vnode的出现

有以下几个原因:

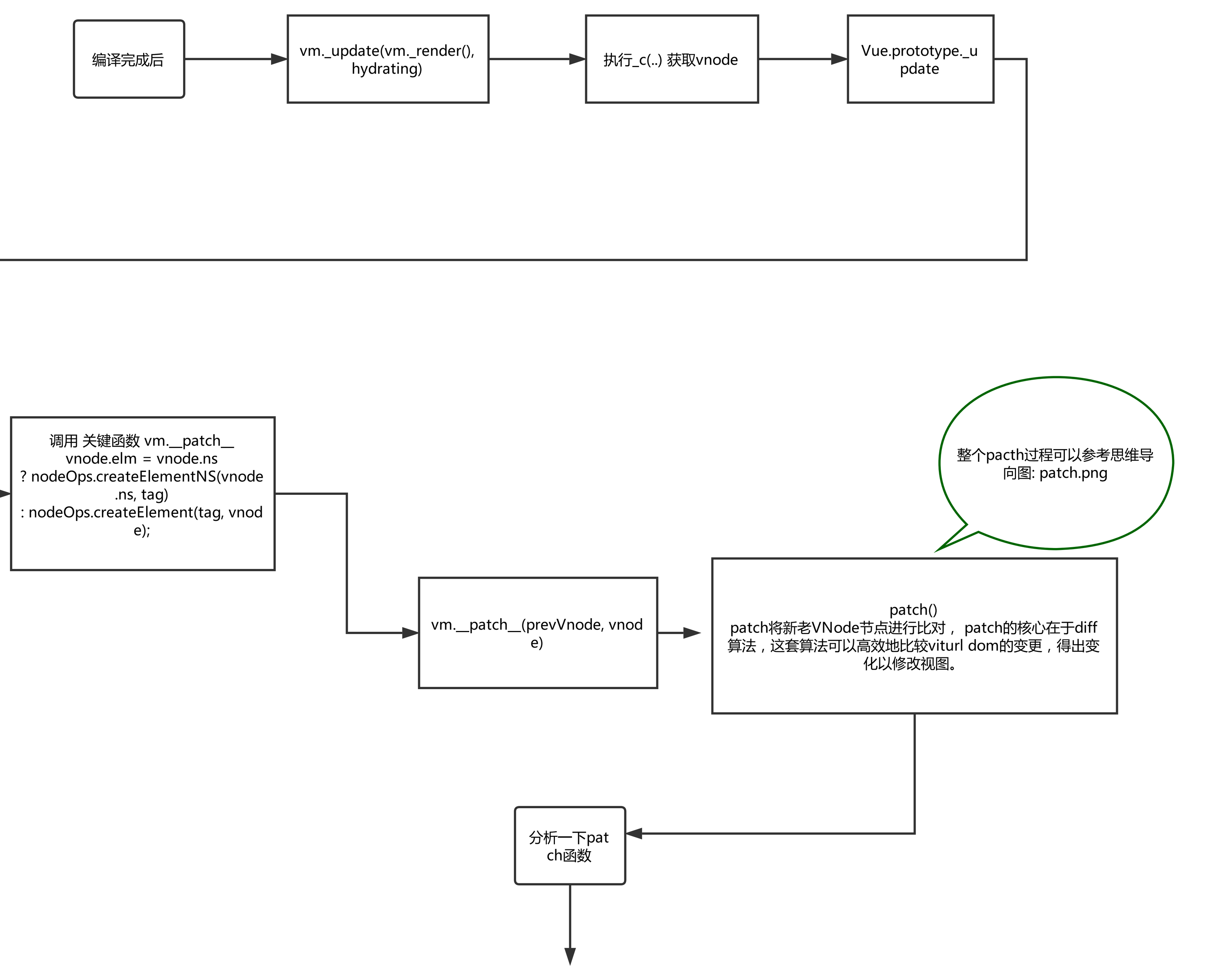
第一. 提高性能, 一个真实的dom的属性往往非常多, 消耗的内存也大,
所以创建简单的js对象去替代复杂的dom对象,

另外不当dom操作, 会引起频繁大量的页面重排, 会造成巨大的性能浪费，
这时候需要vdom 的patch 来 寻找本次更新dom的较优的操作来更新页面.
virtual dom很多时候都不是最优的操作，但具有普适性。

第二. 解决了 复杂程序状态更新后 对应的视图更新问题,
而且当页面结构很庞大，结构很复杂时，手工更新dom 和 dom操作优化会花去大量时间，
而且可维护性也不高，也不能保证每个人都有手工优化的能力,
virtual dom的解决方案应运而生
它具有普适性，在效率、可维护性之间达平衡。

