Diff算法

`diff`算法，在新旧父节点类型相同，且都有多个子节点的情况下才会发挥作用。

无key

对于没有设置key的diff比较，属于直截了当的进行。依次遍历新旧子节点，进行patch操作，就地重复利用旧节点，不存在节点的移动问题。具体为遍历新旧`children`中长度较短的那一个，这样我们能够做到尽可能多的应用`patch`函数进行更新，然后再对比新旧`children`的长度，如果新的 children 更长，则说明有新的节点需要添加，否则说明有旧的节点需要移除。

有key

React

从头开始进行节点比较初始化`lastIndex = 0`，如果相同的key对应的新旧index不一样则表明需要移动旧DOM的位置，同时记录`lastIndex= Math.max(lastIndex, index)`中的最大值，如果当前匹配的Vnode对应的真实node的索引小于lastIndex，则表示该节点是需要进行移动的节点，同时需要移动的位置为上一个节点的后面，通过记录lastIndex则可以完成vnode的对比和移动操作。

对于在旧的DOM不存在的Vnode，进行mount()添加操作，添加的位置在前一个元素的后面

对于没有被使用的旧的的DOM，直接删除即可。

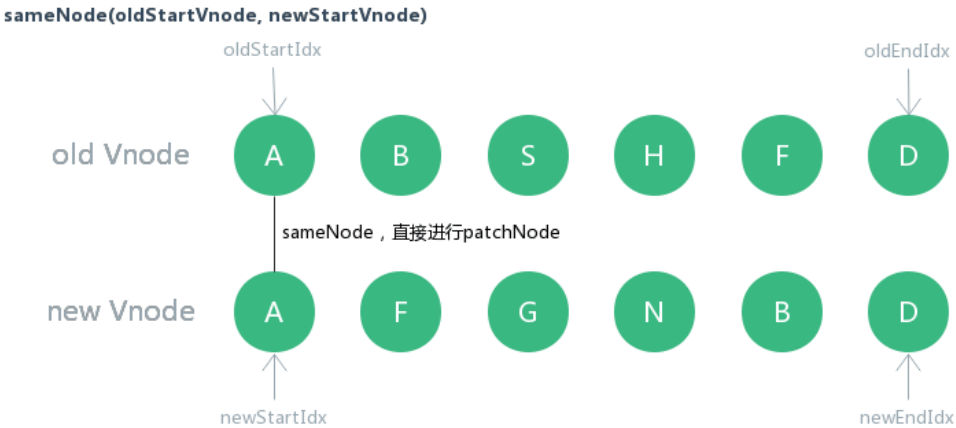
Vue

双指针比较，对新旧vnode数组同时添加首尾的指针。

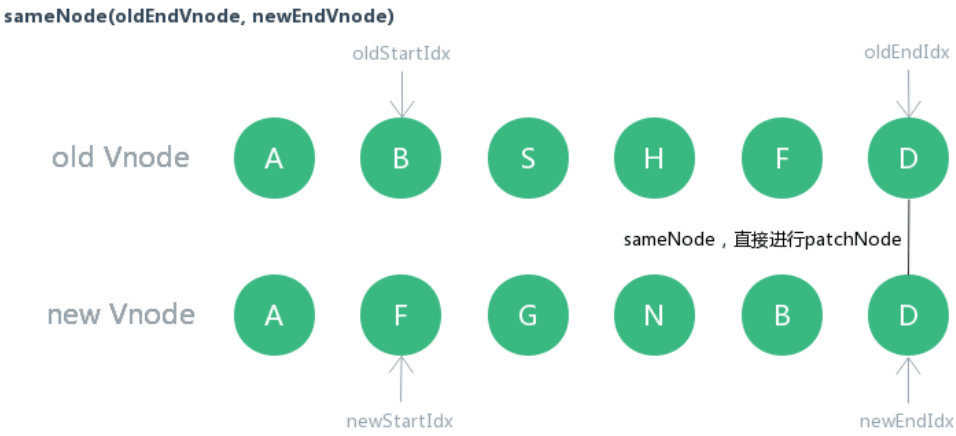
首先给旧vnode数组定义首尾指针`oldStartIdx`和`oldEndIdx`，给新vnode数组定义首尾指针`newStartIdx`和`newEndIdx`。通过每个指针可以取得对应的`vnode`，为`oldStartVnode`，`oldEndVnode`，`newStartVnode`，`newEndVnode`。

