

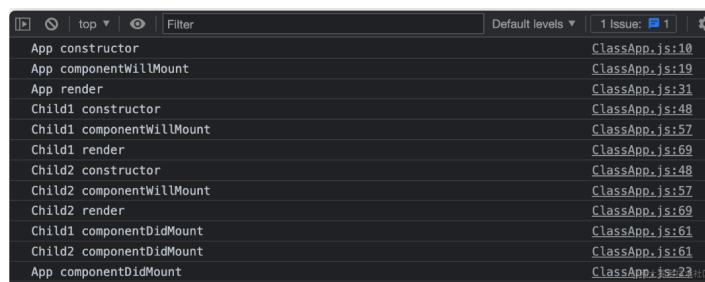
## 函数组件

### 1. 函数组件中的生命周期替代方案?

- 数组组件中不存在类组件的生命周期，但可以通过hooks替代实现
- `useEffect` & `useLayoutEffect` :
  - `useEffect` 是异步执行的，不会阻塞浏览器渲染
  - `useLayoutEffect` 是同步执行的，在DOM更新之后，浏览器渲染绘制之前
  - 需要修改 DOM，改变页面布局就使用 `useLayoutEffect`，否则都使用 `useEffect`
- `useInsertionEffect` :
  - 主要用于CSS in JS (样式代码)，如果在`useLayoutEffect` 中使用样式代码的话，此时DOM已经更新了，会造成浏览器再次重新布局，所以应该在DOM变化之前执行，因此有了此hook
- `componentDidMount` 替代方案:
  - `useEffect(()=>{`  
`}, [])` 利用 `useEffect` 第二个参数dep数组为空时，`useEffect` 的回调只会执行一次
- `componentWillUnmount` 替代方案:
  - `useEffect(()=>{ return ()=>{ 移除监听器, 定时器等 } }, [])`
- `componentWillReceiveProps` 替代方案:
  - `useEffect(()=>{}, [props])`
  - `useEffect` 的dep数组可以起到监听作用，但本质上不完全算是替代
- `componentDidUpdate` 替代方案:
  - `componentDidUpdate` 是同步执行的，而 `useEffect` 是异步执行的，但它们的执行时机都是在 commit 阶段
  - 只需要不传递 `useEffect` 的第二个参数即可在每次函数组件执行时执行 `useEffect` 里的回调

### 2. 父子组件生命周期函数调用顺序?

- **首次渲染**----旧的生命周期函数调用顺序:



App constructor	ClassApp.js:10
App componentWillMount	ClassApp.js:19
App render	ClassApp.js:31
Child1 constructor	ClassApp.js:48
Child1 componentWillMount	ClassApp.js:57
Child1 render	ClassApp.js:69
Child2 constructor	ClassApp.js:48
Child2 componentWillMount	ClassApp.js:57
Child2 render	ClassApp.js:69
Child1 componentDidMount	ClassApp.js:61
Child2 componentDidMount	ClassApp.js:61
App componentDidMount	ClassApp.js:73

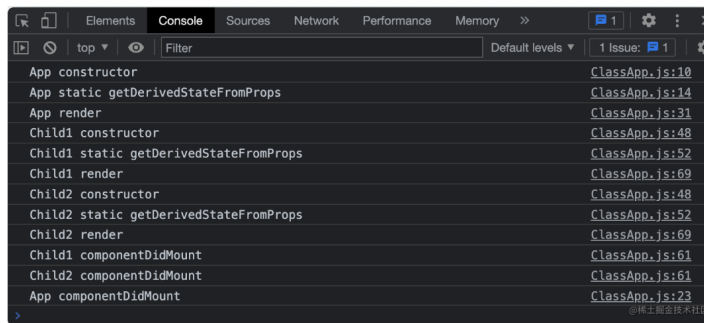
其中 `constructor`、`componentWillMount`、`render` 为 `render` 阶段执行的生命周期函数，`componentDidMount` 为 `commit` 阶段执行的生命周期函数。

