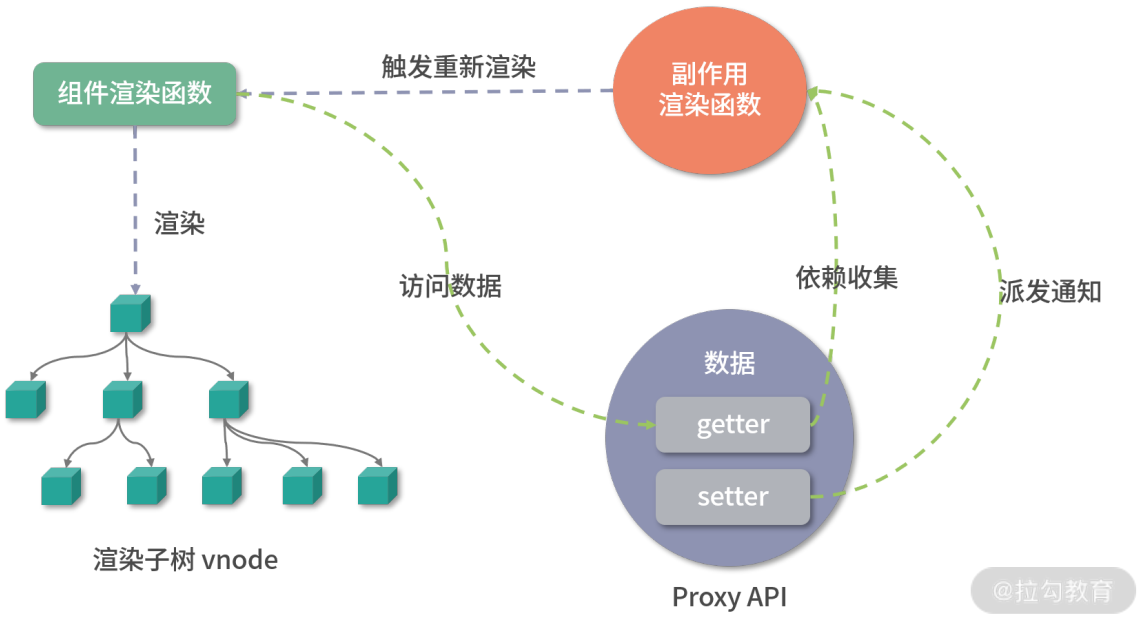
接着对 rawValue 做了一层转换，如果 rawValue 是对象或者数组类型，那么把它转换成一个 reactive 对象。

最后定义一个对 value 属性做 getter 和 setter 劫持的对象并返回，get 部分就是执行 track 函数做依赖收集然后返回它的值；set 部分就是设置新值并且执行 trigger 函数派发通知。

总结

好的，到这里我们这一节的学习也要结束啦，我希望通过这节课的学习，你能搞明白响应式 API 的实现原理，知道什么时候收集依赖，什么时候派发更新，以及副作用函数的作用和设计原理。我还希望你能知道 reactive、readonly、ref 三种 API 的区别和各自的使用场景，这样你就可以在今后的开发中对它们应用自如啦。

最后我们通过一张图来看一下整个响应式 API 实现和组件更新的关系：



这幅图是不是很眼熟？没错，它和前面 Vue.js 2.x 的响应式原理图很接近，其实 Vue.js 3.0 在响应式的实现思路和 Vue.js 2.x 差别并不大，主要就是 **劫持数据的方式改成用 Proxy 实现**， **以及收集的依赖由 watcher 实例变成了组件副作用渲染函数** 。

最后，给你留一道思考题目，为什么说 Vue.js 3 的响应式 API 实现和 Vue.js 2.x 相比性能要好，具体好在哪里呢？它又有哪些不足呢？欢迎你在留言区与我分享。

> **本节课的相关代码在源代码中的位置如下：**
> packages/reactivity/src/baseHandlers.ts
> packages/reactivity/src/effect.ts
> packages/reactivity/src/reactive.ts
> packages/reactivity/src/ref.ts