# vdom diff
# 算法原理概述

stormqx

# Diff

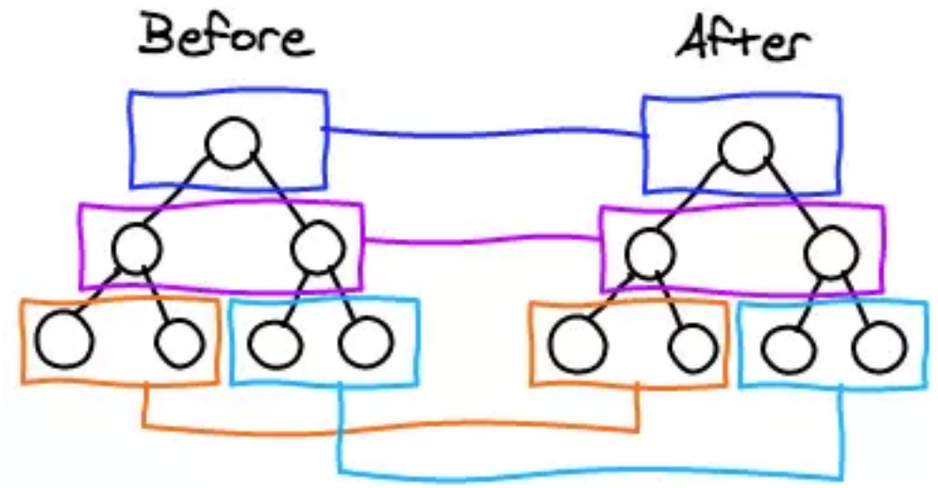
- Diff算法是vdom的加速器
- Diff算法会计算出vdom中真正变化的部分
- Diff算法只会对变化的部分进行原生DOM操作

# 三个假设

1. Web UI中DOM节点跨层级的移动操作很少
2. 不同type的元素生成不同的树
3. 在两次渲染中，同一层级的一组子节点，能通过 key 属性判断哪一个子元素保持不变

# 三个策略

1. 逐级比较




2. 不同类型的组件，直接替换

3. 子节点使用diff算法

# 分析oldChildren和newChildren都有多个节点的情况


# 朴素思路：

1. 遍历旧的子节点，将其全部移除

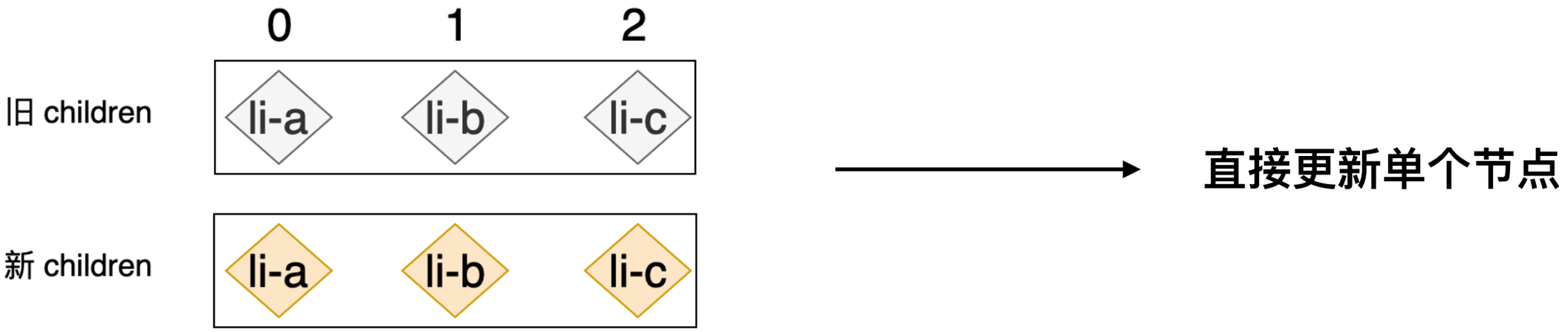
2. 再遍历新的子节点，将其全部添加

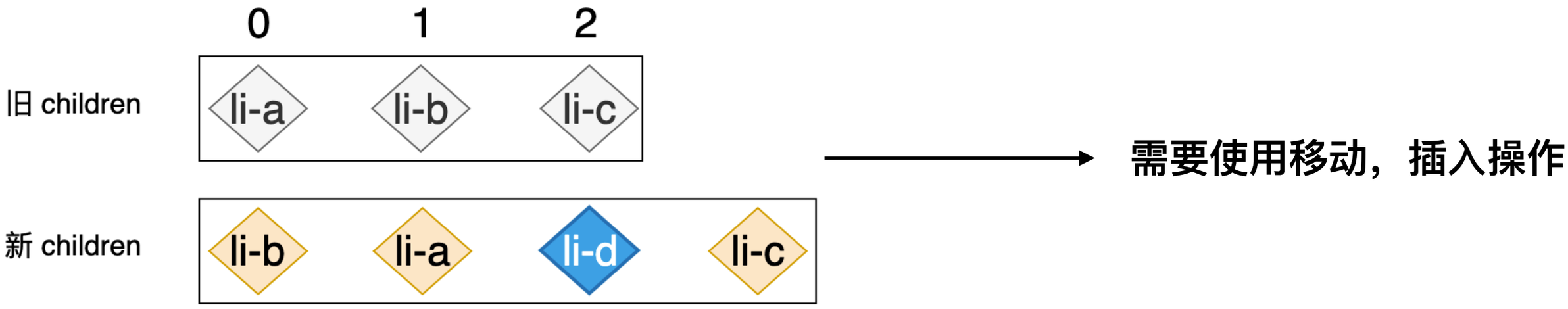
# 理想状态

- 尽可能的复用DOM元素，利用**key**
- 相同key值的节点可以通过**移动**来更新
- 新出现节点通过**插入**操作更新
- 消失节点通过**删除**操作更新

# React(v16以前)

据说v16 diff很快...


