复习

整体流程

响应化

https://www.processon.com/view/link/5d1eb5a0e4b0fdb331d3798c

异步更新

```
<span id="s">{{foo}}</span>

s.innerHTML // foo
this.foo = 'bar'
s.innerHTML // foo
this.$nextTick(()=>{
  s.innerHTML // bar
})
```

$mount src\platforms\web\runtime\index.js

挂载时执行mountComponent，将dom内容追加至el

```
Vue.prototype.$mount = function (
  el?: string | Element, // 可选参数
  hydrating?: boolean
): Component {
  el = el && inBrowser ? query(el) : undefined
  return mountComponent(this, el, hydrating)
}
```

mountComponent core/instance/lifecycle

创建组件更新函数，创建组件watcher实例

```
updateComponent = () => {
      // 首先执行vm._render() 返回VNode
      // 然后VNode作为参数执行update做dom更新
      vm._update(vm._render(), hydrating)
    }

new Watcher(vm, updateComponent, noop, {
    before () {
      if (vm._isMounted && !vm._isDestroyed) {
        callHook(vm, 'beforeUpdate')
      }
    }
  }, true /* isRenderWatcher */)
```

_render() src\core\instance\render.js

获取组件vnode

```
const { render, _parentVnode } = vm.$options;
vnode = render.call(vm._renderProxy, vm.$createElement);
```

_update src\core\instance\lifecycle.js

执行patching算法，初始化或更新vnode至$el

```
if (!prevVnode) {
      // initial render
      // 如果没有老vnode，说明在初始化
      vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly */)
    } else {
      // updates
      // 更新周期直接diff，返回新的dom
      vm.$el = vm.__patch__(prevVnode, vnode)
    }
```

__patch__ src\platforms\web\runtime\patch.js

定义组件实例补丁方法

```
Vue.prototype.__patch__ = inBrowser ? patch : noop
```

createPatchFunction src\core\vdom\patch.js

创建浏览器平台特有patch函数，主要负责dom更新操作

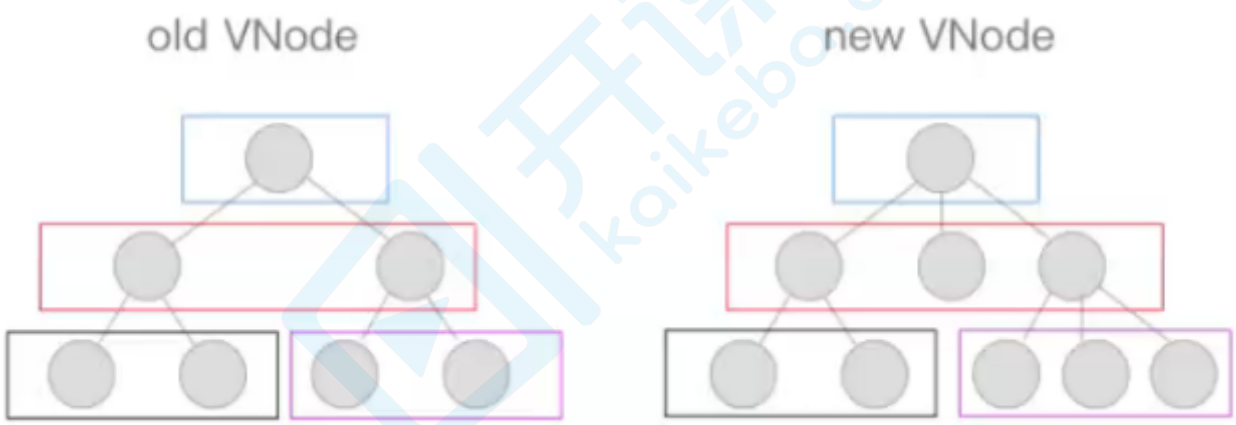
```
// 扩展操作：把通用模块和浏览器中特有模块合并
const modules = platformModules.concat(baseModules)

// 工厂函数：创建浏览器特有的patch函数，这里主要解决跨平台问题
export const patch: Function = createPatchFunction({ nodeOps, modules })
```

patch

那么patch如何工作的呢?

