Sunday • 更新于 2022-10-19

上一节 06：总结：com... 08：框架实现：... 下一节

07：源码阅读：响应性的数据监听器 watch，跟踪源码实现逻辑

我们可以点击 这里 来查看 `watch` 的官方文档。

`watch` 的实现和 `computed` 有一些相似的地方，但是作用却与 `computed` 大有不同。 `watch` 可以**监听响应式数据的变化，从而触发指定的函数**

在 `vue3` 中使用 `watch` 的代码如下所示：

代码块

```
watch(() => obj.name, (value, oldValue) => {
  console.log('watch 监听被触发');
  console.log('oldValue', oldValue);
  console.log('value', value);
}, {
  immediate: true,
  deep: true
})
```

上述代码中 `watch` 函数接收三个参数：

1. 监听的响应式对象
2. 回调函数 `cb`
3. 配置对象： `options`
    1. `immediate`： `watch` 初始化完成后被立刻触发一次
    2. `deep`：深度监听

由此可见， `watch` 函数颇为复杂，所以我们在跟踪 `watch` 的源码实现时，应当分步骤进行跟踪。

基础的 `watch` 实例

修改 `packages/vue/examples/imooc/watch.html` 实例代码如下：

代码块

```
<script>
  const { reactive, watch } = Vue

  const obj = reactive({
    name: '张三'
  })

  watch(obj, (value, oldValue) => {
    console.log('watch 监听被触发');
    console.log('value', value);
  })

  setTimeout(() => {
    obj.name = '李四'
  }, 2000);
</script>
```

在以上代码中：

1. 首先通过 `reactive` 函数构建了响应性的实例
2. 然后触发 `watch`
3. 最后触发 `proxy` 的 `setter`

摒弃掉之前熟悉的 `reactive`，我们从 `watch` 函数开始跟踪：

watch 函数

1. 在 `packages/runtime-core/src/apiWatch.ts` 中找到 `watch` 函数，开始 `debugger`：
2. 执行 `doWatch` 函数：
    1. 进入 `doWatch` 函数
    2. 因为 `source` 为 `reactive` 类型数据，所以 `getter = () => source`，目前 `source` 为 `proxy` 实例，即：

        代码块

        ```
        getter = () => Proxy{name: '张三'}
        ```

    3. 紧接着，指定 `deep = true`，即：`source` **为** `reactive` **时，默认添加** `options.deep = true`
    4. 执行 `if (cb && deep)`，条件满足：
        1. 创建新的常量 `baseGetter = getter`
    5. 执行 `let oldValue = isMultiSource ? [] : INITIAL_WATCHER_VALUE`：
        1. 其中 `isMultiSource` 表示是否有多个源，我们当前只有一个源，所以 `oldValue = INITIAL_WATCHER_VALUE`
        2. `INITIAL_WATCHER_VALUE = {}`
    6. 执行 `const job: SchedulerJob = () => {...}`，我们知道 `Scheduler` 是一个调度器，`SchedulerJob` 其实就是一个调度器的处理函数，在之前我们接触了一下 `Scheduler` 调度器，但是并没有进行深入了解，那么这里将涉及到调度器的比较复杂的一些概念，所以后面我们想要实现 `watch`，还需要 **深入的了解下调度器的概念**，现在我们暂时先不需要管它。
    7. 接下来还是 **调度器** 概念，直接执行：`let scheduler: EffectScheduler = () => queuePreFlushCb(job)`
    8. `6、7` 结合，将得到一个完整的调度器函数 `scheduler`，该函数被触发时，会返回 `queuePreFlushCb(job)` **函数执行的结果**
    9. 代码继续执行得到一个 `ReactiveEffect` 的实例，**注意：** 该实例包含一个完善的调度器 `scheduler`
    10. 代码继续执行，进入如下判断逻辑：

        代码块

        ```
        // cb 是 watch 的第二个参数
        if (cb) {
          // immediate 是 options 中的 immediate，表示：watch 是否立刻执行。
          // 那么根据这个这个概念和一下代码，其实可以猜测：《 job() 触发，表示 watch 被立刻执行
          if (immediate) {
            job()
          } else {
            // 不包含 immediate，则通过 effect.run() 获取旧值。
            // 根据我们前面创建 effect 的代码可知，run() 的执行其实是 getter 的执行。
            // 所以此处可以理解为 getter 被触发，则获取了 oldValue
            // 我们的代码将执行 else
            oldValue = effect.run()
          }
        ```

