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

在平时的开发工作中，我们经常使用侦听器帮助我们去观察某个数据的变化然后去执行一段逻辑。

在 Vue.js 2.x 中，你可以通过 watch 选项去初始化一个侦听器，称作 watcher：

```
export default {

  watch: {

    a(newVal, oldVal) {

      console.log('new: %s,00 old: %s', newVal, oldVal)

    }

  }

}
```

当然你也可以通过 $watch API 去创建一个侦听器：

```
const unwatch = vm.$watch('a', function(newVal, oldVal) {

  console.log('new: %s, old: %s', newVal, oldVal)

})
```

与 watch 选项不同，通过 $watch API 创建的侦听器 watcher 会返回一个 unwatch 函数，你可以随时执行它来停止这个 watcher 对数据的侦听，而对于 watch 选项创建的侦听器，它会随着组件的销毁而停止对数据的侦听。

在 Vue.js 3.0 中，虽然你仍可以使用 watch 选项，但针对 Composition API，Vue.js 3.0 提供了 watch API 来实现侦听器的效果。

那么，接下来就随我一起来学习 watch API 吧。

watch API 的用法

我们先来看 Vue.js 3.0 中 watch API 有哪些用法。

1.watch API 可以**侦听一个 getter 函数**，但是它必须返回一个响应式对象，当该响应式对象更新后，会执行对应的回调函数。

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

const state = reactive({ count: 0 })

watch(() => state.count, (count, prevCount) => {

})
```

2.watch API 也可以直接**侦听一个响应式对象**，当响应式对象更新后，会执行对应的回调函数。

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

const count = ref(0)

watch(count, (count, prevCount) => {

})
```

3.watch API 还可以直接**侦听多个响应式对象**，任意一个响应式对象更新后，就会执行对应的回调函数。

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

const count = ref(0)

const count2 = ref(1)

watch([count, count2], ([count, count2], [prevCount, prevCount2]) => {

})
```

watch API 实现原理

侦听器的言下之意就是，当侦听的对象或者函数发生了变化则自动执行某个回调函数，这和我们前面说过的副作用函数 effect 很像， 那它的内部实现是不是依赖了 effect 呢？带着这个疑问，我们来探究 watch API 的具体实现：

```
function watch(source, cb, options) {

if ((process.env.NODE_ENV !== 'production') && !isFunction(cb)) {
```

```
    warn(`\`watch(fn, options?)\` signature has been moved to a separate API. ` +
      `Use \`watchEffect(fn, options?)\` instead. \`watch\` now only ` +
      `supports \`watch(source, cb, options?) signature.`)
  }
return doWatch(source, cb, options)
}
function doWatch(source, cb, { immediate, deep, flush, onTrack, onTrigger } = EMPTY_OBJ) {
}
```

从代码中可以看到，watch 函数内部调用了 doWatch 函数，调用前会在非生产环境下判断第二个参数 cb 是不是一个函数，如果不是则会报警告以告诉用户应该使用 watchEffect(fn, options) API，watchEffect API 也是侦听器相关的 API，稍后我们会详细介绍。

这个 doWatch 函数很长，所以我只贴出了需要理解的部分，我用注释将这个函数的实现逻辑拆解成了几个步骤。可以看到，内部确实创建了 effect 副作用函数。接下来，就随我一步步看它具体做了哪些事情吧。

标准化 source

我们先来看 watch 函数的第一个参数 source。

通过前文知道 source 可以是 getter 函数，也可以是响应式对象甚至是响应式对象数组，所以我们需要标准化 source，这是**标准化 source 的流程**：

```
const warnInvalidSource = (s) => {
  warn(`Invalid watch source: `, s, `A watch source can only be a getter/effect function, a ref, ` +
    `a reactive object, or an array of these types.`)
}
const instance = currentInstance
```

```
let getter

if (isArray(source)) {

  getter = () => source.map(s => {

if (isRef(s)) {

return s.value

    }

else if (isReactive(s)) {

return traverse(s)

    }

else if (isFunction(s)) {

return callWithErrorHandling(s, instance, 2 )

    }

else {

      (process.env.NODE_ENV !== 'production') && warnInvalidSource(s)

    }

  })

}

else if (isRef(source)) {

  getter = () => source.value
```

```
}

else if (isReactive(source)) {

  getter = () => source

  deep = true

}

else if (isFunction(source)) {

if (cb) {

    getter = () => callWithErrorHandling(source, instance, 2 )

  }

else {

  }

}

else {

  getter = NOOP

  (process.env.NODE_ENV !== 'production') && warnInvalidSource(source)

}

if (cb && deep) {

const baseGetter = getter

  getter = () => traverse(baseGetter())