- 首先【依次】执行父组件 `render` 阶段的生命周期函数;
- 然后【依次】执行子组件 `render` 阶段的生命周期函数;
- 最后【交叉】执行子组件和父组件 `commit` 阶段的生命周期函数。

- React Fiber 树的构建、更新类似于树的先序遍历（深度优先搜索）。在“递归”时，执行 `render` 阶段的生命周期函数；在“回溯”时，执行 `commit` 阶段的生命周期函数
- 于 `render` 阶段的生命周期函数，其顺序是 父组件 -> 子组件；而对于 `commit` 阶段的生命周期函数，其顺序是 子组件 -> 父组件。

- **首次渲染**---新的生命周期函数调用:

■

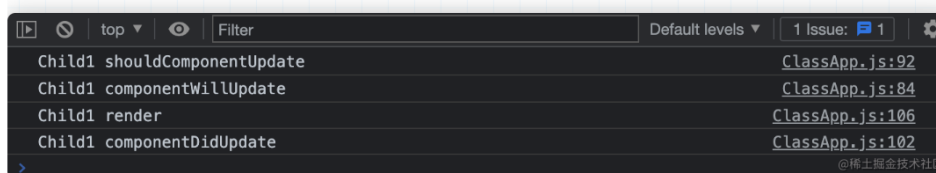


- 可以看到，在首次渲染时，`getDerivedStateFromProps` 的执行顺序基本上替代了 `componentWillMount` 的执行顺序。

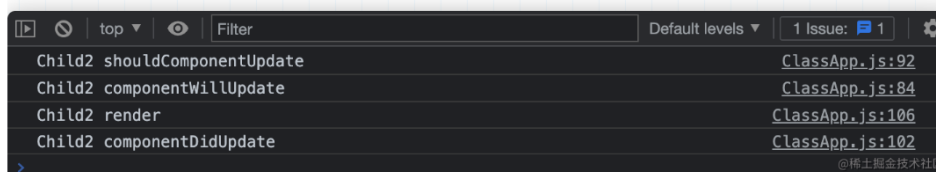
○ 更新时---的调用顺序：

■ 子组件状态改变造成的更新

- （旧生命周期）：
- 当点击文字 Child1 时，其执行结果如下：

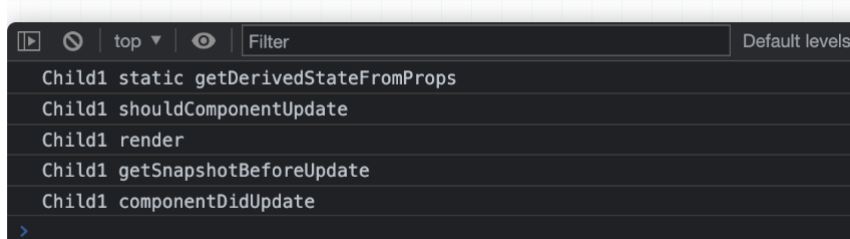


当点击文字 Child2 时，其执行结果如下：

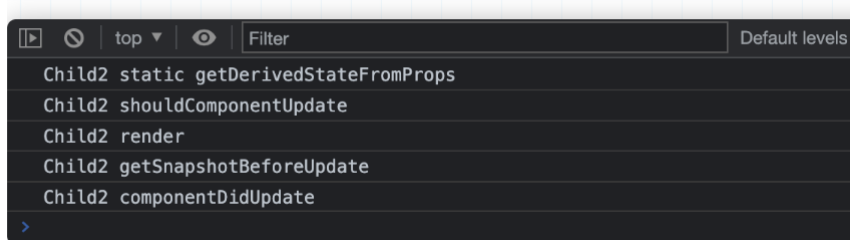


- 注意：此时并没有调用 `componentWillReceiveProps` 生命周期函数，因为使用 `this.setState` 触发组件更新时，并不会调用此生命周期钩子，只有 **props 改变** 或者 **父组件更新导致子组件重新渲染** 时，才会执行这个生命周期钩子，看它的名字也知道它仅和 props 有关。