0
1
2
旧 children
li-a
li-b
li-c
直接更新单个节点
新 children
li-a
li-b
li-c

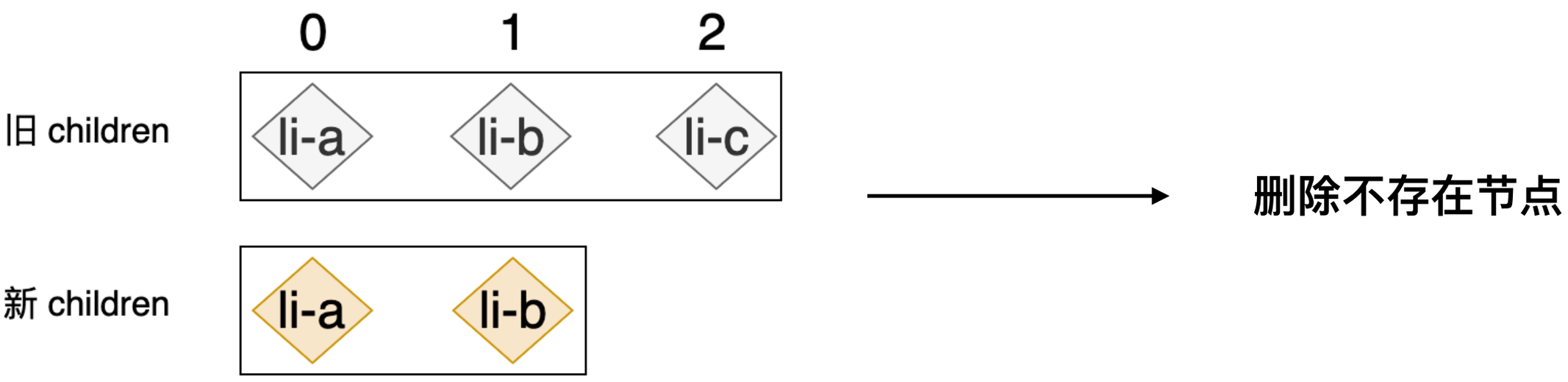



# 如何移动?

**如果在寻找的过程中遇到的索引呈现递增趋势，则说明新旧children中节点顺序相同，不需要移动操作。**

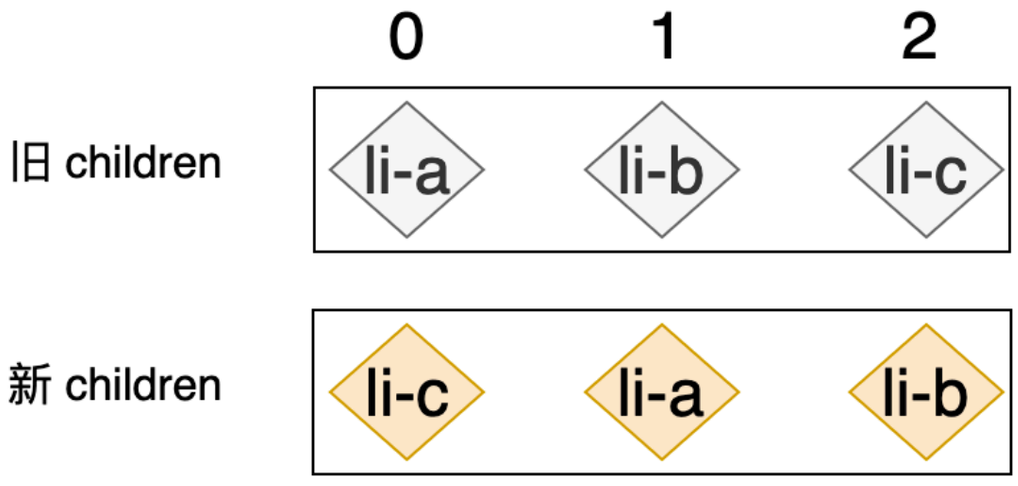
**维护了一个变量lastIndex，存储寻找过程中遇到的最大索引值**

- 节点b不需要移动。
- 节点a移动到节点b后。
- 节点d插入到节点a后。
- 节点c不需要移动。




**需要再遍历一次旧children，确保不存在节点被删除。**

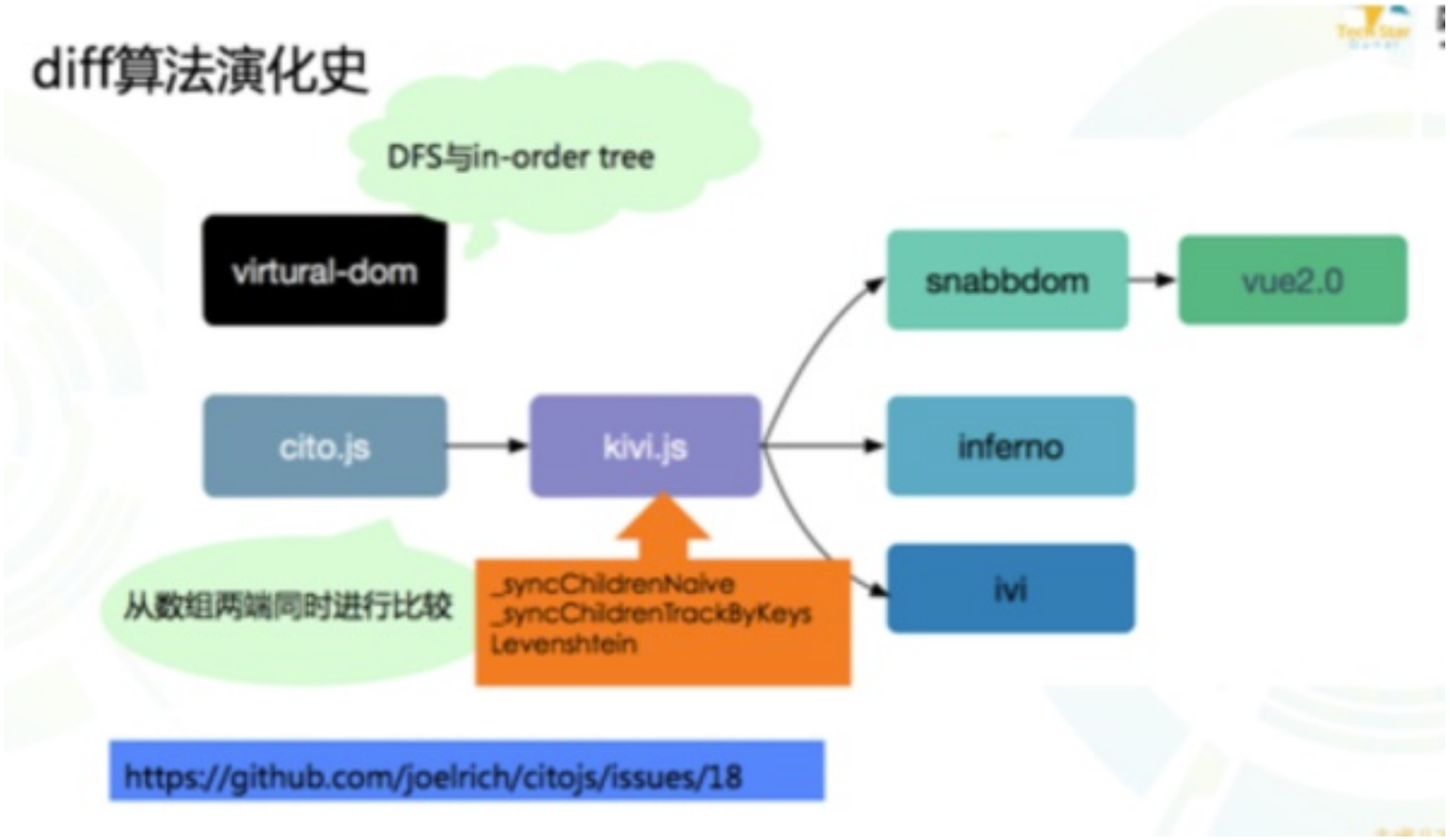
# 缺点：



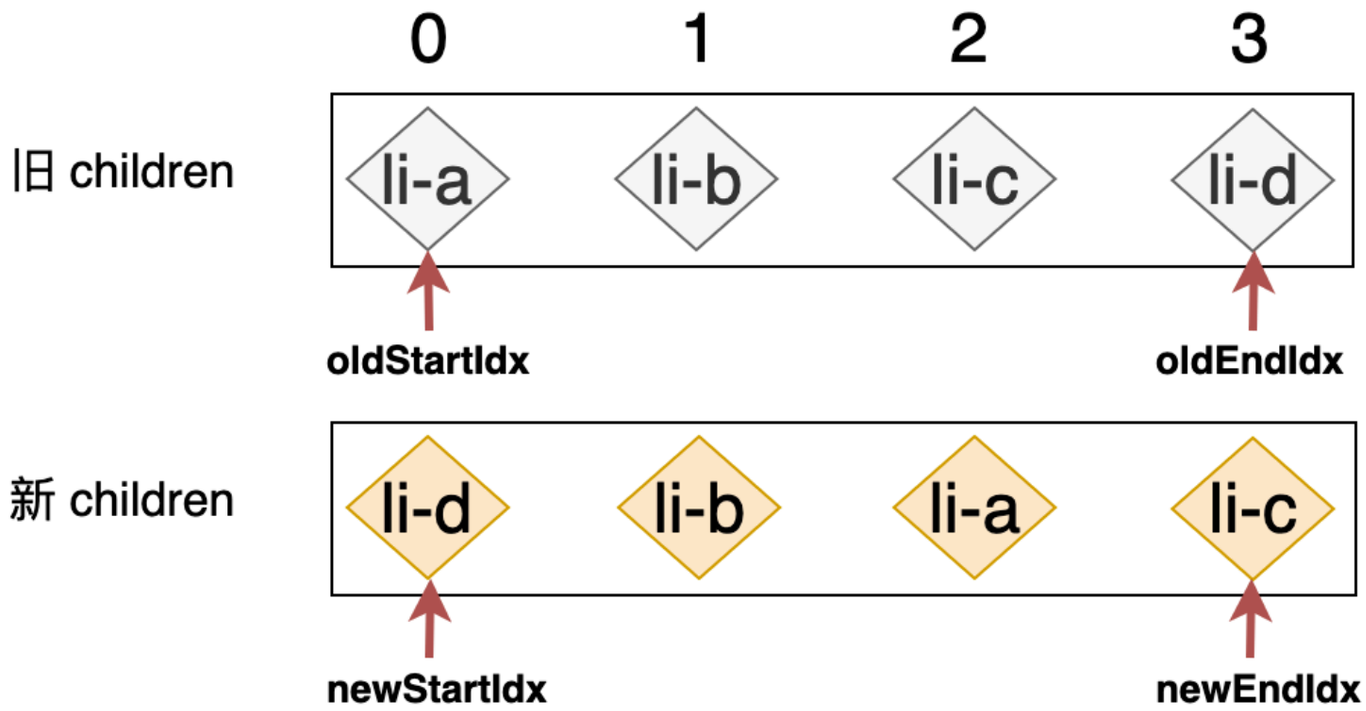

- 上述结构，react会进行两次移动操作
