JavaScript | 复制代码

```javascript
export function initState (vm: Component) {
  // 初始化组件的watcher列表
  vm._watchers = []
  const opts = vm.$options
  // 初始化props
  if (opts.props) initProps(vm, opts.props)
  // 初始化methods方法
  if (opts.methods) initMethods(vm, opts.methods)
  if (opts.data) {
    // 初始化data
    initData(vm)
  } else {
    observe(vm._data = {}, true /* asRootData */)
  }
  if (opts.computed) initComputed(vm, opts.computed)
  if (opts.watch && opts.watch !== nativeWatch) {
    initWatch(vm, opts.watch)
  }
}
```

我们主要看初始化 `data` 的方法为 `initData` ，它与 `initState` 在同一文件上

JavaScript | 复制代码

```
function initData (vm: Component) {
  let data = vm.$options.data
  // 获取到组件上的data
  data = vm._data = typeof data === 'function'
    ? getData(data, vm)
    : data || {}
  if (!isPlainObject(data)) {
    data = {}
    process.env.NODE_ENV !== 'production' && warn(
      'data functions should return an object:\n' +
      'https://vuejs.org/v2/guide/components.html#data-Must-Be-a-Function'
,
      vm
    )
  }
  // proxy data on instance
  const keys = Object.keys(data)
  const props = vm.$options.props
  const methods = vm.$options.methods
  let i = keys.length
  while (i--) {
    const key = keys[i]
    if (process.env.NODE_ENV !== 'production') {
      // 属性名不能与方法名重复
      if (methods && hasOwn(methods, key)) {
        warn(
          `Method "${key}" has already been defined as a data property.`,
          vm
        )
      }
    }
    // 属性名不能与state名称重复
    if (props && hasOwn(props, key)) {
      process.env.NODE_ENV !== 'production' && warn(
        `The data property "${key}" is already declared as a prop. ` +
        `Use prop default value instead.`,
        vm
      )
    } else if (!isReserved(key)) { // 验证key值的合法性
      // 将_data中的数据挂载到组件vm上,这样就可以通过this.xxx访问到组件上的数据
      proxy(vm, `_data`, key)
    }
  }
  // observe data
  // 响应式监听data是数据的变化
```

```
45      observe(data, true /* asRootData */)
46    }
```

仔细阅读上面的代码，我们可以得到以下结论

- 初始化顺序：`props` 、`methods` 、`data`
- `data` 定义的时候可选择函数形式或者对象形式（组件只能为函数形式）

关于数据响应式在这就不展开详细说明

上文提到挂载方法是调用 `vm.$mount` 方法

源码位置：

JavaScript | 复制代码

```
Vue.prototype.$mount = function (
  el?: string | Element,
  hydrating?: boolean
): Component {
  // 获取或查询元素
  el = el && query(el)

  /* istanbul ignore if */
  // vue 不允许直接挂载到body或页面文档上
  if (el === document.body || el === document.documentElement) {
    process.env.NODE_ENV !== 'production' && warn(
      `Do not mount Vue to <html> or <body> - mount to normal elements instead.`
    )
    return this
  }

  const options = this.$options
  // resolve template/el and convert to render function
  if (!options.render) {
    let template = options.template
    // 存在template模板，解析vue模板文件
    if (template) {
      if (typeof template === 'string') {
        if (template.charAt(0) === '#') {
          template = idToTemplate(template)
          /* istanbul ignore if */
          if (process.env.NODE_ENV !== 'production' && !template) {
            warn(
              `Template element not found or is empty: ${options.template}`,
              this
            )
          }
        }
      } else if (template.nodeType) {
        template = template.innerHTML
      } else {
        if (process.env.NODE_ENV !== 'production') {
          warn('invalid template option:' + template, this)
        }
        return this
      }
    } else if (el) {
      // 通过选择器获取元素内容
```

```
      template = getOuterHTML(el)
    }
    if (template) {
      /* istanbul ignore if */
      if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
        mark('compile')
      }
      /**
       *  1.将temmplate解析ast tree
       *  2.将ast tree转换成render语法字符串
       *  3.生成render方法
       */
      const { render, staticRenderFns } = compileToFunctions(template, {
        outputSourceRange: process.env.NODE_ENV !== 'production',
        shouldDecodeNewlines,
        shouldDecodeNewlinesForHref,
        delimiters: options.delimiters,
        comments: options.comments
      }, this)
      options.render = render
      options.staticRenderFns = staticRenderFns

      /* istanbul ignore if */
      if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
        mark('compile end')
        measure(`vue ${this._name} compile`, 'compile', 'compile end')
      }
    }
  }
  return mount.call(this, el, hydrating)
}
```

阅读上面代码，我们能得到以下结论：

- 不要将根元素放到 `body` 或者 `html` 上
- 可以在对象中定义 `template/render` 或者直接使用 `template` 、 `el` 表示元素选择器
- 最终都会解析成 `render` 函数，调用 `compileToFunctions` ，会将 `template` 解析成 `render` 函数

