☰ 逐行剖析 Vue.js 源码

1. 前言

在上一篇文章中，我们介绍了当新的 `VNode` 与旧的 `oldVNode` 都是元素节点并且都包含子节点时， `Vue` 对子节点是

先外层循环 `newChildren` 数组，再内层循环 `oldChildren` 数组，每循环外层 `newChildren` 数组里的一个子节点，就去内层 `oldChildren` 数组里找看有没有与之相同的子节点，最后根据不同的情况作出不同的操作。

在上一篇文章的结尾我们也说了，这种方法虽然能够解决问题，但是还存在可优化的地方。比如当包含的子节点数量很多时，这样循环算法的时间复杂度就会变的很大，不利于性能提升。当然， `Vue` 也意识到了这点，并对此也进行了优化，那么本篇文章，就来学习一下关于子节点更新的优化问题 `Vue` 是如何做的。

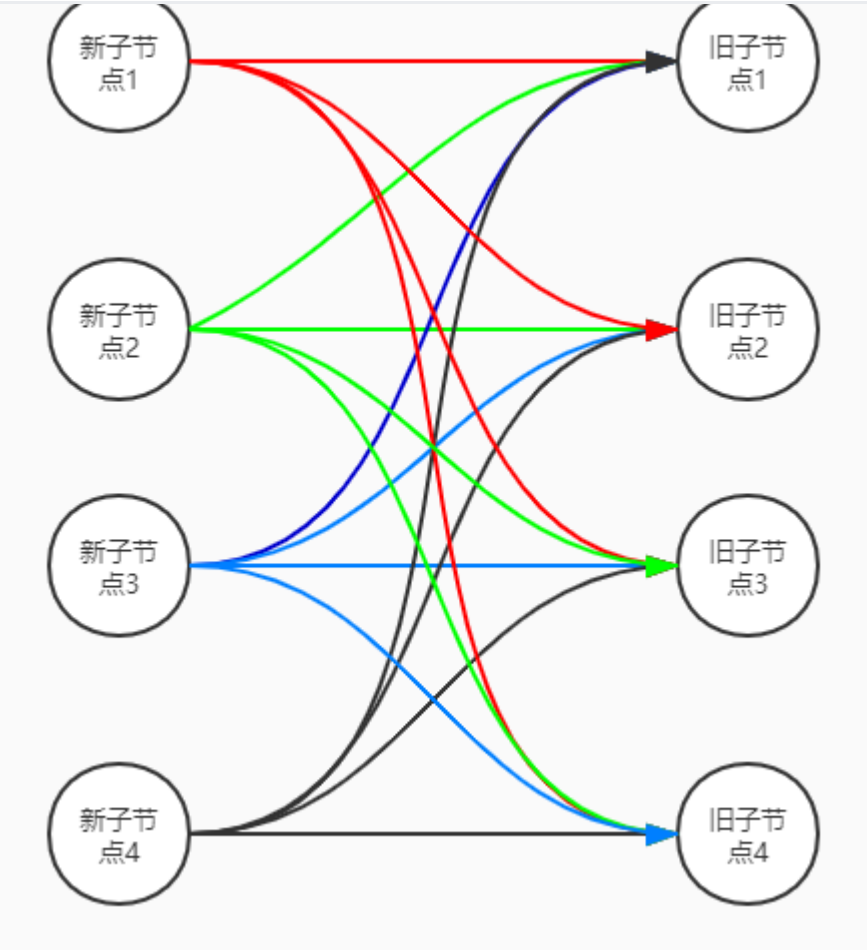
2. 优化策略介绍

假如我们现有一份新的 `newChildren` 数组和旧的 `oldChildren` 数组，如下所示：

```js
newChildren = ['新子节点1','新子节点2','新子节点3','新子节点4']
oldChildren = ['旧子节点1','旧子节点2','旧子节点3','旧子节点4']
```

如果按照优化之前的解决方案，那么我们接下来的操作应该是这样的：先循环 `newChildren` 数组，拿到第一个新子节点1，然后用第一个新子节点1去跟 `oldChildren` 数组里的旧子节点逐一对比，如果运气好一点，刚好 `oldChildren` 数组里的第一个旧子节点1与第一个新子节点1相同，那就皆大欢喜，直接处理，不用再往下循环了。那如果运气坏一点，直到循环到 `oldChildren` 数组里的第四个旧子节点4才与第一个新子节点1相同，那此时就会多循环了4次。我们不妨把情况再设想的极端一点，如果 `newChildren` 数组和 `oldChildren` 数组里前三个节点都没有变化，只是第四个节点发生了变化，那么我们就会循环16次，只有在第16次

