

全部开发者教程

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

08：框架实现：深入 scheduler 调度系统实现机制


09：框架实现：初步实现 watch 数据监听器

10：问题分析：watch 下的依赖收集

11：框架实现：完成 watch 数据监听器的依赖收集

12：总结：watch 数据侦听器

13：总结



Sunday • 更新于 2022-10-19

◀ 上一节 09：框架实现：...

11：框架实现：... 下一节 ▶

10：问题分析：watch 下的依赖收集原则

现在我们还差一步就可以完成 `watch` 的响应式数据监听了，那么这一步是什么呢？

根据代码可知，`watch` 内部本质上也是通过：`ReactiveEffect + scheduler` 进行实现的。

那么对于 `ReactiveEffect` 而言，我们知道，他需要拥有两个先决条件才可以完成响应性：

1. 依赖收集

2. 触发依赖

那么对于我们当前的代码而言，我们在 `setTimeout` 中，触发了 **触发依赖** 操作。但是我们在哪里进行的 **依赖收集**呢？

答案是：**没有**

这就是我们为什么没有办法触发 `watch` 监听的原因。

那么这个依赖收集我们应该怎么做呢？

不知道大家还记不记得，我们之前在看源码的时候，看到过一个 `traverse` 方法。

之前的时候，我们一直没有看过该方法，那么现在我们可以来说一下它了。

它的源码在 `packages/runtime-core/src/apiWatch.ts` 中：

查看源代码可以发现，这里的代码其实有些 **莫名其妙**，他好像什么都没有做，只是在 **循环的进行** `xx.x.value` 的形式，我们知道 `xxx.value` 这个行为，我们把它叫做 `getter` 行为。并且这样会产生 **副作用**，那就是 **依赖收集**！。

所以我们知道了，对于 `traverse` 方法而言，它就是一个不断在触发响应式数据 **依赖收集** 的方法。

我们可以通过该方法来触发依赖收集，然后在两秒之后，触发依赖，完成 `scheduler` 的回调。

09：框架实现：初步实现 watch 数据监听器

◀ 上一节

下一节 ▶

11：框架实现：完成 watch 数据监听器的依...

✎ 我要提出意见反馈

索引目录

10：问题分析：w