对 `template` 的解析步骤大致分为以下几步：

- 将 `html` 文档片段解析成 `ast` 描述符
- 将 `ast` 描述符解析成字符串
- 生成 `render` 函数

生成 `render` 函数，挂载到 `vm` 上后，会再次调用 `mount` 方法

源码位置：src\platforms\web\runtime\index.js

JavaScript | 复制代码

```
// public mount method
Vue.prototype.$mount = function (
  el?: string | Element,
  hydrating?: boolean
): Component {
  el = el && inBrowser ? query(el) : undefined
  // 渲染组件
  return mountComponent(this, el, hydrating)
}
```

调用 `mountComponent` 渲染组件

JavaScript | 复制代码

```
export function mountComponent (
  vm: Component,
  el: ?Element,
  hydrating?: boolean
): Component {
  vm.$el = el
  // 如果没有获取解析的render函数，则会抛出警告
  // render是解析模板文件生成的
  if (!vm.$options.render) {
    vm.$options.render = createEmptyVNode
    if (process.env.NODE_ENV !== 'production') {
      /* istanbul ignore if */
      if ((vm.$options.template && vm.$options.template.charAt(0) !== '#') ||
        vm.$options.el || el) {
        warn(
          'You are using the runtime-only build of Vue where the template ' +
          'compiler is not available. Either pre-compile the templates into ' +
          'render functions, or use the compiler-included build.',
          vm
        )
      } else {
        // 没有获取到vue的模板文件
        warn(
          'Failed to mount component: template or render function not defined.',
          vm
        )
      }
    }
  }
  // 执行beforeMount钩子
  callHook(vm, 'beforeMount')

  let updateComponent
  /* istanbul ignore if */
  if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
    updateComponent = () => {
      const name = vm._name
      const id = vm._uid
      const startTag = `vue-perf-start:${id}`
      const endTag = `vue-perf-end:${id}`
```

```
        mark(startTag)
        const vnode = vm._render()
        mark(endTag)
        measure(`vue ${name} render`, startTag, endTag)

        mark(startTag)
        vm._update(vnode, hydrating)
        mark(endTag)
        measure(`vue ${name} patch`, startTag, endTag)
      }
    } else {
      // 定义更新函数
      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)
      }
    }
    // we set this to vm._watcher inside the watcher's constructor
    // since the watcher's initial patch may call $forceUpdate (e.g. inside child
    // component's mounted hook), which relies on vm._watcher being already defined
    // 监听当前组件状态，当有数据变化时，更新组件
    new Watcher(vm, updateComponent, noop, {
      before () {
        if (vm._isMounted && !vm._isDestroyed) {
          // 数据更新引发的组件更新
          callHook(vm, 'beforeUpdate')
        }
      }
    }, true /* isRenderWatcher */)
    hydrating = false

    // manually mounted instance, call mounted on self
    // mounted is called for render-created child components in its inserted hook
    if (vm.$vnode == null) {
      vm._isMounted = true
      callHook(vm, 'mounted')
    }
    return vm
  }
```

阅读上面代码，我们得到以下结论：

- 会触发 `beforeCreate` 钩子

- 定义 `updateComponent` 渲染页面视图的方法
- 监听组件数据，一旦发生变化，触发 `beforeUpdate` 生命钩子

`updateComponent` 方法主要执行在 `vue` 初始化时声明的 `render`， `update` 方法

`render` 的作用主要是生成 `vnode`

源码位置：src\core\instance\render.js

JavaScript | 复制代码

```
// 定义vue 原型上的render方法
Vue.prototype._render = function (): VNode {
    const vm: Component = this
    // render函数来自于组件的option
    const { render, _parentVnode } = vm.$options

    if (_parentVnode) {
        vm.$scopedSlots = normalizeScopedSlots(
            _parentVnode.data.scopedSlots,
            vm.$slots,
            vm.$scopedSlots
        )
    }

    // set parent vnode. this allows render functions to have access
    // to the data on the placeholder node.
    vm.$vnode = _parentVnode
    // render self
    let vnode
    try {
        // There's no need to maintain a stack because all render fns are called
        // separately from one another. Nested component's render fns are called
        // when parent component is patched.
        currentRenderingInstance = vm
        // 调用render方法，自己的独特的render方法， 传入createElement参数，生成vNode
        vnode = render.call(vm._renderProxy, vm.$createElement)
    } catch (e) {
        handleError(e, vm, `render`)
        // return error render result,
        // or previous vnode to prevent render error causing blank component
        /* istanbul ignore else */
        if (process.env.NODE_ENV !== 'production' && vm.$options.renderError) {
            try {
                vnode = vm.$options.renderError.call(vm._renderProxy, vm.$createElement, e)
            } catch (e) {
                handleError(e, vm, `renderError`)
                vnode = vm._vnode
            }
        } else {
```

```
            vnode = vm._vnode
        }
    } finally {
        currentRenderingInstance = null
    }
    // if the returned array contains only a single node, allow it
    if (Array.isArray(vnode) && vnode.length === 1) {
        vnode = vnode[0]
    }
    // return empty vnode in case the render function errored out
    if (!(vnode instanceof VNode)) {
        if (process.env.NODE_ENV !== 'production' && Array.isArray(vnode)) {
            warn(
                'Multiple root nodes returned from render function. Render function ' +
                'should return a single root node.',
                vm
            )
        }
        vnode = createEmptyVNode()
    }
    // set parent
    vnode.parent = _parentVnode
    return vnode
}
```