首先说一下patch的核心diff算法：通过**同层的树节点进行比较**而非对树进行逐层搜索遍历的方式，所以时间复杂度只有O(n)，是一种相当高效的算法。

同层级只做三件事：增删改。具体规则是：new VNode不存在就删；old VNode不存在就增；都存在就比较类型，类型不同直接替换、类型相同执行更新；




```
 /*createPatchFunction的返回值，一个patch函数*/
  return function patch (oldVnode, vnode, hydrating, removeOnly, parentElm,
refElm) {
    /*vnode不存在则删*/
    if (isUndef(vnode)) {
      if (isDef(oldVnode)) invokeDestroyHook(oldVnode)
      return
    }

    let isInitialPatch = false
    const insertedVnodeQueue = []

    if (isUndef(oldVnode)) {
      /*oldVnode不存在则创建新节点*/
      isInitialPatch = true
```

开课吧web全栈架构师

```
      createElm(vnode, insertedVnodeQueue, parentElm, refElm)
    } else {
      /*oldVnode有nodeType，说明传递进来一个DOM元素*/
      const isRealElement = isDef(oldVnode.nodeType)

      if (!isRealElement && sameVnode(oldVnode, vnode)) {
        /*是组件且是同一个节点的时候打补丁*/
        patchVnode(oldVnode, vnode, insertedVnodeQueue, removeOnly)
      } else {
        /*传递进来oldVnode是dom元素*/
        if (isRealElement) {
          // 将该dom元素清空
          oldVnode = emptyNodeAt(oldVnode)
        }

        /*取代现有元素：*/
        const oldElm = oldVnode.elm
        const parentElm = nodeOps.parentNode(oldElm)
        //创建一个新的dom
        createElm(
          vnode,
          insertedVnodeQueue,
          oldElm._leaveCb ? null : parentElm,
          nodeOps.nextSibling(oldElm)
        )

        if (isDef(parentElm)) {
          /*移除老节点*/
          removeVnodes(parentElm, [oldVnode], 0, 0)
        } else if (isDef(oldVnode.tag)) {
          /*调用destroy钩子*/
          invokeDestroyHook(oldVnode)
        }
      }
    }

    /*调用insert钩子*/
    invokeInsertHook(vnode, insertedVnodeQueue, isInitialPatch)
    return vnode.elm
  }
```

