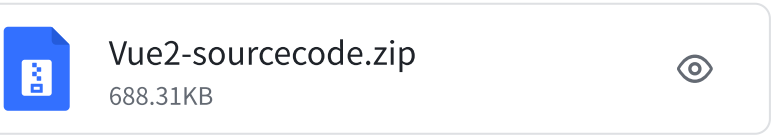
5. Vue2 核心模块源码解析

1. 课程资料



2. 课程目标

💡
- 初级：
  - 掌握 Vue2 数据驱动的思路；
  - 掌握双端比较的使用；
- 中级：
  - 掌握 Vue2 的整体源码的核心执行流程；
  - 能够掌握 Vue2 diff 的使用；
- 高级：
  - 深入理解 Vue2 核心源码，熟练掌握 Vue2 完整的执行逻辑；
  - 完整掌握 Vue2 diff 的执行思路；

3. 课程大纲

1. 前置知识；
2. 数据驱动；
3. Vue2 Diff 算法；

4. 前置知识

4.1 Flow

Flow 是 facebook 出品的 JavaScript 静态类型检查工具。Vue.js 的源码利用了 Flow 做了静态类型检查，也就是文件顶部出现的

```
/* @flow */
```

4.1.1 使用Flow的原因

1. JavaScript 是动态类型语言，它的灵活性有目共睹，但是过于灵活的副作用是很容易就写出非常隐蔽的隐患代码，在编译期甚至看上去都不会报错，但在运行阶段就可能出现各种奇怪的 bug；
2. 类型检查是当前动态类型语言的发展趋势，可以帮助我们在编译期尽早发现（由类型错误引起的）bug，又不影响代码运行（不需要运行时动态检查类型），使编写 JavaScript 具有和编写 Java 等强类型语言相近的体验；
3. 项目越复杂就越需要通过工具的手段来保证项目的维护性和增强代码的可读性。 Vue.js 在做 2.0 重构的时候，在 ES2015 的基础上，除了 ESLint 保证代码风格之外，也引入了 Flow 做静态类型检查。之所以选择 Flow，主要是因为 Babel 和 ESLint 都有对应的 Flow 插件以支持语法，可以完全沿用现有的构建配置，非常小成本的改动就可以拥有静态类型检查的能力；

Vue2.0选用Flow的具体原因，当然Vue3.0还是还是用TS重构了。

4.1.2 Flow 的工作方式

通常类型检查分成 2 种方式：

- 类型推断：通过变量的使用上下文来推断出变量类型，然后根据这些推断来检查类型；
- 类型注释：事先注释好我们期待的类型，Flow 会基于这些注释来判断；

4.1.2.1 类型推断

它不需要任何代码修改即可进行类型检查，最小化开发者的工作量。它不会强制你改变开发习惯，因为它会自动推断出变量的类型。这就是所谓的类型推断，Flow 最重要的特性之一。

通过一个简单例子说明一下：

```
/*@flow*/

function split(str) {
  return str.split(' ')
}

split(11)
```

Flow 检查上述代码后会报错，因为函数 split 期待的参数是字符串，而我们输入了数字；

4.1.2.2 类型注释

如上所述，类型推断是 Flow 最有用的特性之一，不需要编写类型注释就能获取有用的反馈。但在某些特定的场景下，添加类型注释可以提供更好更明确的检查依据。

考虑如下代码：

```
/*@flow*/

function add(x, y){
  return x + y
}

add('Hello', 11)
```

Flow 检查上述代码时检查不出任何错误，因为从语法层面考虑， + 既可以用在字符串上，也可以用在数字上，我们并没有明确指出 `add()` 的参数必须为数字。

在这种情况下，我们可以借助类型注释来指明期望的类型。类型注释是以冒号 : 开头，可以在函数参数，返回值，变量声明中使用。

如果我们在上段代码中添加类型注释，就会变成如下：

```
/*@flow*/

function add(x: number, y: number): number {
  return x + y
}

add('Hello', 11)
```

现在 Flow 就能检查出错误，因为函数参数的期待类型为数字，而我们提供了字符串。

上面的例子是针对函数的类型注释。接下来我们来看看 Flow 能支持的一些常见的类型注释。

数组

```
/*@flow*/

var arr: Array<number> = [1, 2, 3]

arr.push('Hello')
```

数组类型注释的格式是 Array<T>，T 表示数组中每项的数据类型。在上述代码中，arr 是每项均为数字的数组。如果我们给这个数组添加了一个字符串，Flow 能检查出错误。

类和对象

```
/*@flow*/

class Bar {
  x: string;           // x 是字符串
  y: string | number;  // y 可以是字符串或者数字
  z: boolean;

  constructor(x: string, y: string | number) {
    this.x = x
    this.y = y
    this.z = false
  }
}

var bar: Bar = new Bar('hello', 4)

var obj: { a: string, b: number, c: Array<string>, d: Bar } = {
  a: 'hello',
  b: 11,
  c: ['hello', 'world'],
  d: new Bar('hello', 3)
}
```

类的类型注释格式如上，可以对类自身的属性做类型检查，也可以对构造函数的参数做类型检查。这里需要注意的是，属性 y 的类型中间用 | 做间隔，表示 y 的类型即可以是字符串也可以是数字。

对象的注释类型类似于类，需要指定对象属性的类型。

Null

若想任意类型 T 可以为 null 或者 undefined，只需类似如下写成 ?T 的格式即可。

```
/*@flow*/

var foo: ?string = null
```

此时，foo 可以为字符串，也可以为 null。

目前我们只列举了 Flow 的一些常见的类型注释。如果想了解所有类型注释，请移步 Flow 的官方文档。

4.1.3 Flow 在 Vue.js 源码中的应用

有时候我们想引用第三方库，或者自定义一些类型，但 Flow 并不认识，因此检查的时候会报错。为了解决这类问题，Flow 提出了一个 libdef 的概念，可以用来识别这些第三方库或者是自定义类型，而 Vue.js 也利用了这一特性。

在 Vue.js 的主目录下有 .flowconfig 文件， 它是 Flow 的配置文件，文件内容为：

```
flow
├── compiler.js         # 编译相关
├── component.js        # 组件数据结构
├── global-api.js       # Global API 结构
├── modules.js          # 第三方库定义
├── options.js          # 选项相关
├── ssr.js              # 服务端渲染相关
├── vnode.js            # 虚拟 node 相关
```

可以看到，Vue.js 有很多自定义类型的定义，在阅读源码的时候，如果遇到某个类型并想了解它完整的数据结构的时候，可以回来翻阅这些数据结构的定义。

4.2 Vue目录结构设计

```
src
├── compiler        # 编译相关
├── core            # 核心代码
├── platforms       # 不同平台的支持
├── server          # 服务端渲染
├── sfc             # .vue 文件解析
├── shared          # 共享代码
```

4.2.1 compiler

包含 Vue.js 所有编译相关的代码。它包括把模板解析成 ast 语法树，ast 语法树优化，代码生成等功能。

编译的工作可以在构建时做（借助 webpack、vue-loader 等辅助插件）；也可以在运行时做，使用包含构建功能的 Vue.js。显然，编译是一项耗性能的工作，所以更推荐前者——离线编译。

4.2.2 core

包含了 Vue.js 的核心代码，包括内置组件、全局 API 封装，Vue 实例化、观察者、虚拟 DOM、工具函数等等。

这里的代码可谓是 Vue.js 的灵魂，也是我们之后需要重点分析的地方。

4.2.3 platform

Vue.js 是一个跨平台的 类MVVM 框架，它可以跑在 web 上，也可以配合 weex 跑在 native 客户端上。platform 是 Vue.js 的入口，2 个目录代表 2 个主要入口，分别打包成运行在 web 上和 weex 上的 Vue.js。

4.2.4 server

Vue.js 2.0 支持了服务端渲染，所有服务端渲染相关的逻辑都在这个目录下。

这部分代码是跑在服务端的 Node.js，不要和跑在浏览器端的 Vue.js 混为一谈。

服务端渲染主要的工作是把组件渲染为服务器端的 HTML 字符串，将它们直接发送到浏览器，最后将静态标记"混合"为客户端上完全交互的应用程序。

4.2.5 sfc

通常我们开发 Vue.js 都会借助 webpack 构建， 然后通过 .vue 单文件来编写组件。

这个目录下的代码逻辑会把 .vue 文件内容解析成一个 JavaScript 的对象。

4.2.6 shared

Vue.js 会定义一些工具方法，这里定义的工具方法都是会被浏览器端的 Vue.js 和服务端的 Vue.js 所共享的。

4.3 Vue源码构建

Vue.js 源码是基于 Rollup 构建的，它的构建相关配置都在 scripts 目录下。

4.3.1 构建脚本

Vue.js 源码构建的脚本如下：（只说跟build相关）

```
"build": "node scripts/build.js",
 "build:ssr": "npm run build -- web-runtime-cjs,web-server-renderer",
 "build:weex": "npm run build -- weex",
```

这里总共有 3 条命令，作用都是构建 Vue.js，后面 2 条是在第一条命令的基础上，添加一些环境参数。

当在命令行运行 npm run build 的时候，实际上就会执行 node scripts/build.js，接下来我们来看看它实际是怎么构建的。

4.3.2 构建过程

在 scripts/build.js 中：

```
let builds = require('./config').getAllBuilds()

// filter builds via command line arg
if (process.argv[2]) {
  const filters = process.argv[2].split(',')
  builds = builds.filter(b => {
    return filters.some(f => b.output.file.indexOf(f) > -1 ||
b._name.indexOf(f) > -1)
  })
} else {
  // filter out weex builds by default
  builds = builds.filter(b => {
    return b.output.file.indexOf('weex') === -1
  })
}

build(builds)
```

这段代码逻辑非常简单，先从配置文件读取配置，再通过命令行参数对构建配置做过滤，这样就可以构建出不同用途的 Vue.js 了。接下来我们看一下配置文件，在 `scripts/config.js` 中：

```
const builds = {
  // Runtime only (CommonJS). Used by bundlers e.g. Webpack & Browserify
  'web-runtime-cjs': {
    entry: resolve('web/entry-runtime.js'),
    dest: resolve('dist/vue.runtime.common.js'),
    format: 'cjs',
    banner
  },
  // Runtime+compiler CommonJS build (CommonJS)
  'web-full-cjs': {
    entry: resolve('web/entry-runtime-with-compiler.js'),
    dest: resolve('dist/vue.common.js'),
    format: 'cjs',
    alias: { he: './entity-decoder' },
    banner
  },
```

```
  // Runtime only (ES Modules). Used by bundlers that support ES Modules,
  // e.g. Rollup & Webpack 2
  'web-runtime-esm': {
    entry: resolve('web/entry-runtime.js'),
    dest: resolve('dist/vue.runtime.esm.js'),
    format: 'es',
    banner
  },
  // Runtime+compiler CommonJS build (ES Modules)
  'web-full-esm': {
    entry: resolve('web/entry-runtime-with-compiler.js'),
    dest: resolve('dist/vue.esm.js'),
    format: 'es',
    alias: { he: './entity-decoder' },
    banner
  },
  // runtime-only build (Browser)
  'web-runtime-dev': {
    entry: resolve('web/entry-runtime.js'),
    dest: resolve('dist/vue.runtime.js'),
    format: 'umd',
    env: 'development',
    banner
  },
  // runtime-only production build (Browser)
  'web-runtime-prod': {
    entry: resolve('web/entry-runtime.js'),
    dest: resolve('dist/vue.runtime.min.js'),
    format: 'umd',
    env: 'production',
    banner
  },
  // Runtime+compiler development build (Browser)
  'web-full-dev': {
    entry: resolve('web/entry-runtime-with-compiler.js'),
    dest: resolve('dist/vue.js'),
    format: 'umd',
    env: 'development',
    alias: { he: './entity-decoder' },
    banner
  },
  // Runtime+compiler production build  (Browser)
  'web-full-prod': {
    entry: resolve('web/entry-runtime-with-compiler.js'),
    dest: resolve('dist/vue.min.js'),
    format: 'umd',
    env: 'production',
```

```
    alias: { he: './entity-decoder' },
    banner
  },
  // ...
}
```

这里列举了一些 Vue.js 构建的配置，对于单个配置，它是遵循 Rollup 的构建规则的。其中 entry 属性表示构建的入口 JS 文件地址，dest 属性表示构建后的 JS 文件地址。format 属性表示构建的格式，cjs 表示构建出来的文件遵循 CommonJS 规范，es 表示构建出来的文件遵循 ES Module 规范。 umd 表示构建出来的文件遵循 UMD 规范。

以 `web-runtime-cjs-dev` 配置为例，它的 entry 是 `resolve('web/entry-runtime.js')` ，先来看一下 resolve 函数的定义。

