

vue源码剖析（二）



复习

<https://www.processon.com/view/link/5da6c108e4b002a6448895c3>

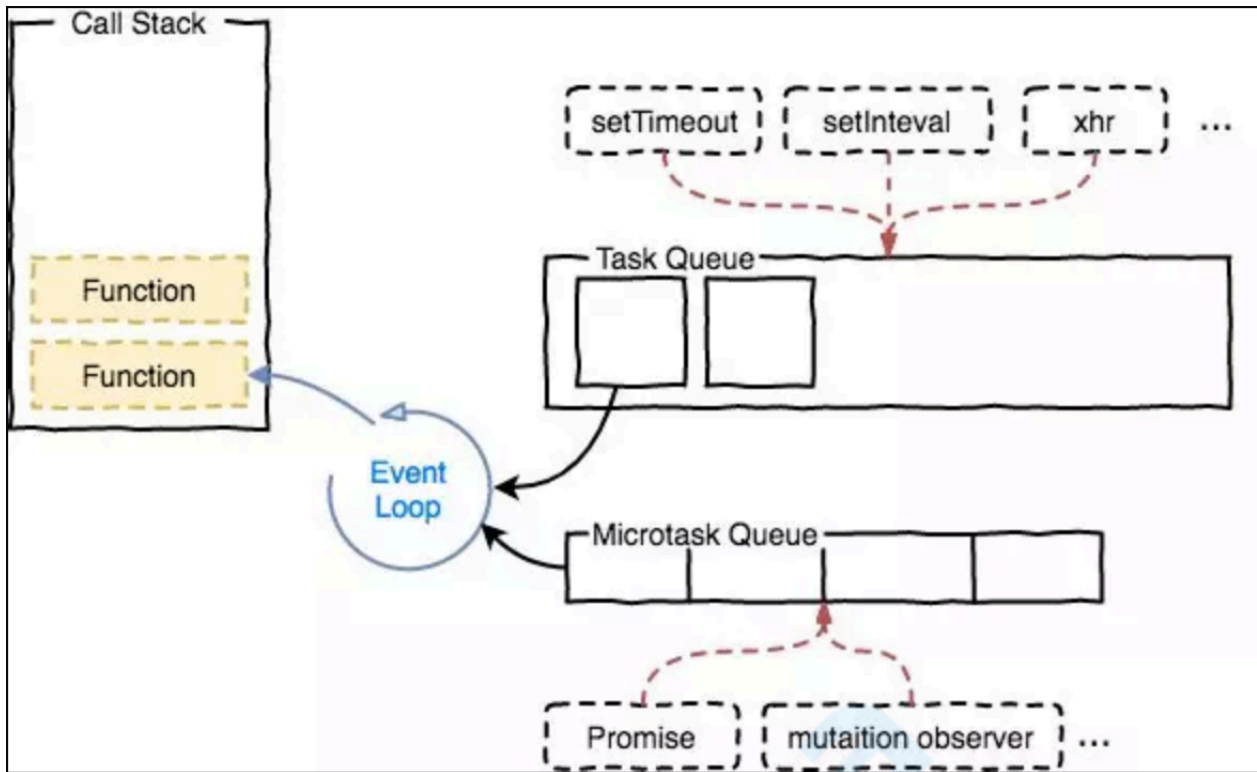
学习目标

- 理解Vue批量异步更新策略
- 掌握虚拟DOM和Diff算法

异步更新队列

Vue高效的秘诀是一套**批量、异步**的更新策略。

概念解释



- 事件循环Event Loop：浏览器为了协调事件处理、脚本执行、网络请求和渲染等任务而制定的工作机制。
- 宏任务Task：代表一个个离散的、独立的工作单元。**浏览器完成一个宏任务，在下一个宏任务执行开始前，会对页面进行重新渲染。**主要包括创建文档对象、解析HTML、执行主线JS代码以及各种事件如页面加载、输入、网络事件和定时器等。
- 微任务：微任务是更小的任务，是在当前宏任务执行结束后立即执行的任务。**如果存在微任务，浏览器会清空微任务之后再重新渲染。**微任务的例子有 Promise 回调函数、DOM变化等。

[体验一下](#)

vue中的具体实现

Promise.then(flushQueue)