- （新生命周期）：
- 当点击文字 Child1 时，其执行结果如下：



当点击文字 Child2 时，其执行结果如下：



- 可以看到，子组件的状态发生改变，只会执行该子组件对应的生命周期函数，而不会执行其父组件或其兄弟组件的生命周期函数。

- 首先执行 `getDerivedStateFromProps`，在这里可以根据 props 更新 state；
- 然后判断该组件是否需要更新，即执行 `shouldComponentUpdate`；
- 需要更新则执行 `render` 函数以及后续生命周期函数，否则跳过后面生命周期函数的执行；
- 在将更改提交至 DOM 树之前执行 `getSnapshotBeforeUpdate`，在这里可以获取 DOM 被更改前的最后一次快照；
- 最后在将更改提交至 DOM 树之后执行 `componentDidUpdate`
- 父组件状态改变 造成更新：
  - （旧生命周期）：

```

App shouldComponentUpdate
App componentWillUpdate
App render
Child1 componentWillReceiveProps
Child1 shouldComponentUpdate
Child1 componentWillUpdate
Child1 render
Child2 componentWillReceiveProps
Child2 shouldComponentUpdate
Child2 componentWillUpdate
Child2 render
Child1 componentDidUpdate
Child2 componentDidUpdate
App componentDidUpdate

```

- 首先依次执行父组件 render 阶段的生命周期函数；
- 然后依次执行子组件 render 阶段的生命周期函数；
- 最后交叉执行子组件和父组件 commit 阶段的生命周期函数。
- 因为是在父组件中调用 `this.setState` 方法触发的更新，并不会执行它的 `componentWillReceiveProps` 生命周期函数，而由于父组件更新导致的子组件更新，是会执行子组件的 `componentWillReceiveProps` 生命周期函数的，这点也在子组件状态改变中提到了。
- （新生命周期）：

- 更新时的调用顺序：

```

App static getDerivedStateFromProps
App shouldComponentUpdate
App render
Child1 static getDerivedStateFromProps
Child1 shouldComponentUpdate
Child1 render
Child2 static getDerivedStateFromProps
Child2 shouldComponentUpdate
Child2 render
Child1 getSnapshotBeforeUpdate
Child2 getSnapshotBeforeUpdate
App getSnapshotBeforeUpdate
Child1 componentDidUpdate
Child2 componentDidUpdate
App componentDidUpdate

```

\* 可以看到，换成 `getDerivedStateFromProps` 后，不管是不是通过调用 `this.setState` 导致的组件更新，都会执行 `getDerivedStateFromProps` 生命周期函数。

### 3. State和props之间的区别是什么？

- `props` (“properties” 的缩写) 和 `state` 都是普通的 JavaScript 对象。它们都是用来保存信息的，这些信息可以控制组件的渲染输出，而它们的一个重要的不同点就是：`props` 是传递给组件的（类似于函数的形参），而 `state` 是在组件内被组件自己管理的（类似于在一个函数内声明的变量）。

#### 4. 什么是受控组件？

- 在 HTML 中，表单元素（如 `<input>`、`<textarea>` 和 `<select>`）通常自己维护 `state`，并根据用户输入进行更新。而在 React 中，可变状态（mutable state）通常保存在组件的 `state` 属性中，并且只能通过使用 `setState()` 来更新。
- 我们可以把两者结合起来，使 React 的 `state` 成为“唯一数据源”。渲染表单的 React 组件还控制着用户输入过程中表单发生的操作。被 React 以这种方式控制取值的表单输入元素就叫做“受控组件”。
- 对于受控组件来说，输入的值始终由 React 的 `state` 驱动。你也可以将 `value` 传递给其他 UI 元素，或者通过其他事件处理函数重置，但这意味着你需要编写更多的代码。