通过指针进行如下的对比规则，递归处理vnode和真实的dom node。

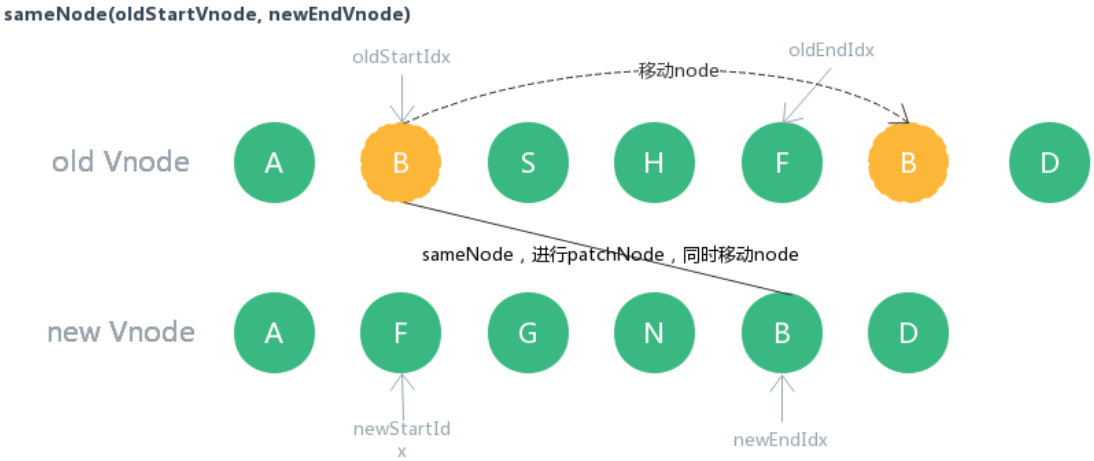
A. 首先将`oldStartVnode.key`与`newStartVnode.key`进行比较，如果相同则进行`patch`操作，同时`oldStartIdx++, newStartIdx++`。否则进入步骤`B`。



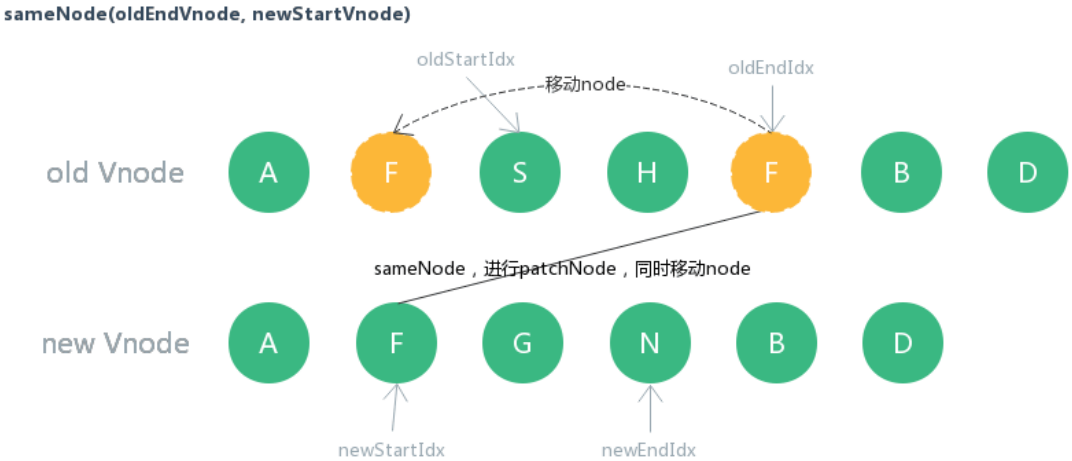
B. 将`oldEndVnode.key`与`newEndVnode.key`进行比较，如果相同则进行`patch`操作，同时`oldEndIdx--, newEndIdx--`。否则进入步骤`C`。