源码目录：scripts/config.js

```
const aliases = require('./alias')
const resolve = p => {
  const base = p.split('/')[0]
  if (aliases[base]) {
    return path.resolve(aliases[base], p.slice(base.length + 1))
  } else {
    return path.resolve(__dirname, '../', p)
  }
}
```

这里的 resolve 函数实现非常简单，它先把 resolve 函数传入的参数 p 通过 / 做了分割成数组，然后取数组第一个元素设置为 base。在我们这个例子中，参数 p 是 `web/entry-runtime.js` ，那么 base 则为 web。base 并不是实际的路径，它的真实路径借助了别名的配置，我们来看一下别名配置的代码，在 scripts/alias 中：

```
const path = require('path')

const resolve = p => path.resolve(__dirname, '../', p)

module.exports = {
  vue: resolve('src/platforms/web/entry-runtime-with-compiler'),
  compiler: resolve('src/compiler'),
  core: resolve('src/core'),
  shared: resolve('src/shared'),
  web: resolve('src/platforms/web'),
```

```
    weex: resolve('src/platforms/weex'),
    server: resolve('src/server'),
    sfc: resolve('src/sfc')
  }
```

很显然，这里 web 对应的真实的路径是 path.resolve(__dirname, '../src/platforms/web')，这个路径就找到了 Vue.js 源码的 web 目录。然后 resolve 函数通过 `path.resolve(aliases[base], p.slice(base.length + 1))` 找到了最终路径，它就是 Vue.js 源码 web 目录下的 entry-runtime.js。因此， `web-runtime-cjs-dev` 配置对应的入口文件就找到了。

它经过 Rollup 的构建打包后，最终会在 dist 目录下生成 `dist/vue.runtime.common.dev.js` 。

4.3.3 Runtime Only VS Runtime + Compiler

通常我们利用 vue-cli 去初始化我们的 Vue.js 项目的时候会询问我们用 `Runtime Only` 版本的还是 `Runtime + Compiler` 版本。下面我们来对比这两个版本。

- Runtime Only

我们在使用 Runtime Only 版本的 Vue.js 的时候，通常需要借助如 webpack 的 vue-loader 工具把 .vue 文件编译成 JavaScript，因为是在编译阶段做的，所以它只包含运行时的 Vue.js 代码，因此代码体积也会更轻量；

- Runtime + Compiler

我们如果没有对代码做预编译，但又使用了 Vue 的 template 属性并传入一个字符串，则需要在客户端编译模板，如下所示：

```
// 需要编译器的版本
new Vue({
  template: '<div>{{ hi }}</div>'
})

// 这种情况不需要
new Vue({
  render (h) {
    return h('div', this.hi)
  }
})
```

在 Vue.js 2.0 中，最终渲染都是通过 render 函数，如果写 template 属性，则需要编译成 render 函数，那么这个编译过程会发生运行时，所以需要带有编译器的版本。

很显然，这个编译过程对性能会有一定损耗，所以通常我们更推荐使用 Runtime-Only 的 Vue.js。

4.4 Vue入口

在 web场景下，我们来分析 Runtime + Compiler 构建出来的 Vue.js，它的入口是 `src/platforms/web/entry-runtime-with-compiler.js`：

```
/* @flow */

import config from 'core/config'
import { warn, cached } from 'core/util/index'
import { mark, measure } from 'core/util/perf'

import Vue from './runtime/index'
import { query } from './util/index'
import { compileToFunctions } from './compiler/index'
import { shouldDecodeNewlines, shouldDecodeNewlinesForHref } from
'./util/compat'

const idToTemplate = cached(id => {
  const el = query(id)
  return el && el.innerHTML
})

const mount = Vue.prototype.$mount
Vue.prototype.$mount = function (
  el?: string | Element,
  hydrating?: boolean
): Component {
  el = el && query(el)

  /* istanbul ignore if */
  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
    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) {
      template = getOuterHTML(el)
    }
    if (template) {
      /* istanbul ignore if */
      if (process.env.NODE_ENV !== 'production' && config.performance && mark)
{
        mark('compile')
      }

      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)
```

```
}

/**
 * Get outerHTML of elements, taking care
 * of SVG elements in IE as well.
 */
function getOuterHTML (el: Element): string {
  if (el.outerHTML) {
    return el.outerHTML
  } else {
    const container = document.createElement('div')
    container.appendChild(el.cloneNode(true))
    return container.innerHTML
  }
}

Vue.compile = compileToFunctions

export default Vue
```

当我们的代码执行 import Vue from 'vue' 的时候，就是从这个入口执行代码来初始化 Vue，来源为：`import Vue from './runtime/index'`，入口在 `src/platforms/web/runtime/index.js`

```
/* @flow */

import Vue from 'core/index'
import config from 'core/config'
import { extend, noop } from 'shared/util'
import { mountComponent } from 'core/instance/lifecycle'
import { devtools, inBrowser } from 'core/util/index'

import {
  query,
  mustUseProp,
  isReservedTag,
  isReservedAttr,
  getTagNamespace,
  isUnknownElement
} from 'web/util/index'

import { patch } from './patch'
import platformDirectives from './directives/index'
```

```
import platformComponents from './components/index'

// install platform specific utils
Vue.config.mustUseProp = mustUseProp
Vue.config.isReservedTag = isReservedTag
Vue.config.isReservedAttr = isReservedAttr
Vue.config.getTagNamespace = getTagNamespace
Vue.config.isUnknownElement = isUnknownElement

// install platform runtime directives & components
extend(Vue.options.directives, platformDirectives)
extend(Vue.options.components, platformComponents)

// install platform patch function
Vue.prototype.__patch__ = inBrowser ? patch : noop

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

// devtools global hook
/* istanbul ignore next */
if (inBrowser) {
  setTimeout(() => {
    if (config.devtools) {
      if (devtools) {
        devtools.emit('init', Vue)
      } else if (
        process.env.NODE_ENV !== 'production' &&
        process.env.NODE_ENV !== 'test'
      ) {
        console[console.info ? 'info' : 'log'](
          'Download the Vue Devtools extension for a better development
experience:\n' +
          'https://github.com/vuejs/vue-devtools'
        )
      }
    }
    if (process.env.NODE_ENV !== 'production' &&
      process.env.NODE_ENV !== 'test' &&
      config.productionTip !== false &&
      typeof console !== 'undefined'
```

```
    ) {
      console[console.info ? 'info' : 'log'](
        `You are running Vue in development mode.\n` +
        `Make sure to turn on production mode when deploying for production.\n`
 +
        `See more tips at https://vuejs.org/guide/deployment.html`
      )
    }
  }, 0)
}

export default Vue
```

里关键的代码是 `import Vue from 'core/index'`，之后的逻辑都是对 Vue 这个对象做一些扩展，可以先不用看，我们来看一下真正初始化 Vue 的地方，在 `src/core/index.js` 中：

```
import Vue from './instance/index'
import { initGlobalAPI } from './global-api/index'
import { isServerRendering } from 'core/util/env'
import { FunctionalRenderContext } from 'core/vdom/create-functional-component'

initGlobalAPI(Vue)

Object.defineProperty(Vue.prototype, '$isServer', {
  get: isServerRendering
})

Object.defineProperty(Vue.prototype, '$ssrContext', {
  get () {
    /* istanbul ignore next */
    return this.$vnode && this.$vnode.ssrContext
  }
})

// expose FunctionalRenderContext for ssr runtime helper installation
Object.defineProperty(Vue, 'FunctionalRenderContext', {
  value: FunctionalRenderContext
})

Vue.version = '__VERSION__'

export default Vue
```

这里有 2 处关键的代码，`import Vue from './instance/index'` 和 `initGlobalAPI(Vue)`，初始化全局 Vue API（我们稍后介绍），我们先来看第一部分，在 `src/core/instance/index.js` 中：

4.4.1 Vue定义

```
import { initMixin } from './init'
import { stateMixin } from './state'
import { renderMixin } from './render'
import { eventsMixin } from './events'
import { lifecycleMixin } from './lifecycle'
import { warn } from '../util/index'

function Vue (options) {
  if (process.env.NODE_ENV !== 'production' &&
    !(this instanceof Vue)
  ) {
    warn('Vue is a constructor and should be called with the `new` keyword')
  }
  this._init(options)
}

initMixin(Vue)
stateMixin(Vue)
eventsMixin(Vue)
lifecycleMixin(Vue)
renderMixin(Vue)

export default Vue
```

在这里，我们会发现，Vue实际上就是一个用 Function 实现的类，这也是为什么我们只能通过 `new Vue` 去实例化它的原因。

Q：为何 Vue 不用 ES6 的 Class 去实现呢?

A：我们往后看这里有很多 xxxMixin 的函数调用，并把 Vue 当参数传入，它们的功能都是给 Vue 的 prototype 上扩展一些方法，Vue 按功能把这些扩展分散到多个模块中去实现，而不是在一个模块里实现所有，这种方式是用 Class 难以实现的。这么做的好处是非常方便代码的维护和管理。

4.4.2 initGlobalAPI

Vue.js 在整个初始化过程中，除了给它的原型 prototype 上扩展方法，还会给 Vue 这个对象本身扩展全局的静态方法，它的定义在 `src/core/global-api/index.js` 中：

```
export function initGlobalAPI (Vue: GlobalAPI) {
  // config
  const configDef = {}
  configDef.get = () => config
  if (process.env.NODE_ENV !== 'production') {
    configDef.set = () => {
      warn(
        'Do not replace the Vue.config object, set individual fields instead.'
      )
    }
  }
  Object.defineProperty(Vue, 'config', configDef)

  // exposed util methods.
  // NOTE: these are not considered part of the public API - avoid relying on
  // them unless you are aware of the risk.
  Vue.util = {
    warn,
    extend,
    mergeOptions,
    defineReactive
  }

  Vue.set = set
  Vue.delete = del
  Vue.nextTick = nextTick

  Vue.options = Object.create(null)
  ASSET_TYPES.forEach(type => {
    Vue.options[type + 's'] = Object.create(null)
  })

  // this is used to identify the "base" constructor to extend all plain-object
  // components with in Weex's multi-instance scenarios.
  Vue.options._base = Vue

  extend(Vue.options.components, builtInComponents)

  initUse(Vue)
  initMixin(Vue)
  initExtend(Vue)
  initAssetRegisters(Vue)
}
```

这里就是在 Vue 上扩展的一些全局方法的定义，Vue 官网中关于全局 API 都可以在这里找到，有一点要注意的是，`Vue.util` 暴露的方法最好不要依赖，因为它可能经常会发生变化，是不稳定的。具体

的全局API后面详细讲解；

5. 数据驱动

Vue.js 一个核心思想是数据驱动。所谓数据驱动，是指视图是由数据驱动生成的，我们对视图的修改，不会直接操作 DOM，而是通过修改数据。它相比我们传统的前端开发，如使用 jQuery 等前端库直接修改 DOM，大大简化了代码量。特别是当交互复杂的时候，只关心数据的修改会让代码的逻辑变的非常清晰，因为 DOM 变成了数据的映射，我们所有的逻辑都是对数据的修改，而不用碰触 DOM，这样的代码非常利于维护。

在 Vue.js 中我们可以采用简洁的模板语法来声明式的将数据渲染为 DOM：

```
<div id="app">
  {{ message }}
</div>

var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

最终它会在页面上渲染出 `Hello Vue` 。接下来，我们会从源码角度来分析 Vue 是如何实现的。在这里，先弄清楚模板和数据如何渲染成最终的 DOM。

5.1 new Vue 时发生了什么

从入口代码开始分析，我们先来分析 new Vue 背后发生了哪些事情。我们都知道，new 关键字在 Javascript 语言中代表实例化是一个对象，而 Vue 实际上是一个类，类在 Javascript 中是用 Function 来实现的，来看一下源码，在 `src/core/instance/index.js` 中。

```
function Vue (options) {
  if (process.env.NODE_ENV !== 'production' &&
    !(this instanceof Vue)
  ) {
    warn('Vue is a constructor and should be called with the `new` keyword')
  }
  this._init(options)
}
```

可以看到 Vue 只能通过 new 关键字初始化，然后会调用 this._init 方法， 该方法在 `initMixin(Vue)` 时注入到prototype上，在 `src/core/instance/init.js` 中定义。

```
Vue.prototype._init = function (options?: Object) {
  const vm: Component = this
  // a uid
  vm._uid = uid++