第三. 可以适配不同的平台，提供一个中间层，js去写ui，再转换成 ios安卓的底层API调用，就像reactNative一样

第四. 可以利用vnode 映射真实的dom结构树, 利用浏览器冒泡原理来实现浏览器事件, 屏蔽浏览器的事件差异,
如: react使用对象池来管理合成事件对象的创建和销毁，
这样减少了垃圾的生成和新对象内存的分配，大大提高了性能



判断两个VNode节点是否是同一个节点，
需要满足以下条件

key相同
tag（当前节点的标签名）相同
isComment（是否为注释节点）相同
是否data（当前节点对应的对象，包含了具体的一些数据信息，是一个VNodeData类型，可以参考VNodeData类型中的数据信息）都有定义
当标签是<input>的时候，type必须相同

```
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)
      )
    )
  )
}
```

第一: 删除操作, 新的 vnode如果没有, 说明是,oldVnode是多余的, 直接调用 销毁钩子

第二: 新增操作, 如果旧的vnode未定义, 可能是创建了一个新的组件, 调用document.createElement(vnode.tag)

第三: 如果 新旧 vnode的类型一致，则进入patchVnode方法

```
return function patch (oldVnode, vnode, hydrating, removeOnly) {
  if (isUndef(vnode)) {
    if (isDef(oldVnode)) { invokeDestroyHook(oldVnode); }
    return
  }

  var isInitialPatch = false;
  var insertedVnodeQueue = [];

  if (isUndef(oldVnode)) {
    // empty mount (likely as component), create new root element
    isInitialPatch = true;
    createElm(vnode, insertedVnodeQueue);
  } else {
    var isRealElement = isDef(oldVnode.nodeType);
    if (!isRealElement && sameVnode(oldVnode, vnode)) {
      // patch existing root node
      patchVnode(oldVnode, vnode, insertedVnodeQueue, 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;
        }
        if (isTrue(hydrating)) {…
        // either not server-rendered, or hydration failed.
        // create an empty node and replace it
        oldVnode = emptyNodeAt(oldVnode);
      }

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

      // create new node
      createElm(
```

第四: 替换操作, 如果新旧节点不一致，
删除旧节点 emptyNodeAt(oldVnode),
新增新节点;

比如 <span> 和 <div> 类型不一样, 是没必要进入diff的

parent?

```
  // update parent placeholder node element, recursively
  if (isDef(vnode.parent)) {
    var ancestor = vnode.parent;
    var patchable = isPatchable(vnode);
    while (ancestor) {
      for (var i = 0; i < cbs.destroy.length; ++i) {
        cbs.destroy[i](ancestor);
      }
      ancestor.elm = vnode.elm;
      if (patchable) {
        for (var i$1 = 0; i$1 < cbs.create.length; ++i$1) {
          cbs.create[i$1](emptyNode, ancestor);
        }
        // #6513
        // invoke insert hooks that may have been merged by create hooks
        // e.g. for directives that uses the "inserted" hook.
        var insert = ancestor.data.hook.insert;
        if (insert.merged) {
          // start at index 1 to avoid re-invoking component mounted hoo
          for (var i$2 = 1; i$2 < insert.fns.length; i$2++) {
            insert.fns[i$2]();
          }
        }
      } else {
        registerRef(ancestor);
      }
      ancestor = ancestor.parent;
    }
  }

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

进入patchVnode

1. 新旧节点完全一致, 引用地址都相同, 不用操作

2. 静态根节点, 而且是复制节点或者isonce属性, 直接返回旧节点的实例

3.更新标签的 属性 class 事件 指令等, 具体操作的函数如下图 cbs.update

4. 如果新旧节点都有孩子列表, 进入diff算法, 更新dom, dom增删改操作
(新节点没有文本情况下)

5. 如果只有新节点孩子, 清空oldvnode的文本, 换成newvnode的孩子
(新节点没有文本情况下)

6.只有旧节点孩子, 直接删除
(新节点没有文本情况下)

7.如果旧节点有文本, 删除
(新节点没有文本情况下)

8. 如果新节点有文本, 更新文本

