

小明无话在

为了实现高效的DOM操作，一套高效的虚拟DOM diff算法显得很有必要。

Vue的diff算法是基于sabbdom改造过来的

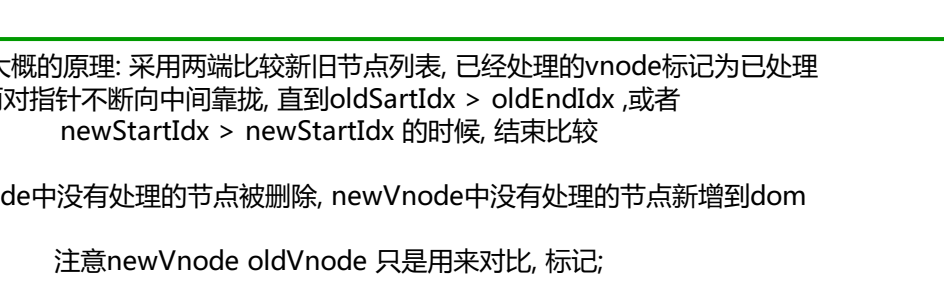
这是一张很经典的图，出自《React's diff algorithm》，Vue的diff算法也同样，  
即仅在同级的vnode间做diff，递归地进行同级vnode的diff，最终实现整个DOM树的更新。

因为跨层级的操作是非常少的，忽略不计，这样时间复杂度就从  $O(n^3)$  变成  $O(n)$   
那同级vnode diff的细节又是怎样的呢？正是本文所要讲的。

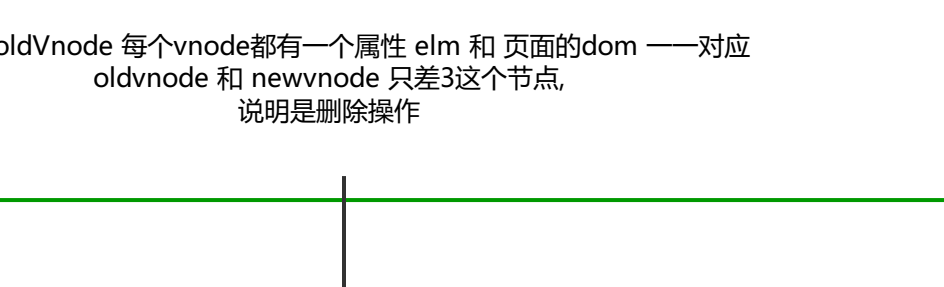
vue的diff算法是两端比较的  
现从一个简单的小例子开始

代码转换成图片 精简如下

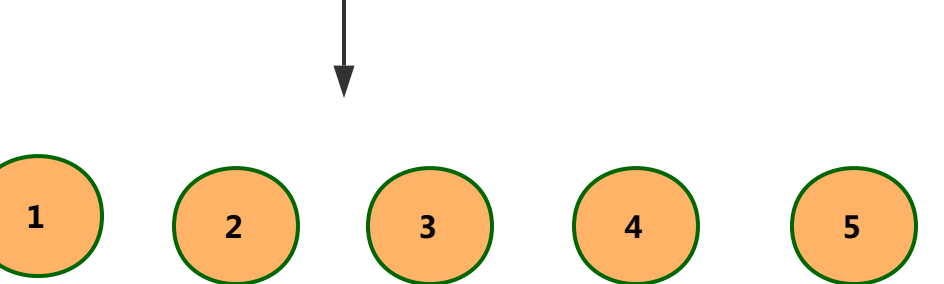
页面的DOM:



oldVnode:



newVnode:



解释下diff的大概的原理: 采用两端比较新旧节点列表, 已经处理的vnode标记为已处理  
两对指针不断向中间靠拢, 直到oldStartIdx > oldEndIdx, 或者  
newStartIdx > newStartIdx 的时候, 结束比较