patchVnode

两个VNode类型相同，就执行更新操作，包括三种类型操作：**属性更新PROPS、文本更新TEXT、子节点更新REORDER**

patchVnode具体规则如下：

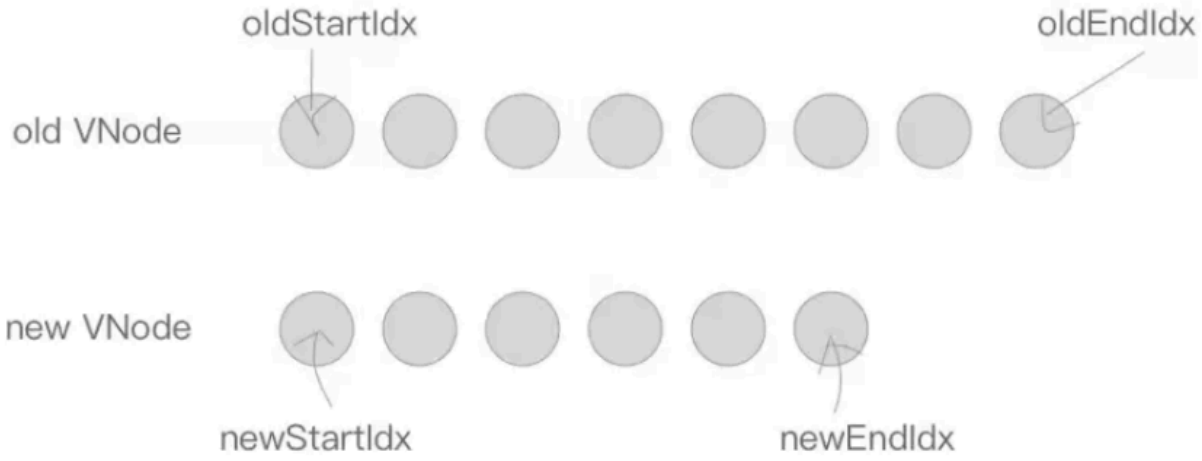
1. 如果新旧VNode都是**静态的**，同时它们的key相同（代表同一节点），并且新的VNode是clone或者是标记了v-once，那么只需要替换elm以及componentInstance即可。
2. 新老节点**均有children**子节点，则对子节点进行diff操作，调用**updateChildren**，这个updateChildren也是diff的核心。
3. 如果**老节点没有子节点而新节点存在子节点**，先清空老节点DOM的文本内容，然后为当前DOM节点加入子节点。
4. 当**新节点没有子节点而老节点有子节点**的时候，则移除该DOM节点的所有子节点。
5. 当**新老节点都无子节点**的时候，只是文本的替换。

```
/*patch VNode节点*/
  function patchVnode (oldVnode, vnode,insertedVnodeQueue,
ownerArray,index,removeOnly) {
    /*两个VNode节点相同则直接返回*/
    if (oldVnode === vnode) {
      return
    }
    if (isDef(vnode.elm) && isDef(ownerArray)) {
      // clone reused vnode
      vnode = ownerArray[index] = cloneVNode(vnode)
    }

    const elm = vnode.elm = oldVnode.elm
    /*
      如果新旧VNode都是静态的，同时它们的key相同（代表同一节点），
      并且新的VNode是clone或者是标记了once（标记v-once属性，只渲染一次），
      那么只需要替换elm以及componentInstance即可。
    */
    if (isTrue(vnode.isStatic) &&
        isTrue(oldVnode.isStatic) &&
        vnode.key === oldVnode.key &&
        (isTrue(vnode.isCloned) || isTrue(vnode.isOnce))) {
      vnode.elm = oldVnode.elm
      vnode.componentInstance = oldVnode.componentInstance
      return
    }

    /*如果存在data.hook.prepatch则要先执行*/
    let i
    const data = vnode.data
    if (isDef(data) && isDef(i = data.hook) && isDef(i = i.prepatch)) {
      i(oldVnode, vnode)
    }

    const oldCh = oldVnode.children
    const ch = vnode.children

    /*执行属性、事件、样式等等更新操作*/
    if (isDef(data) && isPatchable(vnode)) {
      for (i = 0; i < cbs.update.length; ++i) cbs.update[i](oldVnode, vnode)
```

```
      if (isDef(i = data.hook) && isDef(i = i.update)) i(oldVnode, vnode)
    }

    /*开始判断children的各种情况*/
    /*如果这个VNode节点没有text文本时*/
    if (isUndef(vnode.text)) {
      if (isDef(oldCh) && isDef(ch)) {
        /*新老节点均有children子节点，则对子节点进行diff操作，调用updateChildren*/
        if (oldCh !== ch) updateChildren(elm, oldCh, ch, insertedVnodeQueue,
removeOnly)
      } else if (isDef(ch)) {
        /*如果老节点没有子节点而新节点存在子节点，先清空elm的文本内容，然后为当前节点加入子
节点*/
        if (isDef(oldVnode.text)) nodeOps.setTextContent(elm, '')
        addVnodes(elm, null, ch, 0, ch.length - 1, insertedVnodeQueue)
      } else if (isDef(oldCh)) {
        /*当新节点没有子节点而老节点有子节点的时候，则移除所有ele的子节点*/
        removeVnodes(elm, oldCh, 0, oldCh.length - 1)
      } else if (isDef(oldVnode.text)) {
        /*当新老节点都无子节点的时候，只是文本的替换，因为这个逻辑中新节点text不存在，所以
清除ele文本*/
        nodeOps.setTextContent(elm, '')
      }
    } else if (oldVnode.text !== vnode.text) {
      /*当新老节点text不一样时，直接替换这段文本*/
      nodeOps.setTextContent(elm, vnode.text)
    }
    /*调用postpatch钩子*/
    if (isDef(data)) {
      if (isDef(i = data.hook) && isDef(i = i.postpatch)) i(oldVnode, vnode)
    }
  }
```

updateChildren

updateChildren主要作用是用一种较高效的方式比对新旧两个VNode的children得出最小操作补丁。执行一个双循环是传统方式，vue中针对web场景特点做了特别的算法优化，我们看图说话：