```
> cbs.update
< ▼(7) [ƒ, ƒ, ƒ, ƒ, ƒ, ƒ, ƒ]
  ▶0: ƒ updateAttrs(oldVnode, vnode)
  ▶1: ƒ updateClass(oldVnode, vnode)
  ▶2: ƒ updateDOMListeners(oldVnode, vnode)
  ▶3: ƒ updateDOMProps(oldVnode, vnode)
  ▶4: ƒ updateStyle(oldVnode, vnode)
  ▶5: ƒ update(oldVnode, vnode)
  ▶6: ƒ updateDirectives(oldVnode, vnode)
   length: 7
  ▶__proto__: Array(0)
>
```

```
function patchVnode (oldVnode, vnode, insertedVnodeQueue, removeOnly) {
  if (oldVnode === vnode) { //1
    return
  }
  var elm = vnode.elm = oldVnode.elm;
  if (isTrue(vnode.isStatic) && isTrue(oldVnode.isStatic) &&
    vnode.key === oldVnode.key &&(isTrue(vnode.isCloned) || isTrue(vnode.isOnce))
  ) { //2
    vnode.componentInstance = oldVnode.componentInstance;
    return
  }

  var i; var data = vnode.data;
  if (isDef(data) && isDef(i = data.hook) && isDef(i = i.prepatch)) {
    i(oldVnode, vnode);
  }

  var oldCh = oldVnode.children; var ch = vnode.children;
  if (isDef(data) && isPatchable(vnode)) { //3
    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); }
  }
  if (isUndef(vnode.text)) {
    if (isDef(oldCh) && isDef(ch)) {//4
      if (oldCh !== ch) {
        updateChildren(elm, oldCh, ch, insertedVnodeQueue, removeOnly); }
    } else if (isDef(ch)) {//5
      if (isDef(oldVnode.text)) { nodeOps.setTextContent(elm, ''); }
      addVnodes(elm, null, ch, 0, ch.length - 1, insertedVnodeQueue);
    } else if (isDef(oldCh)) {//6
      removeVnodes(elm, oldCh, 0, oldCh.length - 1);
    } else if (isDef(oldVnode.text)) {//7
      nodeOps.setTextContent(elm, '');
    }
  } else if (oldVnode.text !== vnode.text) {
    nodeOps.setTextContent(elm, vnode.text);
  }
```

进入diff算法, 也就是调用updateChildren方法

diff这里比较复杂, 文字说明不直观, 可以结合diff.pdf 来看,

这里先稍微解释下, 来看第一次循环, 可能发生的情况

首先 在 000处,: 定义了 两对指针, 和两对vnode, 分别指向新旧类别的两端

后面如果处理过的节点, 指针就可以往中间移动

111处: 如果 newch的 key和旧的vnode 的key重复, 发一个警告

0.1处: 如果 旧节点列表的第一个孩子是空或者未定义,

指针往右边移动一位, 相应的 oldStartVnode 也就是第二个节点了

0.2 处: 同样如果旧节点列表的最后一个孩子是空或者未定义,

指针左边移动一位, 向中间靠拢, 相应的

oldEndVnode也就是倒数第二个节点

1.1 1.2 处: 开始 或者 结束节点类型一致, 直接移动指针, 不用操作dom;

可以复用, 那么不需要对dom操作, 愉快的移动指针

新旧节点都往前迈一步;

因为新旧节点可能还有孩子, 所以继续调用patchVnode对各自孩子继续diff dom操作

123处: 旧节点的第一个 和 新节点的最后一个类型相同, 可以复用,

只需要在真实dom中 (oldSatrtVnode.elm对应的就是真实dom)

把elm(start) 移动到 elm(end) 之后

旧的第一个指针往→移动，新的最后一个指针往左移动一位

321处: 类似操作

```
function updateChildren (parentElm, oldCh, newCh, insertedVnodeQueue, removeOnly) {
    var oldStartIdx = 0; var oldEndIdx = oldCh.length - 1;  // 000
    var oldStartVnode = oldCh[0];
    var oldEndVnode = oldCh[oldEndIdx];

    var newStartIdx = 0;var newEndIdx = newCh.length - 1;
    var newStartVnode = newCh[0];
    var newEndVnode = newCh[newEndIdx];
    var oldKeyToIdx, idxInOld, vnodeToMove, refElm;