最后 oldVnode中没有处理的节点被删除, newVnode中没有处理的节点新增到dom

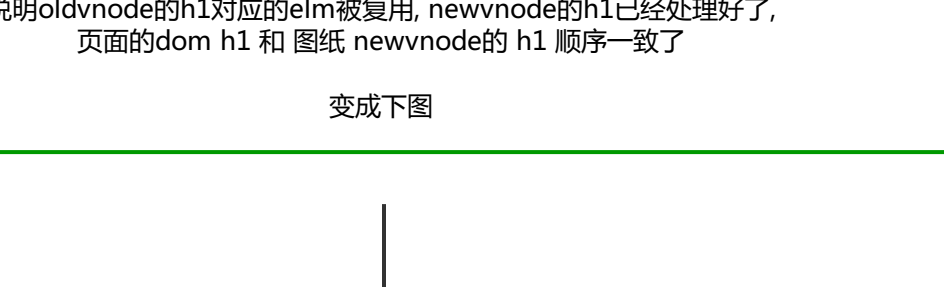
注意newVnode oldVnode 只是用来对比, 标记;

它们本身是不会移动或者被删除的, 可以理解为蓝图, 页面的dom是已经存在的按钮。

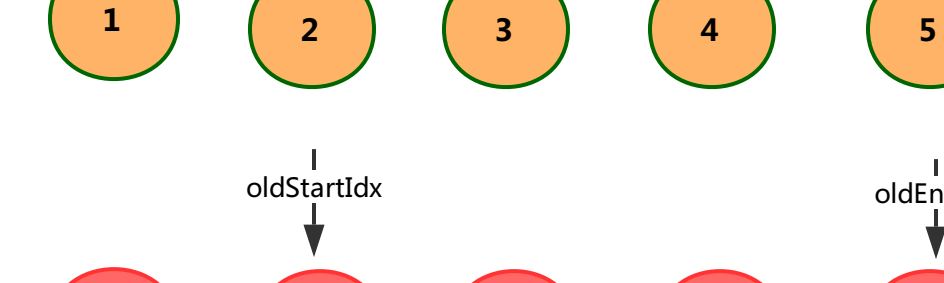
也就是说它们的顺序diff前后保持不变的  
移动, 更新操作的是oldVnode对应的elm, 也就是已经挂载页面中的真实DOM节点

一开始 oldVnode 每个vnode都有一个属性 elm 和 页面的dom 一一对应  
oldvnode 和 newvnode 只差3这个节点,  
说明是删除操作

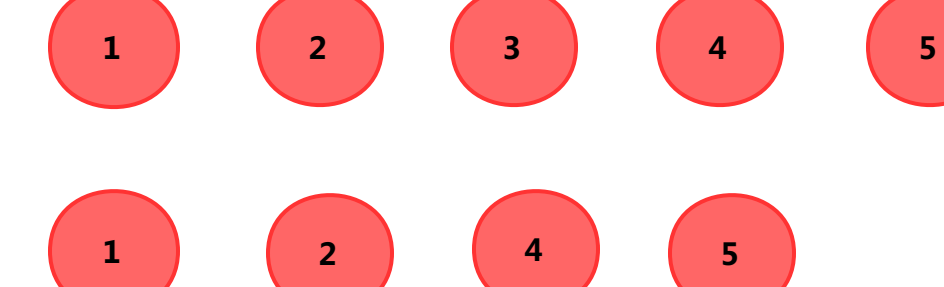
页面的DOM:



oldVnode:



newVnode:

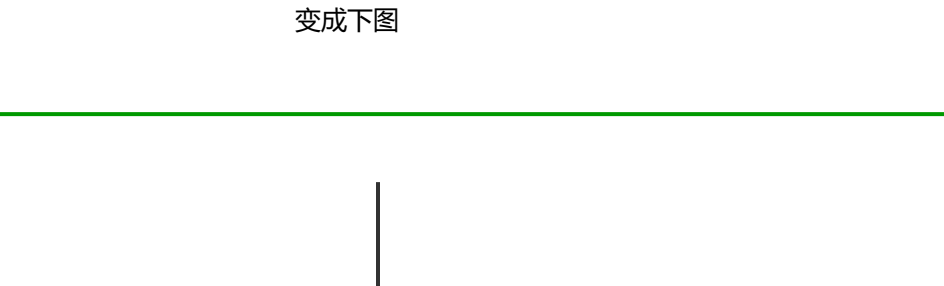


开始定义了两对 指针 分别指向新旧节点 首尾  
进入第一次对比, 发现 新旧列表的第一个节点相同, 都是 h1  
则, 不需要操作dom, oldStartIdx++; newStartIdx++; 都向右移一位

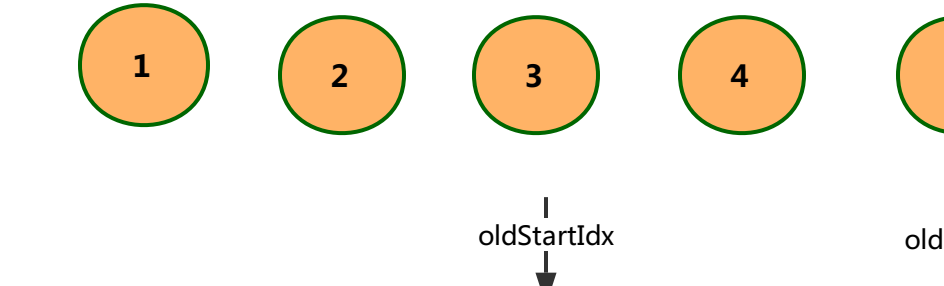
说明oldvnode的h1对应的elm被复用, newvnode的h1已经处理好了,  
页面的dom h1 和 图纸 newvnode的 h1 顺序一致了

变成下图

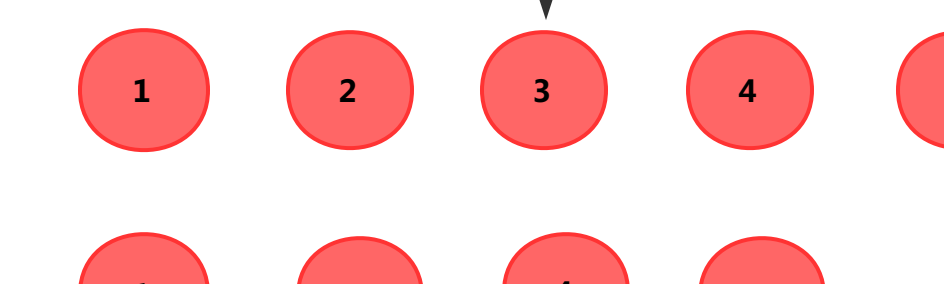
页面的DOM:



oldVnode:



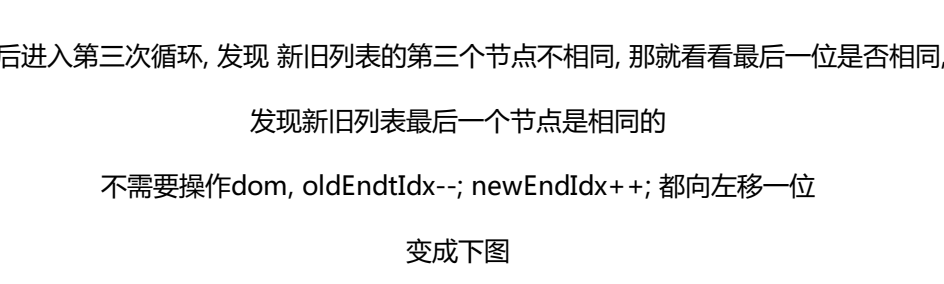
newVnode:



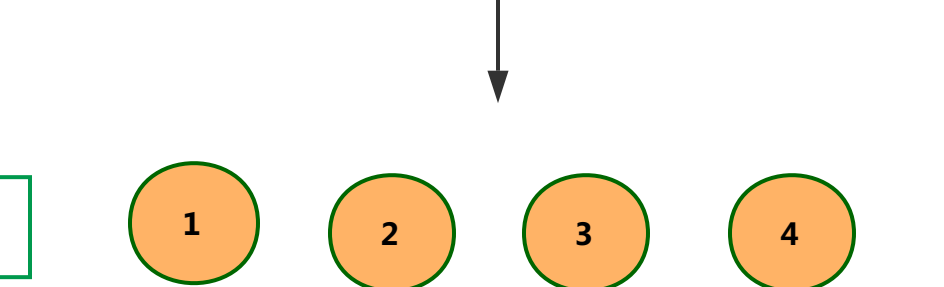
然后第二次对比, 发现 新旧列表的第二个节点相同, 都是 h2  
则, 不需要操作dom, oldStartIdx++; newStartIdx++; 都向右移一位

变成下图

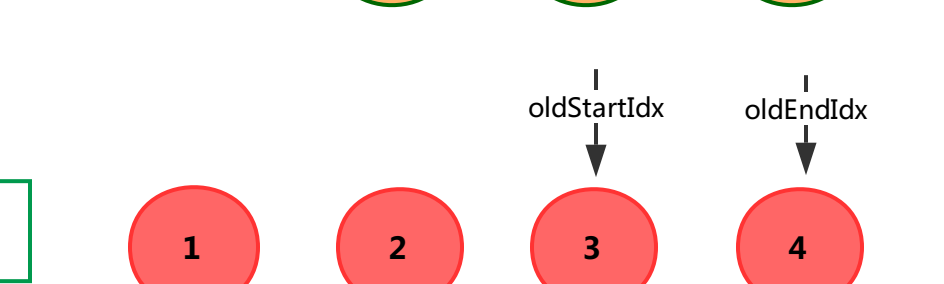
页面的DOM:



oldVnode:



newVnode:

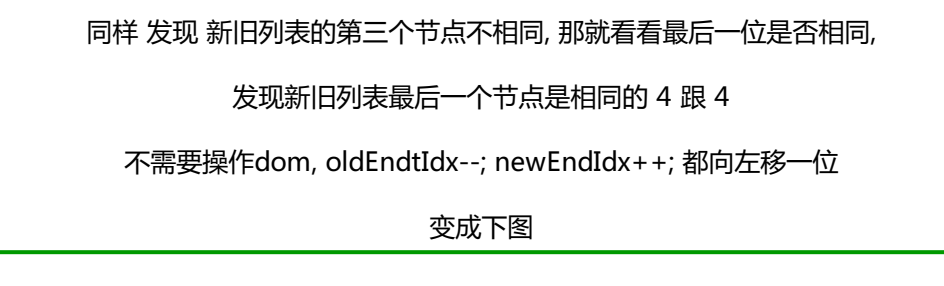


然后进入第三次循环, 发现 新旧列表的第三个节点不相同, 那就看看最后一位是否相同,  
发现新旧列表最后一个节点是相同的

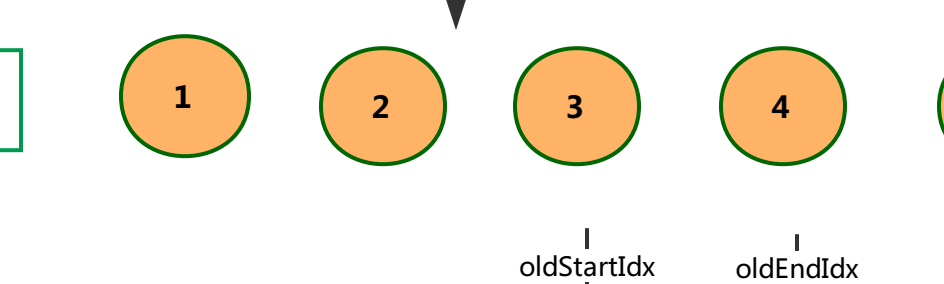
不需要操作dom, oldEndIdx--; newEndIdx++; 都向左移一位