`_update` 主要功能是调用 `patch`，将 `vnode` 转换为真实 `DOM`，并且更新到页面中

源码位置：src\core\instance\lifecycle.js

JavaScript | 复制代码

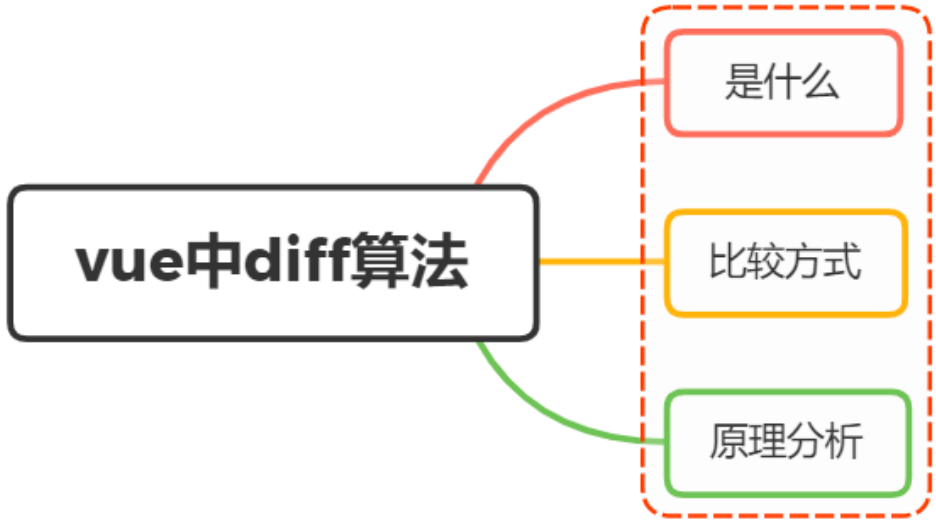
```javascript
Vue.prototype._update = function (vnode: VNode, hydrating?: boolean) {
    const vm: Component = this
    const prevEl = vm.$el
    const prevVnode = vm._vnode
    // 设置当前激活的作用域
    const restoreActiveInstance = setActiveInstance(vm)
    vm._vnode = vnode
    // Vue.prototype.__patch__ is injected in entry points
    // based on the rendering backend used.
    if (!prevVnode) {
      // initial render
      // 执行具体的挂载逻辑
      vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly */)
    } else {
      // updates
      vm.$el = vm.__patch__(prevVnode, vnode)
    }
    restoreActiveInstance()
    // update __vue__ reference
    if (prevEl) {
      prevEl.__vue__ = null
    }
    if (vm.$el) {
      vm.$el.__vue__ = vm
    }
    // if parent is an HOC, update its $el as well
    if (vm.$vnode && vm.$parent && vm.$vnode === vm.$parent._vnode) {
      vm.$parent.$el = vm.$el
    }
    // updated hook is called by the scheduler to ensure that children are
    // updated in a parent's updated hook.
  }
```

12.3. 结论

- `new Vue` 的时候调用会调用 `_init` 方法
  - 定义 `$set` 、 `$get` 、 `$delete` 、 `$watch` 等方法
  - 定义 `$on` 、 `$off` 、 `$emit` 、 `$off` 等事件
  - 定义 `_update` 、 `$forceUpdate` 、 `$destroy` 生命周期
- 调用 `$mount` 进行页面的挂载

- 挂载的时候主要是通过 `mountComponent` 方法
- 定义 `updateComponent` 更新函数
- 执行 `render` 生成虚拟 `DOM`
- `_update` 将虚拟 `DOM` 生成真实 `DOM` 结构，并且渲染到页面中

13. 你了解vue的diff算法吗?



13.1. 是什么

`diff` 算法是一种通过同层的树节点进行比较的高效算法

其有两个特点：

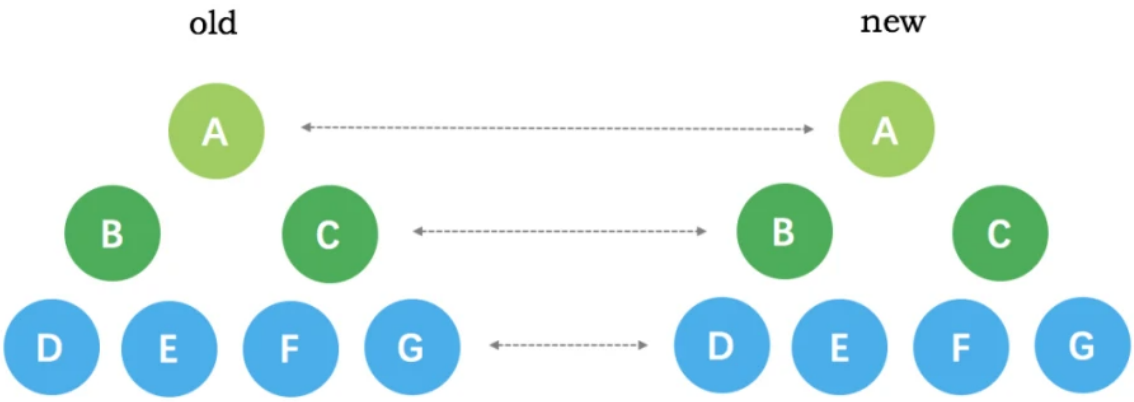
- 比较只会在同层级进行, 不会跨层级比较
- 在diff比较的过程中，循环从两边向中间比较