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

  // a flag to avoid this being observed
  vm._isVue = true
  // merge options
  if (options && options._isComponent) {
    // optimize internal component instantiation
    // since dynamic options merging is pretty slow, and none of the
    // internal component options needs special treatment.
    initInternalComponent(vm, options)
  } else {
    vm.$options = mergeOptions(
      resolveConstructorOptions(vm.constructor),
      options || {},
      vm
    )
  }
  /* istanbul ignore else */
  if (process.env.NODE_ENV !== 'production') {
    initProxy(vm)
  } else {
    vm._renderProxy = vm
  }
  // expose real self
  vm._self = vm
  initLifecycle(vm)
  initEvents(vm)
  initRender(vm)
  callHook(vm, 'beforeCreate')
  initInjections(vm) // resolve injections before data/props
  initState(vm)
  initProvide(vm) // resolve provide after data/props
```

```
    callHook(vm, 'created')

    /* istanbul ignore if */
    if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
      vm._name = formatComponentName(vm, false)
      mark(endTag)
      measure(`vue ${vm._name} init`, startTag, endTag)
    }

    if (vm.$options.el) {
      vm.$mount(vm.$options.el)
    }
  }
```

Vue 初始化主要就干了几件事情，合并配置，初始化生命周期，初始化事件中心，初始化渲染，初始化 data、props、computed、watcher 等等。

Vue 的初始化逻辑写的非常清楚，把不同的功能逻辑拆成一些单独的函数执行，让主线逻辑一目了然，由于我们这一章的目标是弄清楚模板和数据如何渲染成最终的 DOM，所以各种初始化逻辑我们先不看。在初始化的最后，检测到如果有 el 属性，则调用 vm.$mount 方法挂载 vm，挂载的目标就是把模板渲染成最终的 DOM，那么接下来我们来分析 Vue 的挂载过程。

5.2 Vue实例挂载的实现

Vue 中我们是通过 $mount 实例方法去挂载 vm 的，$mount 方法在多个文件中都有定义，如 `src/platform/web/entry-runtime-with-compiler.js`、`src/platform/web/runtime/index.js`、`src/platform/weex/runtime/index.js`。因为 $mount 这个方法的实现是和平台、构建方式都相关的。接下来我们重点分析带 compiler 版本的 $mount 实现，因为抛开 webpack 的 vue-loader，我们在纯前端浏览器环境分析 Vue 的工作原理，有助于我们对原理理解的深入。

先来看一下 `src/platform/web/entry-runtime-with-compiler.js` 文件中定义：

```
  const mount = Vue.prototype.$mount
  Vue.prototype.$mount = function (
    el?: string | Element,
    hydrating?: boolean
  ): Component {
    el = el && query(el)

    /* istanbul ignore if */
    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
      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) {
        template = getOuterHTML(el)
      }
      if (template) {
        /* istanbul ignore if */
        if (process.env.NODE_ENV !== 'production' && config.performance && mark)
{
          mark('compile')
        }

        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)
}
```

这段代码首先缓存了原型上的 `$mount` 方法，再重新定义该方法

1. 它对 el 做了限制，Vue 不能挂载在 body、html 这样的根节点上；
2. 如果没有定义 render 方法，则会把 el 或者 template 字符串转换成 render 方法。在 Vue 2.0 版本中，所有 Vue 的组件的渲染最终都需要 render 方法，无论我们是用单文件 .vue 方式开发组件，还是写了 el 或者 template 属性，最终都会转换成 render 方法；
3. 根据生成的template函数，会执行在线编译的过程，是调用 `compileToFunctions` 方法实现的，在后续的编译过程中讲解。最后，调用原先原型上的 $mount 方法挂载；

原先原型上的 $mount 方法在 `src/platform/web/runtime/index.js` 中定义，之所以这么设计完全是为了复用，因为它是可以被 `runtime only` 版本的 Vue 直接使用的。

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

$mount 方法支持传入 2 个参数，第一个是 el，它表示挂载的元素，可以是字符串，也可以是 DOM 对象，如果是字符串在浏览器环境下会调用 query 方法转换成 DOM 对象的。第二个参数是和服务端渲染相关，在浏览器环境下我们不需要传第二个参数。

$mount 方法实际上会去调用 mountComponent 方法，这个方法定义在 `src/core/instance/lifecycle.js` 文件中：

```
export function mountComponent (
  vm: Component,
  el: ?Element,
  hydrating?: boolean
): Component {
  vm.$el = el
  if (!vm.$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 {
        warn(
          'Failed to mount component: template or render function not
defined.',
          vm
        )
      }
    }
  }
  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 {
        updateComponent = () => {
          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
      new Watcher(vm, updateComponent, noop, {
        before () {
          if (vm._isMounted) {
            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
    }
```

这里面核心就是先实例化一个渲染Watcher，在它的回调函数中会调用 `updateComponent` 方法，在此方法中调用 vm._render 方法先生成虚拟 Node，最终调用 vm._update 更新 DOM。

Watcher 在这里起到两个作用：（后面会详细讲解）

1. 初始化的时候会执行回调函数；
2. 当 vm 实例中的监测的数据发生变化的时候执行回调函数；

函数最后判断当根节点vm.$vnode为null时，执行mount初始化；接下来详细讲_render（生成VNode）和_update（更新DOM）

5.3 render

Vue 的 `_render` 方法是实例的一个私有方法，它用来把实例渲染成一个虚拟 Node。它的定义在 `src/core/instance/render.js` 文件中：

在 `src/core/instance/init.js` 中，执行 `renderMixin` ，注入到到Vue.prototype上：

```
Vue.prototype._render = function (): VNode {
  const vm: Component = this
  const { render, _parentVnode } = vm.$options

  // reset _rendered flag on slots for duplicate slot check
  if (process.env.NODE_ENV !== 'production') {
    for (const key in vm.$slots) {
      // $flow-disable-line
      vm.$slots[key]._rendered = false
    }
  }

  if (_parentVnode) {
    vm.$scopedSlots = _parentVnode.data.scopedSlots || emptyObject
  }

  // set parent vnode. this allows render functions to have access
  // to the data on the placeholder node.
  vm.$vnode = _parentVnode
  // render self
  let vnode
  try {
    vnode = render.call(vm._renderProxy, vm.$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') {
      if (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
      }
    } else {
      vnode = vm._vnode
    }
  }
  // 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
    }
```

最关键的是 render 方法的调用，在之前的 mounted 方法的实现中，会把 template 编译成 render 方法，具体的编译方法在后面讲

在_render 函数中的 render 方法的调用中：

```
vnode = render.call(vm._renderProxy, vm.$createElement)
```

render 函数中的 createElement 方法就是 vm.$createElement 方法，其中vm.$createElement的方法是在initRender中定义的，其中vm.$createElement调用了createElement，另一个方法也调用了，这个方法是被模板编译成的 render 函数使用，而 vm.$createElement 是用户手写 render 方法使用的， 这俩个方法支持的参数相同，并且内部都调用了 createElement 方法

```
 vm._c = (a, b, c, d) => createElement(vm, a, b, c, d, false)
vm.$createElement = (a, b, c, d) => createElement(vm, a, b, c, d, true)
```

此方法，也是Vue中提到的render的第一个参数 `createElement`

```
render: function (createElement) {
  return createElement('div', {
     attrs: {
        id: 'app'
      },
  }, this.message)
}
```

5.4 VDOM

在了解createElement原理之前，先了解下Virtual DOM是什么。

在浏览器中，一个最简单的div所包含的元素也是很多的，因为浏览器的标准就把 DOM 设计的非常复杂。当我们频繁的去做 DOM 更新，会产生一定的性能问题。

```
const div = document.createElement('div')
let str = '';
for (const key in div) str += key + ' '
```

而 Virtual DOM 就是用一个原生的 JS 对象去描述一个 DOM 节点，所以它比创建一个 DOM 的代价要小很多。在 Vue.js 中，Virtual DOM 是用 VNode 这么一个 Class 去描述，它是定义在 `src/core/vdom/vnode.js` 中的。

```
export default class VNode {
  tag: string | void;
  data: VNodeData | void;
  children: ?Array<VNode>;
  text: string | void;
  elm: Node | void;
  ns: string | void;
  context: Component | void; // rendered in this component's scope
  key: string | number | void;
  componentOptions: VNodeComponentOptions | void;
  componentInstance: Component | void; // component instance
  parent: VNode | void; // component placeholder node

  // strictly internal
  raw: boolean; // contains raw HTML? (server only)
  isStatic: boolean; // hoisted static node
  isRootInsert: boolean; // necessary for enter transition check
  isComment: boolean; // empty comment placeholder?
  isCloned: boolean; // is a cloned node?
  isOnce: boolean; // is a v-once node?
  asyncFactory: Function | void; // async component factory function
  asyncMeta: Object | void;
  isAsyncPlaceholder: boolean;
  ssrContext: Object | void;
  fnContext: Component | void; // real context vm for functional nodes
  fnOptions: ?ComponentOptions; // for SSR caching
  fnScopeId: ?string; // functional scope id support

  constructor (
    tag?: string,
    data?: VNodeData,
```

```
    children?: ?Array<VNode>,
    text?: string,
    elm?: Node,
    context?: Component,
    componentOptions?: VNodeComponentOptions,
    asyncFactory?: Function
  ) {
    this.tag = tag
    this.data = data
    this.children = children
    this.text = text
    this.elm = elm
    this.ns = undefined
    this.context = context
    this.fnContext = undefined
    this.fnOptions = undefined
    this.fnScopeId = undefined
    this.key = data && data.key
    this.componentOptions = componentOptions
    this.componentInstance = undefined
    this.parent = undefined
    this.raw = false
    this.isStatic = false
    this.isRootInsert = true
    this.isComment = false
    this.isCloned = false
    this.isOnce = false
    this.asyncFactory = asyncFactory
    this.asyncMeta = undefined
    this.isAsyncPlaceholder = false
  }

  // DEPRECATED: alias for componentInstance for backwards compat.
  /* istanbul ignore next */
  get child (): Component | void {
    return this.componentInstance
  }
}
```

实际上 Vue.js 中 Virtual DOM 是借鉴了一个开源库 snabbdom 的实现，然后加入了一些 Vue.js本身的东西，在后面的内容里，会在VNode的create、diff、patch等阶段讲解VNode的操作；

5.5 createElement

Vue.js 利用 createElement 方法创建 VNode，它定义在 `src/core/vdom/create-element.js` 中：

```
// wrapper function for providing a more flexible interface
// without getting yelled at by flow
export function createElement (
  context: Component,
  tag: any,
  data: any,
  children: any,
  normalizationType: any,
  alwaysNormalize: boolean
): VNode | Array<VNode> {
  if (Array.isArray(data) || isPrimitive(data)) {
    normalizationType = children
    children = data
    data = undefined
  }
  if (isTrue(alwaysNormalize)) {
    normalizationType = ALWAYS_NORMALIZE
  }
  return _createElement(context, tag, data, children, normalizationType)
}

export function _createElement (
  context: Component,
  tag?: string | Class<Component> | Function | Object,
  data?: VNodeData,
  children?: any,
  normalizationType?: number
): VNode | Array<VNode> {
  if (isDef(data) && isDef((data: any).__ob__)) {
    process.env.NODE_ENV !== 'production' && warn(
      `Avoid using observed data object as vnode data:
${JSON.stringify(data)}\n` +
      'Always create fresh vnode data objects in each render!',
      context
    )
    return createEmptyVNode()
  }
  // object syntax in v-bind
  if (isDef(data) && isDef(data.is)) {
    tag = data.is
  }
  if (!tag) {
    // in case of component :is set to falsy value
    return createEmptyVNode()
  }
  // warn against non-primitive key
```

```
  if (process.env.NODE_ENV !== 'production' &&
    isDef(data) && isDef(data.key) && !isPrimitive(data.key)
  ) {
    if (!__WEEX__ || !('@binding' in data.key)) {
      warn(
        'Avoid using non-primitive value as key, ' +
        'use string/number value instead.',
        context
      )
    }
  }
  // support single function children as default scoped slot
  if (Array.isArray(children) &&
    typeof children[0] === 'function'
  ) {
    data = data || {}
    data.scopedSlots = { default: children[0] }
    children.length = 0
  }
  if (normalizationType === ALWAYS_NORMALIZE) {
    children = normalizeChildren(children)
  } else if (normalizationType === SIMPLE_NORMALIZE) {
    children = simpleNormalizeChildren(children)
  }
  let vnode, ns
  if (typeof tag === 'string') {
    let Ctor
    ns = (context.$vnode && context.$vnode.ns) || config.getTagNamespace(tag)
    if (config.isReservedTag(tag)) {
      // platform built-in elements
      if (process.env.NODE_ENV !== 'production' && isDef(data) &&
isDef(data.nativeOn) && data.tag !== 'component') {
        warn(
          `The .native modifier for v-on is only valid on components but it
was used on <${tag}>.`,
          context
        )
      }
      vnode = new VNode(
        config.parsePlatformTagName(tag), data, children,
        undefined, undefined, context
      )
    } else if ((!data || !data.pre) && isDef(Ctor =
resolveAsset(context.$options, 'components', tag))) {
      // component
      vnode = createComponent(Ctor, data, context, children, tag)
    } else {
```