- 实际上，只需要将节点c移到最前面即可

# Vue2

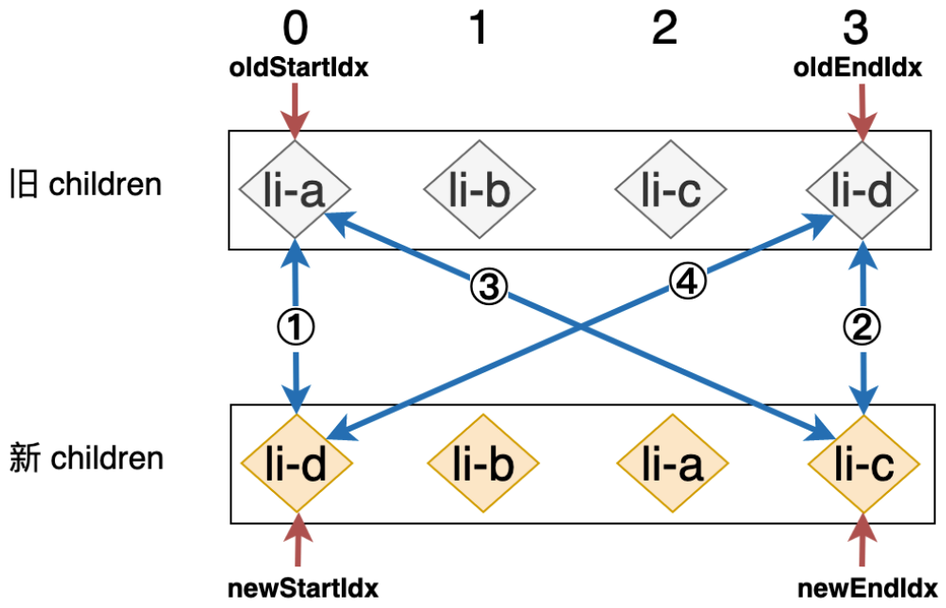
# 双端比较




- 最早出现的是virtual-dom库，很学院派(DFS + in-order tree)。
- 接下来cito.js横空出世，提出双端比较，diff速度拉高好几个层级。
- kivi.js在cito.js基础上，提出了两项优化。
    1. 使用key实现移动追踪
    2. 基于key的编辑长度矩离算法应用（算法复杂度 为O(n^2)）
- 但是编辑距离太复杂了，snabbdom 将 kivi.js 进行简化，去掉编辑长度矩离算法，调整两端比较算法。速度略有损失，但可读性大大提高。再之后，vue2直接把snabbdom库的vdom合掉了。
- 最后是inferno，使用多种优化方案将性能提至最高，因此其作者便邀请进react core team，负责react的性能优化了。这个我后面会详细。