- 异步：只要侦听到数据变化，Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据变更。
- 批量：如果同一个 watcher 被多次触发，只会被推入到队列中一次。去重对于避免不必要的计算和 DOM 操作是非常重要的。然后，在下一个的事件循环“tick”中，Vue 刷新队列执行实际工作。
- 异步策略：Vue 在内部对异步队列尝试使用原生的 `Promise.then`、`MutationObserver` 或 `setImmediate`，如果执行环境都不支持，则会采用 `setTimeout` 代替。

update() core\observer\watcher.js

dep.notify()之后watcher执行更新，执行入队操作

queueWatcher(watcher) core\observer\scheduler.js

执行watcher入队操作

nextTick(flushSchedulerQueue) core\util\next-tick.js

nextTick按照特定异步策略执行队列操作

测试代码：03-timerFunc.html

watcher中update执行三次，但run仅执行一次，且数值变化对dom的影响也不是立竿见影的。

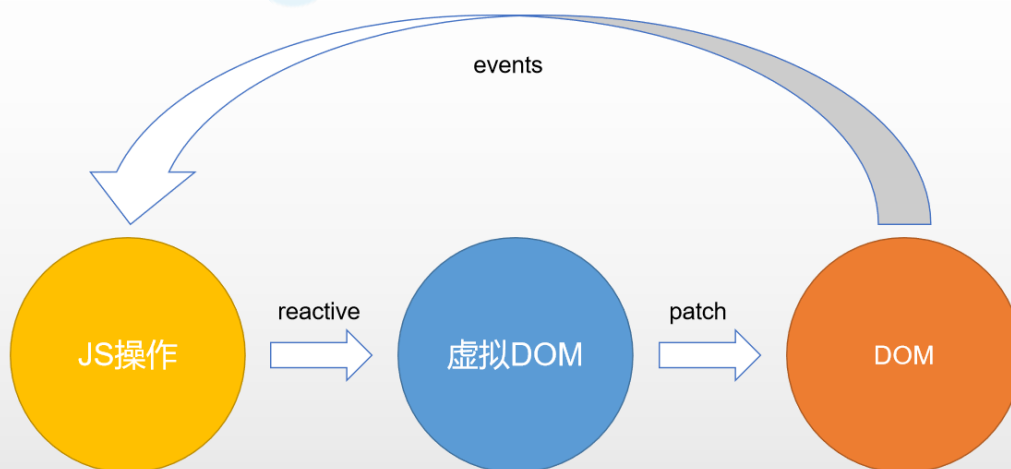
请大家研究一下相关API： `vm.$nextTick(cb)`

虚拟DOM

概念

虚拟DOM（Virtual DOM）是对DOM的JS抽象表示，它们是**JS对象**，能够描述DOM结构和关系。应用的各种状态变化会作用于虚拟DOM，最终映射到DOM上。

虚拟DOM的概念



体验虚拟DOM：

vue中虚拟dom基于snabbdom实现，安装snabbdom并体验

```
<!DOCTYPE html>
<html lang="en">
<head></head>

<body>
  <div id="app"></div>

  <!--安装并引入snabbdom-->
  <script src="../../node_modules/snabbdom/dist/snabbdom.js"></script>

  <script>

    // 之前编写的响应式函数
    function defineReactive(obj, key, val) {
      Object.defineProperty(obj, key, {
        get() {
          return val
        },
        set(newVal) {
          val = newVal
          // 通知更新
          update()
        }
      })
    }

    // 导入patch的工厂init, h是产生vnode的工厂
    const { init, h } = snabbdom
    // 获取patch函数
    const patch = init([])

    // 上次vnode, 由patch()返回
    let vnode;

    // 更新函数, 将数据操作转换为dom操作, 返回新vnode
    function update() {
      if (!vnode) {
        // 初始化, 没有上次vnode, 传入宿主元素和vnode
        vnode = patch(app, render())
      }
      else {
        // 更新, 传入新旧vnode对比并做更新
        vnode = patch(vnode, render())
      }
    }
  </script>
</body>
</html>
```

```

    }
  }

  // 渲染函数, 返回vnode描述dom结构
  function render() {
    return h('div', obj.foo)
  }

  // 数据
  const obj = {}

  // 定义响应式
  defineReactive(obj, 'foo', '')

  // 赋一个日期作为初始值
  obj.foo = new Date().toLocaleTimeString()

  // 定时改变数据, 更新函数会重新执行
  setInterval(() => {
    obj.foo = new Date().toLocaleTimeString()
  }, 1000);
</script>
</body>

</html>