☰ 逐行剖析 Vue.js 源码

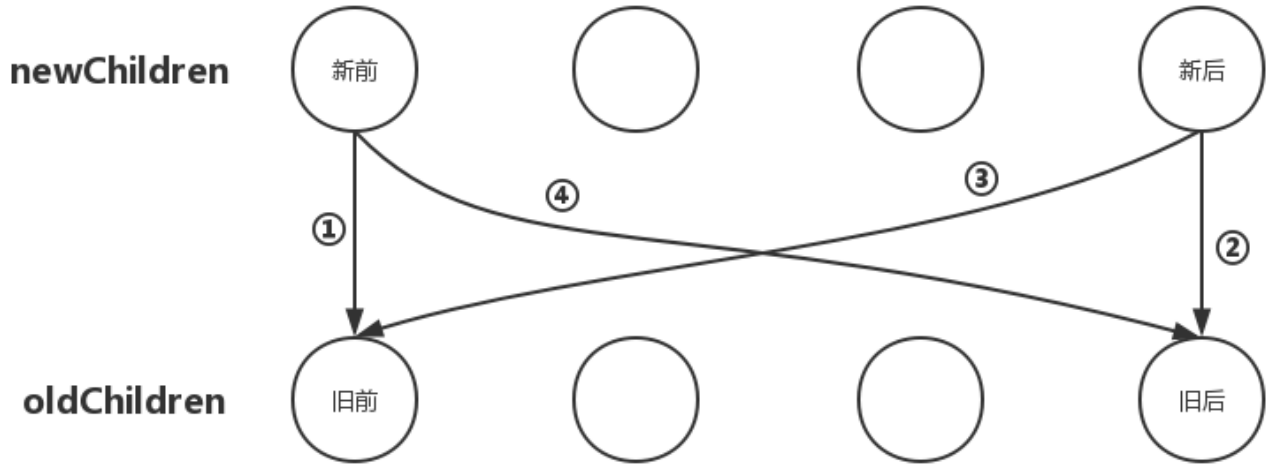


上面例子中只有四个子节点，好像还看不出来有什么缺陷，但是当子节点数量很多的时候，算法的时间复杂度就会非常高，很不利于性能提升。

那么我们该怎么优化呢？其实我们可以这样想，我们不要按顺序去循环 `newChildren` 和 `oldChildren` 这两个数组，可以先比较这两个数组里特殊位置的子节点，比如：

- 先把 `newChildren` 数组里的所有未处理子节点的第一个子节点和 `oldChildren` 数组里所有未处理子节点的第一个子节点做比对，如果相同，那就直接进入更新节点的操作；
- 如果不同，再把 `newChildren` 数组里所有未处理子节点的最后一个子节点和 `oldChildren` 数组里所有未处理子节点的最后一个子节点做比对，如果相同，那就直接进入更新节点的操作；
- 如果不同，再把 `newChildren` 数组里所有未处理子节点的最后一个子节点和 `oldChildren` 数组里所有未处理子节点的第一个子节点做比对，如果相同，那就直接进入更新节点的操作，更新完后再将 `oldChildren` 数组里的该节点移动到与 `newChildren` 数组里节点相同的位置；
- 如果不同，再把 `newChildren` 数组里所有未处理子节点的第一个子节点和 `oldChildren` 数组里所有未处理子节点的最后一个子节点做比对，如果相同，那就直接进入更新节点的操作，更新完后再将 `oldChildren` 数组里的该节点移动到与 `newChildren` 数组里节点相同的位置；