#### 5. 什么是非受控组件？

- 在大多数情况下，我们推荐使用 [受控组件](#) 来处理表单数据。在一个受控组件中，表单数据是由 React 组件来管理的。另一种替代方案是使用非受控组件，这时表单数据将交由 DOM 节点来处理。

#### 6. Context 适用场景？

- Context 主要应用场景在于很多不同层级的组件需要访问同样一些的数据。请谨慎使用，因为这会使得组件的复用性变差。

**如果你只是想避免层层传递一些属性，[组件组合 \(component composition\)](#) 有时候是一个比 context 更好的解决方案。**

- `const MyContext=React.createContext('defaultValue')`

- `<MyContext.Provider value={/* 某个值 */}>`

- ```
<MyContext.Consumer>
  {value => /* 基于 context 值进行渲染 */}
</MyContext.Consumer>
```

这种方法需要一个函数作为子元素（function as a child）。这个函数接收当前的 `context` 值，并返回一个 React 节点。

#### ◦ Context.displayName

`context` 对象接受一个名为 `displayName` 的 property，类型为字符串。React DevTools 使用该字符串来确定 `context` 要显示的内容。

#### 7. 性能优化？

- UI 更新需要昂贵的 DOM 操作，因此 React 内部使用了几种巧妙的技术来最小化 DOM 操作次数

#### 8. Portal

- Portal 提供了一种将子节点渲染到存在于父组件以外的 DOM 节点的优秀的方案。例如全局弹框

#### 9. Profiler

- `Profiler` 测量一个 React 应用多久渲染一次以及渲染一次的“代价”。它的目的是识别出应用中渲染较慢的部分，或是可以使用[类似 memoization 优化](#)的部分，并从相关优化中获益。

#### 10. 协调

- 设计动机：

- 在某一时间节点调用 React 的 `render()` 方法，会创建一棵由 React 元素组成的树。在下次 state 或 props 更新时，相同的 `render()` 方法会返回一棵不同的树。React 需要基于这两棵树之间的差别来判断如何高效的更新 UI，以保证当前 UI 与最新的树保持同步
- React 在以下两个假设的基础之上提出了一套  $O(n)$  的启发式算法：
  1. 两个不同类型的元素会产生出不同的树；
  2. 开发者可以使用 `key` 属性标识哪些子元素在不同的渲染中可能是不变的。
- diff 算法：
  - **对比不同类型的元素**：当根节点为不同类型的元素时，React 会拆卸原有的树并且建立起新的树。
  - **对比同一类型的元素**：当对比两个相同类型的 React 元素时，React 会保留 DOM 节点，仅比对及更新有改变的属性。
  - **对比同一类型的组件元素**：当一个组件更新时，组件实例会保持不变，因此可以在不同的渲染时保持 state 一致。React 将更新该组件实例的 props 以保证与最新的元素保持一致，并且调用该实例的 `UNSAFE_componentWillReceiveProps()`、`UNSAFE_componentWillUpdate()` 以及 `componentDidUpdate()` 方法。然后调用 `render()` 方法，diff 算法将在之前的结果以及新的结果中进行递归。
  - 对子节点进行递归：默认情况下，当递归 DOM 节点的子元素时，React 会同时遍历两个子元素的列表；当产生差异时，生成一个 mutation。

## 11. Web Components

- React 和 [Web Components](#) 为了解决不同的问题而生。Web Components 为可复用组件提供了强大的封装，而 React 则提供了声明式的解决方案，使 DOM 与数据保持同步。两者旨在互补。作为开发人员，可以自由选择使用 Web Components 中使用 React，或者在 React 中使用 Web Components，或者两者共存。

大多数开发者在使用 React 时，不使用 Web Components，但可能你会需要使用，尤其是在使用 Web Components 编写的第三方 UI 组件时。

