全部开发者教程

11：框架实现：生命回调钩子中访问响应性数据

12：源码阅读：响应性数据改变，触发组件的响应性变化

13：框架实现：响应性数据改变，触发组件的响应性变化

14：源码阅读：composition API，setup 函数挂载逻辑

15：框架实现：composition API，setup 函数挂载逻辑

16：总结

第十二章：runtime 运行时 - diff 算法核心实现

01：前言

02：前置知识：VNode 虚拟节点 key 属性的作用

03：源码阅读：场景一：自前向后的 diff 对比

04：框架实现：场景一：自前

Sunday • 更新于 2022-10-19

上一节 02：前置知识：… 04：框架实现：… 下一节

03：源码阅读：场景一：自前向后的 diff 对比

我们创建如下测试实例 `packages/vue/examples/imooc/runtime/render-element-diff.html`：

代码块

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

  const vnode = h('ul', [
    h('li', {
      key: 1
    }, 'a'),
    h('li', {
      key: 2
    }, 'b'),
    h('li', {
      key: 3
    }, 'c'),
  ])
  // 挂载
  render(vnode, document.querySelector('#app'))

  // 延迟两秒，生成新的 vnode，进行更新操作
  setTimeout(() => {
    const vnode2 = h('ul', [
      h('li', {
        key: 1
      }, 'a'),
      h('li', {
        key: 2
      }, 'b'),
      h('li', {
        key: 3
      }, 'd')
    ])
    render(vnode2, document.querySelector('#app'))
  }, 2000);
</script>
```

在上面的测试实例中，我们利用 `vnode 2` 更新 `vnode` 节点。

它们的子节点都是一个 `ARRAY_CHILDREN` ，需要注意的是它们的 **子节点具备相同顺序下的相同 `vnode` (`type` 、 `key` 相等) **。唯一不同的地方在于 **第三个 `li` 标签显示的内容不同**

那么我们来看一下这种情况下 `vue` 是如何来处理对应的 `diff` 的。

在 `patchKeyedChildren` 中，进行 `debugger` ，等待两秒，进入 `debugger`：

1. 进入 `patchKeyedChildren` 方法，此时各参数的值为：

索引目录

03：源码阅读：场

```
const patchKeyedChildren = (
  c1: VNode[],  c1 = (3) [{…}, {…}, {…}]
  c2: VNodeArrayChildren,  c2 = (3) [{…}, {…}, {…}]
  container: RendererElement,  container = ul {__vnode: {…}, __vuePa
  parentAnchor: RendererNode | null,  parentAnchor = null
  parentComponent: ComponentInternalInstance | null,  parentComponen
  parentSuspense: SuspenseBoundary | null,  parentSuspense = null
  isSVG: boolean,  isSVG = false
  slotScopeIds: string[] | null,  slotScopeIds = null
  optimized: boolean  optimized = false
) => {
```

1. 其中 `c1` 表示为：旧的子节点，即： `oldChildren`
2. `c2` 表示为：新的子节点，即： `newChildren`

2. 执行 `let i = 0`，声明了一个 **计数变量 `i`，初始为 `0`**
3. 执行 `const l2 = c2.length`。此时的 `l2` 表示为 **新的子节点的长度**，即： `newChildrenLength`
4. 执行 `let e1 = c1.length - 1`。此时的 `e1` 表示为 **旧的子节点最大（最后一个）下标**，即： `oldChildrenEnd`
5. 执行 `let e2 = l2 - 1`。此时的 `e2` 表示为 **新的子节点最大（最后一个）下标，**即： `newChildrenEnd`
6. 执行 `while` 循环： `while (i <= e1 && i <= e2)`
    1. **第一次** 进入 `while` 循环：
        1. 此时 `n1` 的值为：

            代码块

            ```
            h('li', {
                key: 1
            }, 'a')
            ```

        2. 此时 `n2` 的值为：

            代码块

            ```
            h('li', {
                key: 1
            }, 'a')
            ```

        3. 那么根据上一小节所说，我们知道，此时 `isSameVNodeType(n1, n2)` 会被判定为 `true`
        4. 所以此时执行 `patch` 方法，进行打补丁即可。
        5. 最后执行： `i++`
        6. 至此，第一次循环完成
    2. **第二次** 进入 `while` 循环：
        1. 根据刚才所知，此时的 `n1` 和 `n2` 依然符合 `isSameVNodeType(n1, n2)` 的判定
        2. 所以，依然会执行 `patch` 方法，进行打补丁。
        3. 最后执行： `i++`
        4. 至此，第二次循环完成
    3. **第三次** 进入 `while` 循环：
        1. 根据刚才所知，此时的 `n1` 和 `n2` 依然符合 `isSameVNodeType(n1, n2)` 的判定
        2. 所以，依然会执行 `patch` 方法，进行打补丁。
        3. 最后执行： `i++`
        4. 至此，第三次循环完成
7. 三次循环全部完成，此时，我们查看浏览器，可以发现 `children` 的 **更新** 操作 **已经完成。**

由以上代码可知：

1. `diff` 所面临的的第一个场景就是：**自前向后的 diff 比对**
2. 在这样的一个比对中，会 **依次获取相同下标的** `oldChild` **和** `newChild` ：
    1. 如果 `oldChild` 和 `newChild` 为 **相同的** `VNode`，则直接通过 `patch` 进行打补丁即可
    2. 如果 `oldChild` 和 `newChild` 为 **不相同的** `VNode`，则会跳出循环
3. 每次处理成功，则会自增 `i` 标记，表示：**自前向后已处理过的节点数量**

02：前置知识：VNode 虚拟节点 key 属性... ◂ 上一节　　下一节 ▸ 04：框架实现：场景一：自前向后的 diff 对比

我要提出意见反馈