```

优点

- 虚拟DOM轻量、快速：当它们发生变化时通过新旧虚拟DOM比对可以得到最小DOM操作量，配合异步更新策略减少刷新频率，从而提升性能

```
patch(vnode, h('div', obj.foo))
```

- 跨平台：将虚拟dom更新转换为不同运行时特殊操作实现跨平台

```

<script src="../../node_modules/snabbdom/dist/snabbdom-style.js"></script>
<script>
  // 增加style模块
  const patch = init([snabbdom_style.default])

  function render() {
    // 添加节点样式描述
    return h('div', { style: { color: 'red' } }, obj.foo)
  }
</script>

```

- 兼容性：还可以加入兼容性代码增强操作的兼容性

必要性

vue 1.0中有细粒度的数据变化侦测，它是不需要虚拟DOM的，但是细粒度造成了大量开销，这对于大型项目来说是不可接受的。因此，vue 2.0选择了中等粒度的解决方案，每一个组件一个watcher实例，这样状态变化时只能通知到组件，再通过引入虚拟DOM去进行比对和渲染。

整体流程

mountComponent() core/instance/lifecycle.js

渲染、更新组件

```
// 定义更新函数
const updateComponent = () => {
  // 实际调用是在lifeCycleMixin中定义的_update和renderMixin中定义的_render
  vm._update(vm._render(), hydrating)
}
```

_render core/instance/render.js

生成虚拟dom

_update core\instance\lifecycle.js

update负责更新dom，转换vnode为dom

__patch__() platforms/web/runtime/index.js

__patch__是在平台特有代码中指定的

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

测试代码，examples\test\04-vdom.html

patch获取

patch是createPatchFunction的返回值，传递nodeOps和modules是web平台特别实现

```
export const patch: Function = createPatchFunction({ nodeOps, modules })
```

platforms\web\runtime\node-ops.js

定义各种原生dom基础操作方法

platforms\web\runtime\modules\index.js

`modules` 定义了属性更新实现

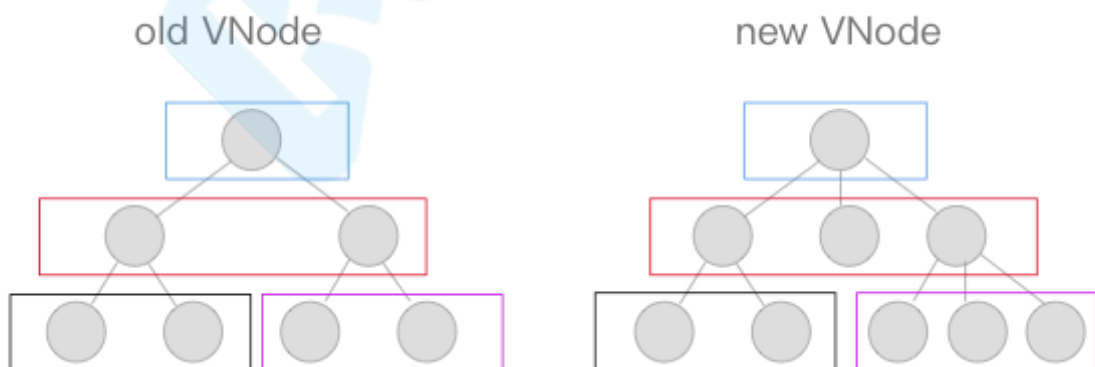
watcher.run() => componentUpdate() => render() => update() => patch()

patch实现

patch core\vdom\patch.js

首先进行树级别比较，可能有三种情况：增删改。

- new VNode不存在就删；
- old VNode不存在就增；
- 都存在就执行diff执行更新



patchVnode

比较两个VNode，包括三种类型操作：属性更新、文本更新、子节点更新

具体规则如下：

1. 新老节点均有**children**子节点，则对子节点进行diff操作，调用**updateChildren**
2. 如果新节点有子节点而老节点没有子节点，先清空老节点的文本内容，然后为其新增子节点。