其过程如下图所示：



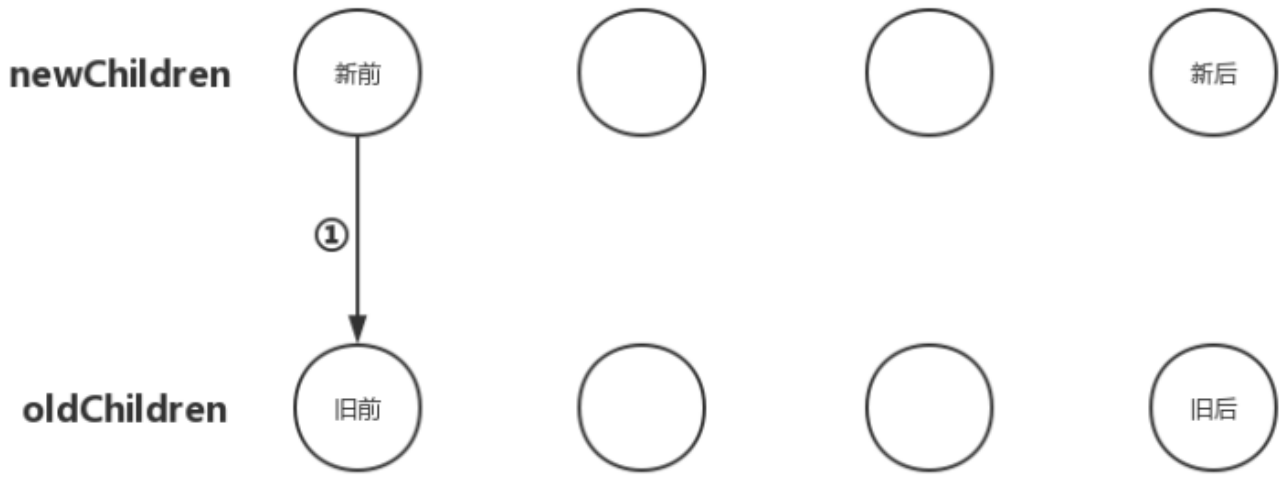
在上图中，我们把：

- `newChildren` 数组里的所有未处理子节点的第一个子节点称为：新前；
- `newChildren` 数组里的所有未处理子节点的最后一个子节点称为：新后；
- `oldChildren` 数组里的所有未处理子节点的第一个子节点称为：旧前；
- `oldChildren` 数组里的所有未处理子节点的最后一个子节点称为：旧后；

OK，有了以上概念以后，下面我们就来看看其具体是如何实施的。

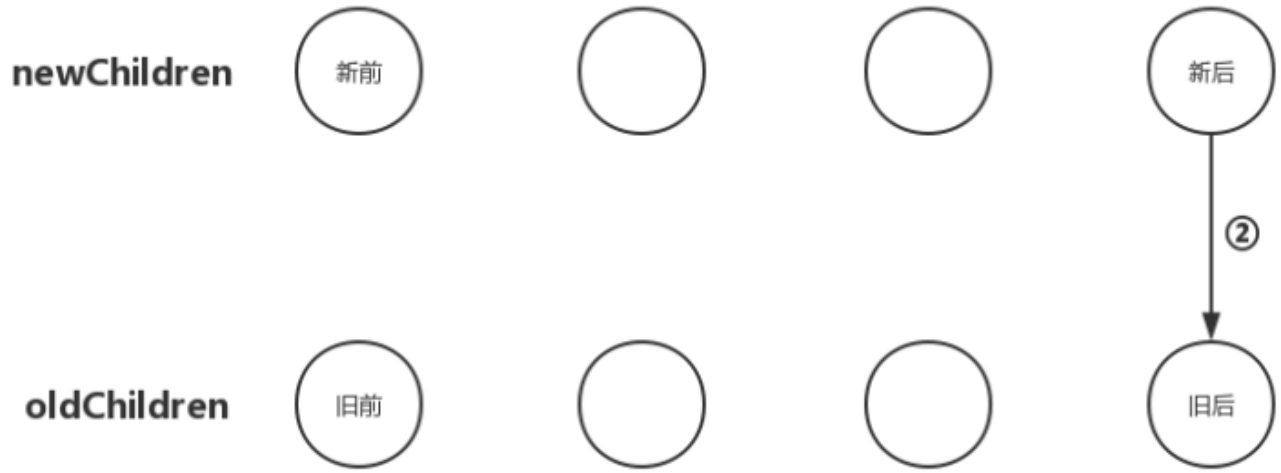
3. 新前与旧前

把 `newChildren` 数组里的所有未处理子节点的第一个子节点和 `oldChildren` 数组里所有未处理子节点的第一个子节点做比对，如果相同，那好极了，直接进入之前文章中说的更新节点的操作并且由于新前与旧前两个节点的位置也相同，无需进行节点移动操作；如果不同，没关系，再尝试后面三种情况。