`diff` 算法在很多场景下都有应用，在 `vue` 中，作用于虚拟 `dom` 渲染成真实 `dom` 的新旧 `VNode` 节点比较

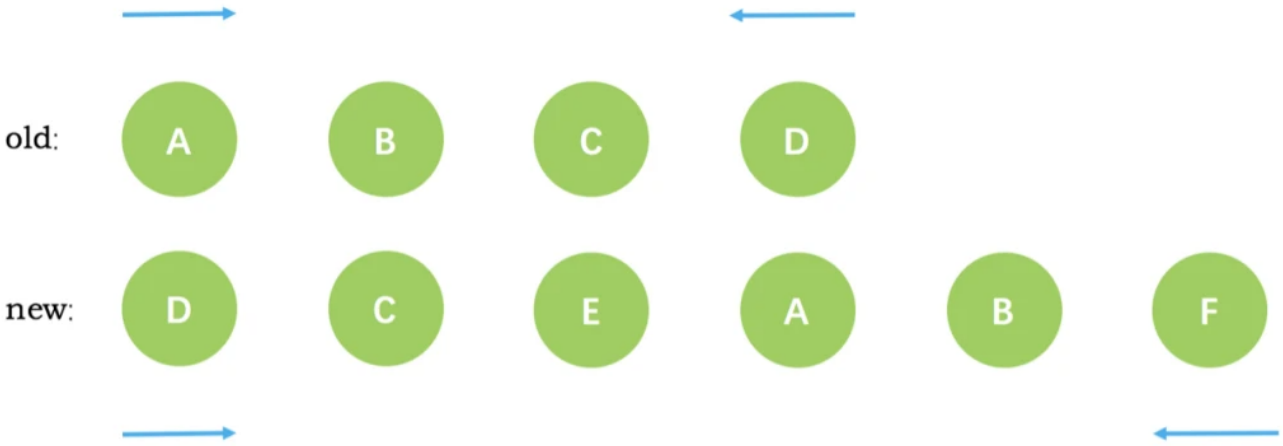
13.2. 比较方式

`diff` 整体策略为：深度优先，同层比较

1. 比较只会在同层级进行, 不会跨层级比较

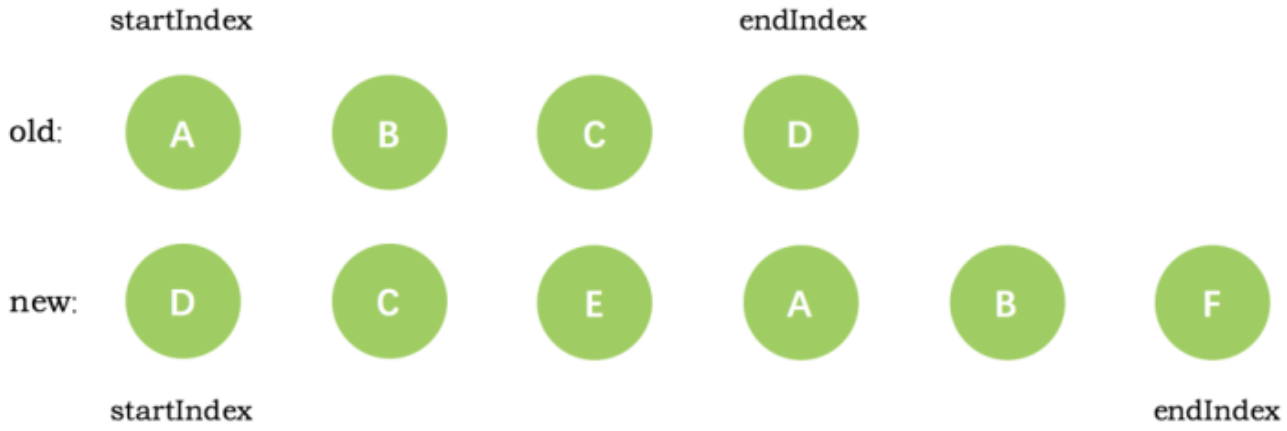


2. 比较的过程中，循环从两边向中间收拢