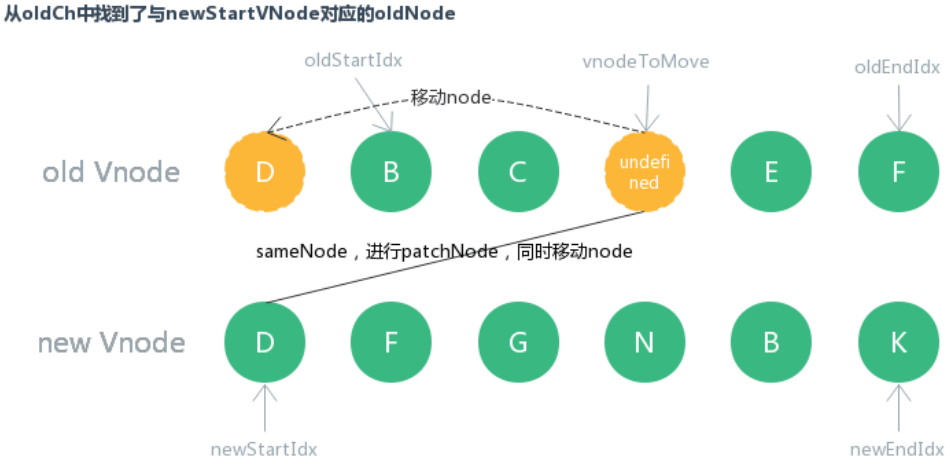
C. 将`oldStartVnode.key`与`newEndVnode.key`进行比较，如果相同则进行`patch`操作`oldStartIdx++, newEndVnode--`，同时将`oldStartVnode`对应的真实node移动到，`oldEndVnode`对应的真实node后面。否则进入步骤`D`。



D. 将`oldEndVnode.key`与`newStartVnode.key`进行比较，如果相同则进行`patch`操作`oldEndIdx--, newStartIdx++`，同时将`oldEndVnode`对应的真实node移动到，`oldStartVnode`对应的真实node前面。否则进入步骤`E`。



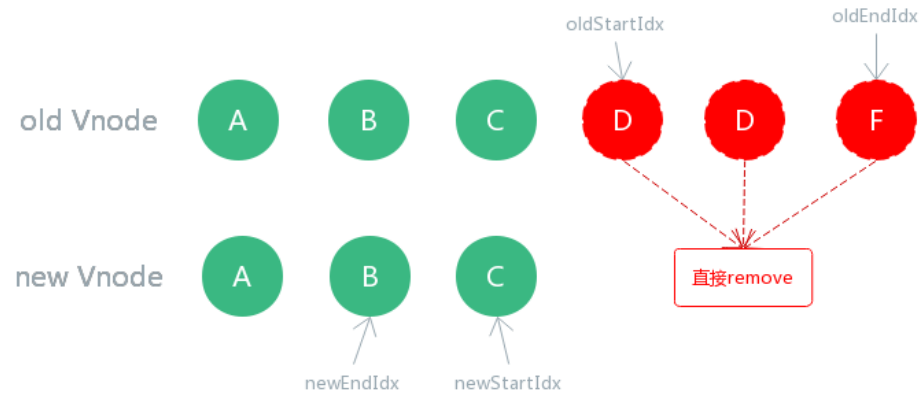
E. 将`newStartVnode.key`与旧vnode中所有的节点进行递归比较，如果找到记录`idxInOld`，同时进行patch操作`newStartIdx++`，同时将`findVnode`对应的真实node移动到`oldStartVnode`对应的真实节点前，最后将`vnodeToMove= undefined`置空，避免后续的重复比较工作。如果没找到，则属于添加节点，在`oldStartVnode`对应的真实节点前添加该node。进入下一次循环。