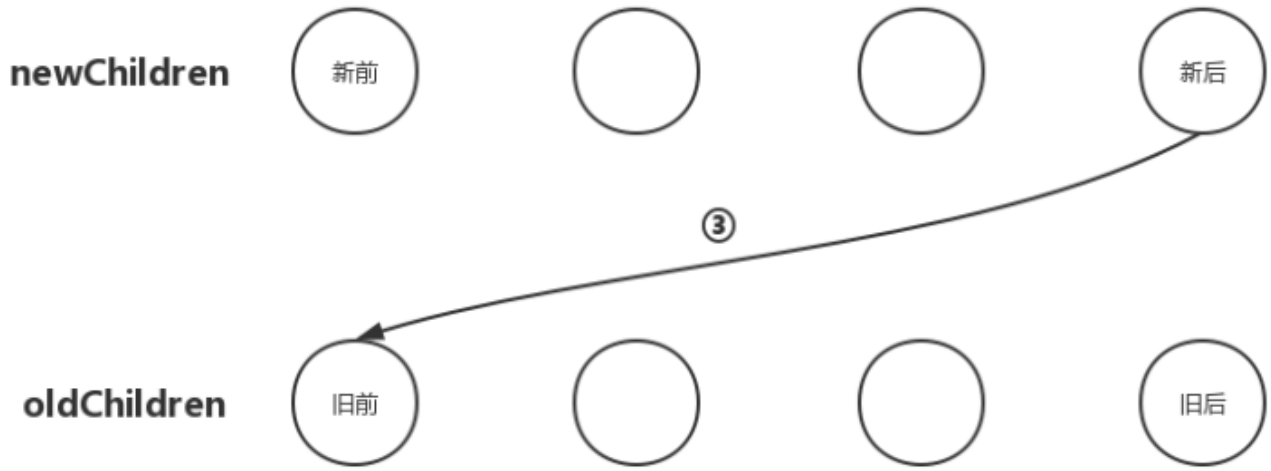
4. 新后与旧后

把 `newChildren` 数组里所有未处理子节点的最后一个子节点和 `oldChildren` 数组里所有未处理子节点的最后一个子节点做比对，如果相同，那就直接进入更新节点的操作并且由于新后与旧后两个节点的位置也相同，无需进行节点移动操作；如果不同，继续往后尝试。

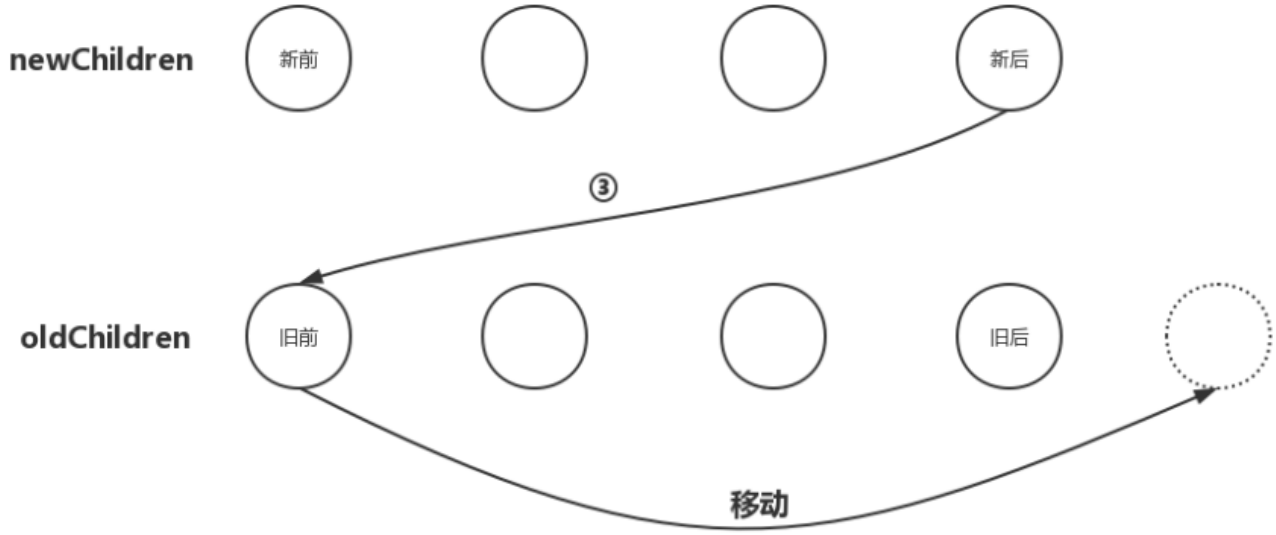


5. 新后与旧前

把 `newChildren` 数组里所有未处理子节点的最后一个子节点和 `oldChildren` 数组里所有未处理子节点的第一个子节点做比对，如果相同，那就直接进入更新节点的操作，更新完后再将 `oldChildren` 数组里的该节点移动到与 `newChildren` 数组里节点相同的位置；



此时，出现了移动节点的操作，移动节点最关键的地方在于找准要移动的位置。我们一再强调，**更新节点要以新 `VNode` 为基准，然后操作旧的 `oldVNode` ，使之最后旧的 `oldVNode` 与新的 `VNode` 相同**。那么现在的情况是： `newChildren` 数组里的最后一个子节点与