3. 当新节点没有子节点而老节点有子节点的时候，则移除该节点的所有子节点。
4. 当新老节点都无子节点的时候，只是文本的替换。

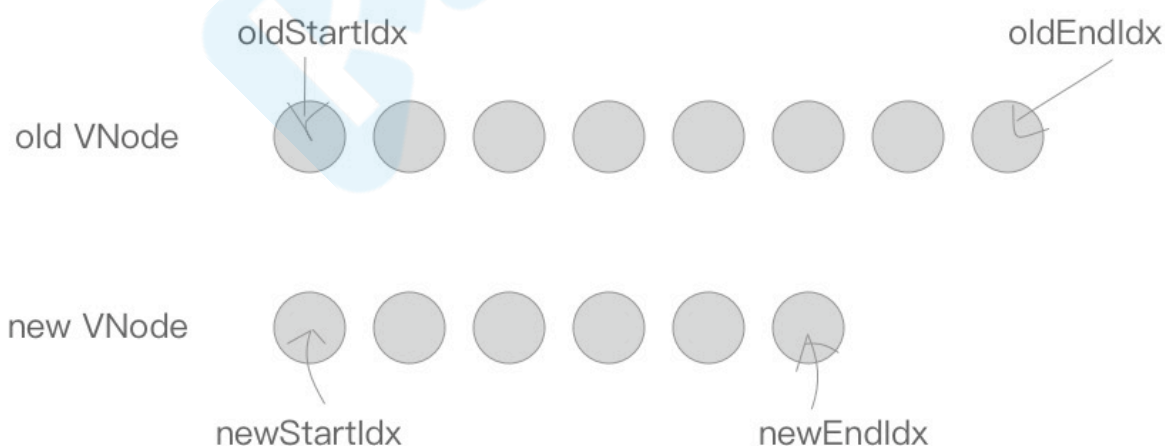
测试，04-vdom.html

```
<div id="demo">
  <h1>虚拟DOM</h1>
  <p>{{foo}}</p>
</div>
```

```
// patchVnode过程分解
// 1.div#demo  updateChildren
// 2.h1        updateChildren
// 3.text      文本相同跳过
// 4.p         updateChildren
// 5.text      setTextContent
```

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：

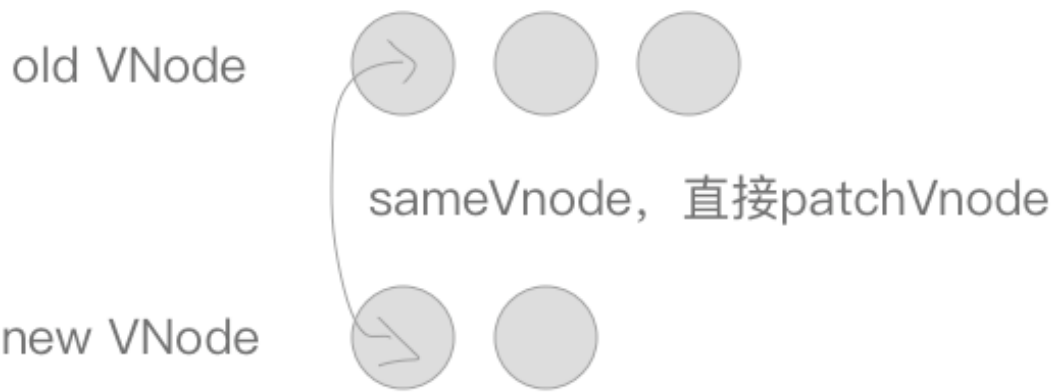


在新老两组VNode节点的左右头尾两侧都有一个变量标记，在遍历过程中这几个变量都会向中间靠拢。当 $\text{oldStartIdx} > \text{oldEndIdx}$ 或者 $\text{newStartIdx} > \text{newEndIdx}$ 时结束循环。