## 12. 何时以及为什么 `setState()` 会批量执行？

## 13. 为什么不直接更新 `this.state`？

## 14. `render()`

- `render(element, container[, callback])`
- 在提供的 `container` 里渲染一个 React 元素，并返回对该组件的引用（或者针对[无状态组件](#)返回 `null`）。
- 如果 React 元素之前已经在 `container` 里渲染过，这将会对其执行更新操作，并仅会在必要时改变 DOM 以映射最新的 React 元素
- 注意：
  - `render()` 会控制你传入容器节点里的内容。当首次调用时，容器节点里的所有 DOM 元素都会被替换，后续的调用则会使用 React 的 DOM 差分算法（DOM diffing algorithm）进行高效的更新。
  - `render()` 不会修改容器节点（只会修改容器的子节点）。可以在不覆盖现有子节点的情况下，将组件插入已有的 DOM 节点中。
  - `render()` 目前会返回对根组件 `ReactComponent` 实例的引用。但是，目前应该避免使用返回的引用，因为它是历史遗留下来的内容，而且在未来版本的 React 中，组件渲染在某些情况下可能会是异步的。如果你真的需要获得对根组件 `ReactComponent` 实例的引用，那么推荐为根元素添加 [callback ref](#)。
  - 使用 `render()` 对服务端渲染容器进行 hydrate 操作的方式已经被废弃，并且会在 React 17 被移除。作为替代，请使用 [hydrateRoot\(\)](#)。

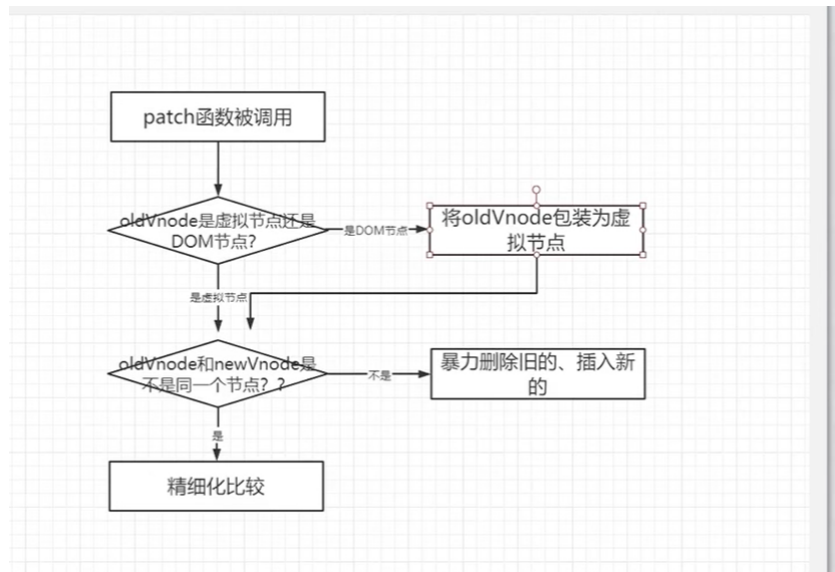
## 15. 产生虚拟节点的h函数?

调用h函数: `h('a',{props:{href:'http://www.baidu.com'}},'百度')`

将得到这样的虚拟节点vnode: `{"sel": "a", "data": {"props": {"href": "http://www.baidu.com"}}}, text: "百度"`