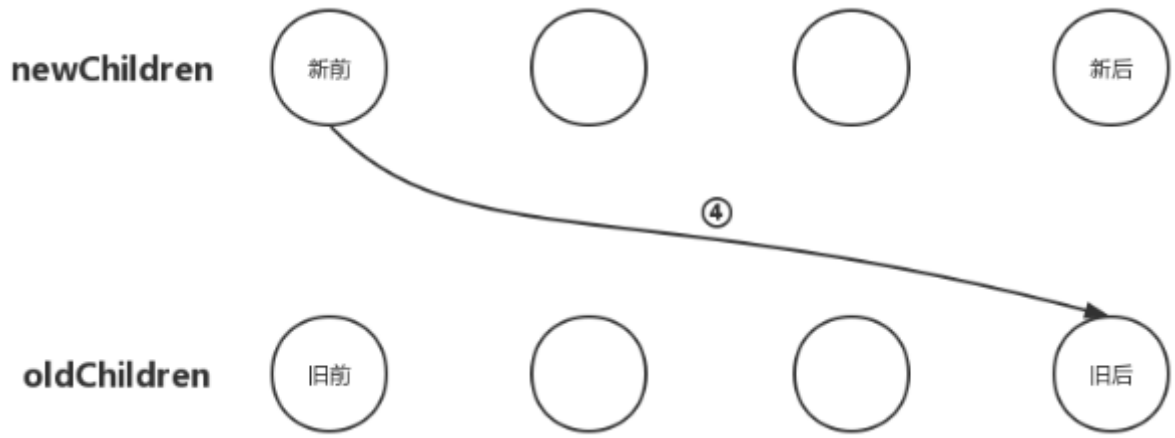


从图中不难看出，我们要把 `oldChildren` 数组里把第一个子节点移动到数组中**所有未处理节点之后**。

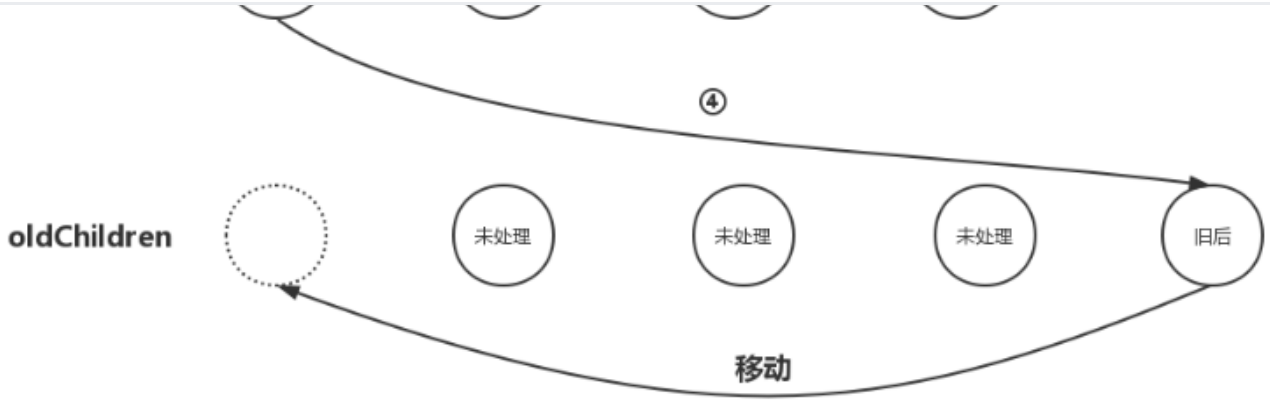
如果对比之后发现这两个节点仍不是同一个节点，那就继续尝试最后一种情况。

6. 新前与旧后

把 `newChildren` 数组里所有未处理子节点的第一个子节点和 `oldChildren` 数组里所有未处理子节点的最后一个子节点做比对，如果相同，那就直接进入更新节点的操作，更新完后再将 `oldChildren` 数组里的该节点移动到与 `newChildren` 数组里节点相同的位置；



同样，这种情况的节点移动位置逻辑与“新后与旧前”的逻辑类似，那就是 `newChildren` 数组里的第一个子节点与 `oldChildren` 数组里的最后一个子节点相同，那么我们就应该在 `oldChildren` 数组里把最后一个子节点移动到第一个子节点的位置，如下图：



