

全部开发者教程

Vue3 源码分析与构建方案

08：总结

第七章：响应系统 - computed && watch

01：开篇

02：源码阅读：computed 的响应性

03：框架实现：构建 ComputedRefImpl，读取计算属性的值

04：框架实现：computed 的响应性：初见调度器，处理脏的状态

05：框架实现：computed 的缓存

06：总结：computed 计算属性

07：源码阅读：响应性的数据监听器 watch，跟踪源码实现逻辑



Sunday • 更新于 2022-10-19

上一节 04：框架实现：... 06：总结：com... 下一节

05：框架实现：computed 的缓存性

我们知道 `computed` 区别于 `function` 最大的地方就是：**computed 具备缓存**，当多次触发计算实行时，那么计算属性只会计算 **一次**。

那么秉承着这样的理念，我们来创建一个测试用例：

1. 创建 `packages/vue/examples/reactivity/computed-cache.html`：

<> 代码块

```
1 <script>
2   const { reactive, computed, effect } = Vue
3
4   const obj = reactive({
5     name: '张三'
6   })
7
8   const computedObj = computed(() => {
9     console.log('计算属性执行计算');
10    return '姓名: ' + obj.name
11  })
12
13  effect(() => {
14    document.querySelector('#app').innerHTML = computedObj.value
15    document.querySelector('#app').innerHTML = computedObj.value
16  })
17
18  setTimeout(() => {
19    obj.name = '李四'
20  }, 2000);
21 </script>
```

运行到浏览器，我们发现当前代码出现了 **死循环** 的问题。

那么这个 **死循环** 是因为什么呢？

如果我们想要实现计算属性的缓存性，又应该如何进行实现呢？

带着这两个问题，我们继续来往下看。

为什么会出现死循环

我们为当前的代码进行 `debugger`，查看出现该问题的原因。我们知道这个死循环是在 **延迟两秒后** 出现的，而延迟两秒之后是 `obj.name` 的调用，即：`reactive` 的 `getter` 行为被触发，也就是 `trigger` 方法触发时：

- 为 `packages/reactivity/src/effect.ts` 中的 `trigger` 方法增加断点，延迟两秒之后，进入断点：
- 此时执行的代码是 `obj.name = '李四'`，所以在 `target` 为 `{name: '李四'}`
- 但是要注意，此时 `targetMap` 中，已经在 **收集过** `effect` 了，此时的 `dep` 中包含一个 **计算属性**的 `effect`：

索引目录

- 05：框架实现：c
- 为什么会出现死
- 如何解决死循环
- 总结



```

this: undefined
▼ dep: Set(1)
  ▼ [[Entries]]
    ▼ 0: ReactiveEffect
      ▼ value: ReactiveEffect
        ▶ computed: ComputedRefImpl {dep: Set(2), __v_isRef: true, _dirty: false,
          fn: () => {
            console.log('计算属性执行计算');
            return '姓名: ' + obj.name;
          }
        ▶ scheduler: f ()
        ▶ [[Prototype]]: Object
      size: 1

```

4. 代码继续向下进行，进入 `triggerEffects(dep)` 方法
5. 在 `triggerEffects(dep)` 方法中，继续进入 `triggerEffect(effect)`
6. 在 `triggerEffect` 中接收到的 `effect`，即为刚才查看的 **计算属性的** `effect`：

```

this: undefined
▼ effect: ReactiveEffect
  ▼ computed: ComputedRefImpl
    ▶ dep: Set(2) {ReactiveEffect, ReactiveEffect}
    ▶ effect: ReactiveEffect {computed: ComputedRefImpl, fn: f, scheduler: f}
    _v_isRef: true
    _dirty: false
    _value: "姓名: 张三"
    value: "姓名: 张三"
    ▶ [[Prototype]]: Object
  fn: () => {
    console.log('计算属性执行计算');
    return '姓名: ' + obj.name;
  }
  ▶ scheduler: f ()
  ▶ [[Prototype]]: Object

```

7. 此时因为 `effect` 中存在 `scheduler`，所以会执行该计算属性的 `scheduler` 函数，在 `scheduler` 函数中，会触发 `triggerRefValue(this)`，而 `triggerRefValue` 则会再次触发 `triggerEffects`。
8. 特别注意：此时 `effects` 的值为 **计算属性实例的** `dep`：

```

▼ effects: Array(2)
  ▶ 0: ReactiveEffect {scheduler: null, fn: f}
  ▶ 1: ReactiveEffect {computed: ComputedRefImpl, fn: f, scheduler: f}
  length: 2
  ▶ [[Prototype]]: Array(0)
effects_1: undefined
effects_1_1: undefined
_a: undefined