    11. 最后 `return` 了一个回调函数：

```
代码块
1     return () => {
2         effect.stop()
3         if (instance && instance.scope) {
4           remove(instance.scope.effects!, effect)
5         }
6       }
```

回调函数中的代码我们 **无需深究**，但是根据 **代码语义** `stop` 停止 、 `remove` 删除 ，可以猜测：**该函数被触发 `watch` 将停止监听，同时删除依赖**

那么至此 `watch` 函数的逻辑执行完成。

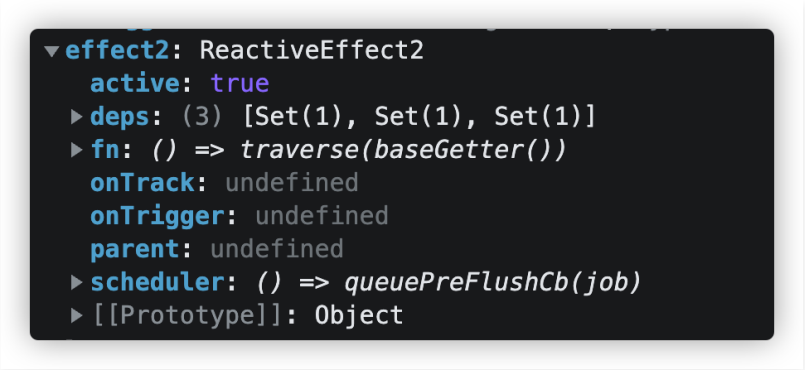
由以上代码可知：

1. `watch` 函数的代码很长，但是逻辑还算清晰
2. 调度器 `scheduler` 在 `watch` 中很关键
3. `scheduler` 、 `ReactiveEffect` 两者之间存在互相作用的关系，一旦 `effect` 触发了 `scheduler` 那么会导致 `queuePreFlushCb(job)` 执行
4. 只要 `job()` 触发，那么就表示 `watch` 触发了一次

reactive 触发 setter

等待两秒， `reactive` 实例将触发 `setter` 行为， `setter` 行为的触发会导致 `trigger` 函数的触发，所以我们可以直接在 `trigger` 中进行 `debugger`

1. 在 `trigger` 中进行 `debugger`
2. 根据我们之前的经验可知， `trigger` 最终会触发到 `triggerEffect` ，所以我们可以 **省略中间** 步骤，直接进入到 `triggerEffect` 中
    1. 进入 `triggerEffect`
    2. 此时 `effect` 为：



    3. 关键其中两个比较重要的变量：
        1. `fn` ： 值为 `traverse(baseGetter())` ：
            1. 根据 `2-4-1` 可知 `baseGetter = getter`
            2. 根据 `2-2` 可知： `getter = () => Proxy{name: 'xx'}`
            3. 所以 `fn = traverse(() => Proxy{name: 'xx'})`
        2. `scheduler` ： 值为 `() => queuePreFlushCb(job)`
            1. 目前已知 `job()` 触发表示 `watch` 被回调一次
    4. 因为 `scheduler` 存在，所以会直接执行 `scheduler` ，即等同于**直接执行** `queuePreFlushCb(job)`
    5. 所以接下来我们 **进入** `queuePreFlushCb` 函数，看看 `queuePreFlushCb` 做了什么：
        1. 进入 `queuePreFlushCb`

2. 触发 `queueCb(cb, ..., pendingPreFlushCbs, ...)` 函数，此时 `cb = job`，即：`cb()` **触发一次，意味着** `watch` **触发一次**

    1. 进入 `queueCb` 函数
    2. 执行 `pendingQueue.push(cb)`， `pendingQueue` 从语义中看表示 **队列**，为一个 **数组**
    3. 执行 `queueFlush()` 函数：

        1. 进入 `queueFlush()` 函数
        2. 执行 `isFlushPending = true`
        3. 执行 `currentFlushPromise = resolvedPromise.then(flushJobs)`