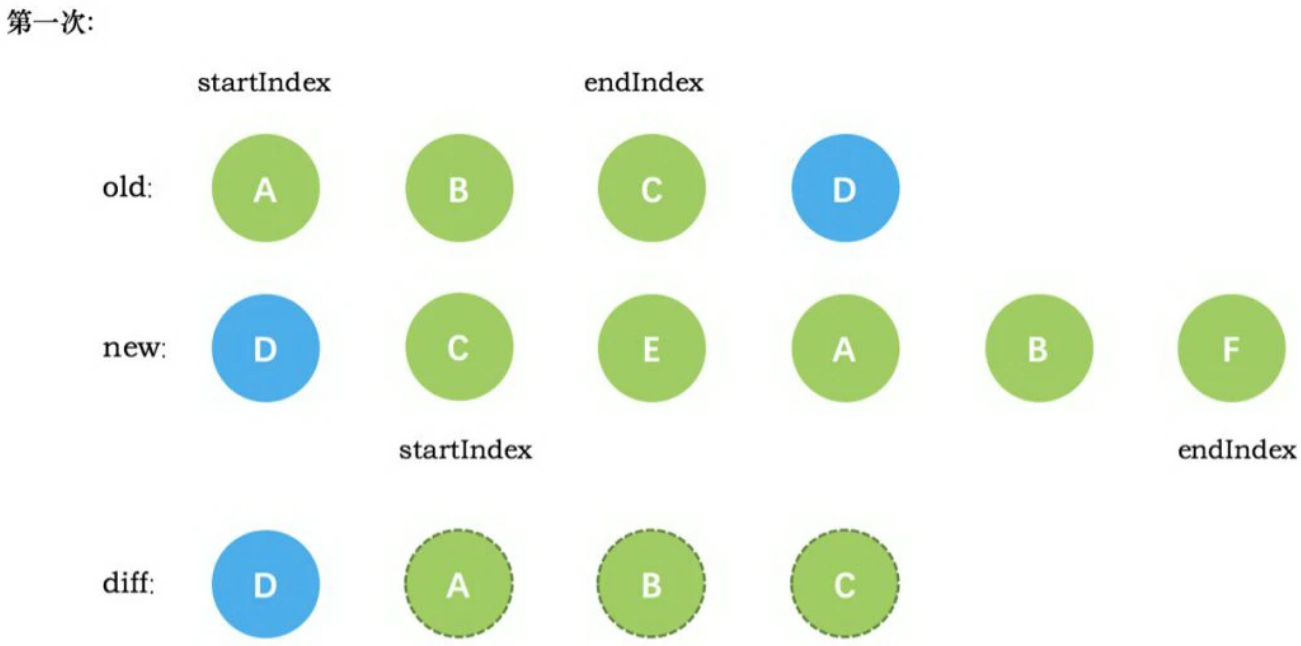


下面举个 `vue` 通过 `diff` 算法更新的例子：

新旧 `VNode` 节点如下图所示：

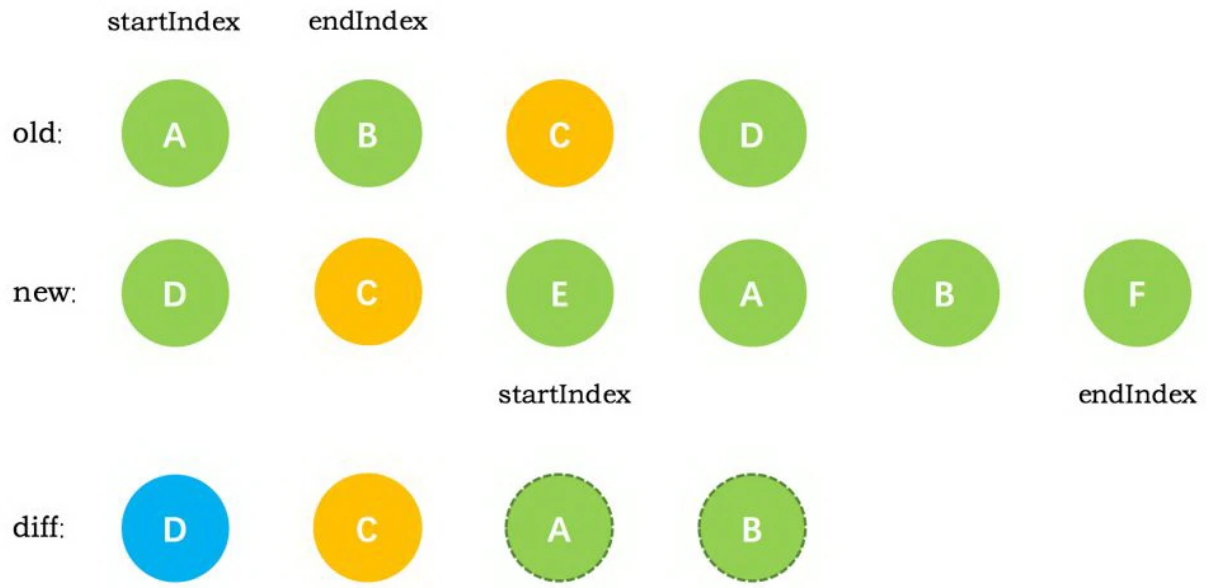


第一次循环后，发现旧节点D与新节点D相同，直接复用旧节点D作为 `diff` 后的第一个真实节点，同时旧节点 `endIndex` 移动到C，新节点的 `startIndex` 移动到了 C