**oldStartIdx、oldEndIdx、newStartIdx 以及 newEndIdx 分别存储旧 children 和新 children 的两个端点的位置索引**



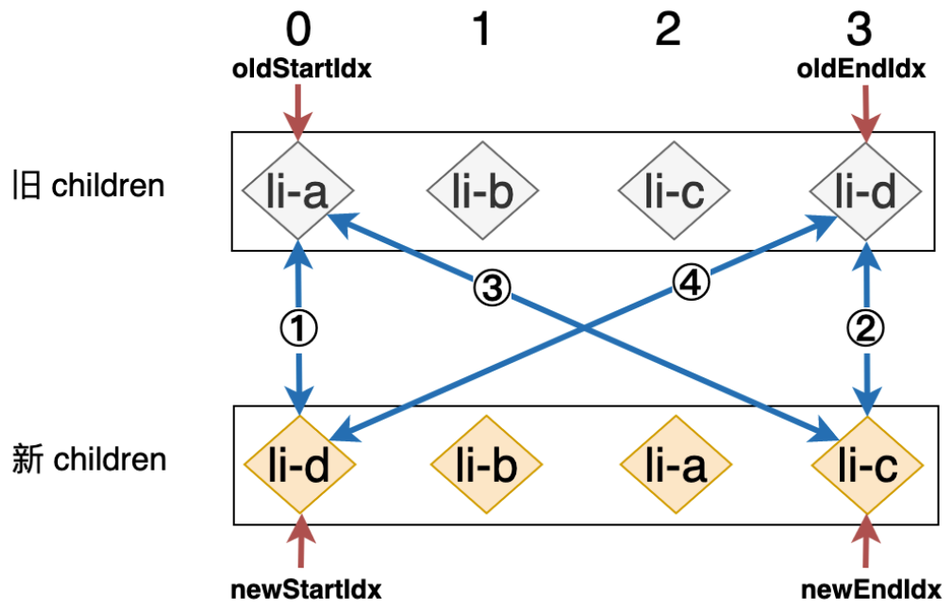

**按照图中所示的顺序依次对双端节点进行比较**
**(前提oldStartIdx <= oldEndIdx, newStartIdx <= newEndIdx)：**
**① 命中， oldStartIdx++, newStartIdx++**
**② 命中，oldEndIdx--, newEndIdx—**
**③ 命中，移动该节点，插入到oldStartIdx前面。将旧children中该位置**
**④ 命中，移动该节点，插入到oldEndIdx后面**

```
while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
  if (oldStartVNode.key === newStartVNode.key) {
  // 步骤一：oldStartVNode 和 newStartVNode 比对
  } else if (oldEndVNode.key === newEndVNode.key) {
    // 步骤二：oldEndVNode 和 newEndVNode 比对
  } else if (oldStartVNode.key === newEndVNode.key) {
    // 步骤三：oldStartVNode 和 newEndVNode 比对
  } else if (oldEndVNode.key === newStartVNode.key) {
    // 步骤四：oldEndVNode 和 newStartVNode 比对
  }
}
```

```
while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
  if (oldStartVNode.key === newStartVNode.key) {
    // 步骤一：oldStartVNode 和 newStartVNode 比对
  } else if (oldEndVNode.key === newEndVNode.key) {
    // 步骤二：oldEndVNode 和 newEndVNode 比对

    // 调用 patch 函数更新
    patch(oldEndVNode, newEndVNode, container)
    // 更新索引，指向下一个位置
    oldEndVNode = prevChildren[--oldEndIdx]
    newEndVNode = newEndVNode[--newEndIdx]
  } else if (oldStartVNode.key === newEndVNode.key) {
    // 步骤三：oldStartVNode 和 newEndVNode 比对
  } else if (oldEndVNode.key === newStartVNode.key) {
    // 步骤四：oldEndVNode 和 newStartVNode 比对

    // 先调用 patch 函数完成更新
    patch(oldEndVNode, newStartVNode, container)
    // 更新完成后，将容器中最后一个子节点移动到最前面，使其成为第一个子节点
    container.insertBefore(oldEndVNode.el, oldStartVNode.el)
    // 更新索引，指向下一个位置
    oldEndVNode = prevChildren[--oldEndIdx]
    newStartVNode = nextChildren[++newStartIdx]
  }
}
```

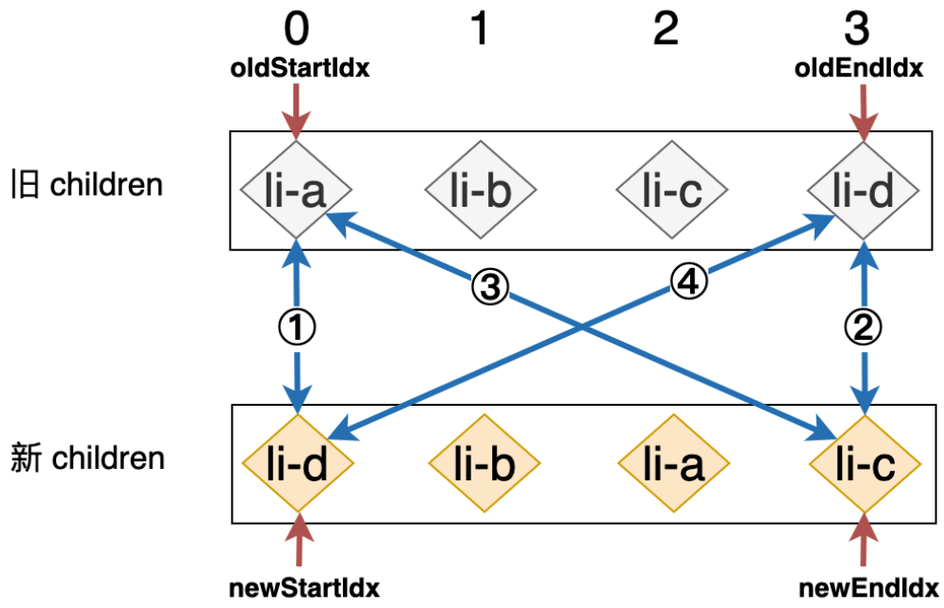



## 如果全部没有命中，怎么办?

1. 根据key，尝试着找newStartIdx节点在旧children中的位置。
2. 如果有，意味着：旧 children中的这个节点所对应的真实 DOM 在新 children的顺序中，已经变成了第一个节点。进行移动操作。