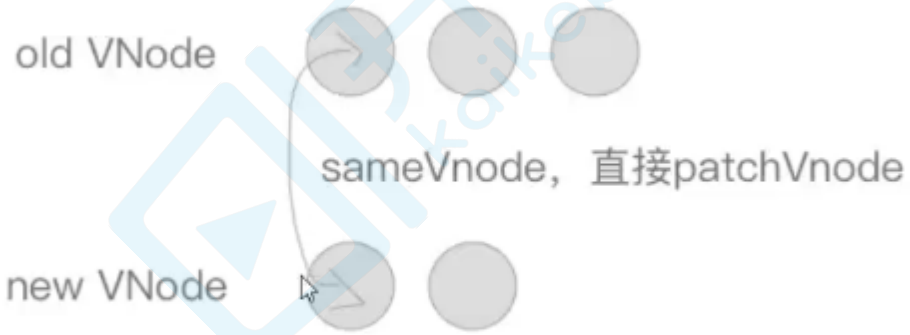


在新老两组VNode节点的左右头尾两侧都有一个变量标记，在**遍历过程中这几个变量都会向中间靠拢**。当**oldStartIdx > oldEndIdx**或者**newStartIdx > newEndIdx**时结束循环。

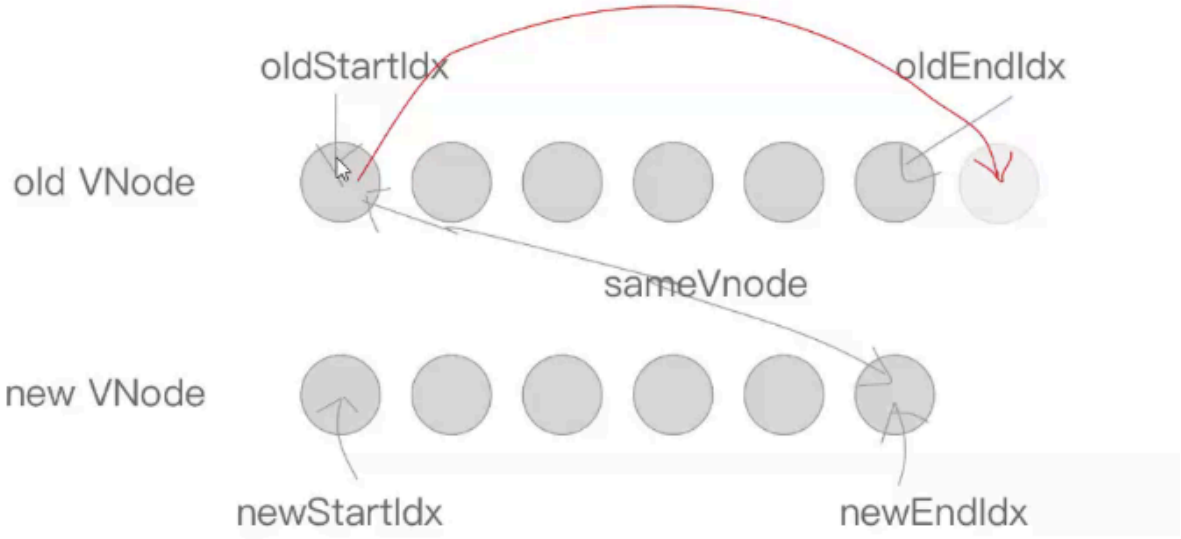
下面是遍历规则：

首先，oldStartVnode、oldEndVnode与newStartVnode、newEndVnode**两两交叉比较**，共有4种比较方法。