变成下图

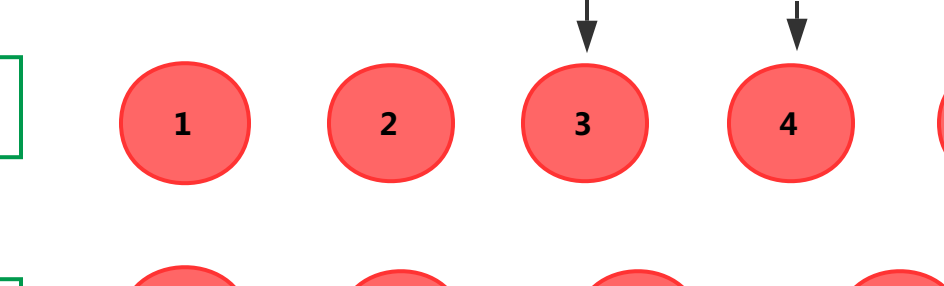
页面的DOM:



oldVnode:



newVnode:



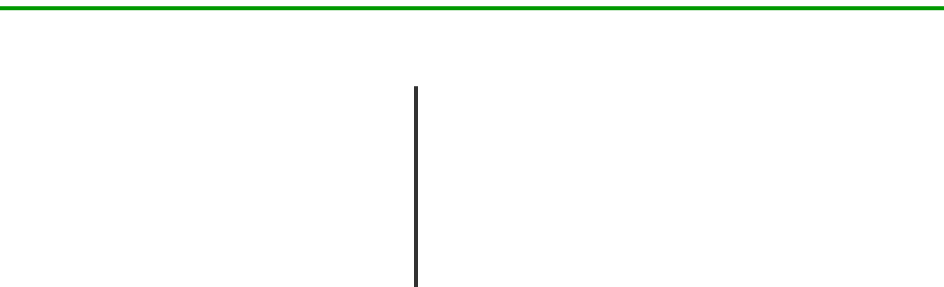
然后进入第四次循环, 可以看到 newStartIdx 和 newEndIdx 已经重合了,  
但 oldStartIdx 和 oldEndIdx还没重合, 所以需要继续比较

同样 发现 新旧列表的第三个节点不相同, 那就看看最后一位是否相同,  
发现新旧列表最后一个节点是相同的 4 跟 4

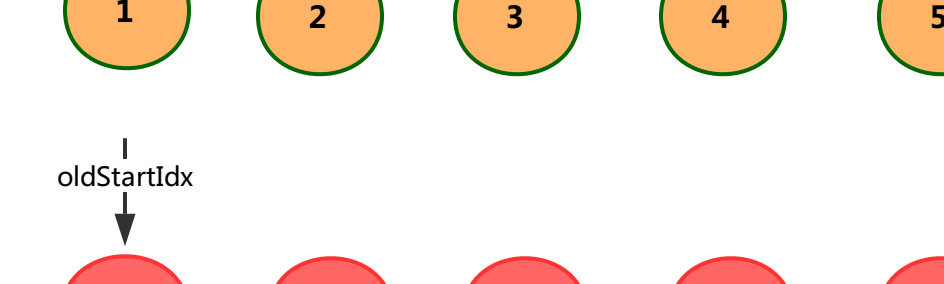
不需要操作dom, oldEndIdx--; newEndIdx++; 都向左移一位