```
            // unknown or unlisted namespaced elements
            // check at runtime because it may get assigned a namespace when its
            // parent normalizes children
            vnode = new VNode(
              tag, data, children,
              undefined, undefined, context
            )
          }
        } else {
          // direct component options / constructor
          vnode = createComponent(tag, data, context, children)
        }
        if (Array.isArray(vnode)) {
          return vnode
        } else if (isDef(vnode)) {
          if (isDef(ns)) applyNS(vnode, ns)
          if (isDef(data)) registerDeepBindings(data)
          return vnode
        } else {
          return createEmptyVNode()
        }
      }
```

_createElement 方法有 5 个参数：

1. context 表示 VNode 的上下文环境，它是 Component 类型；
2. tag 表示标签，它可以是一个字符串，也可以是一个 Component；
3. data 表示 VNode 的数据，它是一个 VNodeData 类型，可以在 flow/vnode.js 中找到它的定义，这里先不展开说；
4. children 表示当前 VNode 的子节点，它是任意类型的，它接下来需要被规范为标准的 VNode 数组；
5. normalizationType 表示子节点规范的类型，类型不同规范的方法也就不一样，它主要是参考 render 函数是编译生成的还是用户手写的。

createElement里核心的流程包括两个：

5.5.1 children的规范化

由于 Virtual DOM 实际上是一个树状结构，每一个 VNode 可能会有若干个子节点，这些子节点应该也是 VNode 的类型。_createElement 接收的第 4 个参数 children 是任意类型的，因此我们需要把它们规范成 VNode 类型。

这里根据 normalizationType 的不同，调用了 `normalizeChildren(children)` 和 `simpleNormalizeChildren(children)` 方法，它们的定义都在 `src/core/vdom/helpers/normalzie-children.js` 中：

```
export function simpleNormalizeChildren (children: any) {
  for (let i = 0; i < children.length; i++) {
    if (Array.isArray(children[i])) {
      return Array.prototype.concat.apply([], children)
    }
  }
  return children
}

export function normalizeChildren (children: any): ?Array<VNode> {
  return isPrimitive(children)
    ? [createTextVNode(children)]
    : Array.isArray(children)
      ? normalizeArrayChildren(children)
      : undefined
}
```

- simpleNormalizeChildren：

1. 调用场景是 render 函数是编译生成的。
2. 理论上编译生成的 children 都已经是 VNode 类型的，但这里有一个例外，就是 functional component 函数式组件返回的是一个数组而不是一个根节点，所以会通过 `Array.prototype.concat` 方法把整个 children 数组打平，让它的深度只有一层。

- normalizeChildren 方法的调用场景有 2 种：

1. 一个场景是 render 函数是用户手写的，当 children 只有一个节点的时候，Vue.js 从接口层面允许用户把 children 写成基础类型用来创建单个简单的文本节点，这种情况会调用 `createTextVNode` 创建一个文本节点的 VNode；
2. 另一个场景是当编译 slot、v-for 的时候会产生嵌套数组的情况，会调用 `normalizeArrayChildren` 方法；

```
function normalizeArrayChildren (children: any, nestedIndex?: string):
Array<VNode> {
  const res = []
  let i, c, lastIndex, last
  for (i = 0; i < children.length; i++) {
    c = children[i]
    if (isUndef(c) || typeof c === 'boolean') continue
```

```
      lastIndex = res.length - 1
      last = res[lastIndex]
      //  nested
      if (Array.isArray(c)) {
        if (c.length > 0) {
          c = normalizeArrayChildren(c, `${nestedIndex || ''}_${i}`)
          // merge adjacent text nodes
          if (isTextNode(c[0]) && isTextNode(last)) {
            res[lastIndex] = createTextVNode(last.text + (c[0]: any).text)
            c.shift()
          }
          res.push.apply(res, c)
        }
      } else if (isPrimitive(c)) {
        if (isTextNode(last)) {
          // merge adjacent text nodes
          // this is necessary for SSR hydration because text nodes are
          // essentially merged when rendered to HTML strings
          res[lastIndex] = createTextVNode(last.text + c)
        } else if (c !== '') {
          // convert primitive to vnode
          res.push(createTextVNode(c))
        }
      } else {
        if (isTextNode(c) && isTextNode(last)) {
          // merge adjacent text nodes
          res[lastIndex] = createTextVNode(last.text + c.text)
        } else {
          // default key for nested array children (likely generated by v-for)
          if (isTrue(children._isVList) &&
            isDef(c.tag) &&
            isUndef(c.key) &&
            isDef(nestedIndex)) {
            c.key = `__vlist${nestedIndex}_${i}__`
          }
          res.push(c)
        }
      }
    }
    return res
  }
```

normalizeArrayChildren 接收 2 个参数：

1. children：表示要规范的子节点；

2. nestedIndex：表示嵌套的索引；

因为单个 child 可能是一个数组类型。 `normalizeArrayChildren` 主要的逻辑就是遍历 children，获得单个节点 c，然后对 c 的类型判断：

1. 如果是一个数组类型，则递归调用 `normalizeArrayChildren`；
2. 如果是基础类型，则通过 `createTextVNode` 方法转换成 VNode 类型；否则就已经是 VNode 类型了，如果 children 是一个列表并且列表还存在嵌套的情况，则根据 nestedIndex 去更新它的 key；

注意：在遍历的过程中，如果存在两个连续的 text 节点，会把它们合并成一个 text 节点

然后，children就变为了VNode的单节点或者数组了

5.5.2 VNode的构建

规范完children后，需要创建一个VNode实例：

```
let vnode, ns
if (typeof tag === 'string') {
  let Ctor
  ns = (context.$vnode && context.$vnode.ns) || config.getTagNamespace(tag)
  if (config.isReservedTag(tag)) {
    // platform built-in elements
    if (process.env.NODE_ENV !== 'production' && isDef(data) &&
isDef(data.nativeOn) && data.tag !== 'component') {
      warn(
        `The .native modifier for v-on is only valid on components but it was
used on <${tag}>.`,
        context
      )
    }
    vnode = new VNode(
      config.parsePlatformTagName(tag), data, children,
      undefined, undefined, context
    )
  } else if ((!data || !data.pre) && isDef(Ctor =
resolveAsset(context.$options, 'components', tag))) {
    // component
    vnode = createComponent(Ctor, data, context, children, tag)
  } else {
    // unknown or unlisted namespaced elements
    // check at runtime because it may get assigned a namespace when its
    // parent normalizes children
    vnode = new VNode(
      tag, data, children,
```

```
        undefined, undefined, context
      )
    }
  } else {
    // direct component options / constructor
    vnode = createComponent(tag, data, context, children)
  }
```

这里先对 tag 做判断：

1. 如果是 string 类型，则接着判断如果是内置的一些节点，则直接创建一个普通 VNode；
2. 如果是为已注册的组件名，则通过 `createComponent` 创建一个组件类型的 VNode，否则创建一个未知的标签的 VNode；
3. 如果是 tag 一个 Component 类型，则直接调用 `createComponent` 创建一个组件类型的 VNode 节点。对于 `createComponent` 创建组件类型的 VNode 的过程，本质上它还是返回了一个 VNode，后面讲解。

因此，createElement 创建 VNode 的过程，每个 VNode 有 children，children 每个元素也是一个 VNode，这样就形成了一个 VNode Tree，它很好的描述了我们的 DOM Tree。

5.6 update

Vue 的 _update 是实例的一个私有方法，它被调用的时机有 2 个，一个是首次渲染，一个是数据更新的时候；（涉及到数据更新的代码会在后面讲到），_update 方法的作用是把 VNode 渲染成真实的 DOM，它的定义在 `src/core/instance/lifecycle.js` 中：

```
Vue.prototype._update = function (vnode: VNode, hydrating?: boolean) {
  const vm: Component = this
  const prevEl = vm.$el
  const prevVnode = vm._vnode
  const prevActiveInstance = activeInstance
  activeInstance = vm
  vm._vnode = vnode
  // Vue.prototype.__patch__ is injected in entry points
  // based on the rendering backend used.
  if (!prevVnode) {
    // initial render
    vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly */)
  } else {
    // updates
    vm.$el = vm.__patch__(prevVnode, vnode)
  }
```

```
    activeInstance = prevActiveInstance
    // 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.
  }
```

_update 的核心就是调用 `vm.__patch__` 方法，这个方法实际上在不同的平台，比如 web 和 weex 上的定义是不一样的，因此在 web 平台中它的定义在 `src/platforms/web/runtime/index.js` 中：

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

在 web 平台上，是否是服务端渲染也会对这个方法产生影响。因为在服务端渲染中，没有真实的浏览器 DOM 环境，所以不需要把 VNode 最终转换成 DOM，因此是一个空函数，而在浏览器端渲染中，它指向了 patch 方法，它的定义在 `src/platforms/web/runtime/patch.js` 中：

```
import * as nodeOps from 'web/runtime/node-ops'
import { createPatchFunction } from 'core/vdom/patch'
import baseModules from 'core/vdom/modules/index'
import platformModules from 'web/runtime/modules/index'

// the directive module should be applied last, after all
// built-in modules have been applied.
const modules = platformModules.concat(baseModules)

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

该方法的定义是调用 `createPatchFunction` 方法的返回值，这里传入了一个对象，包含 nodeOps 参数和 modules 参数。其中，nodeOps 封装了一系列 DOM 操作的方法，modules 定义了

一些模块的钩子函数的实现，我们这里先不详细介绍，来看一下 createPatchFunction 的实现，它定义在 `src/core/vdom/patch.js` 中：

```
const hooks = ['create', 'activate', 'update', 'remove', 'destroy']

export function createPatchFunction (backend) {
  let i, j
  const cbs = {}

  const { modules, nodeOps } = backend

  for (i = 0; i < hooks.length; ++i) {
    cbs[hooks[i]] = []
    for (j = 0; j < modules.length; ++j) {
      if (isDef(modules[j][hooks[i]])) {
        cbs[hooks[i]].push(modules[j][hooks[i]])
      }
    }
  }

  // ...

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

    let isInitialPatch = false
    const insertedVnodeQueue = []

    if (isUndef(oldVnode)) {
      // empty mount (likely as component), create new root element
      isInitialPatch = true
      createElm(vnode, insertedVnodeQueue)
    } else {
      const 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)) {
            if (hydrate(oldVnode, vnode, insertedVnodeQueue)) {
              invokeInsertHook(vnode, insertedVnodeQueue, true)
              return oldVnode
            } else if (process.env.NODE_ENV !== 'production') {
              warn(
                'The client-side rendered virtual DOM tree is not matching ' +
                'server-rendered content. This is likely caused by incorrect '
+
                'HTML markup, for example nesting block-level elements inside '
 +
                '<p>, or missing <tbody>. Bailing hydration and performing ' +
                'full client-side render.'
              )
            }
          }
          // either not server-rendered, or hydration failed.
          // create an empty node and replace it
          oldVnode = emptyNodeAt(oldVnode)
        }

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

        // create new node
        createElm(
          vnode,
          insertedVnodeQueue,
          // extremely rare edge case: do not insert if old element is in a
          // leaving transition. Only happens when combining transition +
          // keep-alive + HOCs. (#4590)
          oldElm._leaveCb ? null : parentElm,
          nodeOps.nextSibling(oldElm)
        )

        // update parent placeholder node element, recursively
        if (isDef(vnode.parent)) {
          let ancestor = vnode.parent
          const patchable = isPatchable(vnode)
          while (ancestor) {
            for (let i = 0; i < cbs.destroy.length; ++i) {
              cbs.destroy[i](ancestor)
            }
```

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

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

        invokeInsertHook(vnode, insertedVnodeQueue, isInitialPatch)
        return vnode.elm
      }
    }
```

createPatchFunction 内部定义了一系列的辅助方法，最终返回了一个 patch 方法，这个方法就赋值给了 vm._update 函数里调用的 `vm.__patch__`。

Q：为何 Vue.js 源码绕了这么一大圈，把相关代码分散到各个目录？

A：

1. 因为patch 是平台相关的，在 Web 和 Weex 环境，它们把虚拟 DOM 映射到 “平台 DOM” 的方法是不同的，并且对 “DOM” 包括的属性模块创建和更新也不尽相同。因此每个平台都有各自的 `nodeOps` 和 `modules` ，它们的代码需要托管在 src/platforms 这个大目录下；
2. 不同平台的 patch 的主要逻辑部分是相同的，所以这部分公共的部分托管在 core 这个大目录下。差异化部分只需要通过参数 `nodeOps` 和 `modules` 来区分：
   a. nodeOps 表示对 “平台 DOM” 的一些操作方法；
   b. modules 表示平台的一些模块，它们会在整个 patch 过程的不同阶段执行相应的钩子函数；

回到 patch 方法本身，它接收 4个参数：

1. oldVnode 表示旧的 VNode 节点，它也可以不存在或者是一个 DOM 对象；
2. vnode 表示执行 _render 后返回的 VNode 的节点；
3. hydrating 表示是否是服务端渲染；
4. removeOnly 是给 transition-group 用的；

先来回顾我们的例子：