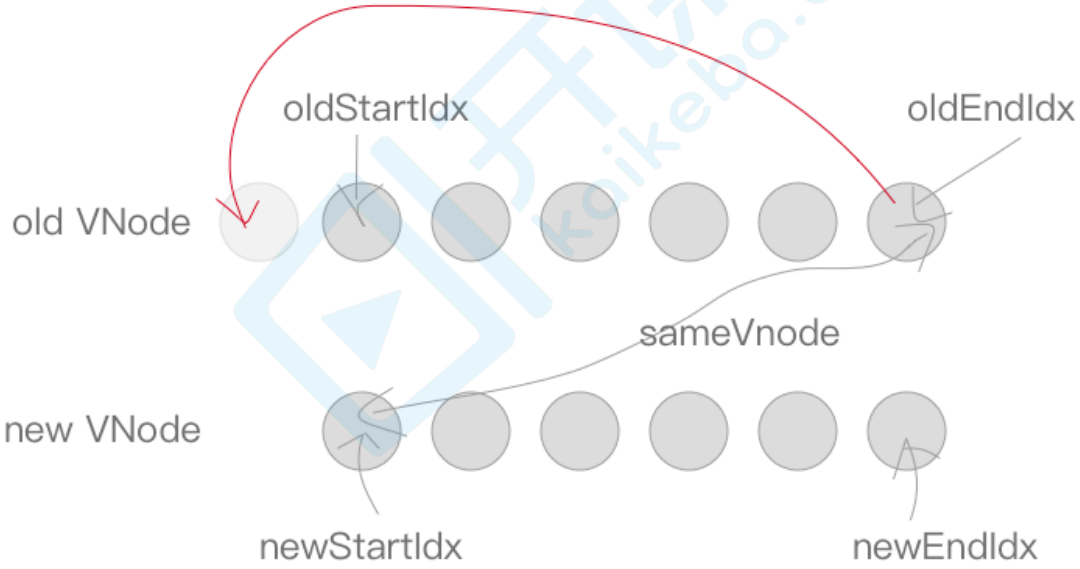
当 oldStartVnode和newStartVnode 或者 oldEndVnode和newEndVnode 满足sameVnode，直接将该VNode节点进行patchVnode即可，不需再遍历就完成了一次循环。如下图，



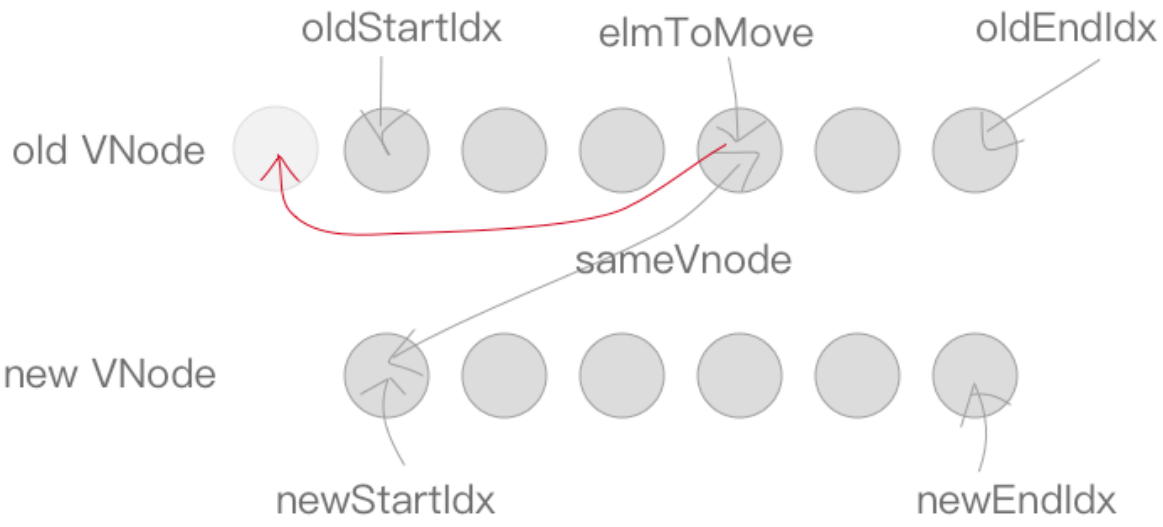
如果oldStartVnode与newEndVnode满足sameVnode。说明oldStartVnode已经跑到了oldEndVnode后面去了，进行patchVnode的同时还需要将真实DOM节点移动到oldEndVnode的后面。