    // removeOnly is a special flag used only by <transition-group>
    // to ensure removed elements stay in correct relative positions
    var canMove = !removeOnly;
    { // during leaving transitions
      checkDuplicateKeys(newCh);  // 111
    }

    while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
      if (isUndef(oldStartVnode)) { // 0.1
        oldStartVnode = oldCh[++oldStartIdx]; // Vnode has been moved left
      } else if (isUndef(oldEndVnode)) {   oldEndVnode = oldCh[--oldEndIdx];
      } else if (sameVnode(oldStartVnode, newStartVnode)) { //1.1
        patchVnode(oldStartVnode, newStartVnode, insertedVnodeQueue);
        oldStartVnode = oldCh[++oldStartIdx];
        newStartVnode = newCh[++newStartIdx];
      } else if (sameVnode(oldEndVnode, newEndVnode)) { //1.2
        patchVnode(oldEndVnode, newEndVnode, insertedVnodeQueue);
        oldEndVnode = oldCh[--oldEndIdx];
        newEndVnode = newCh[--newEndIdx];
      } else if (sameVnode(oldStartVnode, newEndVnode)) { // Vnode moved right // 123
        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  //321
        patchVnode(oldEndVnode, newStartVnode, insertedVnodeQueue);
        canMove && nodeOps.insertBefore(parentElm, oldEndVnode.elm, oldStartVnode.elm);
        oldEndVnode = oldCh[--oldEndIdx];
        newStartVnode = newCh[++newStartIdx];
      } else { //4
```

```
function createKeyToOldIdx (children, beginIdx, endIdx) {
  var i, key;
  var map = {};
  for (i = beginIdx; i <= endIdx; ++i) {
    key = children[i].key;
    if (isDef(key)) { map[key] = i; }
  }
  return map
}
```

4处: 经过上面首首 尾尾 首尾 尾首 都比较过后, 发现没有匹配到,

那就根据key来找节点了, 看old 节点列表的中间部分有没有符合的,

如果没有oldkey表, 则 createKeyToOldIdx(oldCh, oldStartIdx, oldEndIdx)

具体的方法如上图, 会生成

{
id0: 0,
id1: 1,
id2: 2
}

id0 id1 id2 是一开始就存在旧节点中的key值, 这里只不过是组装了一下

如果新节点也有 id1, 那下面就可以直接复用 old节点这个idx : 即oldKeyToIdx[newStqrtVnode Key]

如果没有key, 只能调用findIdxInOld 遍历旧节点 数组 去一个一个对比了,
具体函数定义如右下图

5: 如果实在复用不了, 创建一个新节点给它, 详情待续....

6. 如果可以复用, 拿到vnodeToMove这个复用节点,
虽然它们key一样, 但为保险起见, 还是判断下是否是相同类型节点
如果是, 则 标注这个旧节点为 undefined, 说明已经利用过,

7. 如果不是 只能参加新节点

8: 经过 5 6 7操作之后, 新节点已经更新到dom,
新vnode往右移动一位即可, 说明已经处理过

9:

10:

```
        newStartVnode = newCh[++newStartIdx];
      } else { //4
        if (isUndef(oldKeyToIdx)) { oldKeyToIdx = createKeyToOldIdx(oldCh, oldStartIdx, oldEndIdx); }
        idxInOld = isDef(newStartVnode.key)
          ? oldKeyToIdx[newStartVnode.key]
          : findIdxInOld(newStartVnode, oldCh, oldStartIdx, oldEndIdx);
        if (isUndef(idxInOld)) { // New element            //5
          createElm(newStartVnode, insertedVnodeQueue, parentElm,
            oldStartVnode.elm, false, newCh, newStartIdx);
        } else {
          vnodeToMove = oldCh[idxInOld];
          if (sameVnode(vnodeToMove, newStartVnode)) { //6
            patchVnode(vnodeToMove, newStartVnode, insertedVnodeQueue);
            oldCh[idxInOld] = undefined;
            canMove && nodeOps.insertBefore(parentElm, vnodeToMove.elm, oldStartVnode.elm);
          } else {  //7
            // same key but different element. treat as new element
            createElm(newStartVnode, insertedVnodeQueue, parentElm, oldStartVnode.elm, false, newCh,
            newStartIdx);
          }
        }  //8
        newStartVnode = newCh[++newStartIdx];
      }
    }
    if (oldStartIdx > oldEndIdx) { //9
      refElm = isUndef(newCh[newEndIdx + 1]) ? null : newCh[newEndIdx + 1].elm;
      addVnodes(parentElm, refElm, newCh, newStartIdx, newEndIdx, insertedVnodeQueue);
    } else if (newStartIdx > newEndIdx) { // 10
      removeVnodes(parentElm, oldCh, oldStartIdx, oldEndIdx);
    }
  }
```

```
function findIdxInOld (node, oldCh, start, end) {
  for (var i = start; i < end; i++) {
    var c = oldCh[i];
    if (isDef(c) && sameVnode(node, c)) { return i }
  }
}
```