```
var app = new Vue({
  el: '#app',
  render: function (createElement) {
    return createElement('div', {
      attrs: {
        id: 'app'
      },
    }, this.message)
  },
  data: {
    message: 'Hello Vue!'
  }
})
```

在 vm._update 的方法里是这么调用 patch 方法的：

```
// initial render
vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly */)
```

结合我们的例子：

1. 我们的场景是首次渲染，所以在执行 patch 函数的时候，传入的 vm.$el 对应的是例子中 id 为 app 的 DOM 对象，这个也就是我们在 index.html 模板中写的 `<div id="app">` ，vm.$el 的赋值是在之前 mountComponent 函数做的；
2. vnode 对应的是调用 render 函数的返回值；
3. hydrating 在非服务端渲染情况下为 false；
4. removeOnly 为 false；

这时候回顾patch的执行过程上

```
const 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)) {
      if (hydrate(oldVnode, vnode, insertedVnodeQueue)) {
        invokeInsertHook(vnode, insertedVnodeQueue, true)
        return oldVnode
      } else if (process.env.NODE_ENV !== 'production') {
        warn(
          'The client-side rendered virtual DOM tree is not matching ' +
          'server-rendered content. This is likely caused by incorrect ' +
          'HTML markup, for example nesting block-level elements inside ' +
          '<p>, or missing <tbody>. Bailing hydration and performing ' +
          'full client-side render.'
        )
      }
    }
    // either not server-rendered, or hydration failed.
    // create an empty node and replace it
    oldVnode = emptyNodeAt(oldVnode)
  }

  // replacing existing element
  const oldElm = oldVnode.elm
```

```
    const parentElm = nodeOps.parentNode(oldElm)

    // create new node
    createElm(
      vnode,
      insertedVnodeQueue,
      // extremely rare edge case: do not insert if old element is in a
      // leaving transition. Only happens when combining transition +
      // keep-alive + HOCs. (#4590)
      oldElm._leaveCb ? null : parentElm,
      nodeOps.nextSibling(oldElm)
    )
  }
```

由于我们传入的 oldVnode 实际上是一个 DOM，所以 isRealElement 为 true，接下来又通过 emptyNodeAt 方法把 oldVnode 转换成 VNode 对象，然后再调用 createElm 方法，来看一下它的实现：

```
  function createElm (
    vnode,
    insertedVnodeQueue,
    parentElm,
    refElm,
    nested,
    ownerArray,
    index
  ) {
    if (isDef(vnode.elm) && isDef(ownerArray)) {
      // This vnode was used in a previous render!
      // now it's used as a new node, overwriting its elm would cause
      // potential patch errors down the road when it's used as an insertion
      // reference node. Instead, we clone the node on-demand before creating
      // associated DOM element for it.
      vnode = ownerArray[index] = cloneVNode(vnode)
    }

    vnode.isRootInsert = !nested // for transition enter check
    if (createComponent(vnode, insertedVnodeQueue, parentElm, refElm)) {
      return
    }

    const data = vnode.data
    const children = vnode.children
    const tag = vnode.tag
    if (isDef(tag)) {
```

```
        if (process.env.NODE_ENV !== 'production') {
          if (data && data.pre) {
            creatingElmInVPre++
          }
          if (isUnknownElement(vnode, creatingElmInVPre)) {
            warn(
              'Unknown custom element: <' + tag + '> - did you ' +
              'register the component correctly? For recursive components, ' +
              'make sure to provide the "name" option.',
              vnode.context
            )
          }
        }

        vnode.elm = vnode.ns
          ? nodeOps.createElementNS(vnode.ns, tag)
          : nodeOps.createElement(tag, vnode)
        setScope(vnode)

        /* istanbul ignore if */
        if (__WEEX__) {
          // ...
        } else {
          createChildren(vnode, children, insertedVnodeQueue)
          if (isDef(data)) {
            invokeCreateHooks(vnode, insertedVnodeQueue)
          }
          insert(parentElm, vnode.elm, refElm)
        }

        if (process.env.NODE_ENV !== 'production' && data && data.pre) {
          creatingElmInVPre--
        }
      } else if (isTrue(vnode.isComment)) {
        vnode.elm = nodeOps.createComment(vnode.text)
        insert(parentElm, vnode.elm, refElm)
      } else {
        vnode.elm = nodeOps.createTextNode(vnode.text)
        insert(parentElm, vnode.elm, refElm)
      }
    }
```

createElm 的作用是通过虚拟节点创建真实的 DOM 并插入到它的父节点中。

1. createComponent 方法目的是尝试创建子组件，后面详细解释，在当前这个 case 下它的返回值为 false；

2. 接下来判断 vnode 是否包含 tag，如果包含，先简单对 tag 的合法性在非生产环境下做校验，看是否是一个合法标签；然后再去调用平台 DOM 的操作去创建一个占位符元素；（ns：nameSpace）

```
vnode.elm = vnode.ns
  ? nodeOps.createElementNS(vnode.ns, tag)
  : nodeOps.createElement(tag, vnode)
```

接下来调用 createChildren 方法去创建子元素，也就是去遍历 `createElm` 的：

```
createChildren(vnode, children, insertedVnodeQueue)

function createChildren (vnode, children, insertedVnodeQueue) {
  if (Array.isArray(children)) {
    if (process.env.NODE_ENV !== 'production') {
      checkDuplicateKeys(children)
    }
    for (let i = 0; i < children.length; ++i) {
      createElm(children[i], insertedVnodeQueue, vnode.elm, null, true,
children, i)
    }
  } else if (isPrimitive(vnode.text)) {
    nodeOps.appendChild(vnode.elm, nodeOps.createTextNode(String(vnode.text)))
  }
}
```

接着再调用 invokeCreateHooks 方法执行所有的 create 的钩子并把 vnode push 到 insertedVnodeQueue 中。

```
 if (isDef(data)) {
  invokeCreateHooks(vnode, insertedVnodeQueue)
}

function invokeCreateHooks (vnode, insertedVnodeQueue) {
  for (let i = 0; i < cbs.create.length; ++i) {
    cbs.create[i](emptyNode, vnode)
  }
  i = vnode.data.hook // Reuse variable
  if (isDef(i)) {
    if (isDef(i.create)) i.create(emptyNode, vnode)
    if (isDef(i.insert)) insertedVnodeQueue.push(vnode)
  }
}
```

最后调用 insert 方法把 DOM 插入到父节点中，因为是递归调用，子元素会优先调用 insert，所以整个 vnode 树节点的插入顺序是先子后父。来看一下 insert 方法，它的定义在 `src/core/vdom/patch.js` 上。

```
insert(parentElm, vnode.elm, refElm)

function insert (parent, elm, ref) {
  if (isDef(parent)) {
    if (isDef(ref)) {
      if (ref.parentNode === parent) {
        nodeOps.insertBefore(parent, elm, ref)
      }
    } else {
      nodeOps.appendChild(parent, elm)
    }
  }
}
```

insert 调用一些 nodeOps 把子节点插入到父节点中，这些辅助方法定义在 `src/platforms/web/runtime/node-ops.js` 中：

```
export function insertBefore (parentNode: Node, newNode: Node, referenceNode:
Node) {
  parentNode.insertBefore(newNode, referenceNode)
}

export function appendChild (node: Node, child: Node) {
  node.appendChild(child)
}
```

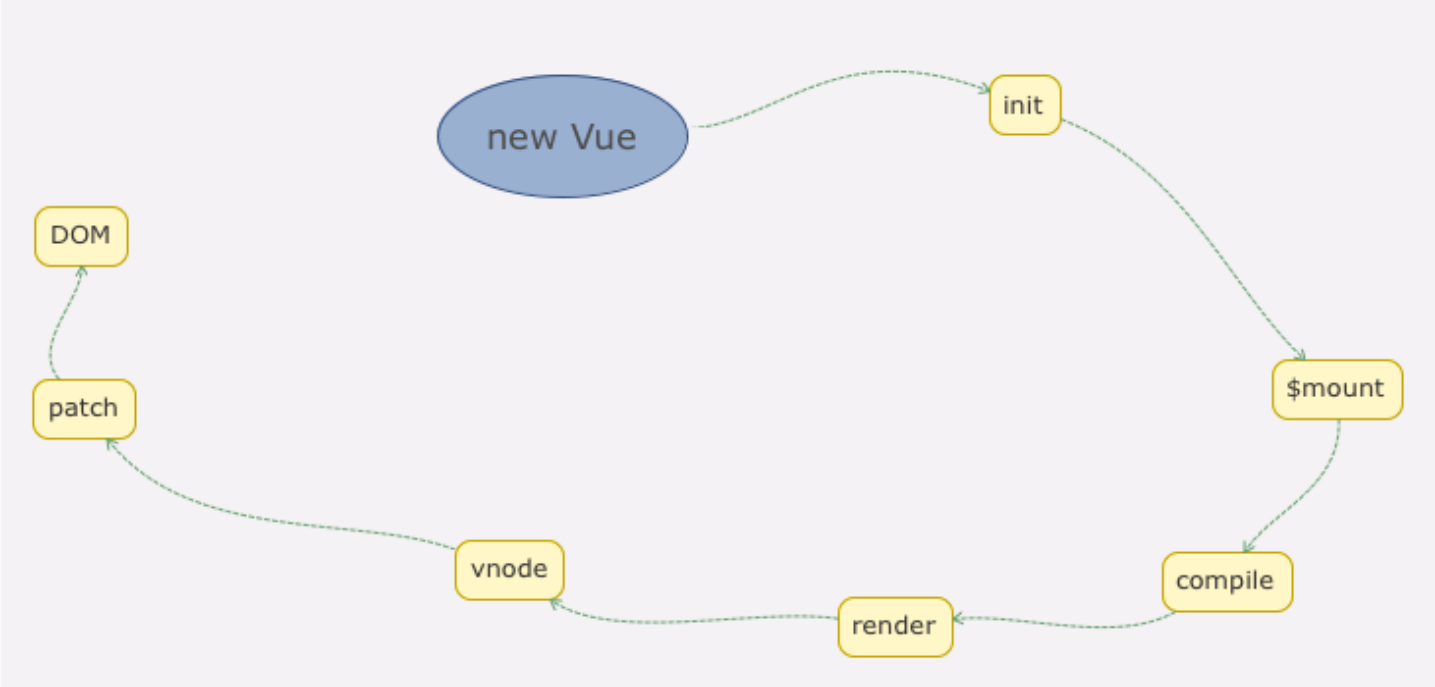
其实在web中，就是调用原生 DOM 的 API 进行 DOM 操作；

在 `createElm` 过程中，如果 vnode 节点不包含 tag，则它有可能是一个注释或者纯文本节点，可以直接插入到父元素中。在我们这个例子中，最内层就是一个文本 vnode，它的 text 值取的就是之前的 this.message 的值 Hello Vue!。

再回到 `patch` 方法，首次渲染我们调用了 createElm 方法，这里传入的 parentElm 是 oldVnode.elm 的父元素，在我们的例子是 id 为 #app div 的父元素，也就是 Body；实际上整个过程就是递归创建了一个完整的 DOM 树并插入到 Body 上；

最后，我们根据之前递归 createElm 生成的 vnode 插入顺序队列，执行相关的 insert 钩子函数；

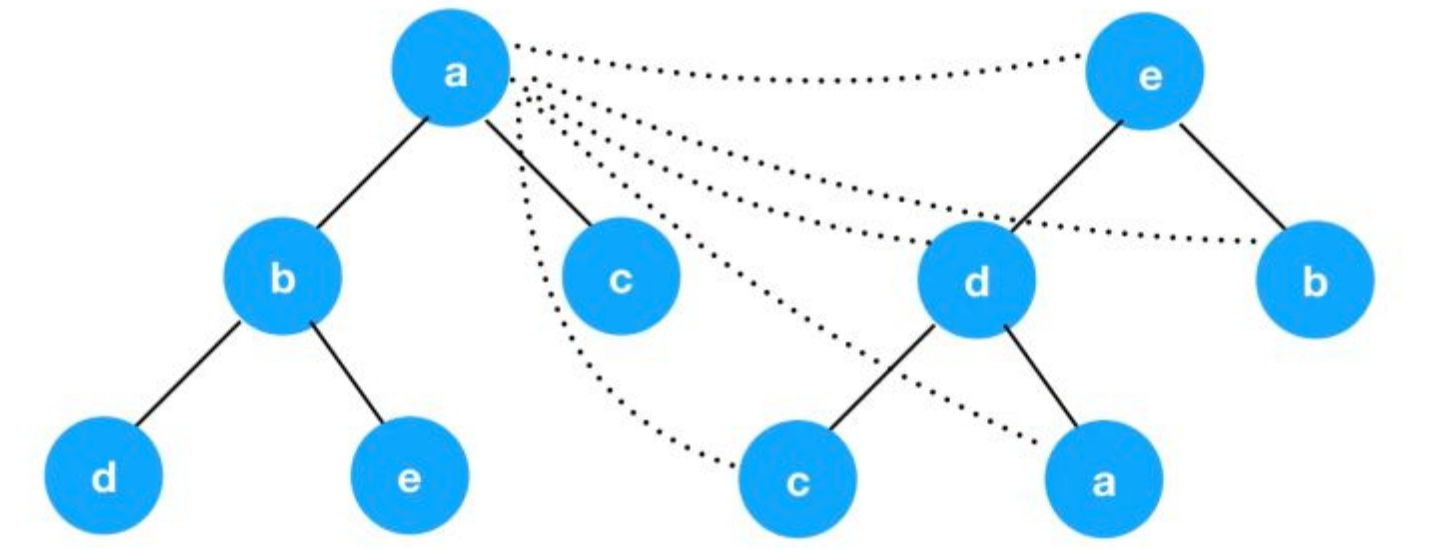
以上就是数据在Vue2中如何渲染成DOM的过程。



6. Vue2 Diff 算法

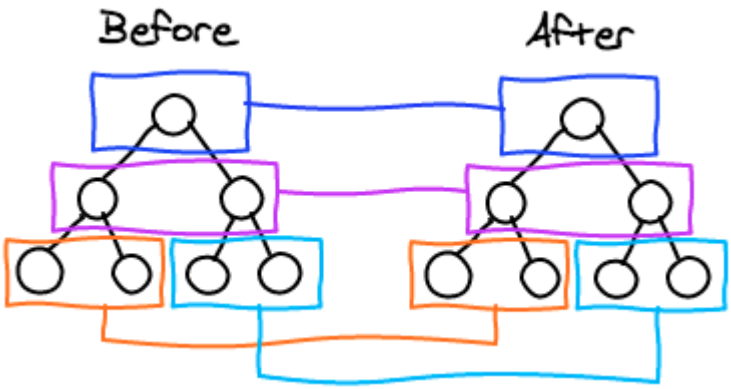
6.1 Diff 算法的目的

diff算法的目的是为了找到哪些节点发生了变化，哪些节点没有发生变化可以复用。如果用最传统的diff算法，如下图所示，每个节点都要遍历另一棵树上的所有节点做比较，这就是o(n^2)的复杂度，加上更新节点时的o(n)复杂度，那就总共达到了o(n^3)的复杂度，这对于一个结构复杂节点数众多的页面，成本是非常大的。



实际上vue和react都对虚拟dom的diff算法做了一定的优化，将复杂度降低到了o(n)级别，具体的策略是：同层的节点才相互比较；

1. 节点比较时，如果类型不同，则对该节点及其所有子节点直接销毁新建；
2. 类型相同的子节点，使用key帮助查找，并且使用算法优化查找效率。其中react和vue2以及vue3的diff算法都不尽相同；



主要对比Vue2和Vue3，掌握为什么要从Vue2升级到Vue3，并代入后续代码，掌握Vue实现diff的流程；

前提：

- `mount(vnode, parent, [refNode])`：通过`vnode`生成真实的DOM节点。parent为其父级的真实DOM节点，`refNode`为真实的DOM节点，其父级节点为parent。如果refNode不为空，vnode生成的DOM节点就会插入到refNode之前；如果refNode为空，那么vnode生成的DOM节点就作为最后一个子节点插入到parent中
- `patch(prevNode, nextNode, parent)`：可以简单的理解为给当前DOM节点进行更新，并且调用diff算法对比自身的子节点；

6.2 双端比较

双端比较就是新列表和旧列表两个列表的头与尾互相对比，，在对比的过程中指针会逐渐向内靠拢，直到某一个列表的节点全部遍历过，对比停止；

6.2.1 patch

先判断是否是首次渲染，如果是首次渲染那么我们就直接createElm即可；如果不是就去判断新老两个节点的元素类型否一样；如果两个节点都是一样的，那么就深入检查他们的子节点。如果两个节点不一样那就说明Vnode完全被改变了，就可以直接替换oldVnode；

先判断是否是首次渲染，如果是首次渲染那么我们就直接createElm即可；如果不是就去判断新老两个节点的元素类型否一样；如果两个节点都是一样的，那么就深入检查他们的子节点。如果两个节点不一样那就说明Vnode完全被改变了，就可以直接替换oldVnode；