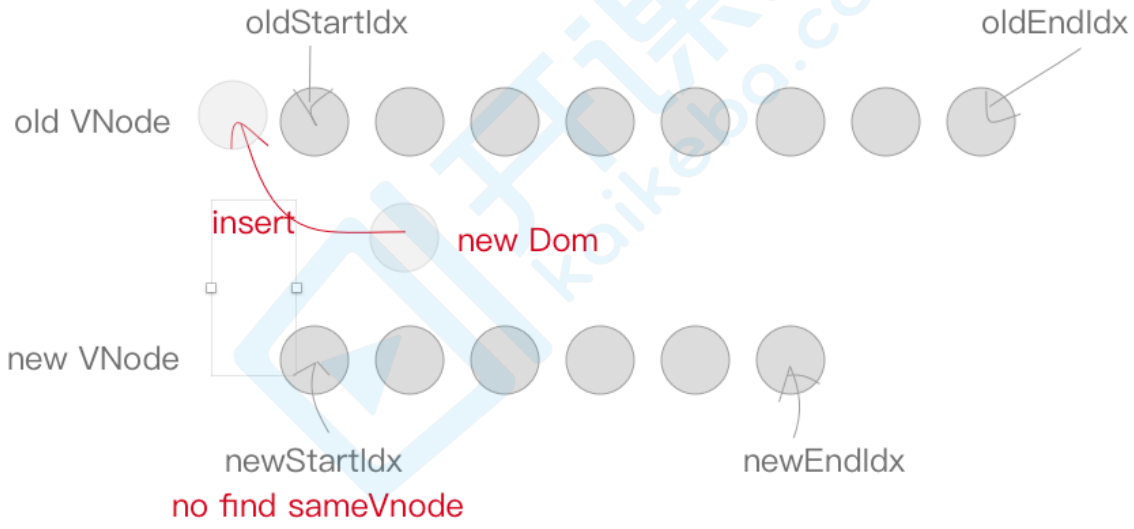
如果oldEndVnode与newStartVnode满足sameVnode，说明oldEndVnode跑到了oldStartVnode的前面，进行patchVnode的同时要将oldEndVnode对应DOM移动到oldStartVnode对应DOM的前面。



如果以上情况均不符合，则在old VNode中找与newStartVnode满足sameVnode的vnodeToMove，若存在执行patchVnode，同时将vnodeToMove对应DOM移动到oldStartVnode对应的DOM的前面。

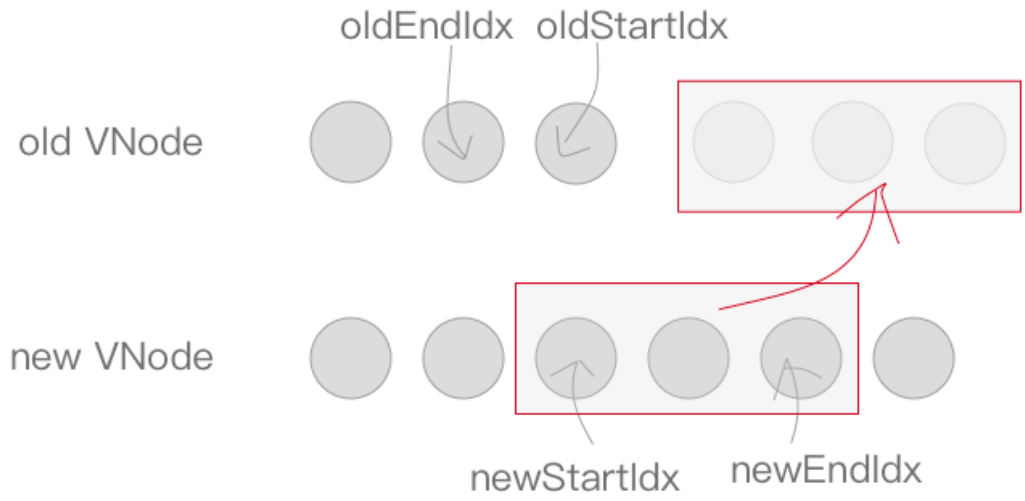


当然也有可能newStartVnode在old VNode节点中找不到一致的key，或者是即便key相同却不是sameVnode，这个时候会调用createElm创建一个新的DOM节点。