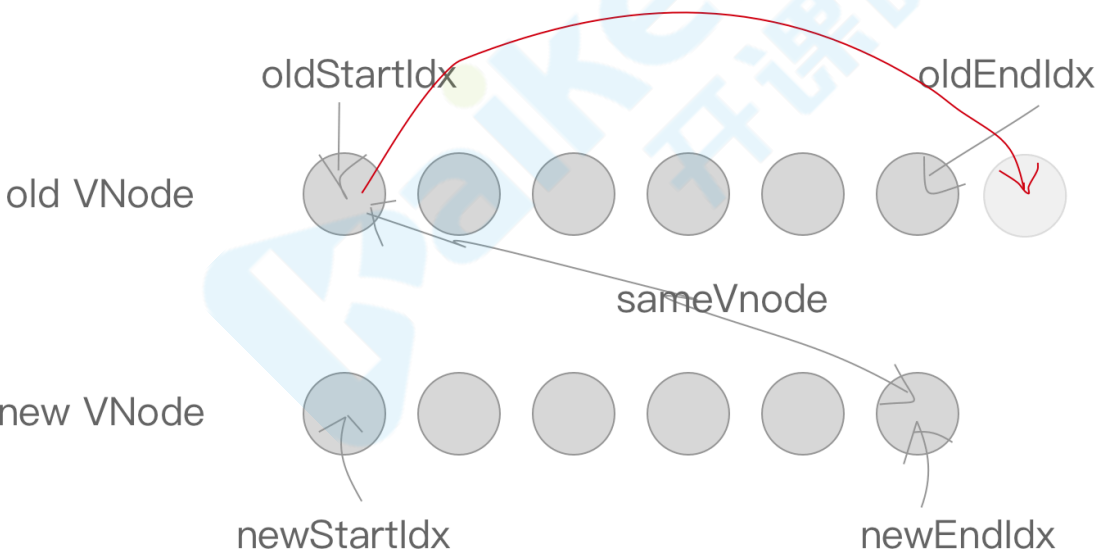
下面是遍历规则：

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

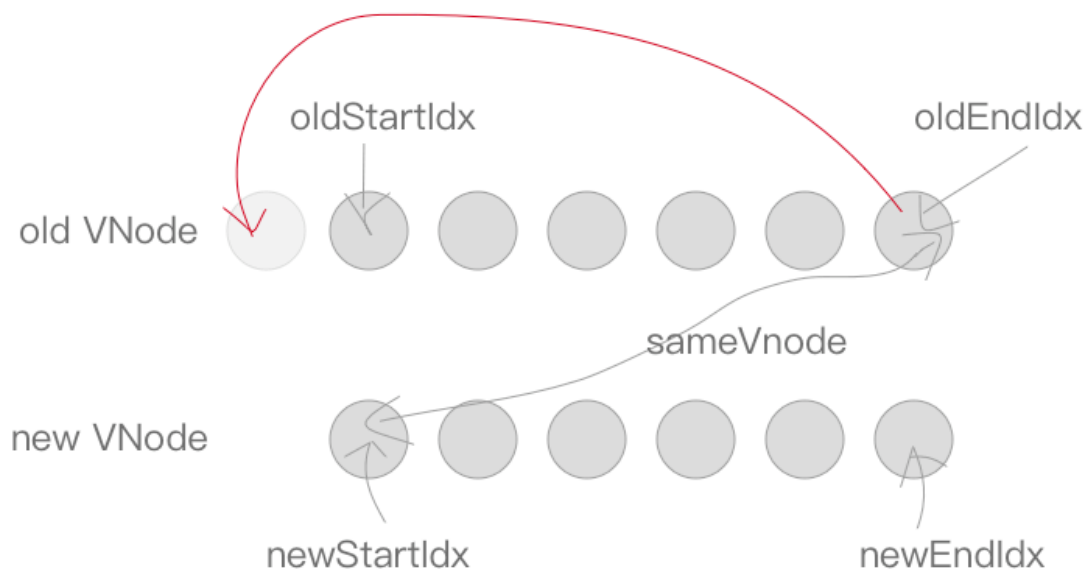
当 oldStartVnode和newStartVnode 或者 oldEndVnode和newEndVnode 满足sameVnode，直接将该VNode节点进行patchVnode即可，不需再遍历就完成了循环。如下图，