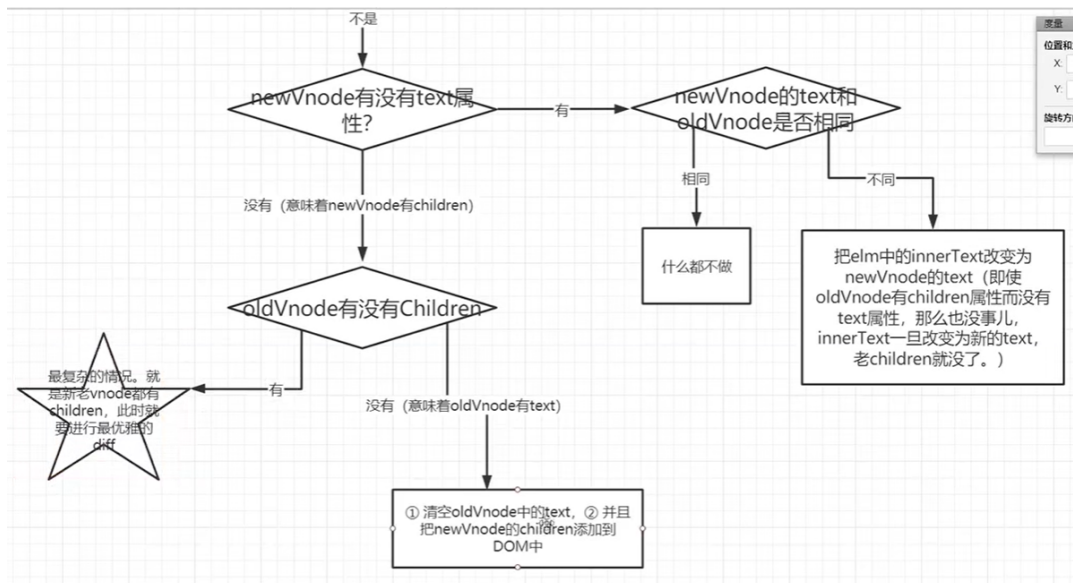
## 16. Diff算法内部细节?

o



o 节点相同时的比较

o



o \* diff中的命中查找: (按照顺序查找, 命中其中一个就不在继续命中了, 如果四都没命中 那么就需要循环来找)