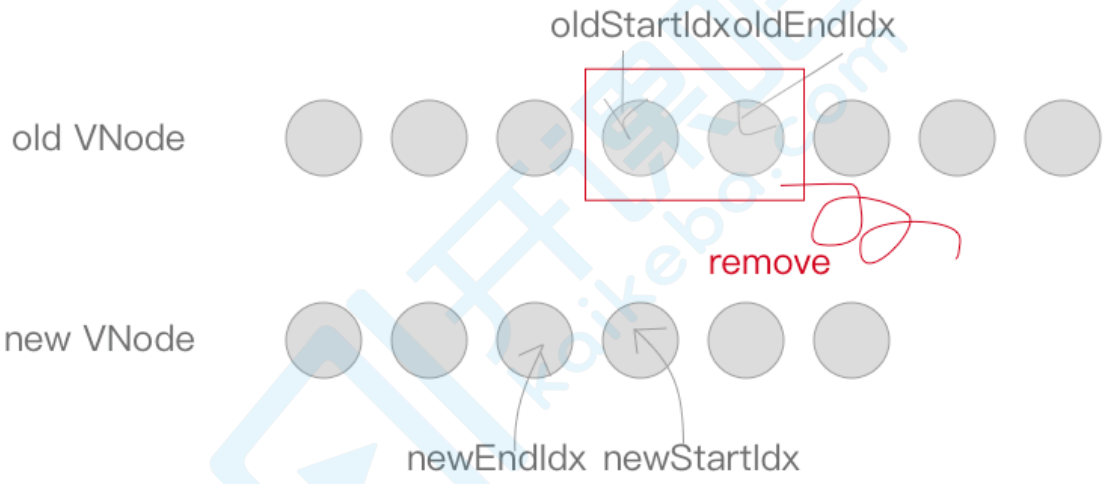


至此循环结束，但是我们还需要处理剩下的节点。

当结束时oldStartIdx > oldEndIdx，这个时候旧的VNode节点已经遍历完了，但是新的节点还没有。说明了新的VNode节点实际上比老的VNode节点多，需要将剩下的VNode对应的DOM插入到真实DOM中，此时调用addVnodes（批量调用createElm接口）。