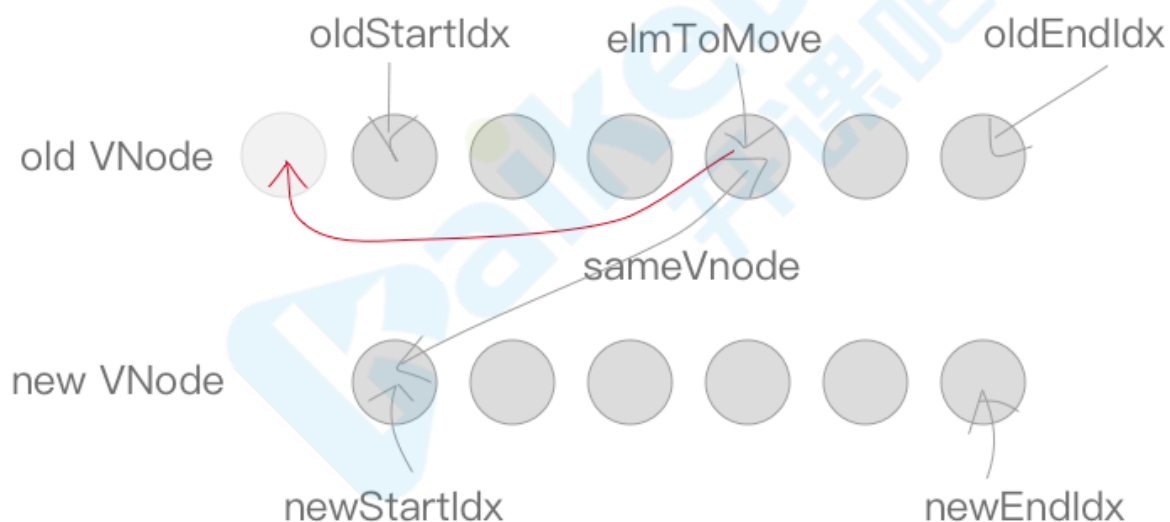
如果oldStartVnode与newEndVnode满足sameVnode。说明oldStartVnode已经跑到了oldEndVnode后面去了，进行patchVnode的同时还需要将真实DOM节点移动到oldEndVnode的后面。



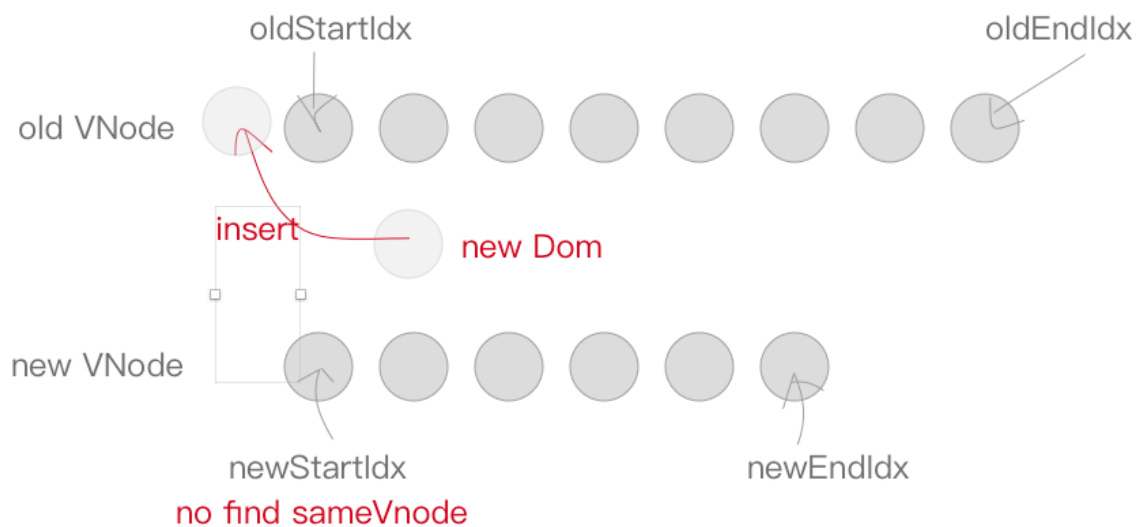
如果oldEndVnode与newStartVnode满足sameVnode，说明oldEndVnode跑到了oldStartVnode的前面，进行patchVnode的同时要将oldEndVnode对应DOM移动到oldStartVnode对应DOM的前面。