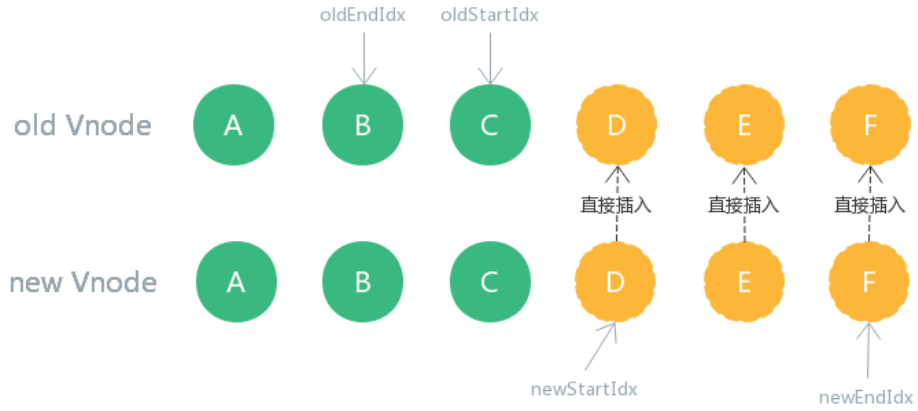
没找到对应的oldVnode

F. 最后如果`oldEndIdx`小于`oldStartIdx`，则表明还有没有处理的新node，遍历`newStartIdx`到`newEndIdx`在`oldStartVNode`对应的真实节点前添加新节点。如果`newEndIdx`小于`newStartIdx`，则表明还有没有被移除的旧node，遍历`oldStartIdx`到`oldEndIdx`对应的真实节点，并将其移除。

newStartIdx > newEndIdx



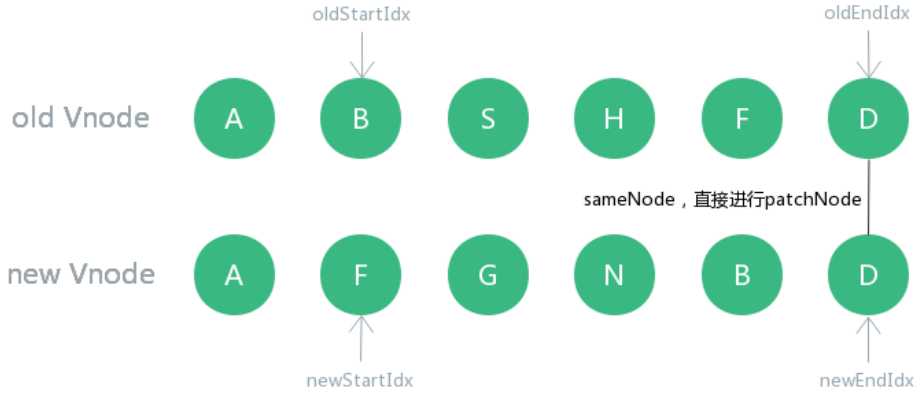
oldStartIdx > oldEndIdx



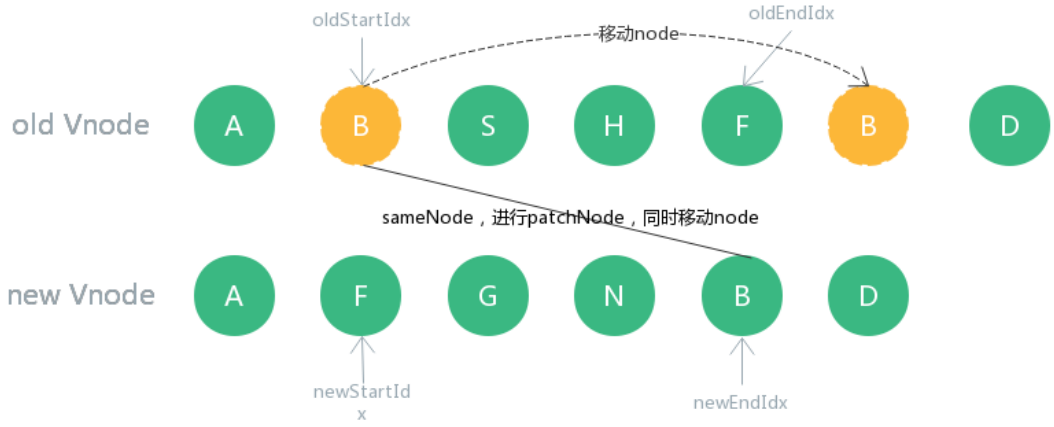
完整示例图：

sameNode(oldStartVnode, newStartVnode)