```

9. 循环 `effects`，从而再次进入 `triggerEffect` 中。
10. 再次进入 `triggerEffect`，此时 `effect` 为 **非计算属性的** `effect`，即 `fn` 函数：

```

▼ effect: ReactiveEffect
  fn: () => {
    document.querySelector('#app').innerHTML = computedObj.value
    document.querySelector('#app').innerHTML = computedObj.value
  }
  scheduler: null
  ▶ [[Prototype]]: Object

```

11. 因为他 **不是** 计算属性的 `effect`，所以会直接执行 `run` 方法。
12. 而我们知道 `run` 方法中，其实就是触发了 `fn` 函数，所以最终会执行：

<> 代码块

```

1  () => {
2    document.querySelector('#app').innerHTML = computedObj.value
3    document.querySelector('#app').innerHTML = computedObj.value
4  }

```

13. 但是在这个 `fn` 函数中，是有触发 `computedObj.value` 的，而 `computedObj.value` 其实是触发了 `c` 的 `get value` 方法

意见反馈

收藏教程

标记书签

14. 那么这次 run 的执行会触发 **两次** computed 的 get value

15. 1. 第一次进入:

1. 进入 computed 的 get value :
2. 首先收集依赖
3. 接下来检查 dirty 脏的状态, 执行 this.effect.run()!
4. 获取最新值, 返回

2. 第二次进入:

1. 进入 computed 的 get value :
2. 首先收集依赖
3. 接下来检查 dirty 脏的状态, **因为在上一次中 dirty 已经为 false **, 所以本次 **不会在触发 this.effect.run()!**
4. 直接返回结束

16. **按说代码应该到这里就结束了, **但是不要忘记, 在刚才我们进入到 triggerEffects 时, effects 是一个数组, 内部还存在一个 computed 的 effect, 所以代码会 **继续** 执行, 再次来到 triggerEffect 中:

1. 此时 effect 为 computed 的 effect:

```
▼ effect: ReactiveEffect
  > computed: ComputedRefImpl {dep: Set(2), __v_isRef: true, _dirty: false, e
  > fn: () => {...}
  > scheduler: f ()
  > [[Prototype]]: Object
```

2. 这会导致, 再次触发 scheduler,
3. scheduler 中还会再次触发 triggerRefValue
4. triggerRefValue 又触发 triggerEffects, **再次生成一个新的 effects 包含两个 effect**, 就像 **第七步** 一样
5. 从而导致 **死循环**

以上逻辑就是为什么会出现死循环的原因。

那么明确好了导致死循环的代码逻辑之后, 接下来就是如何解决这个死循环的问题呢?

PS: 这里大家要注意: vue-next-mini 是一个学习 vue 3 核心源代码的库, 所以它在一些复杂业务中会存在各种 bug。而这样的 bug 在 vue3 的源码中处理完善的逻辑非常非常复杂, 我们不可能完全按照 vue 3 的标准来处理。

所以我们秉承着 **最少代码的实现逻辑** 来解决对应的 bug, 它 **并不是一个完善的方案** (相比于 vue 3 源代码), 但是我们可以保证它是 vue 3 的源逻辑, 并且是合理的!

如何解决死循环

想要解决这个死循环的问题, 其实比较简单, 我们只需要在 packages/reactivity/src/effect.ts 中的 triggerEffects 中修改如下代码:

<> 代码块

```
1 export function triggerEffects(dep: Dep) {
2   // 把 dep 构建为一个数组
3   const effects = isArray(dep) ? dep : [...dep]
4   // 依次触发
5   // for (const effect of effects) {
6   //   triggerEffect(effect)
7   // }
8
9   // 不在依次触发, 而是先触发所有的计算属性依赖, 再触发所有的非计算属性依赖
10  for (const effect of effects) {
```

意见反馈

收藏教程

标记书签



```

13         }
14     }
15     for (const effect of effects) {
16         if (!effect.computed) {
17             triggerEffect(effect)
18         }
19     }
20 }