如果以上情况均不符合，则在old VNode中找与newStartVnode相同的节点，若存在执行patchVnode，同时将elmToMove移动到oldStartIdx对应的DOM的前面。

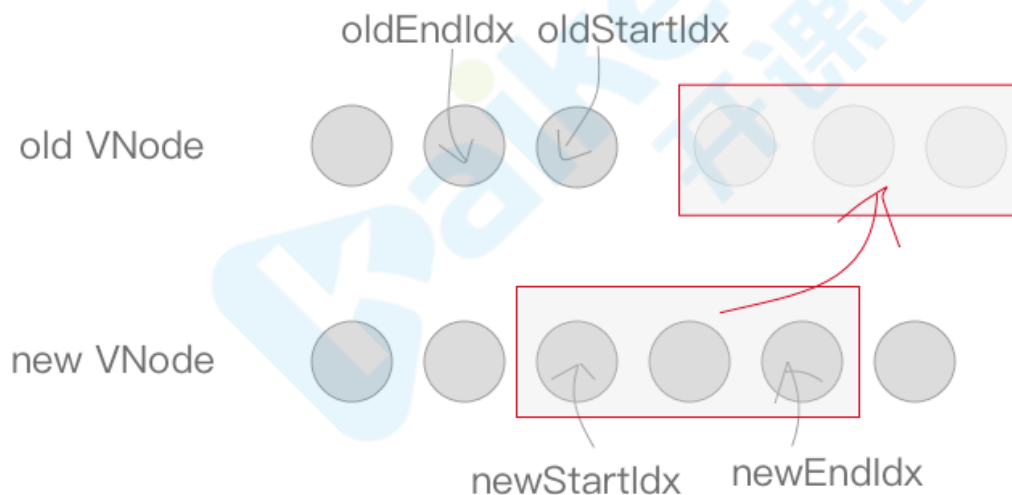


当然也有可能newStartVnode在old VNode节点中找不到一致的sameVnode，这个时候会调用createElm创建一个新的DOM节点。

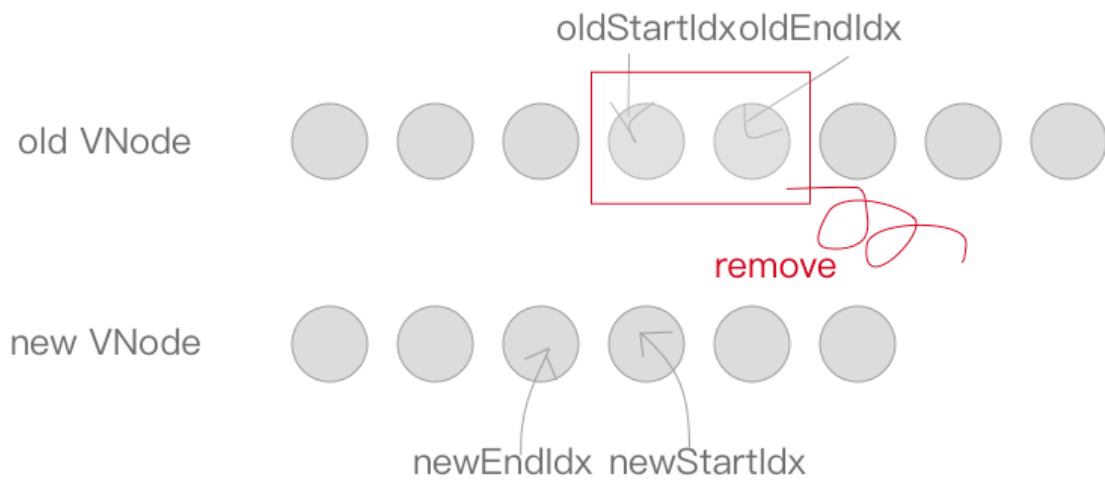


至此循环结束，但是我们还需要处理剩下的节点。

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



作业

将vue异步更新过程绘制为思维导图

思考拓展

- 节点属性是如何更新的
- key是怎么起作用的