从图中不难看出，我们要把 `oldChildren` 数组里把最后一个子节点移动到数组中**所有未处理节点之前**。

OK，以上就是子节点对比更新优化策略种的4种情况，如果以上4种情况逐个试遍之后要是还没找到相同的节点，那就再通过之前的循环方式查找。

7. 回到源码

思路分析完，逻辑理清之后，我们再回到源码里看看，验证一下源码实现的逻辑是否跟我们分析的一样。源码如下：

```js
// 循环更新子节点
  function updateChildren (parentElm, oldCh, newCh, insertedVnodeQueue,
    let oldStartIdx = 0                // oldChildren开始索引
    let oldEndIdx = oldCh.length - 1   // oldChildren结束索引
    let oldStartVnode = oldCh[0]        // oldChildren中所有未处理节点中的第
    let oldEndVnode = oldCh[oldEndIdx]   // oldChildren中所有未处理节点中的

    let newStartIdx = 0                // newChildren开始索引
    let newEndIdx = newCh.length - 1   // newChildren结束索引
    let newStartVnode = newCh[0]        // newChildren中所有未处理节点中的第
    let newEndVnode = newCh[newEndIdx]  // newChildren中所有未处理节点中的最

    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
    const canMove = !removeOnly

    if (process.env.NODE_ENV !== 'production') {
```

逐行剖析 Vue.js 源码

```
24      // 以"新前"、"新后"、"旧前"、"旧后"的方式开始比对节点
25      while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
26        if (isUndef(oldStartVnode)) {
27          oldStartVnode = oldCh[++oldStartIdx] // 如果oldStartVnode不存在，
28        } else if (isUndef(oldEndVnode)) {
29          oldEndVnode = oldCh[--oldEndIdx]
30        } else if (sameVnode(oldStartVnode, newStartVnode)) {
31          // 如果新前与旧前节点相同，就把两个节点进行patch更新
32          patchVnode(oldStartVnode, newStartVnode, insertedVnodeQueue)
33          oldStartVnode = oldCh[++oldStartIdx]
34          newStartVnode = newCh[++newStartIdx]
35        } else if (sameVnode(oldEndVnode, newEndVnode)) {
36          // 如果新后与旧后节点相同，就把两个节点进行patch更新
37          patchVnode(oldEndVnode, newEndVnode, insertedVnodeQueue)
38          oldEndVnode = oldCh[--oldEndIdx]
39          newEndVnode = newCh[--newEndIdx]
40        } else if (sameVnode(oldStartVnode, newEndVnode)) { // Vnode move
41          // 如果新后与旧前节点相同，先把两个节点进行patch更新，然后把旧前节点移动到
42          patchVnode(oldStartVnode, newEndVnode, insertedVnodeQueue)
43          canMove && nodeOps.insertBefore(parentElm, oldStartVnode.elm, n
44          oldStartVnode = oldCh[++oldStartIdx]
45          newEndVnode = newCh[--newEndIdx]
46        } else if (sameVnode(oldEndVnode, newStartVnode)) { // Vnode move
47          // 如果新前与旧后节点相同，先把两个节点进行patch更新，然后把旧后节点移动到
48          patchVnode(oldEndVnode, newStartVnode, insertedVnodeQueue)
49          canMove && nodeOps.insertBefore(parentElm, oldEndVnode.elm, old
50          oldEndVnode = oldCh[--oldEndIdx]
51          newStartVnode = newCh[++newStartIdx]
52        } else {
53          // 如果不属于以上四种情况，就进行常规的循环比对patch
54          if (isUndef(oldKeyToIdx)) oldKeyToIdx = createKeyToOldIdx(oldCh,
55          idxInOld = isDef(newStartVnode.key)
56            ? oldKeyToIdx[newStartVnode.key]
57            : findIdxInOld(newStartVnode, oldCh, oldStartIdx, oldEndIdx)
58          // 如果在oldChildren里找不到当前循环的newChildren里的子节点
59          if (isUndef(idxInOld)) { // New element
60            // 新增节点并插入到合适位置
61            createElm(newStartVnode, insertedVnodeQueue, parentElm, oldSta
62          } else {
63            // 如果在oldChildren里找到了当前循环的newChildren里的子节点
64            vnodeToMove = oldCh[idxInOld]
65            // 如果两个节点相同
66            if (sameVnode(vnodeToMove, newStartVnode)) {
67              // 调用patchVnode更新节点
```