但是，当结束时newStartIdx > newEndIdx时，说明新的VNode节点已经遍历完了，但是老的节点还有剩余，需要从文档中删 的节点删除。




```
  function updateChildren (parentElm, oldCh, newCh, insertedVnodeQueue,
removeOnly) {
    let oldStartIdx = 0
    let newStartIdx = 0
    let oldEndIdx = oldCh.length - 1
    let oldStartVnode = oldCh[0]
    let oldEndVnode = oldCh[oldEndIdx]
    let newEndIdx = newCh.length - 1
    let newStartVnode = newCh[0]
    let newEndVnode = newCh[newEndIdx]
    let oldKeyToIdx, idxInOld, elmToMove, refElm

    // 确保移除元素在过度动画过程中待在正确的相对位置，仅用于<transition-group>
    const canMove = !removeOnly

    // 循环条件：任意起始索引超过结束索引就结束
```

```
    while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
      if (isUndef(oldStartVnode)) {
        oldStartVnode = oldCh[++oldStartIdx] // Vnode has been moved left
      } else if (isUndef(oldEndVnode)) {
        oldEndVnode = oldCh[--oldEndIdx]
      } else if (sameVnode(oldStartVnode, newStartVnode)) {
        /*分别比较oldCh以及newCh的两头节点4种情况，判定为同一个VNode，则直接patchVnode
即可*/
        patchVnode(oldStartVnode, newStartVnode, insertedVnodeQueue)
        oldStartVnode = oldCh[++oldStartIdx]
        newStartVnode = newCh[++newStartIdx]
      } else if (sameVnode(oldEndVnode, newEndVnode)) {
        patchVnode(oldEndVnode, newEndVnode, insertedVnodeQueue)
        oldEndVnode = oldCh[--oldEndIdx]
        newEndVnode = newCh[--newEndIdx]
      } else if (sameVnode(oldStartVnode, newEndVnode)) { // Vnode moved right
        patchVnode(oldStartVnode, newEndVnode, insertedVnodeQueue)
        canMove && nodeOps.insertBefore(parentElm, oldStartVnode.elm,
nodeOps.nextSibling(oldEndVnode.elm))
        oldStartVnode = oldCh[++oldStartIdx]
        newEndVnode = newCh[--newEndIdx]
      } else if (sameVnode(oldEndVnode, newStartVnode)) { // Vnode moved left
        patchVnode(oldEndVnode, newStartVnode, insertedVnodeQueue)
        canMove && nodeOps.insertBefore(parentElm, oldEndVnode.elm,
oldStartVnode.elm)
        oldEndVnode = oldCh[--oldEndIdx]
        newStartVnode = newCh[++newStartIdx]
      } else {
        /*
          生成一个哈希表，key是旧VNode的key，值是该VNode在旧VNode中索引
        */
        if (isUndef(oldKeyToIdx)) oldKeyToIdx = createKeyToOldIdx(oldCh,
oldStartIdx, oldEndIdx)
        /*如果newStartVnode存在key并且这个key在oldVnode中能找到则返回这个节点的索引*/
        idxInOld = isDef(newStartVnode.key) ? oldKeyToIdx[newStartVnode.key] :
null
        if (isUndef(idxInOld)) {
          /*没有key或者是该key没有在老节点中找到则创建一个新的节点*/
          createElm(newStartVnode, insertedVnodeQueue, parentElm,
oldStartVnode.elm)
          newStartVnode = newCh[++newStartIdx]
        } else {
          /*获取同key的老节点*/
          elmToMove = oldCh[idxInOld]
          if (sameVnode(elmToMove, newStartVnode)) {
            /*如果新VNode与得到的有相同key的节点是同一个VNode则进行patchVnode*/
            patchVnode(elmToMove, newStartVnode, insertedVnodeQueue)
            /*因为已经patchVnode进去了，所以将这个老节点赋值undefined，之后如果还有新节
点与该节点key相同可以检测出来提示已有重复的key*/
```

```
            oldCh[idxInOld] = undefined
            /*当有标识位canMove实可以直接插入oldStartVnode对应的真实DOM节点前面*/
            canMove && nodeOps.insertBefore(parentElm, newStartVnode.elm,
oldStartVnode.elm)
            newStartVnode = newCh[++newStartIdx]
          } else {
            /*当新的VNode与找到的同样key的VNode不是sameVNode的时候（比如说tag不一样或者是有不一样type的input标签），创建一个新的节点*/
            createElm(newStartVnode, insertedVnodeQueue, parentElm,
oldStartVnode.elm)
            newStartVnode = newCh[++newStartIdx]
          }
        }
      }
    }
    if (oldStartIdx > oldEndIdx) {
      /*全部比较完成以后，发现oldStartIdx > oldEndIdx的话，说明老节点已经遍历完了，新节点比老节点多，所以这时候多出来的新节点需要一个一个创建出来加入到真实DOM中*/
      refElm = isUndef(newCh[newEndIdx + 1]) ? null : newCh[newEndIdx + 1].elm
      addVnodes(parentElm, refElm, newCh, newStartIdx, newEndIdx,
insertedVnodeQueue)
    } else if (newStartIdx > newEndIdx) {
      /*如果全部比较完成以后发现newStartIdx > newEndIdx，则说明新节点已经遍历完了，老节点多余新节点，这个时候需要将多余的老节点从真实DOM中移除*/
      removeVnodes(parentElm, oldCh, oldStartIdx, oldEndIdx)
    }
  }
```