}
```

其实，source 标准化主要是根据 source 的类型，将其变成 ~标准成~ getter 函数。具体来说：

1. 如果 source 是 ref 对象，则创建一个访问 source.value 的 getter 函数;
2. 如果 source 是 reactive 对象，则创建一个访问 source 的 getter 函数，并设置 deep 为 true（deep 的作用我稍后会说）；
3. 如果 source 是一个函数，则会进一步判断第二个参数 cb 是否存在，对于 watch API 来说，cb 是一定存在且是一个回调函数，这种情况下，getter 就是一个简单的对 source 函数封装的函数。

如果 source 不满足上述条件，则在非生产环境下报警告，提示 source 类型不合法。

我们来看一下最终标准化生成的 getter 函数，它会返回一个响应式对象，在后续创建 effect runner 副作用函数需要用到，每次执行 runner 就会把 getter 函数返回的响应式对象作为 watcher 求值的结果，effect runner 的创建流程我们后续会详细分析，这里不需要深入了解。

最后我们来关注一下 deep 为 true 的情况。此时，我们会发现生成的 getter 函数会被 traverse 函数包装一层。traverse 函数的实现很简单，即通过递归的方式访问 value 的每一个子属性。那么，为什么要递归访问每一个子属性呢?

其实 deep 属于 watcher 的一个配置选项，Vue.js 2.x 也支持，表面含义是深度侦听，实际上是通过遍历对象的每一个子属性来实现。举个例子你就明白了：

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

const state = reactive({

  count: {

    a: {

      b: 1

    }

  }

})

watch(state.count, (count, prevCount) => {

  console.log(count)

})

state.count.a.b = 2
```

这里，我们利用 reactive API 创建了一个嵌套层级较深的响应式对象 state，然后再调用 watch API 侦听 state.count 的变化。接下来我们修改内部属性 state.count.a.b 的值，你会发现 watcher 的回调函数执行了，为什么会执行呢？

学过响应式章节，我们知道只有对象属性先被访问触发了依赖收集，再去修改这个属性，才可以通知对应的依赖更新。而从上述业务代码来看，我们修改 state.count.a.b 的值时并没有访问它，但还是触发了 watcher 的回调函数。

根本原因是，当我们执行 watch 函数的时候，我们知道如果侦听的是一个 reactive 对象，那么内部会设置 deep 为 true，然后执行 traverse 去递归访问对象深层子属性，这个时候就会访问 state.count.a.b 触发依赖收集，这里收集的依赖是 watcher 内部创建的 effect runner。因此，当我们再去修改 state.count.a.b 的时候，就会通知这个 effect，所以最终会执行 watcher 的回调函数。

当我们侦听一个通过 reactive API 创建的响应式对象时，内部会执行 traverse 函数，如果这个对象非常复杂，比如嵌套层级很深，那么递归 traverse 就会有一定的性能耗时。因此如果我们需要侦听这个复杂响应式对象内部的某个具体属性，就可以想办法减少 traverse 带来的性能损耗。

比如刚才的例子，我们就可以直接侦听 state.count.a.b 的变化：

```
watch(state.count.a, (newVal, oldVal) => {


console.log(newVal)


})


state.count.a.b = 2
```

这样就可以减少内部执行 traverse 的次数。你可能会问，直接侦听 state.count.a.b 可以吗？答案是不行，因为 state.count.a.b 已经是一个基础数字类型了，不符合 source 要求的参数类型，所以会在非生产环境下报警告。

那么有没有办法优化使得 traverse 不执行呢？答案是可以的。我们可以侦听一个 getter 函数：

```
watch(() => state.count.a.b, (newVal, oldVal) => {


console.log(newVal)


})


state.count.a.b = 2
```

这样函数内部会访问并返回 state.count.a.b，一次 traverse 都不会执行并且依然可以侦听到它的变化从而执行 watcher 的回调函数。

构造回调函数

处理完 watch API 第一个参数 source 后，接下来处理第二个参数 cb。

cb 是一个回调函数，它有三个参数：第一个 newValue 代表新值；第二个 oldValue 代表旧值。第三个参数 onInvalidate，我打算放在后面介绍。

其实这样的 API 设计非常好理解，即侦听一个值的变化，如果值变了就执行回调函数，回调函数里可以访问到新值和旧值。

接下来我们来看一下构造回调函数的处理逻辑：

```
let cleanup

const onInvalidate = (fn) => {

  cleanup = runner.options.onStop = () => {

    callWithErrorHandling(fn, instance, 4 )

  }

}

let oldValue = isArray(source) ? [] : INITIAL_WATCHER_VALUE

const applyCb = cb

  ? () => {

if (instance && instance.isUnmounted) {

return

    }

const newValue = runner()

if (deep || hasChanged(newValue, oldValue)) {

if (cleanup) {

        cleanup()

      }
```

```
      callWithAsyncErrorHandling(cb, instance, 3 , [

        newValue,

        oldValue === INITIAL_WATCHER_VALUE ? undefined : oldValue,

        onInvalidate

      ])

      oldValue = newValue

    }

  }

  : void 0
```

onInvalidate 函数用来注册无效回调函数，我们暂时不需要关注它，我们需要重点来看 applyCb。 这个函数实际上就是对 cb 做一层封装，当侦听的值发生变化时就会执行 applyCb 方法，我们来分析一下它的实现。