变成下图

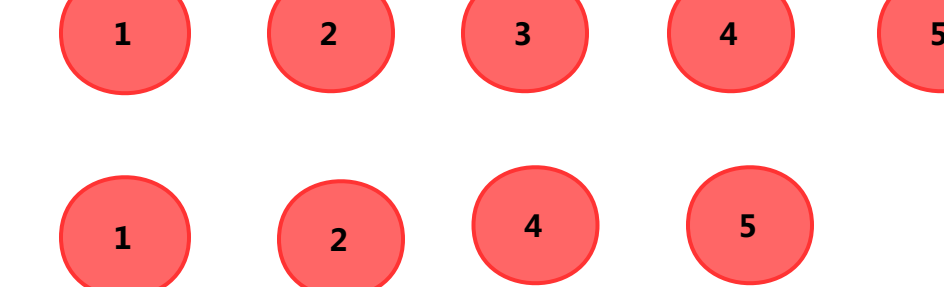
页面的DOM:



oldVnode:



newVnode:



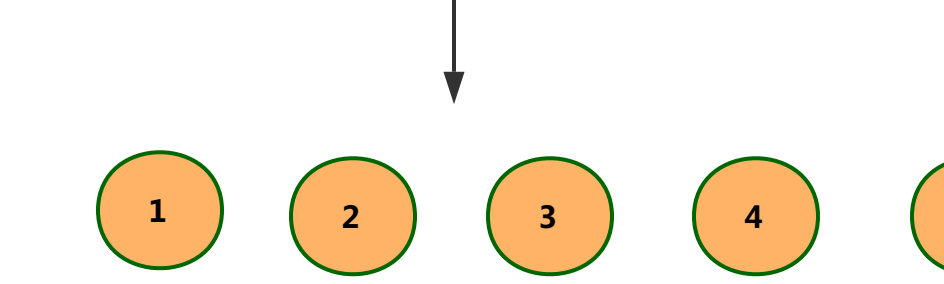
到了这里, 发现 newEndIdx < newStartIdx, 触发了结束比较的条件, diff完成, 删除oldStartIdx 对应的 elm 3 即可  
还有新增阶段也类似, 只不过, 最后是 oldEndIdx < oldStartIdx, 新增 newStartIdx即可, 反过来而已,

自己画个图, 或者调试一下就很容易理解

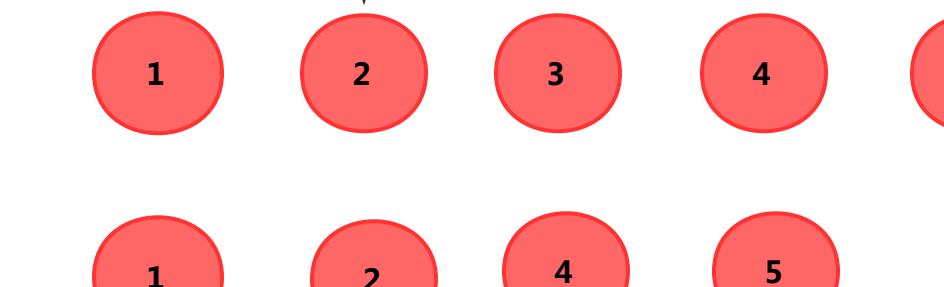
为什么要用两端比较, 先处理首尾相同的节点呢?

下面用单指针来分析一下, for循环 newVnode, 看看会发生哪些操作

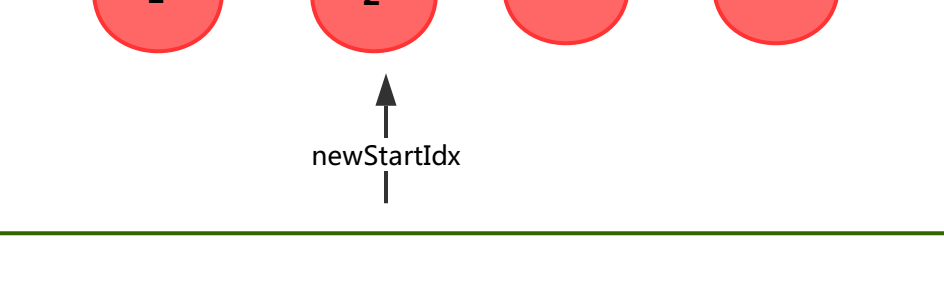
页面的DOM:



oldVnode:

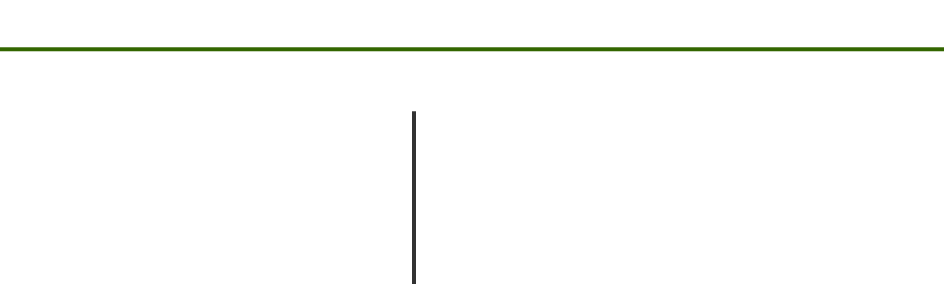


newVnode:

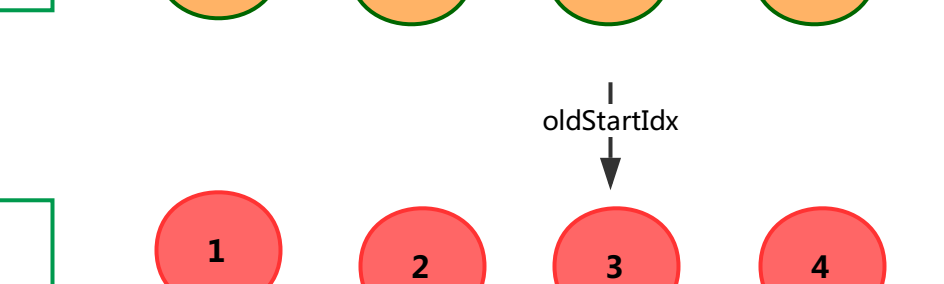


第一次发现 新旧节点第一个相同, 向右移动, 如下图

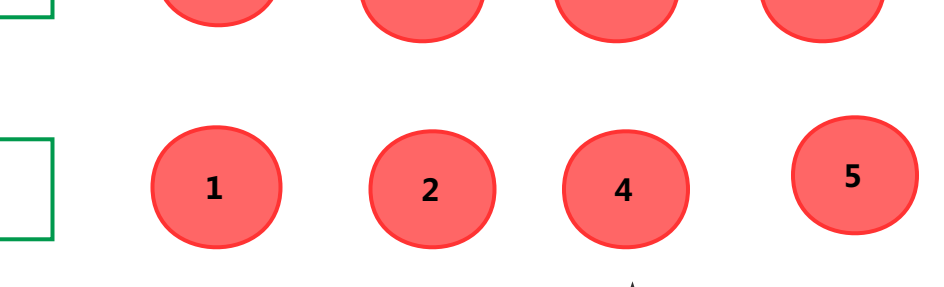
页面的DOM:



oldVnode:

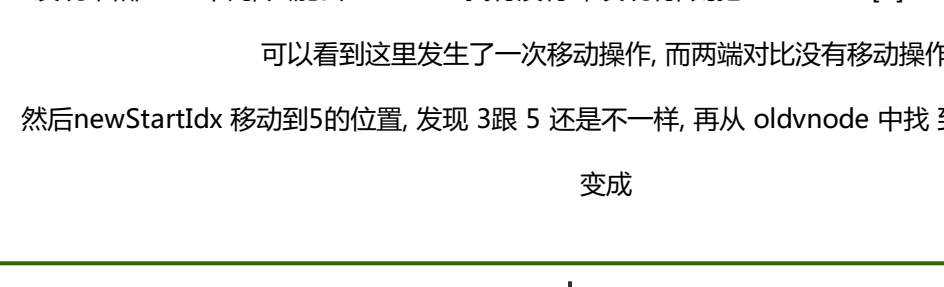


newVnode:

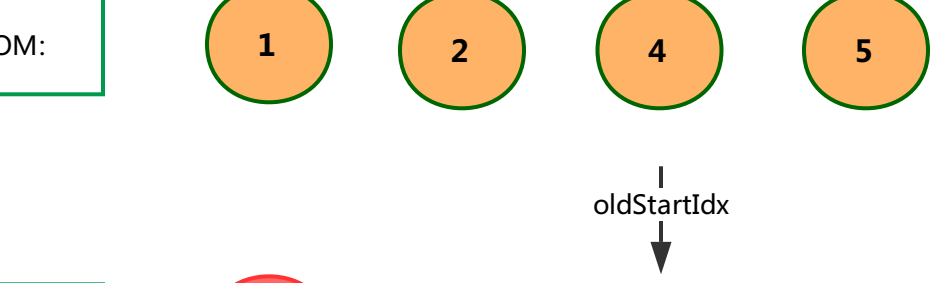


发现首部还是相同 都是2, 继续右移一位

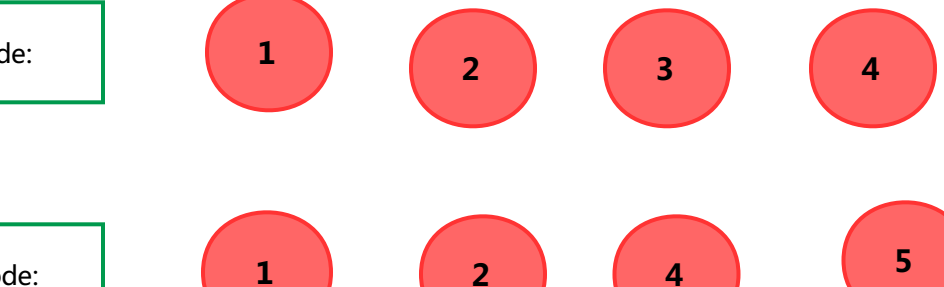
页面的DOM:



oldVnode:



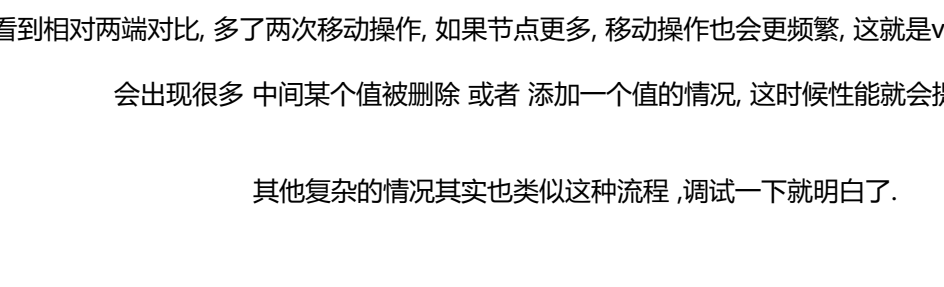
newVnode:



发现 首部 3 4 不同, 只能去oldVnode找有没有4, 发现有, 则把 oldVnode[4].elm 移动到 3.elm前面,  
可以看到这里发生了一次移动操作, 而两端对比没有移动操作

然后newStartIdx 移动到5的位置, 发现 3跟 5 还是不一样, 再从 oldVnode 中找到 5节点, 移动到3前面  
变成

页面的DOM:



oldVnode:



newVnode:



dom节点发生了变化, newStartIdx也指向了空, diff完成, 把oldStartIdx对应的dom节点3移除即可;

可以看到相对两端对比, 多了两次移动操作, 如果节点更多, 移动操作也会更频繁, 这就是vue两端比较算法的好处

会出现很多 中间某个值被删除 或者 添加一个值的情况, 这时候性能就会提升很多

其他复杂的情况其实也类似这种流程, 调试一下就明白了。