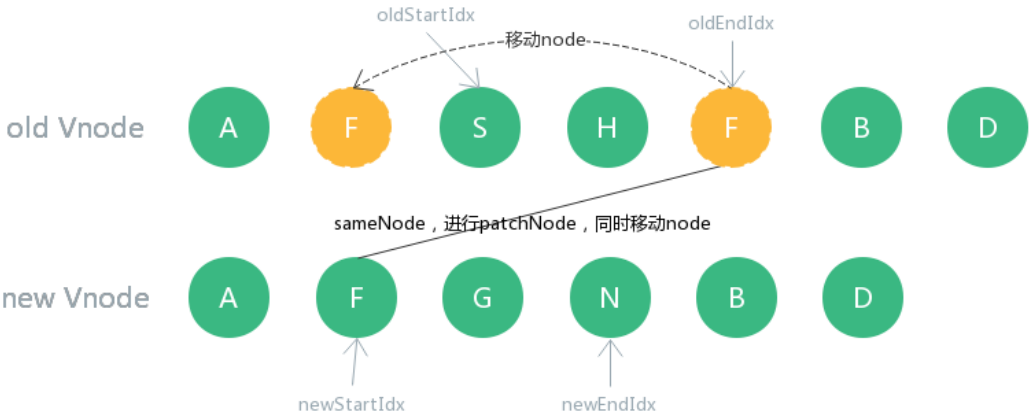
sameNode(oldEndVnode, newEndVnode)



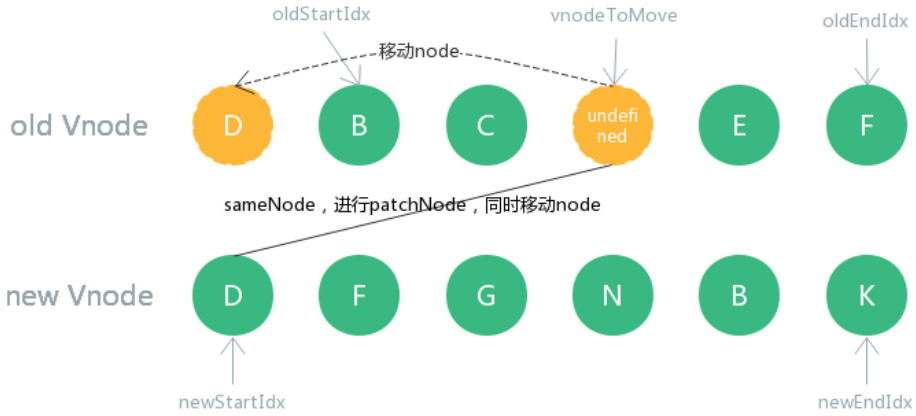
sameNode(oldStartVnode, newEndVnode)



sameNode(oldEndVnode, newStartVnode)