首先，watch API 和组件实例相关，因为通常我们会在组件的 setup 函数中使用它，当组件销毁后，回调函数 cb 不应该被执行而是直接返回。

接着，执行 runner 求得新值，这里实际上就是执行前面创建的 getter 函数求新值。

最后进行判断，如果是 deep 的情况或者新旧值发生了变化，则执行回调函数 cb，传入参数 newValue 和 oldValue。注意，第一次执行的时候旧值的初始值是空数组或者 undefined。执行完回调函数 cb 后，把旧值 oldValue 再更新为 newValue，这是为了下一次的比对。

创建 scheduler

接下来我们要分析创建 scheduler 过程。

scheduler 的作用是根据某种调度的方式去执行某种函数，在 watch API 中，主要影响到的是回调函数的执行方式。我们来看一下它的实现逻辑：

```
const invoke = (fn) => fn()

let scheduler

if (flush === 'sync') {

  scheduler = invoke
```

```
}

else if (flush === 'pre') {

  scheduler = job => {

if (!instance || instance.isMounted) {

      queueJob(job)

    }

else {

      job()

    }

  }

}

else {

  scheduler = job => queuePostRenderEffect(job, instance && instance.suspense)

}
```

Watch API 的参数除了 source 和 cb，还支持第三个参数 options，不同的配置决定了 watcher 的不同行为。前面我们也分析了 deep 为 true 的情况，除了 source 为 reactive 对象时会默认把 deep 设置为 true，你也可以主动传入第三个参数，把 deep 设置为 true。

这里，scheduler 的创建逻辑受到了第三个参数 Options 中的 flush 属性值的影响，不同的 flush 决定了 watcher 的执行时机。

- 当 flush 为 sync 的时候，表示它是一个同步 watcher，即当数据变化时同步执行回调函数。
- 当 flush 为 pre 的时候，回调函数通过 queueJob 的方式在组件更新之前执行，如果组件还没挂载，则同步执行确保回调函数在组件挂载之前执行。
- 如果没设置 flush，那么回调函数通过 queuePostRenderEffect 的方式在组件更新之后执行。

queueJob 和 queuePostRenderEffect 在这里不是重点，所以我们放到后面介绍。总之，你现在要记住，**watcher 的回调函数是通过一定的调度方式执行的。**

创建 effect

前面的分析我们提到了 runner，它其实就是 watcher 内部创建的 effect 函数，接下来，我们来分析它逻辑：

```
const runner = effect(getter, {

  lazy: true,

  computed: true,

  onTrack,

  onTrigger,

  scheduler: applyCb ? () => scheduler(applyCb) : scheduler

})

recordInstanceBoundEffect(runner)

if (applyCb) {

if (immediate) {

    applyCb()

  }

else {

    oldValue = runner()

  }

}

else {
```

```
  runner()

}
```

这块代码逻辑是整个 watcher 实现的核心部分，即通过 effect API 创建一个副作用函数 runner，我们需要关注以下几点。

- **runner 是一个 computed effect。**因为 computed effect 可以优先于普通的 effect（比如组件渲染的 effect）先运行，这样就可以实现当配置 flush 为 pre 的时候，watcher 的执行可以优先于组件更新。
- **runner 执行的方式。**runner 是 lazy 的，它不会在创建后立刻执行。第一次手动执行 runner 会执行前面的 getter 函数，访问响应式数据并做依赖收集。注意，此时 activeEffect 就是 runner，这样在后面更新响应式数据时，就可以触发 runner 执行 scheduler 函数，以一种调度方式来执行回调函数。
- **runner 的返回结果。**手动执行 runner 就相当于执行了前面标准化的 getter 函数，getter 函数的返回值就是 watcher 计算出的值，所以我们第一次执行 runner 求得的值可以作为 oldValue。
- **配置了 immediate 的情况。**当我们配置了 immediate，创建完 watcher 会立刻执行 applyCb 函数，此时 oldValue 还是初始值，在 applyCb 执行时也会执行 runner 进而执行前面的 getter 函数做依赖收集，求得新值。

返回销毁函数

最后，会返回侦听器销毁函数，也就是 watch API 执行后返回的函数。我们可以通过调用它来停止 watcher 对数据的侦听。

```
return () => {

  stop(runner)

if (instance) {

    remove(instance.effects, runner)

  }

}

function stop(effect) {

if (effect.active) {

    cleanup(effect)
```

```
if (effect.options.onStop) {

      effect.options.onStop()

    }

    effect.active = false

  }

}
```

销毁函数内部会执行 stop 方法让 runner 失活，并清理 runner 的相关依赖，这样就可以停止对数据的侦听。并且，如果是在组件中注册的 watcher，也会移除组件 effects 对这个 runner 的引用。

好了，到这里我们对 watch API 的分析就可以告一段落了。侦听器的内部设计很巧妙，我们可以侦听响应式数据的变化，内部创建 effect runner，首次执行 runner 做依赖收集，然后在数据发生变化后，以某种调度方式去执行回调函数。

> **本节课的相关代码在源代码中的位置如下：**
> packages/runtime-core/src/apiWatch.ts