   因为移动的是中间节点，需要将旧children中该节点置空，旧children双端索引扫描的时候就可以直接跳过。
3. 如果没有，意味着：该节点是一个全新的节点，插入即可。
4. 将newStartIdx++。

```
while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
  if (oldStartVNode.key === newStartVNode.key) {
    // 步骤一：oldStartVNode 和 newStartVNode 比对
  } else if (oldEndVNode.key === newEndVNode.key) {
    // 步骤二：oldEndVNode 和 newEndVNode 比对
  } else if (oldStartVNode.key === newEndVNode.key) {
    // 步骤三：oldStartVNode 和 newEndVNode 比对
  } else if (oldEndVNode.key === newStartVNode.key) {
    // 步骤四：oldEndVNode 和 newStartVNode 比对
  } else if {
    // 遍历旧 children，试图寻找与 newStartVNode 拥有相同 key 值的元素
    const idxInOld = prevChildren.findIndex(
      node => node.key === newStartVNode.key
    )
    if(idxInOld) {
      // vnodeToMove 就是在旧 children 中找到的节点，该节点所对应的真实 DOM 应该被移动到最前面
      const vnodeToMove = prevChildren[idxInOld]
      // 调用 patch 函数完成更新
      patch(vnodeToMove, newStartVNode, container)
      // 把 vnodeToMove.el 移动到最前面，即 oldStartVNode.el 的前面
      container.insertBefore(vnodeToMove.el, oldStartVNode.el)
      // 由于旧 children 中该位置的节点所对应的真实 DOM 已经被移动，所以将其设置为 undefined
      prevChildren[idxInOld] = undefined
    } else {
      // 使用 mount 函数挂载新节点
      mount(newStartVNode, container, false, oldStartVNode.el)
    }
    newStartVNode = nextChildren[++newStartIdx]
  }
}
```




按照图中所示的顺序依次对双端节点进行比较

(前提oldStartIdx <= oldEndIdx, newStartIdx <= newEndIdx)：

1. 若oldStartIdx为空，oldStartIdx++
2. 若oldEndIdx为空，oldEndIdx--

增加两步判断

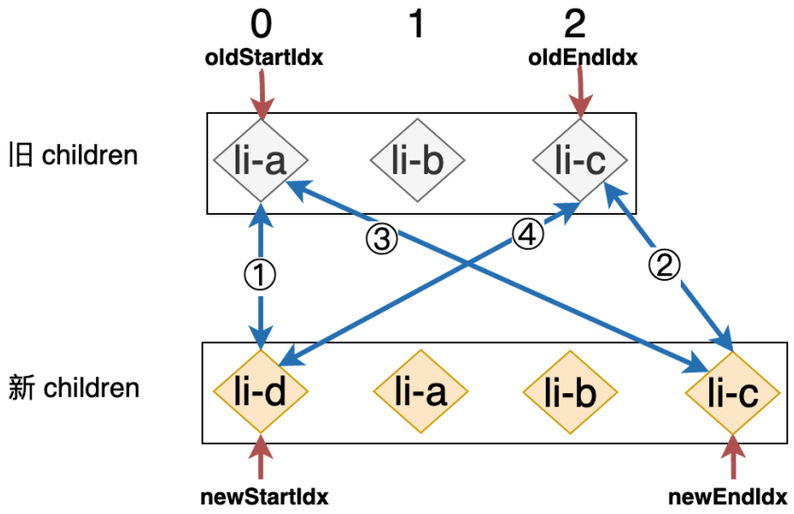
① 命中， oldStartIdx++, newStartIdx++

② 命中，oldEndIdx--, newEndIdx—

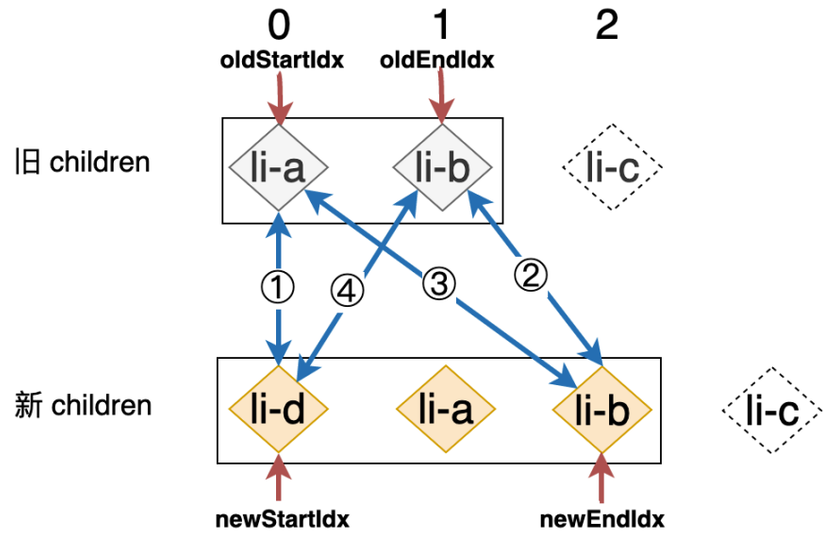
③ 命中，移动该节点，插入到oldStartIdx前面。将旧children中该位置

④ 命中，移动该节点，插入到oldEndIdx后面

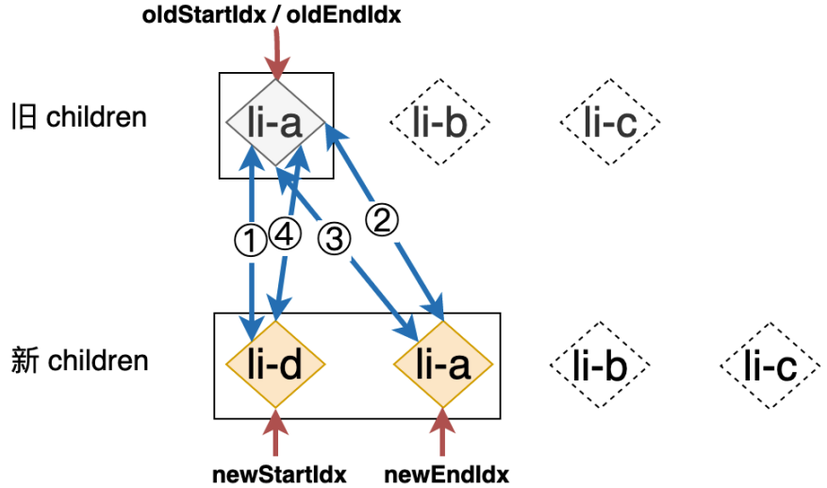
```
while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
  if (!oldStartVNode) {
    oldStartVNode = prevChildren[++oldStartIdx]
  } else if (!oldEndVNode) {
    oldEndVNode = prevChildren[--oldEndIdx]
  } else if (oldStartVNode.key === newStartVNode.key) {
    // 步骤一：oldStartVNode 和 newStartVNode 比对
  } else if (oldEndVNode.key === newEndVNode.key) {
    // 步骤二：oldEndVNode 和 newEndVNode 比对
  } else if (oldStartVNode.key === newEndVNode.key) {
    // 步骤三：oldStartVNode 和 newEndVNode 比对
  } else if (oldEndVNode.key === newStartVNode.key) {
    // 步骤四：oldEndVNode 和 newStartVNode 比对
  } else if {
    // 其他情况
  }
}
```