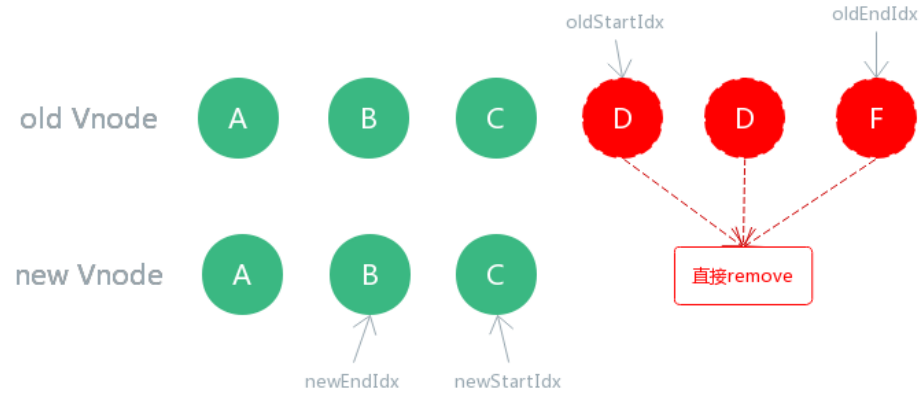


从oldCh中找到了与newStartVNode对应的oldNode

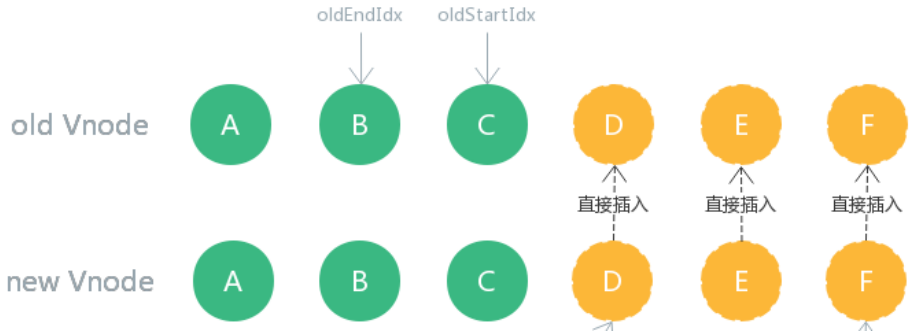


没有找到sameVnode

newStartIdx > newEndIdx



oldStartIdx > oldEndIdx



源码V2.6.9

```
// vue patch 主逻辑源码，文件地址：src/core/vdom/patch.js
function patch(oldVnode, vnode, hydrating, removeOnly) {
   // 省略代码。。。

    // oldVnode为空代表组件首次创建节点的时候
    // 注意：new Vue()操作首次__patch__时会传入vm.$el为容器节点，oldVnode不为空
    if (isUndef(oldVnode)) {
        // empty mount (likely as component), create new root element
        // 首次patch，直接创建新的根节点
        isInitialPatch = true
        createElm(vnode, insertedVnodeQueue)
    } else {
        // 通过是否包含nodeType属性判断当前oldVnode是VNode实例还是已存在的html Element
        const isRealElement = isDef(oldVnode.nodeType)
        if (!isRealElement && sameVnode(oldVnode, vnode)) {
            // patch existing root node
            // 对比当前根节点vnode
            patchVnode(oldVnode, vnode, insertedVnodeQueue, null, null,
removeOnly)
        } else {
            if (isRealElement) {
                // mounting to a real element
                // check if this is server-rendered content and if we can
perform
                // a successful hydration.
                if (oldVnode.nodeType === 1 && oldVnode.hasAttribute(SSR_ATTR))
{
                    oldVnode.removeAttribute(SSR_ATTR)
                    hydrating = true
                }

                oldVnode = emptyNodeAt(oldVnode)
            }

            // replacing existing element
            const oldElm = oldVnode.elm
            const parentElm = nodeOps.parentNode(oldElm)

            // create new node
            // 创建真实DOM
            createElm(
                vnode,
                insertedVnodeQueue,
                // extremely rare edge case: do not insert if old element is in
a
                // leaving transition. Only happens when combining transition +
                // keep-alive + HOCs. (#4590)
                oldElm._leaveCb ? null : parentElm,
                nodeOps.nextSibling(oldElm)
            )
```

```
            // update parent placeholder node element, recursively
            if (isDef(vnode.parent)) {
                let ancestor = vnode.parent
                const patchable = isPatchable(vnode)
                while (ancestor) {
                    for (let i = 0; i < cbs.destroy.length; ++i) {
                        cbs.destroy[i](ancestor)
                    }
                    ancestor.elm = vnode.elm
                    if (patchable) {
                        for (let i = 0; i < cbs.create.length; ++i) {
                            cbs.create[i](emptyNode, ancestor)
                        }
                        // #6513
                        // invoke insert hooks that may have been merged by
create hooks.
                        // e.g. for directives that uses the "inserted" hook.
                        const insert = ancestor.data.hook.insert
                        if (insert.merged) {
                            // start at index 1 to avoid re-invoking component
mounted hook
                            for (let i = 1; i < insert.fns.length; i++) {
                                insert.fns[i]()
                            }
                        }
                    } else {
                        registerRef(ancestor)
                    }
                    ancestor = ancestor.parent
                }
            }

            // destroy old node
            if (isDef(parentElm)) {
                removeVnodes(parentElm, [oldVnode], 0, 0)
            } else if (isDef(oldVnode.tag)) {
                invokeDestroyHook(oldVnode)
            }
        }
    }

    // 执行所有组件的插入时设定的钩子函数
    invokeInsertHook(vnode, insertedVnodeQueue, isInitialPatch)
    return vnode.elm
}
```