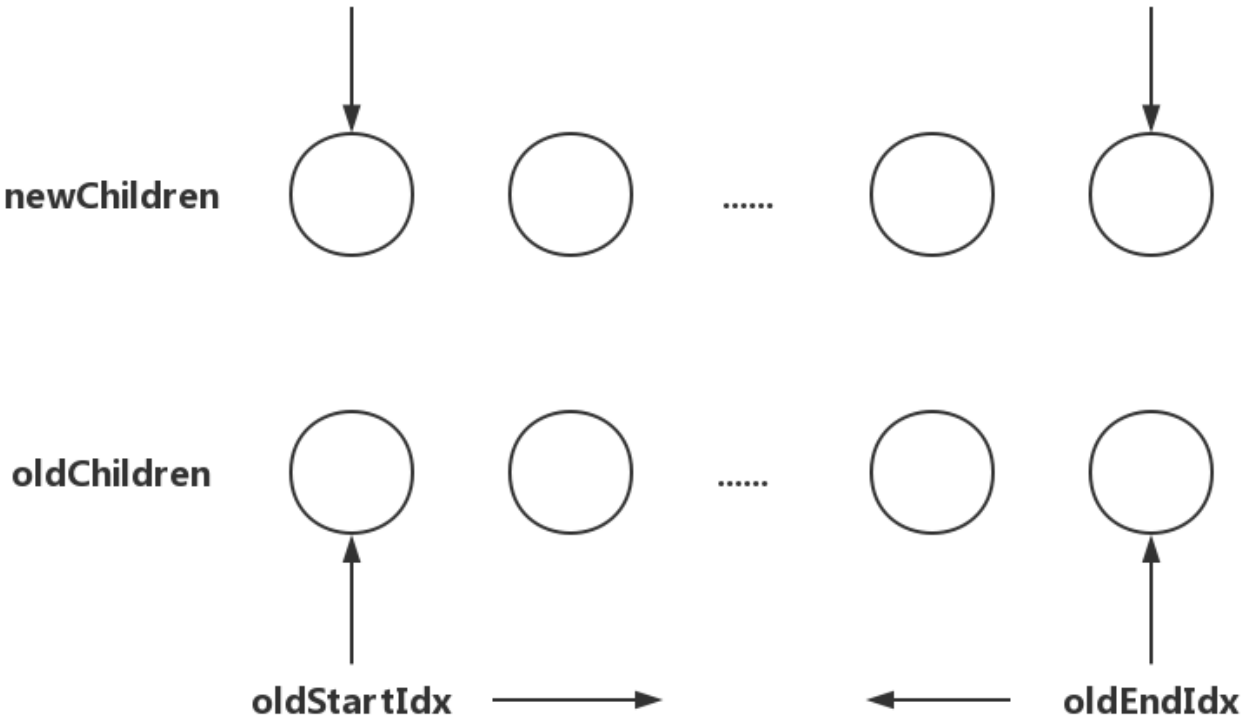
逐行剖析 Vue.js 源码

```
            // canMove表示是否需要移动节点，如果为true表示需要移动，则移动节点，
            canMove && nodeOps.insertBefore(parentElm, vnodeToMove.elm,
          } else {
            // same key but different element. treat as new element
            createElm(newStartVnode, insertedVnodeQueue, parentElm, oldS
          }
        }
        newStartVnode = newCh[++newStartIdx]
      }
    }
    if (oldStartIdx > oldEndIdx) {
      /**
       * 如果oldChildren比newChildren先循环完毕，
       * 那么newChildren里面剩余的节点都是需要新增的节点，
       * 把[newStartIdx, newEndIdx]之间的所有节点都插入到DOM中
       */
      refElm = isUndef(newCh[newEndIdx + 1]) ? null : newCh[newEndIdx +
      addVnodes(parentElm, refElm, newCh, newStartIdx, newEndIdx, inser
    } else if (newStartIdx > newEndIdx) {
      /**
       * 如果newChildren比oldChildren先循环完毕，
       * 那么oldChildren里面剩余的节点都是需要删除的节点，
       * 把[oldStartIdx, oldEndIdx]之间的所有节点都删除
       */
      removeVnodes(parentElm, oldCh, oldStartIdx, oldEndIdx)
    }
  }

```

读源码之前，我们先有这样一个概念：那就是在我们前面所说的优化策略中，节点有可能是从前面对比，也有可能是从后面对比，对比成功就会进行更新处理，也就是说我们有可能处理第一个，也有可能处理最后一个，那么我们在循环的时候就不能简单从前往后或从后往前循环，而是要从两边向中间循环。

☰ 逐行剖析 Vue.js 源码



首先，我们先准备4个变量：

- **newStartIdx:** `newChildren` 数组里开始位置的下标；
- **newEndIdx:** `newChildren` 数组里结束位置的下标；
- **oldStartIdx:** `oldChildren` 数组里开始位置的下标；
- **oldEndIdx:** `oldChildren` 数组里结束位置的下标；

在循环的时候，每处理一个节点，就将下标向图中箭头所指的方向移动一个位置，开始位置所表示的节点被处理后，就向后移动一个位置；结束位置所表示的节点被处理后，就向前移动一个位置；由于我们的优化策略都是新旧节点两两更新的，所以一次更新将会移动两个节点。说的再直白一点就是： `newStartIdx` 和 `oldStartIdx` 只能往后移动（只会加）， `newEndIdx` 和 `oldEndIdx` 只能往前移动（只会减）。