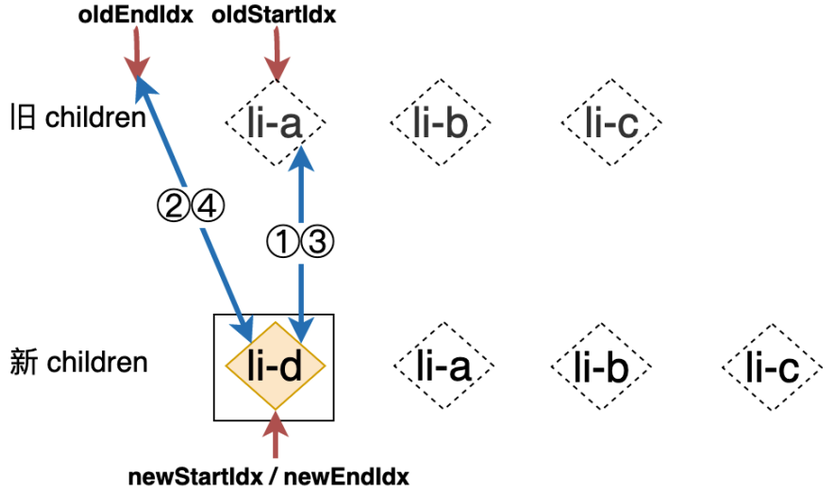

步骤1




步骤2

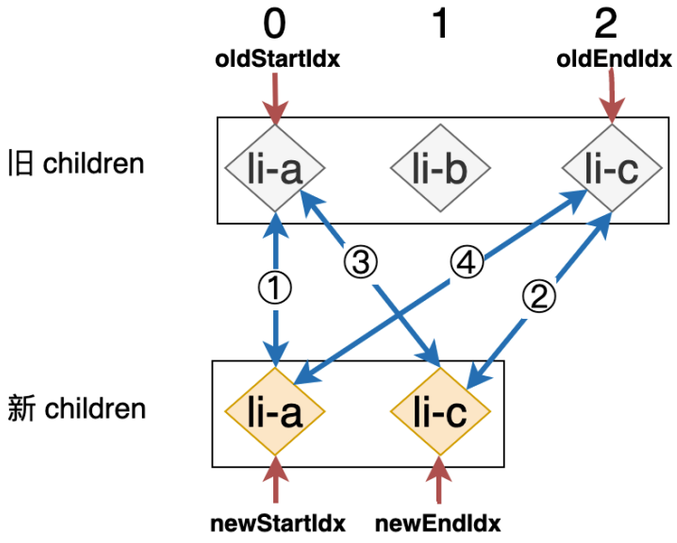



步骤3

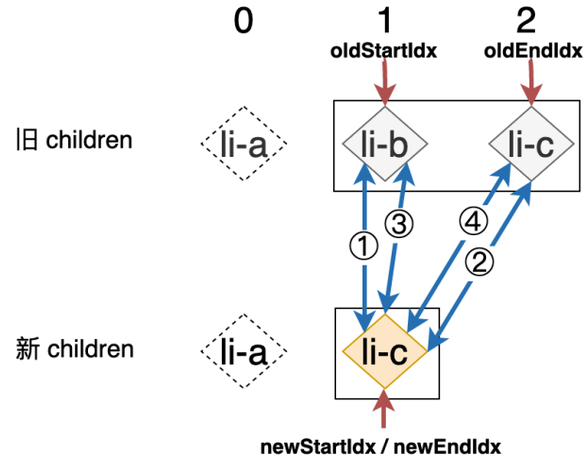



步骤4

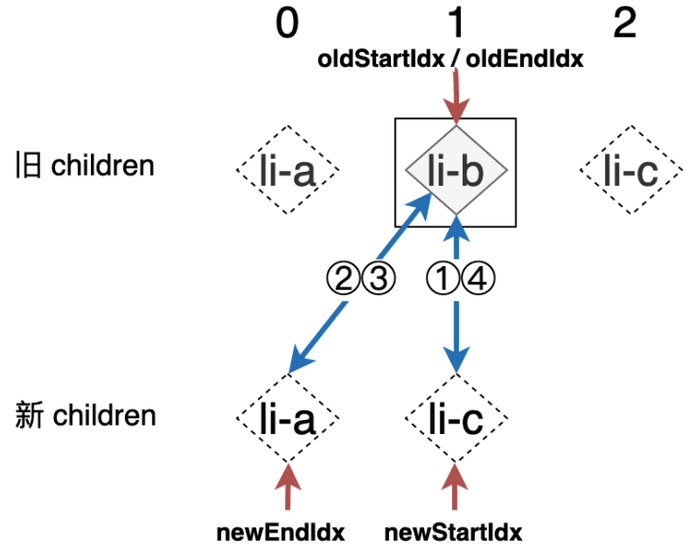
**还存在没有被处理的全新节点**




步骤1




步骤2




步骤3

**还存在没有被移除的节点**

```
while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
  if (!oldStartVNode) {
    // 处理不存在节点
  } else if (!oldEndVNode) {
    // 处理不存在节点
  } else if (oldStartVNode.key === newStartVNode.key) {
    // 步骤一：oldStartVNode 和 newStartVNode 比对
  } else if (oldEndVNode.key === newEndVNode.key) {
    // 步骤二：oldEndVNode 和 newEndVNode 比对
  } else if (oldStartVNode.key === newEndVNode.key) {
    // 步骤三：oldStartVNode 和 newEndVNode 比对
  } else if (oldEndVNode.key === newStartVNode.key) {
    // 步骤四：oldEndVNode 和 newStartVNode 比对
  } else {
    // 其他情况
  }
}
if (oldEndIdx < oldStartIdx) {
  // 添加新节点
  for (let i = newStartIdx; i <= newEndIdx; i++) {
    mount(nextChildren[i], container, false, oldStartVNode.el)
  }
} else if (newEndIdx < newStartIdx) {
  // 移除操作
  for (let i = oldStartIdx; i <= oldEndIdx; i++) {
    container.removeChild(prevChildren[i].el)
  }
}
```

# 总结

- 双端比较可以避免极端情况
- 算法比较折中，适合大多数场景
- 时间复杂度O(n)
- 不能保证移动的次数最少

# Inferno

前端开发 JavaScript React

# 为什么inferno.js这么快?

他

最近在做各个框架的性能测试，发现inferno.js速度快的可怕，几乎已经和原生js持平了，其他框架或多或少的要和它有一定的性能差距，这是因为什么呢?