patchVnode

```
function patchVnode(
        oldVnode,
        vnode,
        insertedVnodeQueue,
        ownerArray,
        index,
        removeOnly
    ) {
```

```
    if (oldVnode === vnode) {
        return
    }

    const elm = vnode.elm = oldVnode.elm

    // 省略部分逻辑

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

    if (isUndef(vnode.text)) {
        if (isDef(oldCh) && isDef(ch)) {
            // oldCh 与 ch 都存在且不相同时，使用 updateChildren 函数来更新子节点
            if (oldCh !== ch) updateChildren(elm, oldCh, ch, insertedVnodeQueue,
removeOnly)
        } else if (isDef(ch)) {
            // 如果只有 ch 存在，表示添加新节点
            if (process.env.NODE_ENV !== 'production') {
                checkDuplicateKeys(ch)
            }
            if (isDef(oldVnode.text)) nodeOps.setTextContent(elm, '')
            addVnodes(elm, null, ch, 0, ch.length - 1, insertedVnodeQueue)
        } else if (isDef(oldCh)) {
            // 如果只有 oldCh 存在，表示更新的是空节点
            removeVnodes(elm, oldCh, 0, oldCh.length - 1)
        } else if (isDef(oldVnode.text)) {
            // 当只有旧节点是文本节点的时候，则清除其节点文本内容
            nodeOps.setTextContent(elm, '')
        }
    } else if (oldVnode.text !== vnode.text) {
        // 如果 vnode 是个文本节点且新旧文本不相同，则直接替换文本内容
        nodeOps.setTextContent(elm, vnode.text)
    }
    if (isDef(data)) {
        // postpatch钩子触发
        if (isDef(i = data.hook) && isDef(i = i.postpatch)) i(oldVnode, vnode)
    }
}
```

