Sunday • 更新于 2022-10-19

上一节 11：框架实现：... 13：框架实现：... 下一节

12：源码阅读：响应性数据改变，触发组件的响应性变化

此时我们已经可以在生命周期回调钩子中访问到对应的响应性数据了，根据响应性数据的概念，**当数据发生变化时，视图应该跟随发生变化**，所以我们接下来就要来看一下 **组件中响应性数据引起的视图改变。**

再来看这一块内容之前，首先我们需要先来明确一些基本的概念：

1. 组件的渲染，本质上是 `render` 函数返回值的渲染。
2. 所谓响应性数据，指的是：
    1. `getter` 时收集依赖
    2. `setter` 时触发依赖

那么根据以上概念，我们所需要做的就是：

1. 在组件的数据被触发 `getter` 时，我们应该收集依赖。那么组件什么时候触发的 `getter` 呢？在 `packages/runtime-core/src/renderer.ts` 的 `setupRenderEffect` 方法中，我们创建了一个 `effect`，并且把 `effect` 的 `fn` 指向了 `componentUpdateFn` 函数。在该函数中，我们触发了 `getter`，然后得到了 `subTree`，然后进行渲染。所以依赖收集的函数为 `componentUpdateFn`。
2. 在组件的数据被触发 `setter` 时，我们应该触发依赖。我们刚才说了，收集的依赖本质上是 `componentUpdateFn` 函数，所以我们在触发依赖时，所触发的也应该是 `componentUpdateFn` 函数。

明确好了以上内容之后，下面我们就可以创建对应的测试案例 `packages/vue/examples/imooc/runtime/redner-component-hook-data-change.html`：

代码块

```
<script>
  const { h, render } = Vue

  const component = {
    data() {
      return {
        msg: 'hello component'
      }
    },
    render() {
      return h('div', this.msg)
    },
    // 组件实例处理完所有与状态相关的选项之后
    created() {
      setTimeout(() => {
        this.msg = '你好，世界'
      }, 2000);
    }
  }

  const vnode = h(component)
  // 挂载
  render(vnode, document.querySelector('#app'))
</script>
```

在 [illegible] 中进行 [illegible] 等待 **第二次** 进入 [illegible] 函数（**注意**），此时我

1. 第二次进入 `componentUpdateFn` 函数，此时因为 **组件已经被挂载**，所以 **不再** 执行 `if (!instance.isMounted)`，而是会直接进入 `else`
2. 执行 `let { next, bu, u, parent, vnode } = instance`，从 `instance` 中获取 `next` 和 `vnode`
    1. 此时拿到的 `next`，表示下一次的 `subTree`，现在为 `null`
    2. 此时拿到的 `vnode`，为当前组件的 `vnode`
3. 执行 `if (next)`，因为 `next = null`，所以会进入 `else`
    1. 进入 `else`
    2. 执行 `next = vnode`。即：下一次的渲染为 `vnode`
4. 代码执行 `const nextTree = renderComponentRoot(instance)`，这个代码我们是熟悉的，并且非常关键，表示我们下一次要渲染的 `VNode`，我们进入 `renderComponentRoot` 方法
    1. 进入 `renderComponentRoot` 方法
    2. 执行 `if (vnode.shapeFlag & ShapeFlags.STATEFUL_COMPONENT)`，因为当前是 **有状态的数据**，所以会进入 `if`
        1. 进入 `if` 之后的代码我们就比较熟悉了：

        代码块

        ```
        result = normalizeVNode(
          render!.call(
            proxyToUse,
            proxyToUse!,
            renderCache,
            props,
            setupState,
            data,
            ctx
          )
        )
        ```

        2. 同样通过 `call` 方法，改变 `this` 指向，触发 `render`。然后通过 `normalizeVNode` 得到 `vnode`
        3. 这次得到的 `vnode` 就是 **下一次要渲染的** `subTree`
5. 跳出 `renderComponentRoot` 方法，此时得到的 `nextTree` 的值为：
6. 代码执行：

    代码块

    ```
    const prevTree = instance.subTree
    instance.subTree = nextTree
    ```

    保存上一次的 `subTree`，同时赋值新的 `subTree`。之所以要保存上一次的 `subTree` 是因为我们后面要进行 **更新** 操作
7. 触发 `patch(...)` 方法，完成 **更新操作**

至此，整个 组件视图的更新完成。

由以上代码可知：

1. 所谓的组件响应性更新，本质上指的是： `componentUpdateFn` 的再次触发，根据新的 **数据** 生成新的 `subTree`，再通过 `path` 进行 **更新** 操作

11：框架实现：生命回调钩子中访问响应性... ◂ 上一节　　下一节 ▸ 13：框架实现：响应性数据改变，触发组件...