主要是两个测试，一个是js-repaint-perfs，见下图；


一个是js-framework-benchmark，见Interactive Results

# 核心思想

- 如果在寻找的过程中遇到的索引呈现递增趋势，则说明新旧children中节点顺序相同，不需要移动操作。
- 双端比较。

TEXT1: I use vue for app development

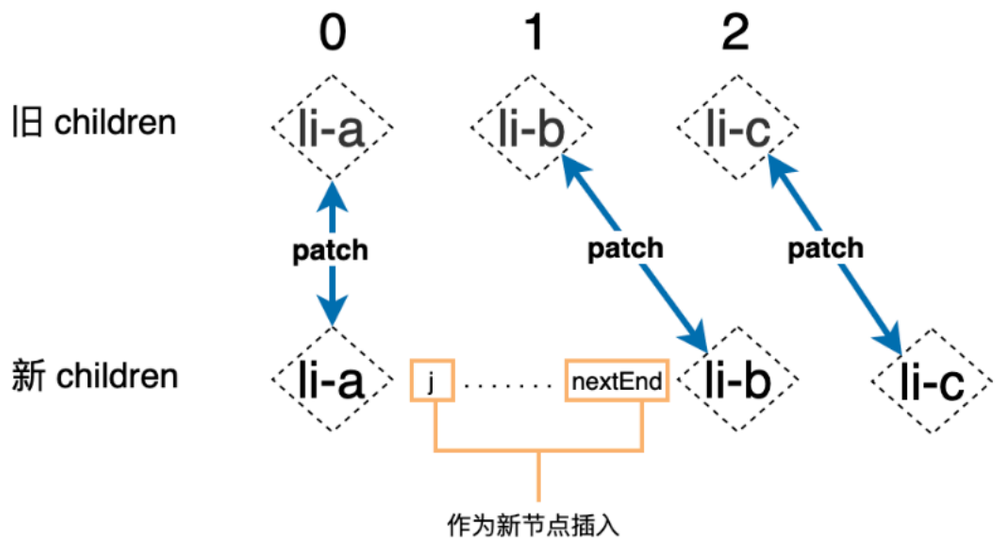
TEXT1: I use react for app development



```
text1: vue
text2: react
```