```
function patch(oldVnode, vnode, hydrating, removeOnly) {
    // 判断新的vnode是否为空
    if (isUndef(vnode)) {
      // 如果老的vnode不为空 卸载所有的老vnode
      if (isDef(oldVnode)) invokeDestroyHook(oldVnode)
      return
    }
    let isInitialPatch = false
    // 用来存储 insert钩子函数，在插入节点之前调用
    const insertedVnodeQueue = []
    // 如果老节点不存在，直接创建新节点
    if (isUndef(oldVnode)) {
      isInitialPatch = true
      createElm(vnode, insertedVnodeQueue)
    } else {
      // 是不是元素节点
      const isRealElement = isDef(oldVnode.nodeType)
      // 当老节点不是真实的DOM节点，并且新老节点的type和key相同，进行patchVnode更新工作
      if (!isRealElement && sameVnode(oldVnode, vnode)) {
        patchVnode(oldVnode, vnode, insertedVnodeQueue, null, null, removeOnly)
      } else {
        // 如果不是同一元素节点的话
        // 当老节点是真实DOM节点的时候
        if (isRealElement) {
          // 如果是元素节点 并且在SSR环境的时候 修改SSR_ATTR属性
          if (oldVnode.nodeType === 1 && oldVnode.hasAttribute(SSR_ATTR)) {
            // 就是服务端渲染的，删掉这个属性
            oldVnode.removeAttribute(SSR_ATTR)
            hydrating = true
          }
          // 这个判断里是服务端渲染的处理逻辑
          if (isTrue(hydrating)) {
            if (hydrate(oldVnode, vnode, insertedVnodeQueue)) {
              invokeInsertHook(vnode, insertedVnodeQueue, true)
              return oldVnode
            }
          }
          // 如果不是服务端渲染的，或者混合失败，就创建一个空的注释节点替换 oldVnode
          oldVnode = emptyNodeAt(oldVnode)
        }

        // 拿到 oldVnode 的父节点
        const oldElm = oldVnode.elm
        const parentElm = nodeOps.parentNode(oldElm)

        // 根据新的 vnode 创建一个 DOM 节点，挂载到父节点上
```

```
createElm(
  vnode,
  insertedVnodeQueue,
  oldElm._leaveCb ? null : parentElm,
  nodeOps.nextSibling(oldElm)
)
// 如果新的 vnode 的根节点存在，就是说根节点被修改了，就需要遍历更新父节点
// 递归 更新父占位符元素
// 就是执行一遍 父节点的 destory 和 create 、insert 的 钩子函数
if (isDef(vnode.parent)) {
  let ancestor = vnode.parent
  const patchable = isPatchable(vnode)
  // 更新父组件的占位元素
  while (ancestor) {
    // 卸载老根节点下的全部组件
    for (let i = 0; i < cbs.destroy.length; ++i) {
      cbs.destroy[i](ancestor)
    }
    // 替换现有元素
    ancestor.elm = vnode.elm
    if (patchable) {
      for (let i = 0; i < cbs.create.length; ++i) {
        cbs.create[i](emptyNode, ancestor)
      }
      // #6513
      // invoke insert hooks that may have been merged by create hooks.
      // e.g. for directives that uses the "inserted" hook.
      const insert = ancestor.data.hook.insert
      if (insert.merged) {
        // start at index 1 to avoid re-invoking component mounted hook
        for (let i = 1; i < insert.fns.length; i++) {
          insert.fns[i]()
        }
      }
    } else {
      registerRef(ancestor)
    }
    // 更新父节点
    ancestor = ancestor.parent
  }
}
// 如果旧节点还存在，就删掉旧节点
if (isDef(parentElm)) {
  removeVnodes([oldVnode], 0, 0)
} else if (isDef(oldVnode.tag)) {
  // 否则直接卸载 oldVnode
  invokeDestroyHook(oldVnode)
```

```
        }
      }
    }
    // 执行 虚拟 dom 的 insert 钩子函数
    invokeInsertHook(vnode, insertedVnodeQueue, isInitialPatch)
    // 返回最新 vnode 的 elm ，也就是真实的 dom节点
    return vnode.elm
  }
```

6.2.2 patchVnode

- 如果 `Vnode` 和 `oldVnode` 指向同一个对象，则直接return即可；
- 将旧节点的真实 DOM 赋值到新节点（真实 dom 连线到新子节点）称为elm，然后遍历调用 update 更新 `oldVnode` 上的所有属性，比如 class,style,attrs,domProps,events...；
- 如果新老节点都有文本节点，并且文本不相同，那么就用 `vnode` .text更新文本内容。
- 如果oldVnode有子节点而 `Vnode` 没有，则直接删除老节点即可；
- 如果oldVnode没有子节点而 `Vnode` 有，则将Vnode的子节点真实化之后添加到DOM中即可。
- 如果两者都有子节点，则执行 `updateChildren` 函数比较子节点。

```
function patchVnode(
    oldVnode, // 老的虚拟 DOM 节点
    vnode, // 新节点
    insertedVnodeQueue, // 插入节点队列
    ownerArray, // 节点数组
    index, // 当前节点的下标
    removeOnly
  ) {
    // 新老节点对比地址一样，直接跳过
    if (oldVnode === vnode) {
      return
    }
    if (isDef(vnode.elm) && isDef(ownerArray)) {
      // clone reused vnode
      vnode = ownerArray[index] = cloneVNode(vnode)
    }
    const elm = vnode.elm = oldVnode.elm
    // 如果当前节点是注释或 v-if 的，或者是异步函数，就跳过检查异步组件
    if (isTrue(oldVnode.isAsyncPlaceholder)) {
      if (isDef(vnode.asyncFactory.resolved)) {
        hydrate(oldVnode.elm, vnode, insertedVnodeQueue)
      } else {
        vnode.isAsyncPlaceholder = true
```

```
      }
      return
    }
    // 当前节点是静态节点的时候，key 也一样，或者有 v-once 的时候，就直接赋值返回
    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)) {
      // 遍历调用 update 更新 oldVnode 所有属性，比如
class,style,attrs,domProps,events...
      // 这里的 update 钩子函数是 vnode 本身的钩子函数
      for (i = 0; i < cbs.update.length; ++i) cbs.update[i](oldVnode, vnode)
      // 这里的 update 钩子函数是我们传过来的函数
      if (isDef(i = data.hook) && isDef(i = i.update)) i(oldVnode, vnode)
    }
    // 如果新节点不是文本节点，也就是说有子节点
    if (isUndef(vnode.text)) {
      // 如果新老节点都有子节点
      if (isDef(oldCh) && isDef(ch)) {
        // 如果新老节点的子节点不一样，就执行 updateChildren 函数，对比子节点
        if (oldCh !== ch) updateChildren(elm, oldCh, ch, insertedVnodeQueue,
removeOnly)
      } else if (isDef(ch)) {
        // 如果新节点有子节点的话，就是说老节点没有子节点

        // 如果老节点是文本节点，就是说没有子节点，就清空
        if (isDef(oldVnode.text)) nodeOps.setTextContent(elm, '')
        // 添加新节点
        addVnodes(elm, null, ch, 0, ch.length - 1, insertedVnodeQueue)
      } else if (isDef(oldCh)) {
        // 如果新节点没有子节点，老节点有子节点，就删除
        removeVnodes(oldCh, 0, oldCh.length - 1)
      } else if (isDef(oldVnode.text)) {
        // 如果老节点是文本节点，就清空
        nodeOps.setTextContent(elm, '')
```