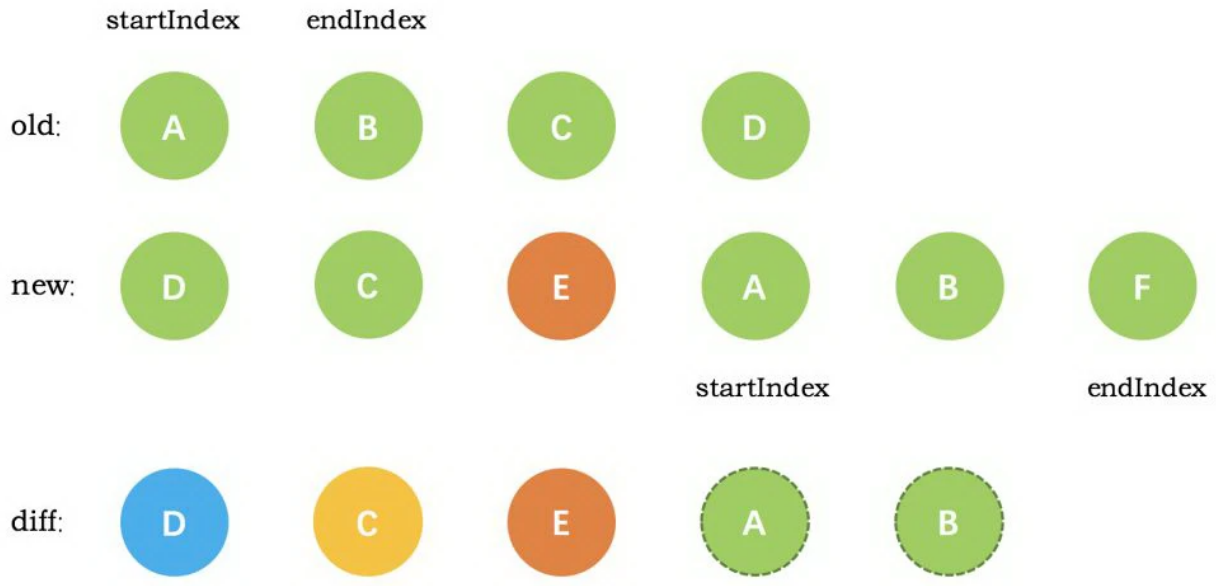
第二次循环后，同样是旧节点的末尾和新节点的开头(都是 C)相同，同理，`diff` 后创建了 C 的真实节点插入到第一次创建的 D 节点后面。同时旧节点的 `endIndex` 移动到了 B，新节点的 `startIndex` 移动到了 E

第二次:



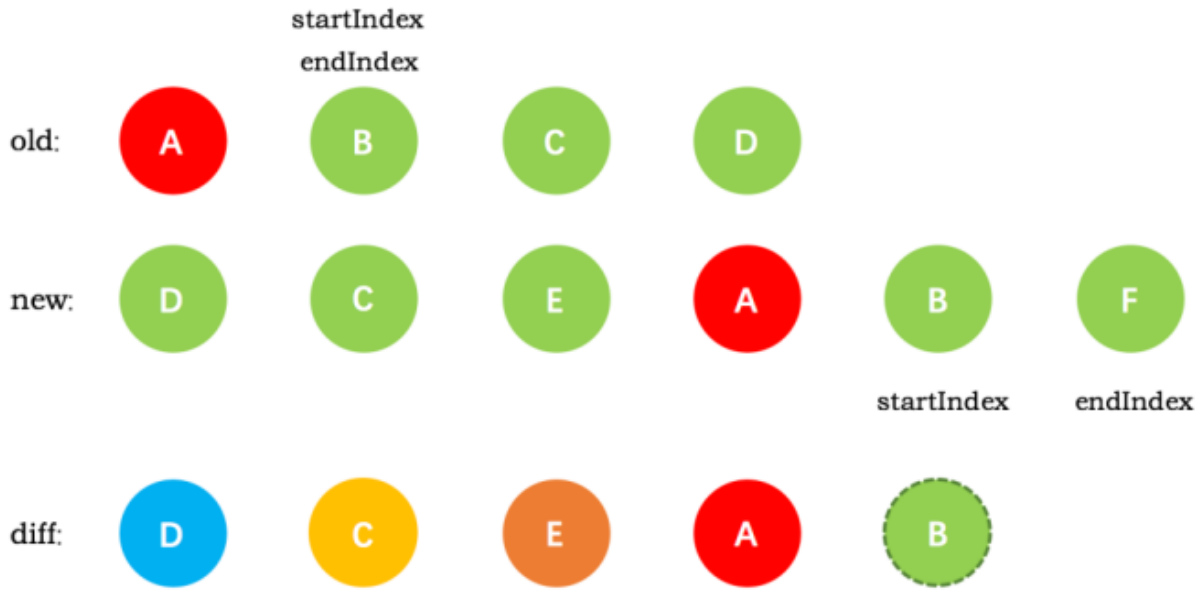
第三次循环中，发现E没有找到，这时候只能直接创建新的真实节点 E，插入到第二次创建的 C 节点之后。同时新节点的 `startIndex` 移动到了 A。旧节点的 `startIndex` 和 `endIndex` 都保持不动

第三次:



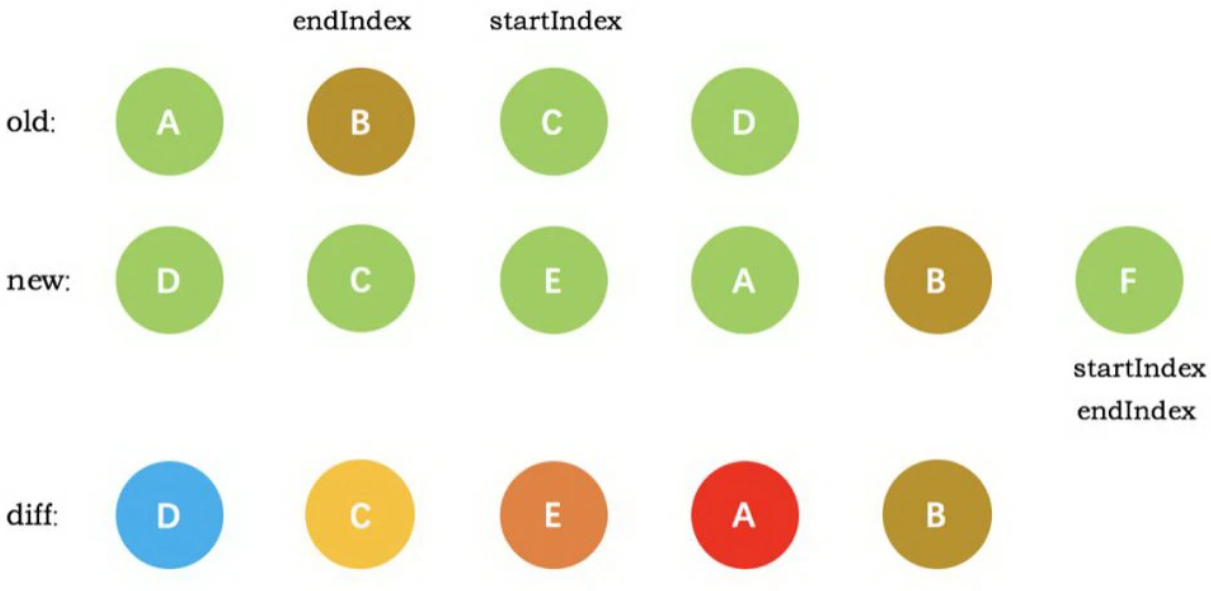
第四次循环中，发现了新旧节点的开头(都是 A)相同，于是 `diff` 后创建了 A 的真实节点，插入到前一次创建的 E 节点后面。同时旧节点的 `startIndex` 移动到了 B，新节点的 `startIndex` 移动到了 B

第四次：



第五次循环中，情形同第四次循环一样，因此 `diff` 后创建了 B 真实节点 插入到前一次创建的 A 节点后面。同时旧节点的 `startIndex` 移动到了 C，新节点的 startIndex 移动到了 F

第五次：



新节点的 `startIndex` 已经大于 `endIndex` 了，需要创建 `newStartIdx` 和 `newEndIdx` 之间的所有节点，也就是节点F，直接创建 F 节点对应的真实节点放到 B 节点后面

第六次:

13.3. 原理分析

当数据发生改变时，`set` 方法会调用 `Dep.notify` 通知所有订阅者 `Watcher`，订阅者就会调用 `patch` 给真实的 `DOM` 打补丁，更新相应的视图

源码位置：src/core/vdom/patch.js

JavaScript | 复制代码

```
function patch(oldVnode, vnode, hydrating, removeOnly) {
    if (isUndef(vnode)) { // 没有新节点，直接执行destory钩子函数
        if (isDef(oldVnode)) invokeDestroyHook(oldVnode)
        return
    }

    let isInitialPatch = false
    const insertedVnodeQueue = []

    if (isUndef(oldVnode)) {
        isInitialPatch = true
        createElm(vnode, insertedVnodeQueue) // 没有旧节点，直接用新节点生成dom元素
    } else {
        const isRealElement = isDef(oldVnode.nodeType)
        if (!isRealElement && sameVnode(oldVnode, vnode)) {
            // 判断旧节点和新节点自身一样，一致执行patchVnode
            patchVnode(oldVnode, vnode, insertedVnodeQueue, null, null, removeOnly)
        } else {
            // 否则直接销毁及旧节点，根据新节点生成dom元素
            if (isRealElement) {

                if (oldVnode.nodeType === 1 && oldVnode.hasAttribute(SSR_ATTR)) {
                    oldVnode.removeAttribute(SSR_ATTR)
                    hydrating = true
                }
                if (isTrue(hydrating)) {
                    if (hydrate(oldVnode, vnode, insertedVnodeQueue)) {
                        invokeInsertHook(vnode, insertedVnodeQueue, true)
                        return oldVnode
                    }
                }
                oldVnode = emptyNodeAt(oldVnode)
            }
            return vnode.elm
        }
    }
}
```

`patch` 函数前两个参数位为 `oldVnode` 和 `Vnode` ，分别代表新的节点和之前的旧节点，主要做了四个判断：

- 没有新节点，直接触发旧节点的 `destory` 钩子

- 没有旧节点，说明是页面刚开始初始化的时候，此时，根本不需要比较了，直接全是新建，所以只调用 `createElm`
- 旧节点和新节点自身一样，通过 `sameVnode` 判断节点是否一样，一样时，直接调用 `patchVnode` 去处理这两个节点
- 旧节点和新节点自身不一样，当两个节点不一样的时候，直接创建新节点，删除旧节点