            1. 查看 `resolvedPromise` 可知：`const resolvedPromise = Promise.resolve()`，即：`promise` **的成功状态**
            2. 我们知道 `promise` 主要存在 三种状态：
            3. 待定（*pending*）：初始状态，既没有被兑现，也没有被拒绝。
            4. 已兑现（*fulfilled*）：意味着操作成功完成。
            5. 已拒绝（*rejected*）：意味着操作失败。
            6. 结合语义，其实可知：`isFlushPending = true` 应该是一个 **标记**，表示 `promise` 进入 `pending` 状态
            7. 而同时我们知道 Promise.resolve() 是一个 **已兑现** 状态的状态切换函数，它是一个 **异步的微任务**，即：**它是一个优先于** `setTimeout(() => {}, 0)` **的异步任务**

        4. 而 `flushJobs` 是将是一个 `.then` 中的回调，即 **异步执行函数**，它会等到 **同步任务执行完成之后** 被触发
        5. 我们可以 **给** `flushJobs` **函数内部增加一个断点**

3. 至此整个 `trigger` 就执行完成

由以上代码可知：

1. 整个 `trigger` 的执行核心是触发了 `scheduler` 调度器，从而触发 `queuePreFlushCb` 函数
2. `queuePreFlushCb` 函数主要做了以下几点事情：

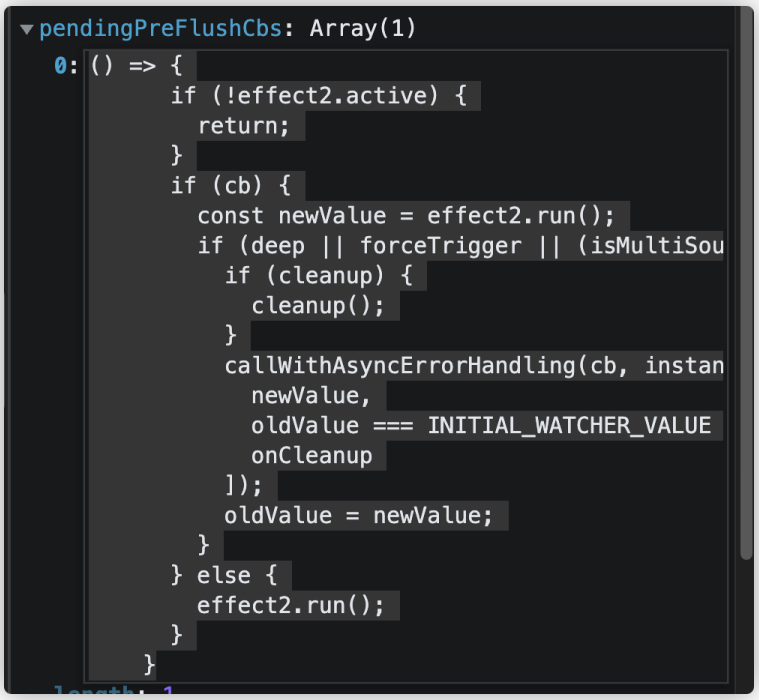
    1. 构建了任务队列 `pendingQueue`
    2. 通过 `Promise.resolve().then` 把 `flushJobs` 函数扔到了微任务队列中

同时因为接下来 **同步任务已经执行完成**，所以 **异步的微任务** 马上就要开始执行，即接下来我们将会进入 `flushJobs` 中。

flushJobs 函数

1. 进入 `flushJobs` 函数
2. 执行 `flushPreFlushCbs(seen)` **函数**，这个函数非常关键，我们来看一下：

    1. 第一行代码执行 `if (pendingPreFlushCbs.length)`，这个 `pendingPreFlushCbs` 此时的值为：



2. 通过截图代码可知， `pendingPreFlushCbs` 为一个数组，其中第一个元素就是 **job** 函数
    1. 从《`reactive 触发 setter 的 2-5-2`》中可以看到传参
3. 执行 `activePreFlushCbs = [...new Set(pendingPreFlushCbs)]`，即：**activePreFlushCbs = pendingPreFlushCbs**
4. 执行 `for` 循环，**执行** `activePreFlushCbs[preFlushIndex]()`，即从 `activePreFlushCbs` 这个数组中，取出一个函数，并执行。
    1. 根据 `2`、`3` 步可知，此时取出并且执行的函数即为：**job** 函数！