```
      }
    } else if (oldVnode.text !== vnode.text) {
      // 如果老节点的文本和新节点的文本不同，就更新文本
      nodeOps.setTextContent(elm, vnode.text)
    }
    if (isDef(data)) {
      if (isDef(i = data.hook) && isDef(i = i.postpatch)) i(oldVnode, vnode)
    }
  }
```

6.2.3 updateChildren

为了方便理解，这里手动实现Vue2中的updateChildren

6.2.3.1 实现思路

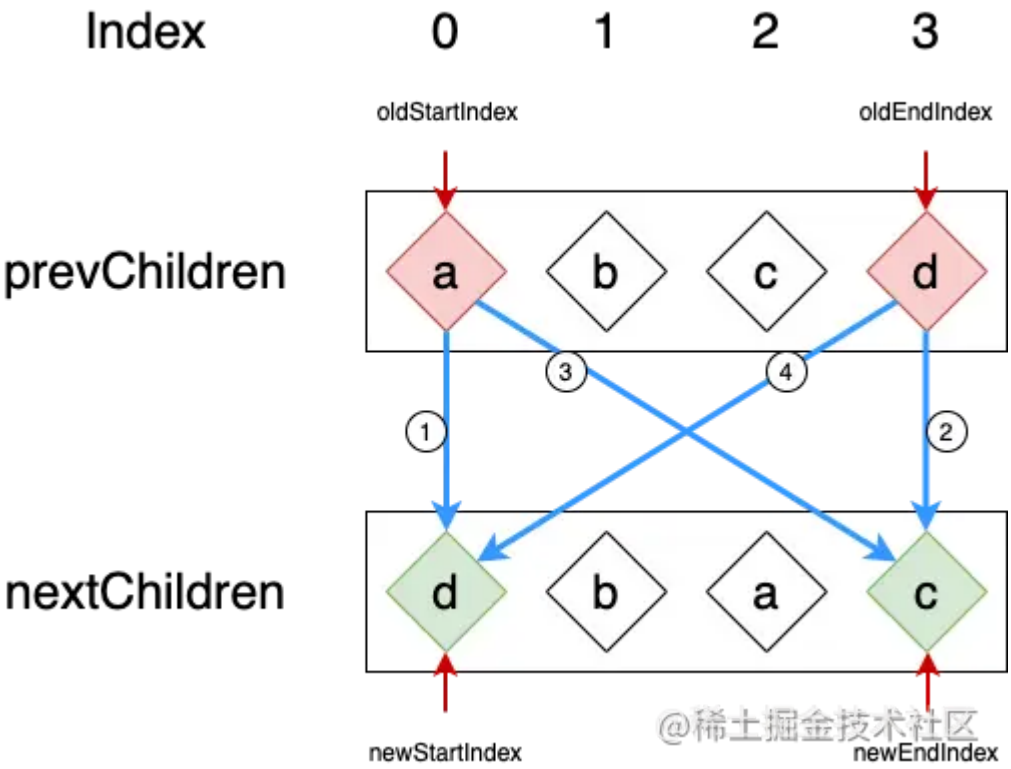
我们先用四个指针指向两个列表的头尾

```
function vue2Diff(prevChildren, nextChildren, parent) {
  let oldStartIndex = 0,
    oldEndIndex = prevChildren.length - 1
    newStartIndex = 0,
    newEndIndex = nextChildren.length - 1;
  let oldStartNode = prevChildren[oldStartIndex],
    oldEndNode = prevChildren[oldEndIndex],
    newStartNode = nextChildren[nextStartIndex],
    newEndNode = nextChildren[nextEndIndex];
}
```

根据四个指针找到四个节点，然后进行对比，那么如何对比呢？我们按照以下四个步骤进行对比

1. 使用旧列表的头一个节点 `oldStartNode` 与新列表的头一个节点 `newStartNode` 对比；
2. 使用旧列表的最后一个节点 `oldEndNode` 与新列表的最后一个节点 `newEndNode` 对比；
3. 使用旧列表的头一个节点 `oldStartNode` 与新列表的最后一个节点 `newEndNode` 对比；
4. 使用旧列表的最后一个节点 `oldEndNode` 与新列表的头一个节点 `newStartNode` 对比；

使用以上四步进行对比，去寻找key相同的可复用的节点，当在某一步中找到了则停止后面的寻找。具体对比顺序如下图：



对比顺序代码结构如下:

```
function vue2Diff(prevChildren, nextChildren, parent) {
  let oldStartIndex = 0,
    oldEndIndex = prevChildren.length - 1
    newStartIndex = 0,
    newEndIndex = nextChildren.length - 1;
  let oldStartNode = prevChildren[oldStartIndex],
    oldEndNode = prevChildren[oldEndIndex],
    newStartNode = nextChildren[newStartIndex],
    newEndNode = nextChildren[newEndIndex];

  if (oldStartNode.key === newStartNode.key) {

  } else if (oldEndNode.key === newEndNode.key) {

  } else if (oldStartNode.key === newEndNode.key) {

  } else if (oldEndNode.key === newStartNode.key) {

  }
}
```

当对比时找到了可复用的节点，我们还是先 `patch` 给元素打补丁，然后将指针进行前/后移一位指针。根据对比节点的不同，我们移动的指针和方向也不同，具体规则如下：

1. 当旧列表的头一个节点 `oldStartNode` 与新列表的头一个节点 `newStartNode` 对比时key相同。那么旧列表的头指针 `oldStartIndex` 与新列表的头指针 `newStartIndex` 同时向后移动一位；
2. 当旧列表的最后一个节点 `oldEndNode` 与新列表的最后一个节点 `newEndNode` 对比时key相同。那么旧列表的尾指针oldEndIndex与新列表的尾指针 `newEndIndex` 同时向前移动一位；
3. 当旧列表的头一个节点 `oldStartNode` 与新列表的最后一个节点 `newEndNode` 对比时key相同。那么旧列表的头指针 `oldStartIndex` 向后移动一位；新列表的尾指针 `newEndIndex` 向前移动一位；
4. 当旧列表的最后一个节点 `oldEndNode` 与新列表的头一个节点 `newStartNode` 对比时key相同。那么旧列表的尾指针 `oldEndIndex` 向前移动一位；新列表的头指针 `newStartIndex` 向后移动一位；

```
function vue2Diff(prevChildren, nextChildren, parent) {
  let oldStartIndex = 0,
    oldEndIndex = prevChildren.length - 1,
    newStartIndex = 0,
    newEndIndex = nextChildren.length - 1;
  let oldStartNode = prevChildren[oldStartIndex],
    oldEndNode = prevChildren[oldEndIndex],
    newStartNode = nextChildren[newStartIndex],
    newEndNode = nextChildren[newEndIndex];

  if (oldStartNode.key === newStartNode.key) {
    patch(oldvStartNode, newStartNode, parent)

    oldStartIndex++
    newStartIndex++
    oldStartNode = prevChildren[oldStartIndex]
    newStartNode = nextChildren[newStartIndex]
  } else if (oldEndNode.key === newEndNode.key) {
    patch(oldEndNode, newEndNode, parent)

    oldEndIndex--
    newEndIndex--
    oldEndNode = prevChildren[oldEndIndex]
    newEndNode = nextChildren[newEndIndex]
  } else if (oldStartNode.key === newEndNode.key) {
    patch(oldStartNode, newEndNode, parent)

    oldStartIndex++
    newEndIndex--
    oldStartNode = prevChildren[oldStartIndex]
    newEndNode = nextChildren[newEndIndex]
  } else if (oldEndNode.key === newStartNode.key) {
```

```
      patch(oldEndNode, newStartNode, parent)

      oldEndIndex--
      nextStartIndex++
      oldEndNode = prevChildren[oldEndIndex]
      newStartNode = nextChildren[newStartIndex]
    }
  }
```

上面提到，要让指针向内靠拢，所以我们需要循环。循环停止的条件是当其中一个列表的节点全部遍历完成，代码如下：

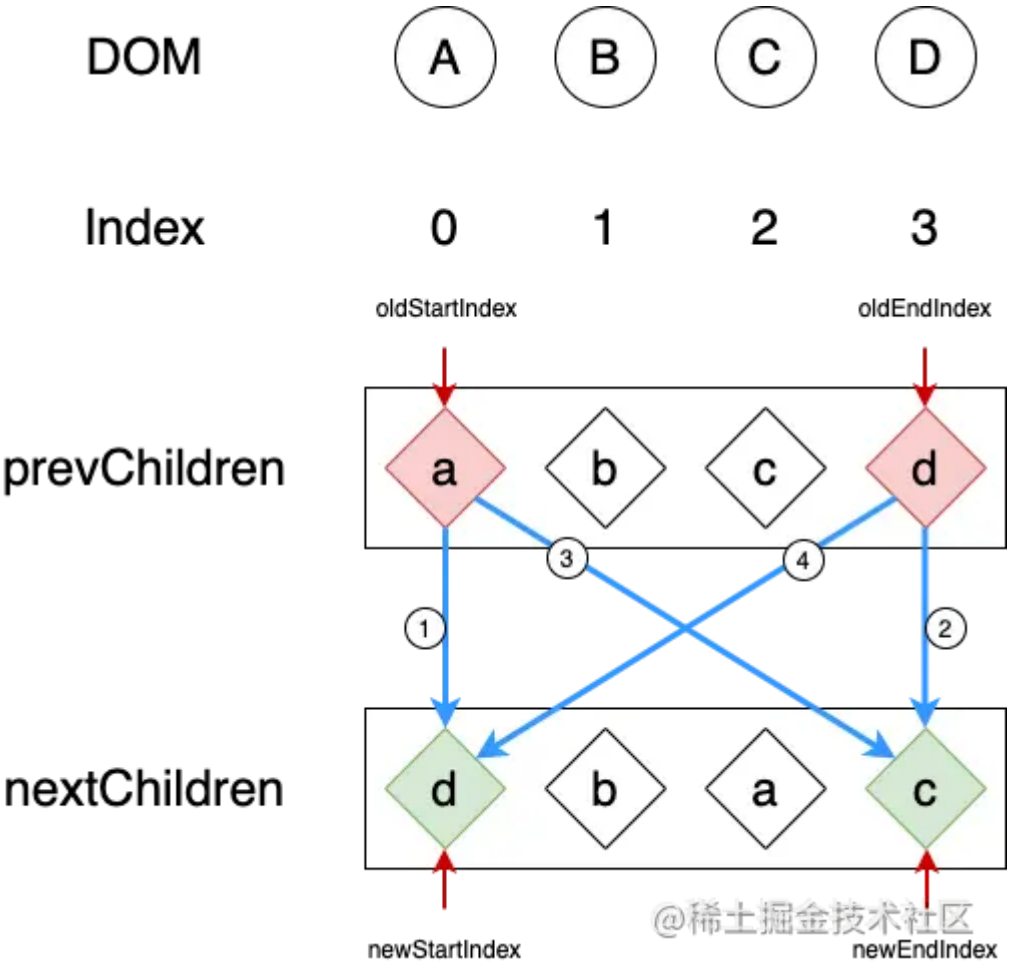
```
  function vue2Diff(prevChildren, nextChildren, parent) {
    let oldStartIndex = 0,
      oldEndIndex = prevChildren.length - 1,
      newStartIndex = 0,
      newEndIndex = nextChildren.length - 1;
    let oldStartNode = prevChildren[oldStartIndex],
      oldEndNode = prevChildren[oldEndIndex],
      newStartNode = nextChildren[newStartIndex],
      newEndNode = nextChildren[newEndIndex];
    while (oldStartIndex <= oldEndIndex && newStartIndex <= newEndIndex) {
      if (oldStartNode.key === newStartNode.key) {
        patch(oldStartNode, newStartNode, parent)

        oldStartIndex++
        newStartIndex++
        oldStartNode = prevChildren[oldStartIndex]
        newStartNode = nextChildren[newStartIndex]
      } else if (oldEndNode.key === newEndNode.key) {
        patch(oldEndNode, newEndNode, parent)

        oldEndIndex--
        newndIndex--
        oldEndNode = prevChildren[oldEndIndex]
        newEndNode = nextChildren[newEndIndex]
      } else if (oldStartNode.key === newEndNode.key) {
        patch(oldvStartNode, newEndNode, parent)

        oldStartIndex++
        newEndIndex--
        oldStartNode = prevChildren[oldStartIndex]
        newEndNode = nextChildren[newEndIndex]
      } else if (oldEndNode.key === newStartNode.key) {
        patch(oldEndNode, newStartNode, parent)
```

