Sunday • 更新于 2022-10-19

上一节 01：前言 03：框架实现：… 下一节

02：源码阅读：ref 复杂数据类型的响应性

和学习 `reactive` 的时候一样，我们首先先来看一下 `ref` 函数下， `vue 3` 源码的执行过程。

1. 创建测试实例 `packages/vue/examples/imooc/ref.html`

代码块 预览 复制

```
<body>
  <div id="app"></div>
</body>
<script>
  const { ref, effect } = Vue

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

  // 调用 effect 方法
  effect(() => {
    document.querySelector('#app').innerText = obj.value.name
  })

  setTimeout(() => {
    obj.value.name = '李四'
  }, 2000);

</script>
```

2. 通过 `Live Server` 运行测试实例
3. `ref` 的代码位于 `packages/reactivity/src/ref.ts` 之下，我们可以在这里打下断点

ref 函数

1. `ref` 函数中，直接触发 `createRef` 函数
2. 在 `createRef` 中，进行了判断如果当前已经是一个 `ref` 类型数据则直接返回，否则 **返回** `RefImpl` **类型的实例**
3. 那么这个 `RefImpl` 是什么呢?
    1. `RefImpl` 是同样位于 `packages/reactivity/src/ref.ts` 之下的一个类
    2. 该类的构造函数中，执行了一个 `toReactive` 的方法，传入了 `value` 并把返回值赋值给了 `this._value` ，那么我们来看看 `toReactive` 的作用：
        1. `toReactive` 方法把数据分成了两种类型：
            1. 复杂数据类型：调用了 `reactive` 函数，即把 `value` 变为响应性的。
            2. 简单数据类型：直接把 `value` 原样返回
    3. 该类提供了一个分别被 `get` 和 `set` 标记的函数 `value`
        1. 当执行 `xxx.value` 时，会触发 `get` 标记

4. 至此 `ref` 函数执行完成。

由以上逻辑可知：

1. 对于 `ref` 而言，主要生成了 `RefImpl` 的实例
2. 在构造函数中对传入的数据进行了处理：
    1. 复杂数据类型：转为响应性的 `proxy` 实例
    2. 简单数据类型：不去处理
3. `RefImpl` 分别提供了 `get value` 、 `set value` 以此来完成对 `getter` 和 `setter` 的监听，注意这里并没有使用 `proxy`

effect 函数

当 `ref` 函数执行完成之后，测试实例开始执行 `effect` 函数。

`effect` 函数我们之前跟踪过它的执行流程，我们知道整个 `effect` 主要做了3 件事情：

1. 生成 `ReactiveEffect` 实例
2. **触发 `fn` 方法，从而激活 `getter`**
3. 建立了 `targetMap` 和 `activeEffect` 之间的联系
    1. `dep.add(activeEffect)`
    2. `activeEffect.deps.push(dep)`

通过以上可知， `effect` 中会触发 `fn` 函数，也就是说会执行 `obj.value.name` ，那么根据 `get value` 机制，此时会触发 `RefImpl` 的 `get value` 方法。

所以我们可以在 `117` 行增加断点，等代码进入 `get value`

get value()

1. 在 `get value` 中会触发 `trackRefValue` 方法
    1. 触发 `trackEffects` 函数，并且在此时为 `ref` 新增了一个 `dep` 属性：

    ```
    代码块
    1    trackEffects(ref.dep || (ref.dep = createDep())...}
    ```

    2. 而 `trackEffects` 其实我们是有过了解的，我们知道 `trackEffects` 主要的作用就是：**收集所有的依赖**
2. 至此 `get value` 执行完成

由以上逻辑可知：

整个 `get value` 的处理逻辑还是比较简单的，主要还是通过之前的 `trackEffects` 属性来收集依赖。

再次触发 get value()

最后就是在两秒之后，修改数据源了：

```
代码块
1    obj.value.name = '李四'
```

但是这里有一个很关键的问题，需要大家进行思考，那就是：**此时会触发 `get value` 还是 `set value` ?**

我们知道以上代码可以被拆解为：

```
代码块
1    const value = obj.value
```

那么通过以上代码我们清晰可知，其实触发的应该是 `get value` 函数。

在 `get value` 函数中：

1. 再次执行 `trackRefValue` 函数：
    1. 但是此时 `activeEffect` 为 `undefined`，所以不会执行后续逻辑
2. 返回 `this._value`：
    1. 通过 **构造函数**，我们可知，此时的 `this._value` 是经过 `toReactive` 函数过滤之后的数据，在当前实例中为 `proxy` 实例。
3. `get value` 执行完成

由以上逻辑可知：

1. `const value` 是 `proxy` 类型的实例，即：**代理对象**，被代理对象为 `{name: '张三'}`
2. 执行 `value.name = '李四'` ，本质上是触发了 `proxy` 的 `setter`
3. 根据 `reactive` 的执行逻辑可知，此时会触发 `trigger` 触发依赖。
4. 至此，修改视图

总结：

由以上逻辑可知：

1. 对于 `ref` 函数，会返回 `RefImpl` 类型的实例
2. 在该实例中，会根据传入的数据类型进行分开处理
    1. 复杂数据类型：转化为 `reactive` 返回的 `proxy` 实例
    2. 简单数据类型：不做处理
3. 无论我们执行 `obj.value.name` 还是 `obj.value.name = xxx` 本质上都是触发了 `get value`
4. 之所以会进行 **响应性** 是因为 `obj.value` 是一个 `reactive` 函数生成的 `proxy`

01：前言 ◂ 上一节　　下一节 ▸ 03：框架实现：ref 函数 - 构建复杂数据类型...

我要提出意见反馈