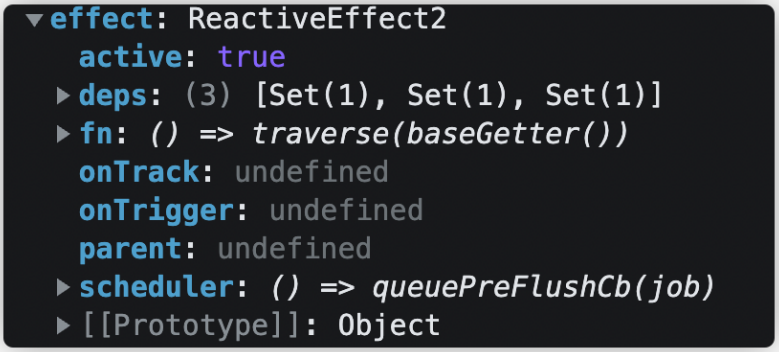
那么到这里， `job` **函数被成功执行**，我们知道 `job` 执行意味着 `watch` 执行，即当前 `watch` 的回调 **即将被执行**

由以上代码可知：

1. `flushJobs` 的主要作用就是触发 `job`，即：触发 `watch`

job 函数

1. 进入 `job` 的执行函数
2. 执行 `const newValue = effect.run()`，此时 `effect` 为：



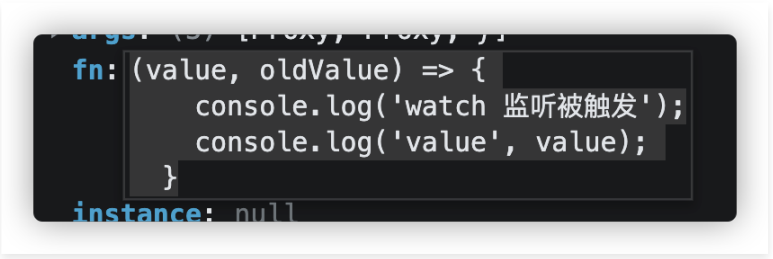
1. 我们知道执行 `run`，本质上是执行 `fn`

3. 结合代码获取到的是 `newValue`，所以我们可以大胆猜测，测试 `fn` 的结果等同于：

```
代码块
1    fn: () => {name: '李四'}
```

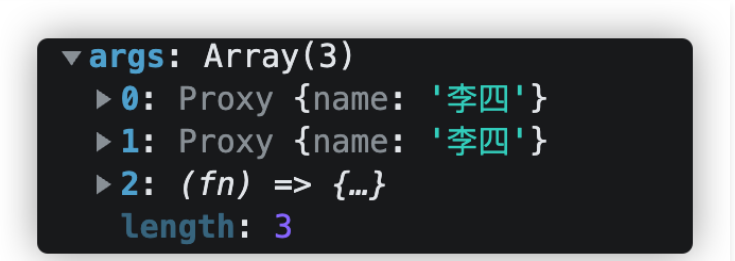
4. 接下来执行：`callWithAsyncErrorHandling(cb ......)`

    1. 进入 `callWithAsyncErrorHandling` 函数：
    2. 函数接收的第一个参数 `fn` 的值为 `watch` 的第一个参数 `db`：



    3. 接下来执行 `callWithErrorHandling(fn ......)`

        1. 进入 `callWithErrorHandling`
        2. 这里的代码就比较简单了，其实就是触发了 `fn(...args)`，即：**watch 的回调被触发**，此时 `args` 的值为：



        3. 但是比较有意思的是，这里执行了一次 `try ... catch`

```
代码块
1       try {
2         res = args ? fn(...args) : fn()
3       } catch (err) {
4         handleError(err, instance, type)
5       }
```

        4. TODO…

截止到此时 `watch` 的回调终于 **被触发了。**

由以上代码可知：

1. `job` 函数的主要作用其实就是有两个：
    1. 拿到 `newValue` 和 `oldValue`
    2. 触发 `fn` 函数执行

总结

到目前为止，整个 `watch` 的逻辑就已经全部理完了。整体氛围了四大块：

1. `watch` 函数本身
2. `reactive` 的 `setter`
3. `flushJobs`
4. `job`

整个 `watch` 还是比较复杂的，主要是因为 `vue` 在内部进行了很多的 **兼容性处理**，使代码的复杂度上升了好几个台阶，我们自己去实现的时候 **会简单很多** 的。

06：总结：computed 计算属性 ◂ 上一节　　下一节 ▸ 08：框架实现：深入 scheduler 调度系统实...

我要提出意见反馈