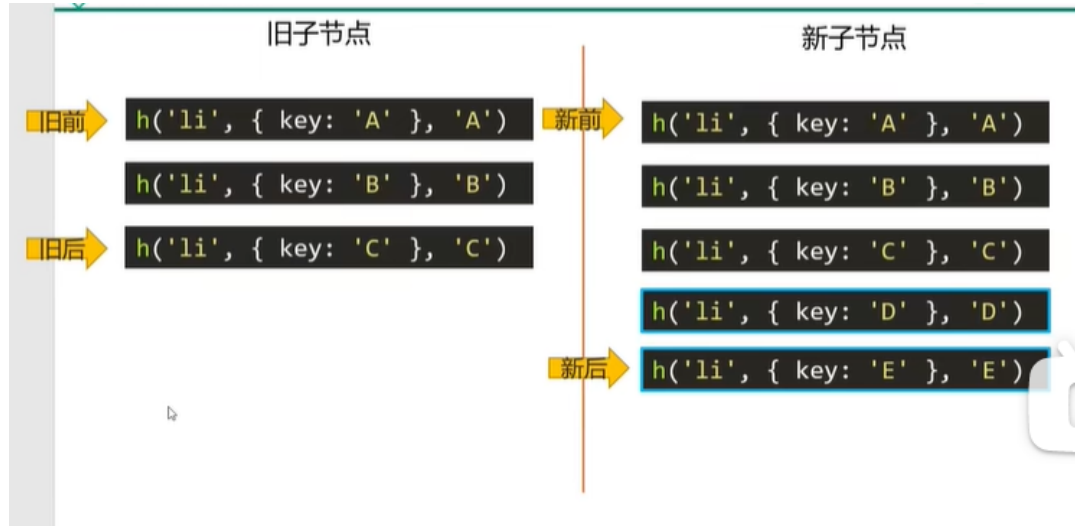
\* 1.新前与旧前 新前: newVnode中的, 所有未处理的开头节点

\* 2.新后与旧后 新后: newVnode中的, 所有未处理的节点中最后一个节点

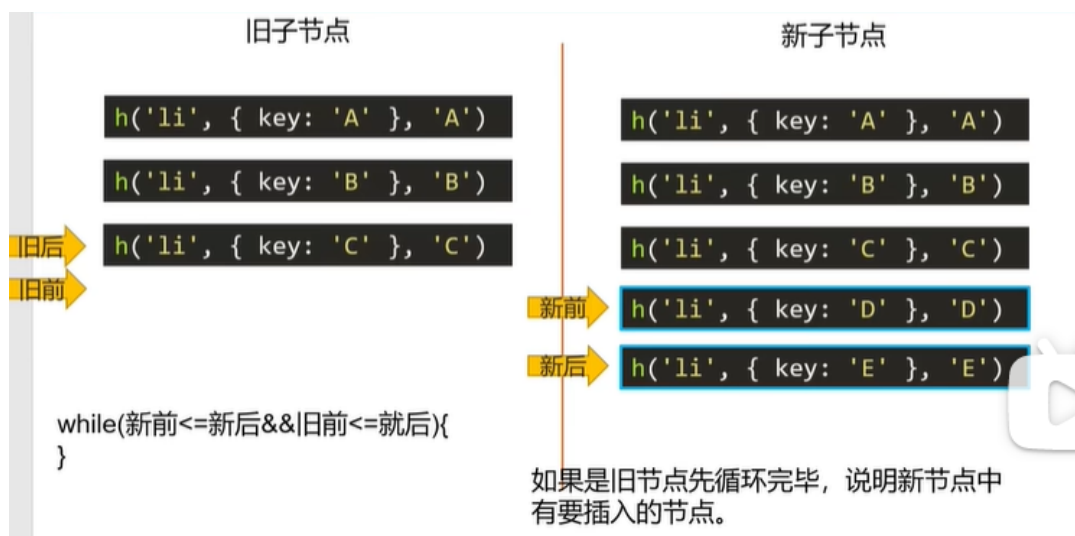
\* 3.新后与旧前 (此种情况下, 将新前指向的节点, 移动到旧后之后)



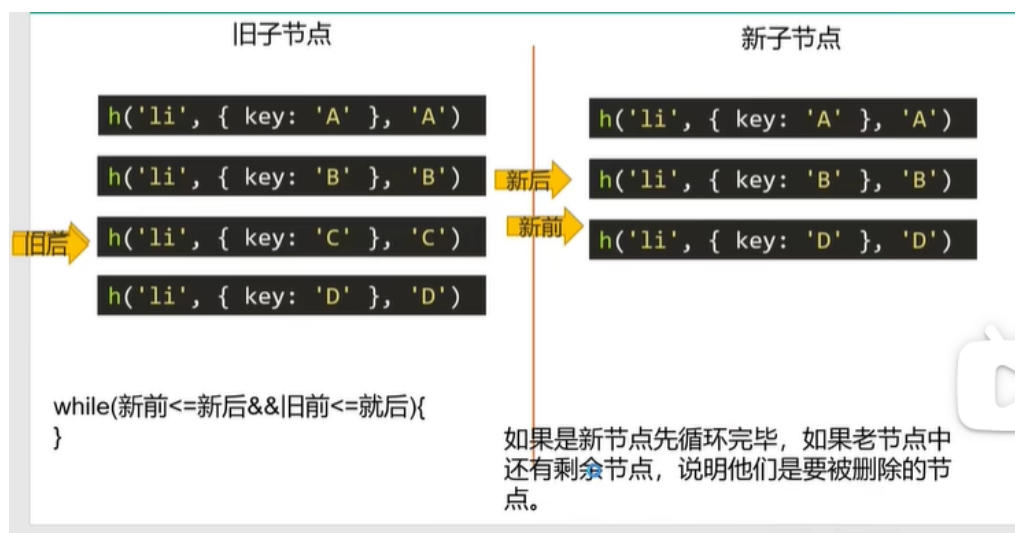
\* 4.新前与旧后 （此种情况下，将新前指向的节点，移动到旧前之前）



#### ■ 有新增节点时



#### ■ 有删除节点时



### 17. React中的fiber特性？

- 增量渲染
- 暂停、中止、重复渲染任务
- 不同更新的优先级
- 并发方面新的基础能力