```

那么为什么这样就可以解决死循环的 bug 呢？

我们再按照刚才的顺序跟踪下代码进行查看：

1. 为 packages/reactivity/src/effect.ts 中的 trigger 方法增加断点，延迟两秒之后，进入断点：
2. 此时执行的代码是 obj.name = '李四'，所以在 target 为 {name: '李四'}
3. 但是要注意，此时 targetMap 中，已经在收集过 effect 了，此时的 dep 中包含一个计算属性的 effect：
4. 代码继续向下进行，进入 triggerEffects(dep) 方法
5. 在 triggerEffects(dep) 方法中，继续进入 triggerEffect(effect)
6. 在 triggerEffect 中接收到的 effect，即为刚才查看的计算属性的 effect：

```

this: undefined
effect: ReactiveEffect
computed: ComputedRefImpl
  dep: Set(2) {ReactiveEffect, ReactiveEffect}
  effect: ReactiveEffect {computed: ComputedRefImpl, fn: f, scheduler: f}
  _v_isRef: true
  _dirty: false
  _value: "姓名: 张三"
  value: "姓名: 张三"
  [[Prototype]]: Object
  fn: () => {
    console.log('计算属性执行计算');
    return '姓名: ' + obj.name;
  }
  scheduler: f ()
  [[Prototype]]: Object

```

7. 此时因为 effect 中存在 scheduler，所以会执行该计算属性的 scheduler 函数，在 scheduler 函数中，会触发 triggerRefValue(this)，而 triggerRefValue 则会再次触发 triggerEffects
8. -----不同从这里开始-----：
9. 因为此时我们在 triggerEffects 中，增加了判断逻辑，所以永远会先触发计算属性的 effect
10. 所以此时再次进入到 triggerEffect 时，此时的 effect 依然为计算属性的 effect：

```

effect: ReactiveEffect
computed: ComputedRefImpl {dep: Set(2), _v_isRef: true, _dirty: true, effect: ReactiveEffect}
fn: () => {
  console.log('计算属性执行计算');
  return '姓名: ' + obj.name;
}
scheduler: f ()

```

11. 从而因为存在 scheduler，所以会执行：

<> 代码块

```

1  () => {
2      // 判断当前脏的状态，如果为 false，表示需要《触发依赖》
3      if (!this._dirty) {
4          // 将脏置为 true，表示
5          this._dirty = true
6          triggerRefValue(this)
7      }
8  })

```

意见反馈

收藏教程

标记书签

12. 但是此时要注意：此时 `_dirty` 脏的状态为 `true`，即：不会触发 `triggerRefValue` 来触发依赖，此次计算属性的 `scheduler` 调度器会直接结束
13. 然后代码跳回到 `triggerEffects` 两次循环中，使用非计算属性的 `effect` 执行 `triggerEffect` 方法
14. 本次进入 `triggerEffect` 时，`effect` 数据如下：

```
▼ effect: ReactiveEffect
  fn: () => {
    document.querySelector('#app').innerHTML = computedObj.value
    document.querySelector('#app').innerHTML = computedObj.value
  }
  scheduler: null
```

15. 那么这次 `run` 的执行会触发两次 `computed` 的 `get value`
16. 所以代码会进入到 `computed` 的 `get value` 中：

1. 第一次进入：

1. 进入 `computed` 的 `get value`：
2. 首先收集依赖
3. 接下来检查 `dirty` 脏的状态，执行 `this.effect.run()!`
4. 获取最新值，返回

2. 第二次进入：

1. 进入 `computed` 的 `get value`：
2. 首先收集依赖
3. 接下来检查 `dirty` 脏的状态，**因为在上一次中 `dirty` 已经为 `false` **，所以本次不会在触发 `this.effect.run()!`
4. 直接返回结束

所有代码逻辑结束。

查看测试实例的打印，`computed` 只计算了一次。

总结

那么到这里我们就解决了计算属性的死循环问题和缓存的问题。

其实解决的方式非常的简单，我们只需要控制 `computed` 的 `effect` 和非 `computed` 的 `effect` 的执行顺序，通过明确的 `dirty` 来控制 `run` 和 `triggerRefValue` 的执行即可。

04: 框架实现: computed 的响应性: 初见... ◀ 上一节 下一节 ▶ 06: 总结: computed 计算属性

✎ 我要提出意见反馈