updateChildren

```
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, vnodeToMove, refElm

    // removeOnly is a special flag used only by <transition-group>
    // to ensure removed elements stay in correct relative positions
    // during leaving transitions
```

```
    // 用于标识oldChild是否可以被移除，在进行patch操作时，transiton-group是不能被移除的
    const canMove = !removeOnly
    // key重复检查
    if (process.env.NODE_ENV !== 'production') {
        checkDuplicateKeys(newCh)
    }

    while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
        // 后面查找匹配的过程中，匹配到会将对应index的oldVnode设为undefined
        if (isUndef(oldStartVnode)) {
            oldStartVnode = oldCh[++oldStartIdx] // Vnode has been moved left
        } else if (isUndef(oldEndVnode)) {
            oldEndVnode = oldCh[--oldEndIdx]
        } else if (sameVnode(oldStartVnode, newStartVnode)) {
            // 相同的vnode，patchVnode
            patchVnode(oldStartVnode, newStartVnode, insertedVnodeQueue, newCh,
newStartIdx)
            oldStartVnode = oldCh[++oldStartIdx]
            newStartVnode = newCh[++newStartIdx]
        } else if (sameVnode(oldEndVnode, newEndVnode)) {
            patchVnode(oldEndVnode, newEndVnode, insertedVnodeQueue, newCh,
newEndIdx)
            oldEndVnode = oldCh[--oldEndIdx]
            newEndVnode = newCh[--newEndIdx]
        } else if (sameVnode(oldStartVnode, newEndVnode)) { // Vnode moved right
            patchVnode(oldStartVnode, newEndVnode, insertedVnodeQueue, newCh,
newEndIdx)
            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, newCh,
newStartIdx)
            canMove && nodeOps.insertBefore(parentElm, oldEndVnode.elm,
oldStartVnode.elm)
            oldEndVnode = oldCh[--oldEndIdx]
            newStartVnode = newCh[++newStartIdx]
        } else {
            if (isUndef(oldKeyToIdx)) oldKeyToIdx = createKeyToOldIdx(oldCh,
oldStartIdx, oldEndIdx)
            // newStartVnode在oldchild中的index，
            // 如果key存在则通过key查找，如果key不存在则遍历所有的oldChild查找与
newStartVnode匹配的sameVnode
            idxInOld = isDef(newStartVnode.key) ?
                oldKeyToIdx[newStartVnode.key] :
                findIdxInOld(newStartVnode, oldCh, oldStartIdx, oldEndIdx)

            // 如果未找到，则为新添加的vnode元素
            if (isUndef(idxInOld)) { // New element
                createElm(newStartVnode, insertedVnodeQueue, parentElm,
oldStartVnode.elm, false, newCh, newStartIdx)
            } else {
                vnodeToMove = oldCh[idxInOld]
                // 如果找到newStartVnode，且相同，则移动idxInOld处的vnode
                if (sameVnode(vnodeToMove, newStartVnode)) {
                    patchVnode(vnodeToMove, newStartVnode, insertedVnodeQueue,
newCh, newStartIdx)
```

```
                    oldCh[idxInOld] = undefined
                    canMove && nodeOps.insertBefore(parentElm, vnodeToMove.elm,
oldStartVnode.elm)
                } else {
                    // same key but different element. treat as new element
                    createElm(newStartVnode, insertedVnodeQueue, parentElm,
oldStartVnode.elm, false, newCh, newStartIdx)
                }
            }
            newStartVnode = newCh[++newStartIdx]
        }
    }

    // 对于未处理的child节点进行处理
    if (oldStartIdx > oldEndIdx) {
        // oldChild全部处理完成，但newChild可能存在遗漏
        refElm = isUndef(newCh[newEndIdx + 1]) ? null : newCh[newEndIdx + 1].elm
        addVnodes(parentElm, refElm, newCh, newStartIdx, newEndIdx,
insertedVnodeQueue)
    } else if (newStartIdx > newEndIdx) {
        // newChild全部处理完成，oldChild可能存在未被匹配项还未移除
        removeVnodes(parentElm, oldCh, oldStartIdx, oldEndIdx)
    }
}
```

`sameVnode()`

```
function sameVnode(a, b) {
    return (
        a.key === b.key && (
            (
                a.tag === b.tag &&
                a.isComment === b.isComment &&
                isDef(a.data) === isDef(b.data) &&
                sameInputType(a, b)
            ) || (
                isTrue(a.isAsyncPlaceholder) &&
                a.asyncFactory === b.asyncFactory &&
                isUndef(b.asyncFactory.error)
            )
        )
    )
}

/**
 * node为input时type相同或者都为文本输入框(text,password,number等)
 */
function sameInputType(a, b) {
    if (a.tag !== 'input') return true
    let i
    const typeA = isDef(i = a.data) && isDef(i = i.attrs) && i.type
    const typeB = isDef(i = b.data) && isDef(i = i.attrs) && i.type
    return typeA === typeB || isTextInputType(typeA) && isTextInputType(typeB)
}

export const isTextInputType =
makeMap('text,number,password,search,email,tel,url');
```

```
/**
 * 通过字符串创建一个map
 * 同时返回一个函数用于判断给定的key是否在map内
 */
function makeMap(str, expectLowerCase) {
    let map = Object.create(null);
    let list = str.split(',');
    for (let i = 0, l = list.length; i < l; i++) {
        map[list[i]] = true;
    }

    return expectLowerCase ?
        val => map[val.toLowerCase()] :
        val => map[val];
}
```