

全部开发者教程

Vue 3 源码分析: 从入门到精通

响应性

08: 总结

第七章: 响应系统 - computed && watch

01: 开篇

02: 源码阅读: computed 的响应

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 01: 开篇 03: 框架实现: ... 下一节

## 02: 源码阅读: computed 的响应性, 跟踪 Vue 3 源码实现逻辑

计算属性 `computed` 会 **基于其响应式依赖被缓存**, 并且在依赖的响应式数据发生变化时 **重新计算**

那么根据计算属性的概念, 我们可以创建对应的测试实例: `packages/vue/examples/imooc/computed.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     return '姓名: ' + obj.name
10  })
11
12  effect(() => {
13    document.querySelector('#app').innerHTML = computedObj.value
14  })
15
16  setTimeout(() => {
17    obj.name = '李四'
18  }, 2000);
19 </script>
```

在以上测试实例中, 程序主要执行了 5 个步骤:

1. 使用 `reactive` 创建响应性数据
2. 通过 `computed` 创建计算属性 `computedObj`, 并且触发了 `obj` 的 `getter`
3. 通过 `effect` 方法创建 `fn` 函数
4. 在 `fn` 函数中, 触发了 `computed` 的 `getter`
5. 延迟触发了 `obj` 的 `setter`

那么在这 5 个步骤中, 有些步骤进行的操作我们是了解的, 所以我们只需要看之前没有了解过得即可。

### computed

`computed` 的代码在 `packages/reactivity/src/computed.ts` 中, 我们可以在这里为 `computed` 函数增加断点:

1. 代码进入 `computed` 函数
2. 执行 `const onlyGetter = isFunction(getterOrOptions)` 方法:
  1. `getterOrOptions` 为传入的第一个参数, 因为我们传入的为函数, 所以 `onlyGetter = true`
3. 执行: `getter = getterOrOptions`, 即: `getter` 为我们传入的函数

#### 索引目录

- 02: 源码阅读: c
- computed
- computed 的 c
- ReactiveEffect
- 总结



意见反馈

收藏教程

标记书签

5. 执行: `new ComputedRefImpl`, **创建 ComputedRefImpl 实例**。那么这里的 `ComputedRefImpl` 是什么呢?

## 6. 进入 ComputedRefImpl

1. 在构造函数中, 可以看到: **\*\*创建了 ReactiveEffect 实例\*\***, 并且传入了两个参数:

1. `getter`: 触发 `computed` 函数时, 传入的第一个参数
2. 匿名函数: 当 `this._dirty` 为 `false` 时, 会触发 `triggerRefValue`, 我们知道 `triggerRefValue` 会 **依次触发依赖**

<> 代码块

```
1  () => {
2    // _dirty 表示 “脏” 的意思, 这里可以理解为 《依赖的响应性数据发生了变化, 计算属性
3    if (!this._dirty) {
4      this._dirty = true
5      triggerRefValue(this)
6    }
7  }
```

2. 而对于 `ReactiveEffect` 而言, 我们之前也是有了解过的:

1. 它位于 `packages/reactivity/src/effect.ts` 文件中
2. 提供了一个 `run` 方法 和一个 `stop` 方法:

1. `run` 方法: 触发 `fn`, 即传入的第一个参数
2. `stop` 方法: 语义上为停止的意思, 目前咱们还没有实现

3. 生成的实例, 我们一般把它叫做 `effect`

3. 执行 `this.effect.computed = this`, 即: **effect 实例** 被挂载了一个新的属性 `computed` 为当前的 `ComputedRefImpl` 的实例。

4. `ReactiveEffect` **构造函数执行完成**

7. 在 `computed` 中返回了 `ComputedRefImpl` 实例

由以上代码可知, 当我们在执行 `computed` 函数时:

1. 定义变量 `getter` 为我们传入的回调函数
2. 生成了 `ComputedRefImpl` 实例, 作为 `computed` 函数的返回值
3. `ComputedRefImpl` 内部, 利用了 `ReactiveEffect` 函数, 并且传入了 **第二个参数**

## computed 的 getter

当 `computed` 代码执行完成之后, 我们在 `effect` 中触发了 `computed` 的 `getter`:

<> 代码块

```
1  computedObj.value
```

根据我们之前在学习 `ref` 的时候可知, `.value` 属性的调用本质上是一个 **get value 的函数调用**, 而 `computedObj` 作为 `computed` 的返回值, 本质上是 `ComputedRefImpl` 的实例, 所以此时会触发 `ComputedRefImpl` 下的 `get value` 函数。

1. 进入 `ComputedRefImpl` 下的 `get value` 函数
2. 执行 `trackRefValue(self)`, 该方法我们是有过了解的, 知道它的作用是: **收集依赖**, 它接收一个 `ref` 作为参数, 该 `ref` 本质上就是 `ComputedRefImpl` 的实例:



```
▼ ref: ComputedRefImpl
  ▶ dep: Set(1) {ReactiveEffect2}
  ▶ effect: ReactiveEffect2 {active: true, deps: Array(1), parent: und
    __v_isReadonly: true
    __v_isRef: true
    _cacheable: true
    _dirty: true
  ▶ _setter: () => {...}
    _value: "姓名: 张三"
    value: (...)
  ▶ [[Prototype]]: Object
```

3. 执行 `self._dirty = false`，我们知道 `_dirty` 是 **脏** 的意思，如果 `_dirty = true` 则会 **触发执行依赖**。在 **当前（标记为 `false` 之前）**，`self._dirty = true`
4. 所以接下来执行 `self.effect.run()!`，执行了 `run` 方法，我们知道 `run` 方法内部其实会触发 `fn` 函数，即：**computed 接收的第一个参数**
5. 接下来把 `self._value = self.effect.run()!`，此时 `self._value` 的值为 `computed` 第一个参数（`fn` 函数）的返回值，即为：**计算属性计算之后的值**
6. 最后执行 `return self._value`，返回计算的值

由以上代码可知：

1. `ComputedRefImpl` 实例本身就没有 **代理监听**，它本质上是一个 `get value` 和 `set value` 的触发
2. 在每一次 `get value` 被触发时，都会主动触发一次 **依赖收集**
3. 根据 `_dirty` 和 `_cacheable` 的状态判断，是否需要触发 `run` 函数
4. `computed` 的返回值，其实是 `run` 函数执行之后的返回值

## ReactiveEffect 的 scheduler

到现在为止，我们貌似已经分析完成了 `computed` 的源码执行逻辑，但是大家仔细来看上面的逻辑分析，可以发现，目前这样的逻辑还存在一些问题。

我们知道对于计算属性而言，当它依赖的响应式数据发生变化时，它将重新计算。那么换句话说就是：**当响应性数据触发 `setter` 时，计算属性需要触发依赖**。

在上面的代码中，我们知道，当《每一次 `get value` 被触发时，都会主动触发一次 **依赖收集**》，但是**触发依赖** 的地方在哪呢？

根据以上代码可知：在 `ComputedRefImpl` 的构造函数中，我们创建了 `ReactiveEffect` 实例，并且传递了第二个参数，该参数为一个回调函数，在这个回调函数中：我们会根据 **脏** 的状态来执行 `triggerRefValue`，即 **触发依赖**，重新计算。

那么这个 `ReactiveEffect` **第二个参数** 是什么呢？它会在什么时候被触发，以 **触发依赖** 呢？

我们来看一下：

1. 进入 `packages/reactivity/src/effect.ts` 中
2. 查看 `ReactiveEffect` 的构造函数，可以第二个参数为 `scheduler`
3. `scheduler` 表示 **调度器** 的意思，我们查看 `packages/reactivity/src/effect.ts` 中 `triggerEffect` 方法，可以发现这里进行了调度器的判定：

<> 代码块

```
1 function triggerEffect(...) {
2   ...
3   if (effect.scheduler) {
4     effect.scheduler()
5   }
6   ...
7 }
```

4. 那么接下来我们就可以跟踪一下代码的实现。

## 跟踪代码

我们知道 **延迟两秒之后** 会触发 `obj.name` 即 `reactive` 的 `setter` 行为，所以我们可以先在 `packages/r`

意见反馈

收藏教程

标记书签

1. 进入 `reactive` 的 `setter` (注意: 这里是延迟两秒之后 `setter` 行为)
2. 跳过之前的相同逻辑之后, 可知, 最后会触发: `trigger(target, TriggerOpTypes.SET, key, value, oldValue)` 方法
3. 进入 `trigger` 方法:
4. 同样跳过之前相同逻辑, 可知, 最后会触发: `triggerEffects(deps[0], eventInfo)` 方法
5. 进入 `triggerEffects` 方法:
6. **这里要注意:** 因为我们在 `ComputedRefImpl` 的构造函数中, 执行了 `this.effect.computed = this`, 所以此时的 `if (effect.computed)` 判断将会为 `true`:

1. 此时我们注意看 `effects`, 此时 `effect` 的值为 `ReactiveEffect` 的实例, 同时 `scheduler` 存在值:

```
▼ effects: Array(1)
  ▼ 0: ReactiveEffect2
    active: true
    ▶ computed: ComputedRefImpl {dep: Set(1), __v_isRef: true, __v_isReadOnly:
    ▶ deps: [Set(1)]
    ▶ fn: () => { return '姓名: ' + obj.name }
    parent: undefined
    scheduler: () => {
      if (!this._dirty) {
        this._dirty = true;
        triggerRefValue(this);
      }
    }
    ▶ [[Prototype]]: Object
```

2. 接下来进入 `triggerEffect`:

1. 在 `triggerEffect` 中
2. 执行 `if (effect.scheduler)` 判断, 因为 `effect` 存在 `scheduler`, 所以会执行 `scheduler` 函数
3. 此时会进入 `ComputedRefImpl` 类的构造函数中, 传递的回调函数

1. 进入 `scheduler` 回调
2. 此时 `this` 的状态如下:

