


```
45     }, 2000);  
</script>
```

该测试实例对应的节点渲染图为：



运行该测试实例，我们来跟踪场景五的逻辑：

```
<> 代码块  
  
1  // 5. unknown sequence  
2  // [i ... e1 + 1]: a b [c d e] f g  
3  // [i ... e2 + 1]: a b [e d c h] f g  
4  // i = 2, e1 = 4, e2 = 5  
5  else {  
6    // 旧子节点的开始索引: oldChildrenStart  
7    const s1 = i  
8    // 新子节点的开始索引: newChildrenStart  
9    const s2 = i  
10  
11    // 5.1 创建一个 <key (新节点的 key):index (新节点的位置)> 的 Map 对象 keyToNewIndexMap。  
12    const keyToNewIndexMap: Map<string | number | symbol, number> = new Map()  
13    // 通过循环为 keyToNewIndexMap 填充值 (s2 = newChildrenStart; e2 = newChildrenEnd)  
14    for (i = s2; i <= e2; i++) {  
15      // 从 newChildren 中根据开始索引获取每一个 child (c2 = newChildren)  
16      const nextChild = (c2[i] = optimized  
17        ? cloneIfMounted(c2[i] as VNode)  
18        : normalizeVNode(c2[i]))  
19      // child 必须存在 key (这也是为什么 v-for 必须要有 key 的原因)  
20      if (nextChild.key != null) {  
21        // key 不可以重复, 否则你将会得到一个错误  
22        if (__DEV__ && keyToNewIndexMap.has(nextChild.key)) {  
23          warn(  
24            `Duplicate keys found during update:`,  
25            JSON.stringify(nextChild.key),  
26            `Make sure keys are unique.`  
27          )  
28        }  
29        // 把 key 和 对应的索引, 放到 keyToNewIndexMap 对象中  
30        keyToNewIndexMap.set(nextChild.key, i)  
31      }  
32    }  
33  
34    // 5.2 循环 oldChildren, 并尝试进行 patch (打补丁) 或 unmount (删除) 旧节点  
35    let j  
36    // 记录已经修复的新节点数量  
37    let patched = 0  
38    // 新节点待修补的数量 = newChildrenEnd - newChildrenStart + 1  
39    const toBePatched = e2 - s2 + 1  
40    // 标记位: 节点是否需要移动  
41    let moved = false
```

意见反馈

收藏教程

标记书签

```

44 // 创建一个 Array 的对象，用来确定最长递增子序列。它的下标表示：《新节点的下标（newIndex），不
45 // 但是，需要特别注意的是：oldIndex 的值应该永远 +1（因为 0 代表了特殊含义，他表示《新节点
46 const newIndexToOldIndexMap = new Array(toBePatched)
47 // 遍历 toBePatched，为 newIndexToOldIndexMap 进行初始化，初始化时，所有的元素为 0
48 for (i = 0; i < toBePatched; i++) newIndexToOldIndexMap[i] = 0
49 // 遍历 oldChildren（s1 = oldChildrenStart; e1 = oldChildrenEnd），获取旧节点（c1 = ol
50 for (i = s1; i <= e1; i++) {
51 // 获取旧节点（c1 = oldChildren）
52 const prevChild = c1[i]
53 // 如果当前 已经处理的节点数量 > 待处理的节点数量，那么就证明：《所有的节点都已经更新完成，剩
54 if (patched >= toBePatched) {
55 // 所有的节点都已经更新完成，剩余的旧节点全部删除即可
56 unmount(prevChild, parentComponent, parentSuspense, true)
57 continue
58 }
59 // 新节点需要存在的位置，需要根据旧节点来寻找（包含已处理的节点。即：n-c 被认为是 1）
60 let newIndex
61 // 旧节点的 key 存在时
62 if (prevChild.key != null) {
63 // 根据旧节点的 key，从 keyToNewIndexMap 中可以获取到新节点对应的位置
64 newIndex = keyToNewIndexMap.get(prevChild.key)
65 } else {
66 // 旧节点的 key 不存在（无 key 节点）
67 // 那么我们就遍历所有的新节点（s2 = newChildrenStart; e2 = newChildrenEnd），找到《没有
68 for (j = s2; j <= e2; j++) {
69 // 找到《没有找到对应旧节点的新节点，并且该新节点可以和旧节点匹配》（s2 = newChildrenSt
70 if (
71     newIndexToOldIndexMap[j - s2] === 0 &&
72     isSameVNodeType(prevChild, c2[j] as VNode)
73 ) {
74 // 如果能找到，那么 newIndex = 该新节点索引
75 newIndex = j
76 break
77 }
78 }
79 }
80 // 最终没有找到新节点的索引，则证明：当前旧节点没有对应的新节点
81 if (newIndex === undefined) {
82 // 此时，直接删除即可
83 unmount(prevChild, parentComponent, parentSuspense, true)
84 }
85 // 没有进入 if，则表示：当前旧节点找到了对应的新节点，那么接下来就是要判断对于该新节点而言，
86 else {
87 // 为 newIndexToOldIndexMap 填充值：下标表示：《新节点的下标（newIndex），不计算已处理的
88 // 因为 newIndex 包含已处理的节点，所以需要减去 s2（s2 = newChildrenStart）表示：不计算
89 newIndexToOldIndexMap[newIndex - s2] = i + 1
90 // maxNewIndexSoFar 会存储当前最大的 newIndex，它应该是一个递增的，如果没有递增，则证明
91 if (newIndex >= maxNewIndexSoFar) {
92 // 持续递增
93 maxNewIndexSoFar = newIndex
94 } else {
95 // 没有递增，则需要移动，moved = true
96 moved = true
97 }
98 // 打补丁
99 patch(
100     prevChild,
101     c2[newIndex] as VNode,
102     container,
103     null,
104     parentComponent,
105     parentSuspense,
106     isSVG,
107     slotScopeIds,
108     optimized
109 )
110 // 自增已处理的节点数量
111 patched++
112 }
113 }
114 }