下面主要讲的是 `patchVnode` 部分

JavaScript | 复制代码

```
function patchVnode (oldVnode, vnode, insertedVnodeQueue, removeOnly) {
    // 如果新旧节点一致，什么都不做
    if (oldVnode === vnode) {
      return
    }

    // 让vnode.el引用到现在的真实dom，当el修改时，vnode.el会同步变化
    const elm = vnode.elm = oldVnode.elm

    // 异步占位符
    if (isTrue(oldVnode.isAsyncPlaceholder)) {
      if (isDef(vnode.asyncFactory.resolved)) {
        hydrate(oldVnode.elm, vnode, insertedVnodeQueue)
      } else {
        vnode.isAsyncPlaceholder = true
      }
      return
    }
    // 如果新旧都是静态节点，并且具有相同的key
    // 当vnode是克隆节点或是v-once指令控制的节点时，只需要把oldVnode.elm和oldVnode.child都复制到vnode上
    // 也不用再有其他操作
    if (isTrue(vnode.isStatic) &&
      isTrue(oldVnode.isStatic) &&
      vnode.key === oldVnode.key &&
      (isTrue(vnode.isCloned) || isTrue(vnode.isOnce))
    ) {
      vnode.componentInstance = oldVnode.componentInstance
      return
    }

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

    const oldCh = oldVnode.children
    const ch = vnode.children
    if (isDef(data) && isPatchable(vnode)) {
      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)
    }
    // 如果vnode不是文本节点或者注释节点
```

```
    if (isUndef(vnode.text)) {
      // 并且都有子节点
      if (isDef(oldCh) && isDef(ch)) {
        // 并且子节点不完全一致，则调用updateChildren
        if (oldCh !== ch) updateChildren(elm, oldCh, ch, insertedVnodeQueue, removeOnly)

        // 如果只有新的vnode有子节点
      } else if (isDef(ch)) {
        if (isDef(oldVnode.text)) nodeOps.setTextContent(elm, '')
        // elm已经引用了老的dom节点，在老的dom节点上添加子节点
        addVnodes(elm, null, ch, 0, ch.length - 1, insertedVnodeQueue)

        // 如果新vnode没有子节点，而vnode有子节点，直接删除老的oldCh
      } else if (isDef(oldCh)) {
        removeVnodes(elm, oldCh, 0, oldCh.length - 1)

        // 如果老节点是文本节点
      } else if (isDef(oldVnode.text)) {
        nodeOps.setTextContent(elm, '')
      }

      // 如果新vnode和老vnode是文本节点或注释节点
      // 但是vnode.text != oldVnode.text时，只需要更新vnode.elm的文本内容就可以
    } else if (oldVnode.text !== vnode.text) {
      nodeOps.setTextContent(elm, vnode.text)
    }
    if (isDef(data)) {
      if (isDef(i = data.hook) && isDef(i = i.postpatch)) i(oldVnode, vnode)
    }
  }
```

`patchVnode` 主要做了几个判断：

- 新节点是否是文本节点，如果是，则直接更新 `dom` 的文本内容为新节点的文本内容
- 新节点和旧节点如果都有子节点，则处理比较更新子节点
- 只有新节点有子节点，旧节点没有，那么不用比较了，所有节点都是全新的，所以直接全部新建就好了，新建是指创建出所有新 `DOM` ，并且添加进父节点
- 只有旧节点有子节点而新节点没有，说明更新后的页面，旧节点全部都不见了，那么要做的，就是把所有的旧节点删除，也就是直接把 `DOM` 删除

子节点不完全一致，则调用 `updateChildren`

JavaScript | 复制代码

```
function updateChildren (parentElm, oldCh, newCh, insertedVnodeQueue, removeOnly) {
    let oldStartIdx = 0 // 旧头索引
    let newStartIdx = 0 // 新头索引
    let oldEndIdx = oldCh.length - 1 // 旧尾索引
    let newEndIdx = newCh.length - 1 // 新尾索引
    let oldStartVnode = oldCh[0] // oldVnode的第一个child
    let oldEndVnode = oldCh[oldEndIdx] // oldVnode的最后一个child
    let newStartVnode = newCh[0] // newVnode的第一个child
    let newEndVnode = newCh[newEndIdx] // newVnode的最后一个child
    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

    // 如果oldStartVnode和oldEndVnode重合，并且新的也都重合了，证明diff完了，循环结束
    while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
      // 如果oldVnode的第一个child不存在
      if (isUndef(oldStartVnode)) {
        // oldStart索引右移
        oldStartVnode = oldCh[++oldStartIdx] // Vnode has been moved left

      // 如果oldVnode的最后一个child不存在
      } else if (isUndef(oldEndVnode)) {
        // oldEnd索引左移
        oldEndVnode = oldCh[--oldEndIdx]

      // oldStartVnode和newStartVnode是同一个节点
      } else if (sameVnode(oldStartVnode, newStartVnode)) {
        // patch oldStartVnode和newStartVnode， 索引左移，继续循环
        patchVnode(oldStartVnode, newStartVnode, insertedVnodeQueue)
        oldStartVnode = oldCh[++oldStartIdx]
        newStartVnode = newCh[++newStartIdx]

      // oldEndVnode和newEndVnode是同一个节点
      } else if (sameVnode(oldEndVnode, newEndVnode)) {
        // patch oldEndVnode和newEndVnode，索引右移，继续循环
        patchVnode(oldEndVnode, newEndVnode, insertedVnodeQueue)
        oldEndVnode = oldCh[--oldEndIdx]
        newEndVnode = newCh[--newEndIdx]

      // oldStartVnode和newEndVnode是同一个节点
```