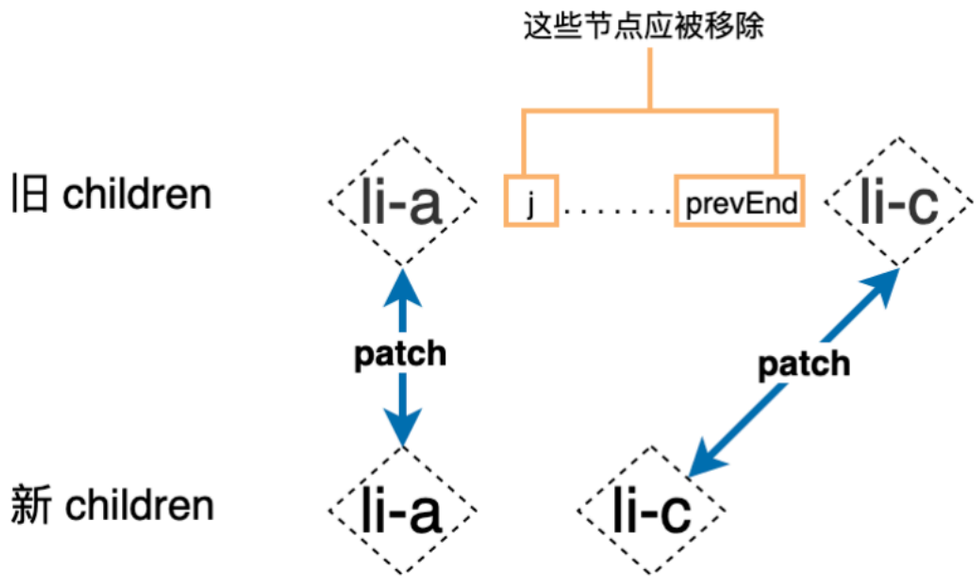
**先去除双端重复的节点**

# 情况一



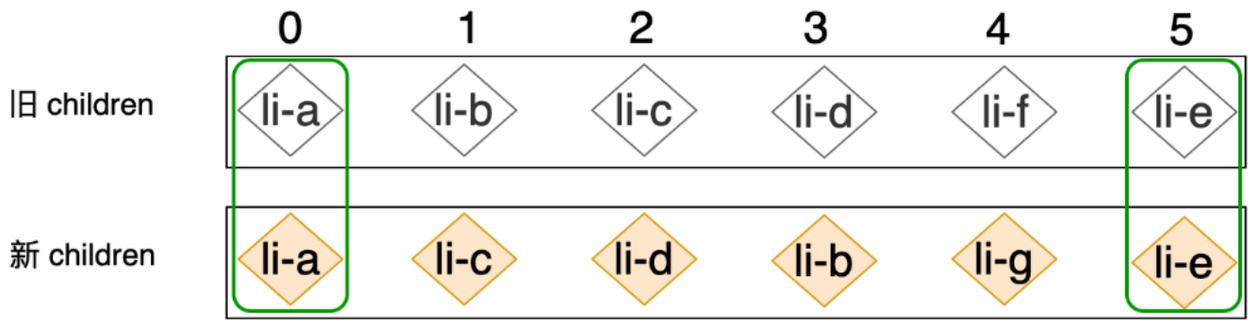

**直接插入新的节点**

# 情况二




**直接删除旧的节点**

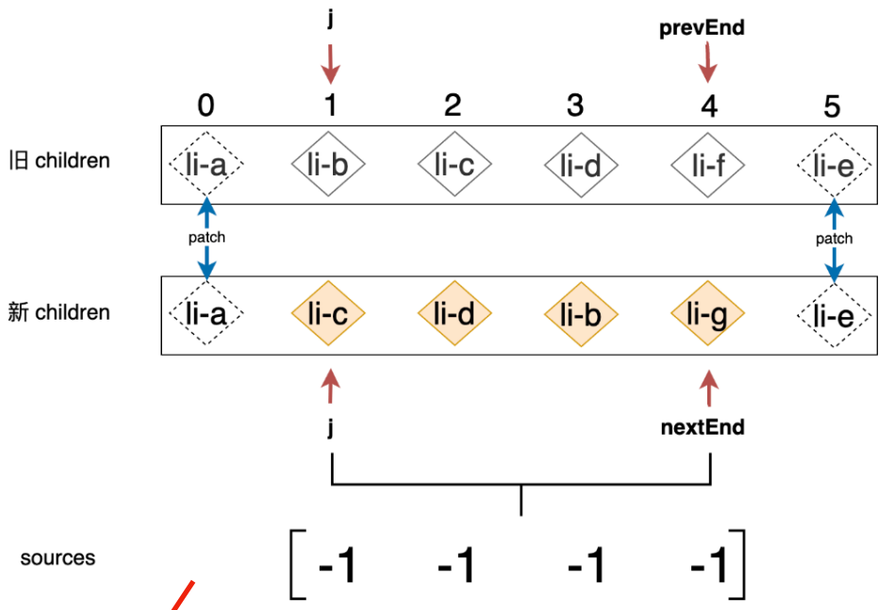
# 情况三



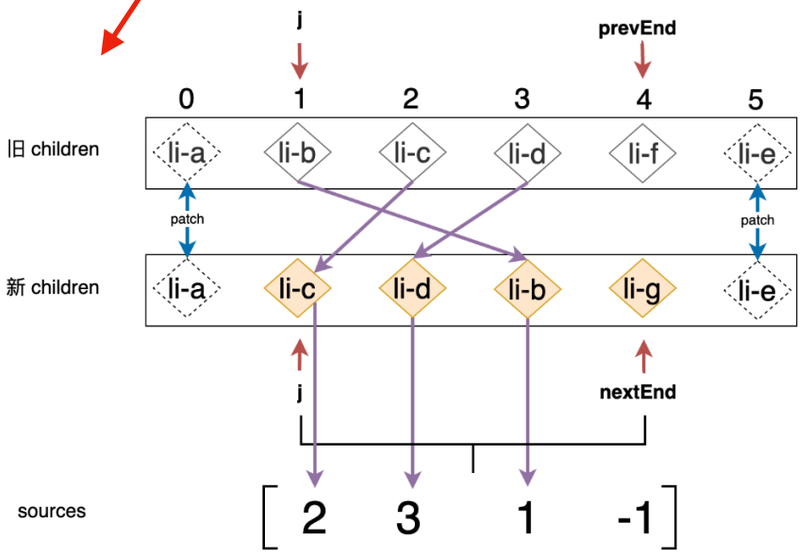

- 如果在寻找的过程中遇到的索引呈现递增趋势，则说明新旧children中节点顺序相同，不需要移动操作。

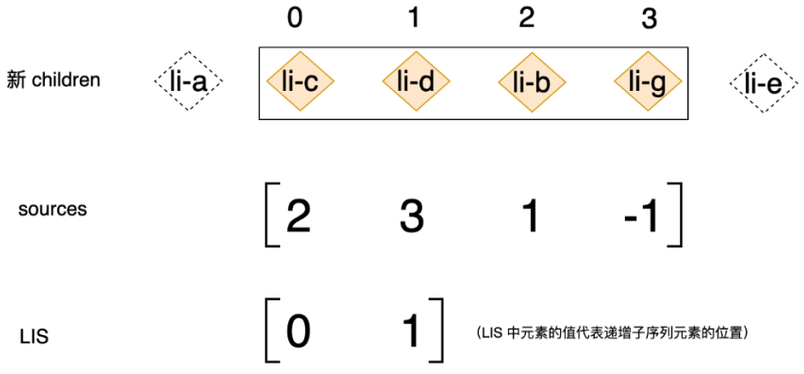
寻找最长递增子序列

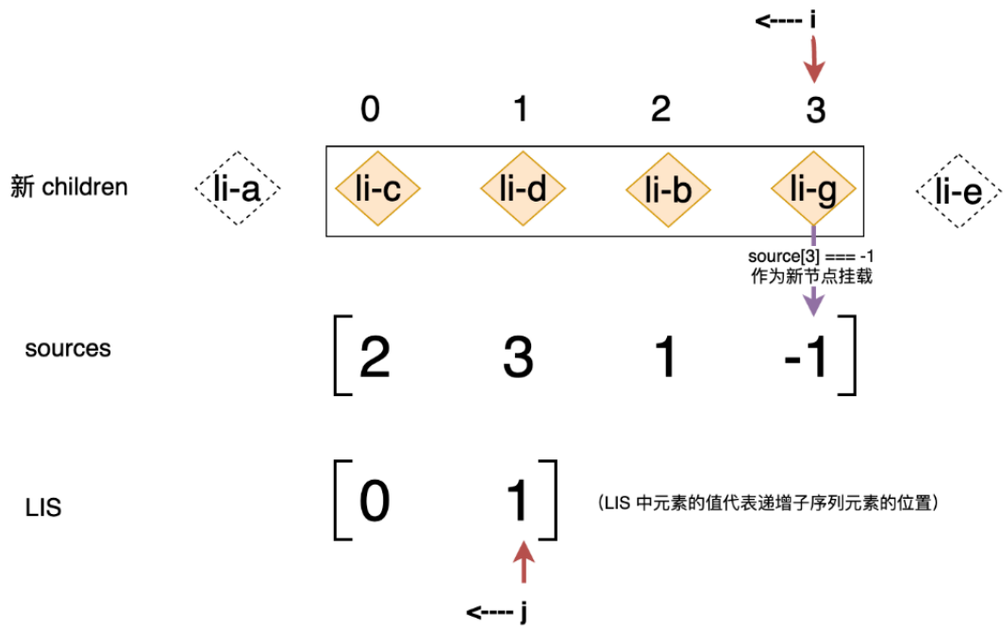
# 怎么做?




1. 维护一个source数组：
   - 存储新 children 中的节点在旧 children 中的位置
   - -1代表新增
2. 维护LIS数组：
   - 代表最长递增子序列
   - LIS中每个元素的值代表递增子序列元素的位置






```
// move变量是在构建source数组的过程中，得到的判断是否需要移动的标记
if (moved) {
  const seq = lis(source)
    // j 指向最长递增子序列的最后一个值
  let j = seq.length - 1
  // 从后向前遍历新 children 中的剩余未处理节点
  for (let i = nextChildren; i >= 0; i--) {
    if (source[i] === -1) {
      // 作为全新的节点挂载
    } else if (i !== seq[j]) {
      // 说明该节点需要移动
    } else {
      // 当 i === seq[j] 时，说明该位置的节点不需要移动
      // 并让 j 指向下一个位置
      j--
    }
  }
}
```

# 总结

- 先用双端比较去重
- 直接计算最长递增子序列，消灭了react15 diff算法的缺点
- 移动的次数可能会少于vue2的双端比较
- 最长递增子序列最快的时间复杂度O(nlgn)

# 谢谢