当开始位置大于结束位置时，表示所有节点都已经遍历过了。

OK，有了这个概念后，我们开始读源码：

1. 如果 `oldStartVnode` 不存在，则直接跳过，将 `oldStartIdx` 加1，比对下一个

```
                                                                    js
1  // 以"新前"、"新后"、"旧前"、"旧后"的方式开始比对节点
2  while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
3   if (isUndef(oldStartVnode)) {
4     oldStartVnode = oldCh[++oldStartIdx]
```

☰ 逐行剖析 Vue.js 源码

2. 如果 `oldEndVnode` 不存在，则直接跳过，将 `oldEndIdx` 减1，比对前一个

```js
else if (isUndef(oldEndVnode)) {
    oldEndVnode = oldCh[--oldEndIdx]
}
```

3. 如果新前与旧前节点相同，就把两个节点进行 `patch` 更新，同时 `oldStartIdx` 和 `newStartIdx` 都加1，后移一个位置

```js
else if (sameVnode(oldStartVnode, newStartVnode)) {
    patchVnode(oldStartVnode, newStartVnode, insertedVnodeQueue)
    oldStartVnode = oldCh[++oldStartIdx]
    newStartVnode = newCh[++newStartIdx]
}
```

4. 如果新后与旧后节点相同，就把两个节点进行 `patch` 更新，同时 `oldEndIdx` 和 `newEndIdx` 都减1，前移一个位置

```js
else if (sameVnode(oldEndVnode, newEndVnode)) {
    patchVnode(oldEndVnode, newEndVnode, insertedVnodeQueue)
    oldEndVnode = oldCh[--oldEndIdx]
    newEndVnode = newCh[--newEndIdx]
}
```

5. 如果新后与旧前节点相同，先把两个节点进行 `patch` 更新，然后把旧前节点移动到 `oldChilren` 中所有未处理节点之后，最后把 `oldStartIdx` 加1，后移一个位置，`newEndIdx` 减1，前移一个位置

```js
else if (sameVnode(oldStartVnode, newEndVnode)) {
    patchVnode(oldStartVnode, newEndVnode, insertedVnodeQueue)
    canMove && nodeOps.insertBefore(parentElm, oldStartVnode.elm, nod
    oldStartVnode = oldCh[++oldStartIdx]
    newEndVnode = newCh[--newEndIdx]
}
```

oldEndIdx 减1，前移一个位置

```js
else if (sameVnode(oldEndVnode, newStartVnode)) { // Vnode moved left
    patchVnode(oldEndVnode, newStartVnode, insertedVnodeQueue)
    canMove && nodeOps.insertBefore(parentElm, oldEndVnode.elm, oldSt
    oldEndVnode = oldCh[--oldEndIdx]
    newStartVnode = newCh[++newStartIdx]
}
```

7. 如果不属于以上四种情况，就进行常规的循环比对 patch

8. 如果在循环中， oldStartIdx 大于 oldEndIdx 了，那就表示 oldChildren 比 newChildren 先循环完毕，那么 newChildren 里面剩余的节点都是需要新增的节点，把 [newStartIdx, newEndIdx] 之间的所有节点都插入到 DOM 中

```js
if (oldStartIdx > oldEndIdx) {
    refElm = isUndef(newCh[newEndIdx + 1]) ? null : newCh[newEndIdx +
    addVnodes(parentElm, refElm, newCh, newStartIdx, newEndIdx, inser
}
```

9. 如果在循环中， newStartIdx 大于 newEndIdx 了，那就表示 newChildren 比 oldChildren 先循环完毕，那么 oldChildren 里面剩余的节点都是需要删除的节点，把 [oldStartIdx, oldEndIdx] 之间的所有节点都删除

```js
else if (newStartIdx > newEndIdx) {
    removeVnodes(parentElm, oldCh, oldStartIdx, oldEndIdx)
}
```

OK,处理完毕，可见源码中的处理逻辑跟我们之前分析的逻辑是一样的。

8. 总结

本篇文章中，我们介绍了 Vue 中子节点更新的优化策略，发现 Vue 为了避免双重循环数据量大时间复杂度升高带来的性能问题，而选择了从子节点数组中的4个特殊位置互相比对，分别是：新前与旧前，新后与旧后，新后与旧前，新前与旧后。对于每一种情况我们都通过图文的形式对其逻辑进行了分析。最后我们回到源码，通过阅读源码来验证我们分析的是否正确。

源码的时候就有比较清晰的思路了。

在 GitHub 上编辑此页

上次更新: 3/24/2020, 5:37:47 AM

← 更新子节点 综述 →