```



```

116 // 仅当节点需要移动的时候，我们才需要生成最长递增子序列，否则只需要有一个空数组即可
117 const increasingNewIndexSequence = moved
118   ? getSequence(newIndexToOldIndexMap)
119   : EMPTY_ARR
120 // j >= 0 表示：初始值为 最长递增子序列的最后下标
121 // j < 0 表示：《不存在》最长递增子序列。
122 j = increasingNewIndexSequence.length - 1
123 // 倒序循环，以便我们可以使用最后修补的节点作为锚点
124 for (i = toBePatched - 1; i >= 0; i--) {
125   // nextIndex（需要更新的新节点下标）= newChildrenStart + i
126   const nextIndex = s2 + i
127   // 根据 nextIndex 拿到要处理的 新节点
128   const nextChild = c2[nextIndex] as VNode
129   // 获取锚点（是否超过了最长长度）
130   const anchor =
131     nextIndex + 1 < l2 ? (c2[nextIndex + 1] as VNode).el : parentAnchor
132   // 如果 newIndexToOldIndexMap 中保存的 value = 0，则表示：新节点没有用对应的旧节点，此时需
133   if (newIndexToOldIndexMap[i] === 0) {
134     // 挂载新节点
135     patch(
136       null,
137       nextChild,
138       container,
139       anchor,
140       parentComponent,
141       parentSuspense,
142       isSVG,
143       slotScopeIds,
144       optimized
145     )
146   }
147   // moved 为 true，表示需要移动
148   else if (moved) {
149     // j < 0 表示：不存在 最长递增子序列
150     // i !== increasingNewIndexSequence[j] 表示：当前节点不在最后位置
151     // 那么此时就需要 move（移动）
152     if (j < 0 || i !== increasingNewIndexSequence[j]) {
153       move(nextChild, container, anchor, MoveType.REORDER)
154     } else {
155       // j 随着循环递减
156       j--
157     }
158   }
159 }
160 }

```

由以上代码可知：

1. 乱序下的 `diff` 是 **最复杂** 的一块场景
2. 它的主要逻辑分为三大步：
 1. 创建一个 `<key（新节点的 key）:index（新节点的位置）>` 的 `Map` 对象 `keyToNewIndexMap`。通过该对象可知：新的 `child`（根据 `key` 判断指定 `child`）更新后的位置（根据对应的 `index` 判断）在哪里
 2. 循环 `oldChildren`，并尝试进行 `patch`（打补丁）或 `unmount`（删除）旧节点
 3. 处理 移动和挂载

13: 源码逻辑：场景五：求解最长递增子序列 ◀ 上一节 下一节 ▶ 15: 框架实现：场景五：乱序下的 `diff` 比对

 我要提出意见反馈