```
▼ this: ComputedRefImpl
  ▶ dep: Set(1) {ReactiveEffect2}
  ▼ effect: ReactiveEffect2
    active: true
    ▶ computed: ComputedRefImpl {dep: Set(1), __v_isRef: true, __v_isReadOnly:
    ▶ deps: [Set(1)]
    ▶ fn: () => { return '姓名: ' + obj.name }
    parent: undefined
    ▶ scheduler: () => {...}
    ▶ [[Prototype]]: Object
    __v_isReadOnly: true
    __v_isRef: true
    cacheable: true
    _dirty: false
    ▶ _setter: () => {...}
    _value: "姓名: 张三"
    value: (...)
    ▶ [[Prototype]]: Object
```

3. 所以会执行 `triggerRefValue` 函数:

1. 进入 `triggerRefValue` 函数
2. 会再次触发 `triggerEffects` 函数, 把当前的 `this.dep` 作为参数传入

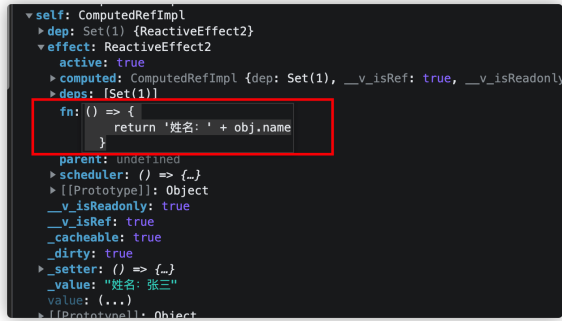
1. 再次进入 `triggerEffects`
2. **注意:** 此时的 `effects` 的值为:

```
▼ effects: Array(1)
  ▼ 0: ReactiveEffect2
    active: true
    ▶ deps: [Set(1)]
    ▶ fn: () => {
      document.querySelector('#app').innerHTML = computedObj.value
    }
    parent: undefined
    scheduler: null
    ▶ [[Prototype]]: Object
    length: 1
    ▶ [[Prototype]]: Array(0)
```

#### 4. 接下来进入 `triggerEffect` :

1. 在 `triggerEffect` 因为 `effect` 不再包含调度器 `scheduler`
2. 所以会直接执行 `fn` 函数
3. `fn` 函数的触发, 标记着 `computedObj.value` 触发, 而我们知道 `computedObj.value` 本质上是 `get value` 函数的触发, 所以代码接下来会触发 `ComputedRefImpl` 的 `get value`
4. 接下来进入 `get value`

1. 进入 `get value`
2. 执行 `self._value = self.effect.run()!`, 而 `run` 函数的执行本质上是 `fn` 函数的执行, 而此时 `fn` 函数为:



```
self: ComputedRefImpl
  dep: Set(1) {ReactiveEffect2}
  effect: ReactiveEffect2
    active: true
    computed: ComputedRefImpl {dep: Set(1), __v_isRef: true, __v_isReadOnly: true}
    deps: [Set(1)]
    fn: () => {
      return '姓名: ' + obj.name
    }
  parent: undefined
  scheduler: () => {[_]}
  [[Prototype]]: Object
  __v_isReadOnly: true
  __v_isRef: true
  __cacheable: true
  __dirty: true
  __setter: () => {[_]}
  __value: "姓名: 张三"
  value: (...)
  [[Prototype]]: Object
```

3. 执行该函数得到计算的值
4. 最后作为 `computedObj.value` 的返回值
5. 省略后续的触发...

至此, 整个 `obj.name` 引发的副作用全部执行完成。

由以上代码可知, 整个的计算属性的逻辑是非常复杂的, 我们来做一下整理:

1. 整个事件有 `obj.name` 开始
2. 触发 `proxy` 实例的 `setter`
3. 执行 `trigger`, **第一次触发依赖**
4. 注意, 此时 `effect` 包含调度器属性, 所以会触发调度器
5. 调度器指向 `ComputedRefImpl` 的构造函数中传入的匿名函数
6. 在匿名函数中会: **再次触发依赖**
7. 即: **两次触发依赖**
8. 最后执行:

<> 代码块

```
1    () => {
2      return '姓名: ' + obj.name
3    }
```

得到值作为 `computedObj` 的值

## 总结

那么到这里我们基本上了解了 `computed` 的执行逻辑, 里面涉及到了一些我们之前没有了解过的概念, 比如 **调度器** `scheduler`, 并且整体的 `computed` 的流程也相当复杂。

所以接下来我们去实现 `computed` 的时候, 会分步骤一步一步执行。