```
            oldEndIndex--
            newStartIndex++
            oldEndNode = prevChildren[oldEndIndex]
            newStartNode = nextChildren[newStartIndex]
          }
        }
      }
```

至此整体的循环我们就全部完成了，下面我们需要考虑这样两个问题：

- 什么情况下DOM节点需要移动；
- DOM节点如何移动；

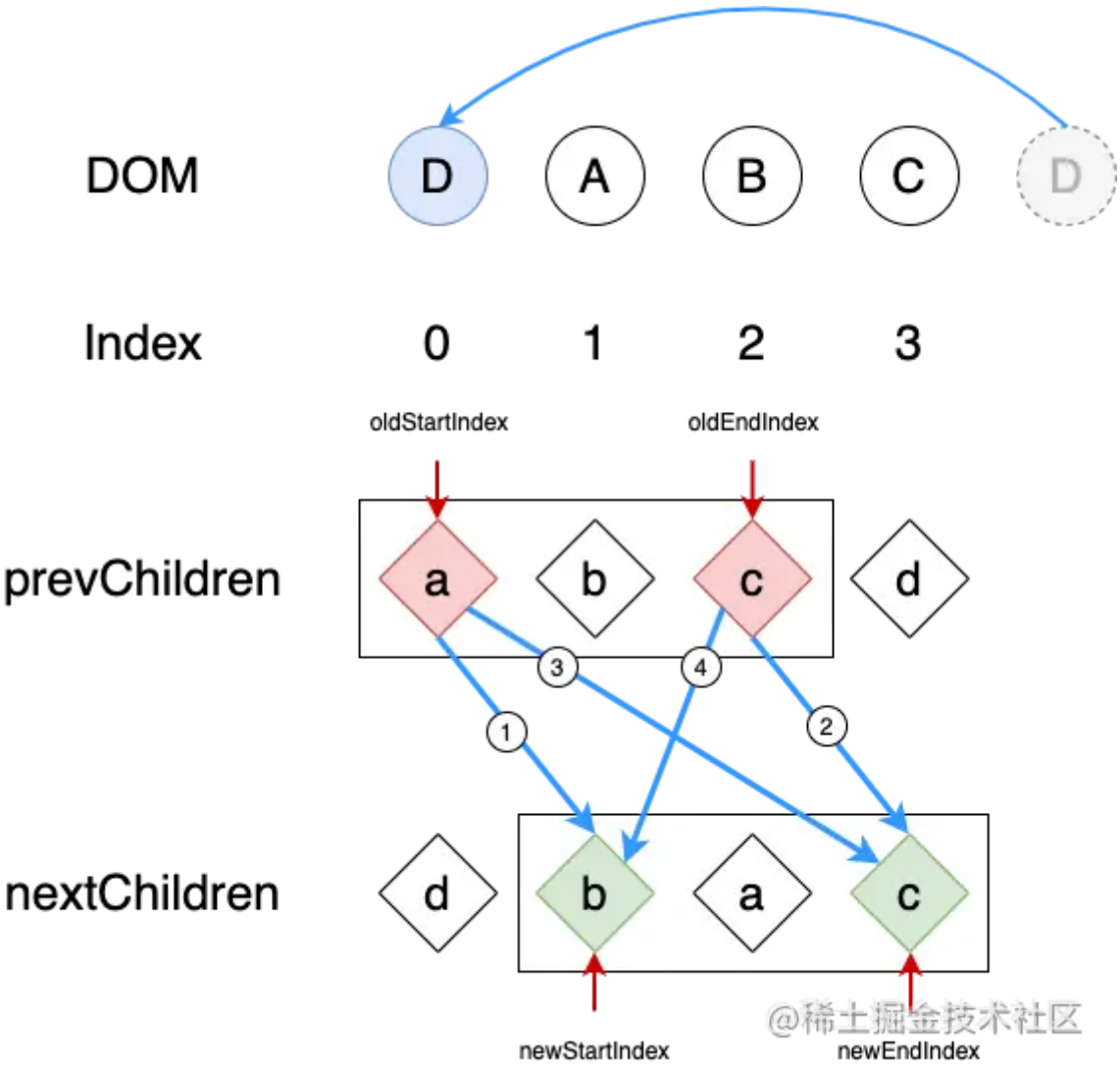
我们来解决第一个问题：什么情况下需要移动，我们还是以上图为例：



当我们在第一个循环时，在第四步发现旧列表的尾节点 `oldEndNode` 与新列表的头节点 `newStartNode` 的key相同，是可复用的DOM节点。通过观察我们可以发现，原本在旧列表末尾的节点，却是新列表中的开头节点，没有人比他更靠前，因为他是第一个，所以我们只需要把当前的节点移动到原本旧列表中的第一个节点之前，让它成为第一个节点即可。

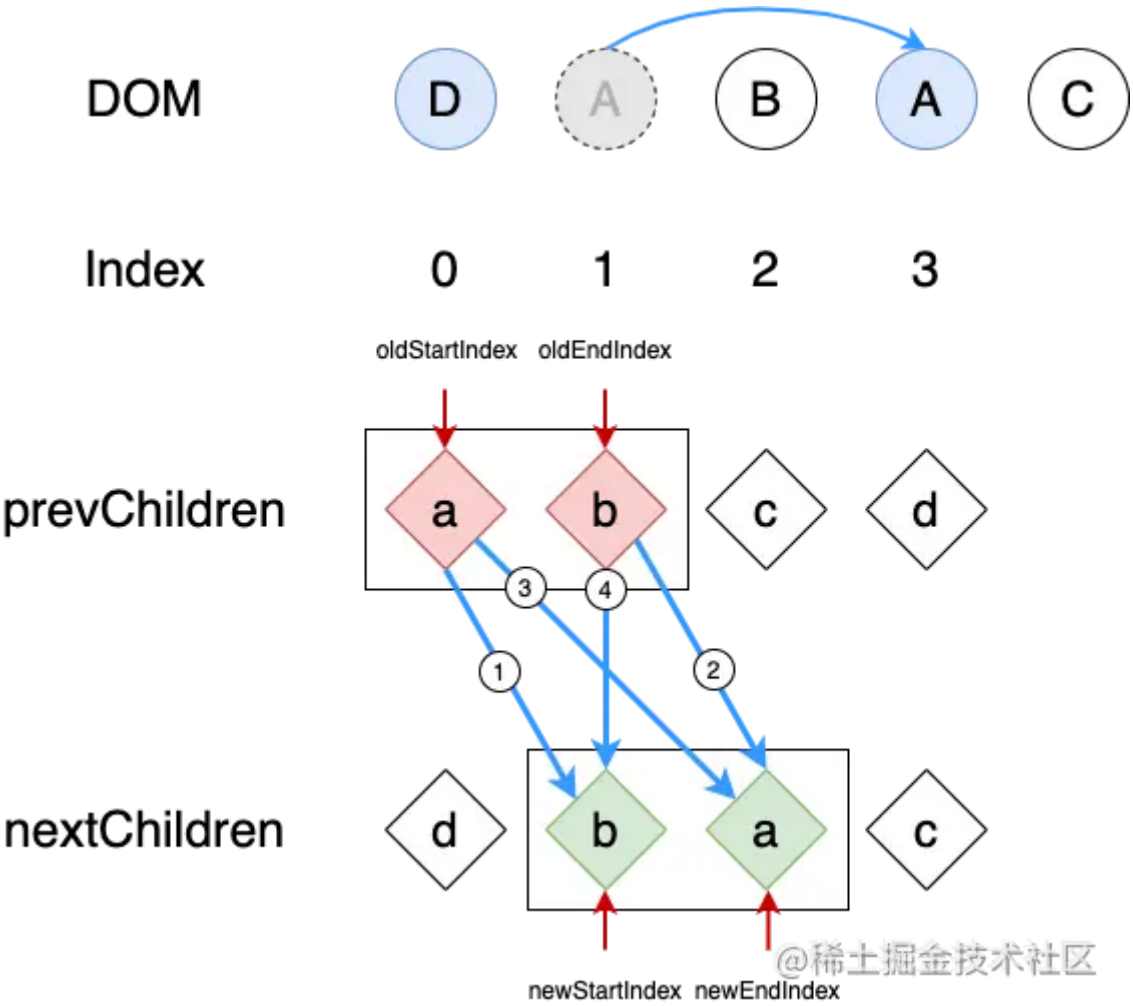
```
function vue2Diff(prevChildren, nextChildren, parent) {
```

```
    // ...
     while (oldStartIndex <= oldEndIndex && newStartIndex <= newEndIndex) {
       if (oldStartNode.key === newStartNode.key) {
          // ...
       } else if (oldEndNode.key === newEndNode.key) {
         // ...
       } else if (oldStartNode.key === newEndNode.key) {
         // ...
       } else if (oldEndNode.key === newStartNode.key) {
         patch(oldEndNode, newStartNode, parent)
         // 移动到旧列表头节点之前
         parent.insertBefore(oldEndNode.el, oldStartNode.el)

         oldEndIndex--
         newStartIndex++
         oldEndNode = prevChildren[oldEndIndex]
         newStartNode = nextChildren[newStartIndex]
       }
     }
   }
```



进入第二次循环，我们在第二步发现，旧列表的尾节点 `oldEndNode` 和新列表的尾节点 `newEndNode` 为复用节点。原本在旧列表中就是尾节点，在新列表中也是尾节点，说明该节点不需要移动，所以我们什么都不需要做。

同理，如果是旧列表的头节点 `oldStartNode` 和新列表的头节点 `newStartNode` 为复用节点，我们也什么都不需要做

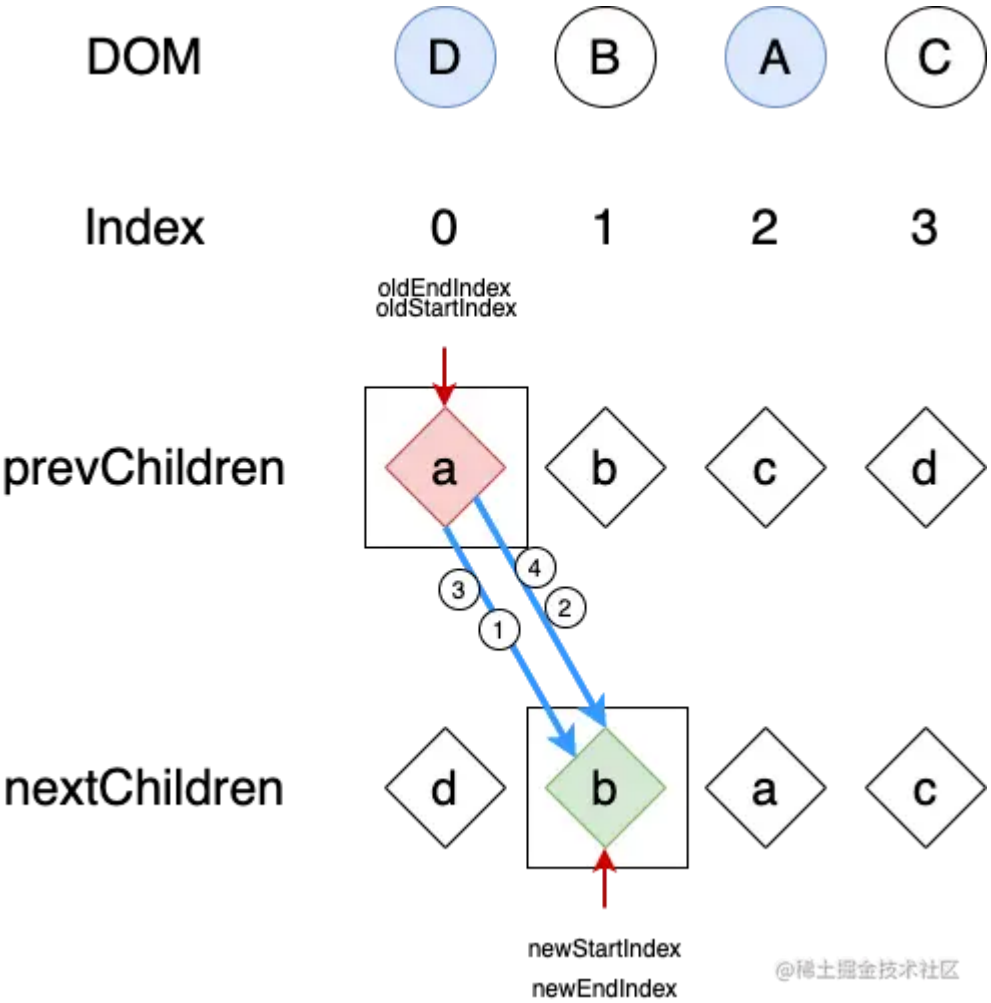


进入第三次循环，我们在第三部发现，旧列表的头节点 `oldStartNode` 和新列表的尾节点 `newEndNode` 为复用节点。，我们只要将DOM-A移动到DOM-B后面就可以了。

依照惯例我们还是解释一下，原本旧列表中是头节点，然后在新列表中是尾节点。那么只要在旧列表中把当前的节点移动到原本尾节点的后面，就可以了。

```
function vue2Diff(prevChildren, nextChildren, parent) {
  // ...
  while (oldStartIndex <= oldEndIndex && newStartIndex <= newEndIndex) {
    if (oldStartNode.key === newStartNode.key) {
      // ...
    } else if (oldEndNode.key === newEndNode.key) {
      // ...
    } else if (oldStartNode.key === newEndNode.key) {
      patch(oldStartNode, newEndNode, parent)
      parent.insertBefore(oldStartNode.el, oldEndNode.el.nextSibling)
```

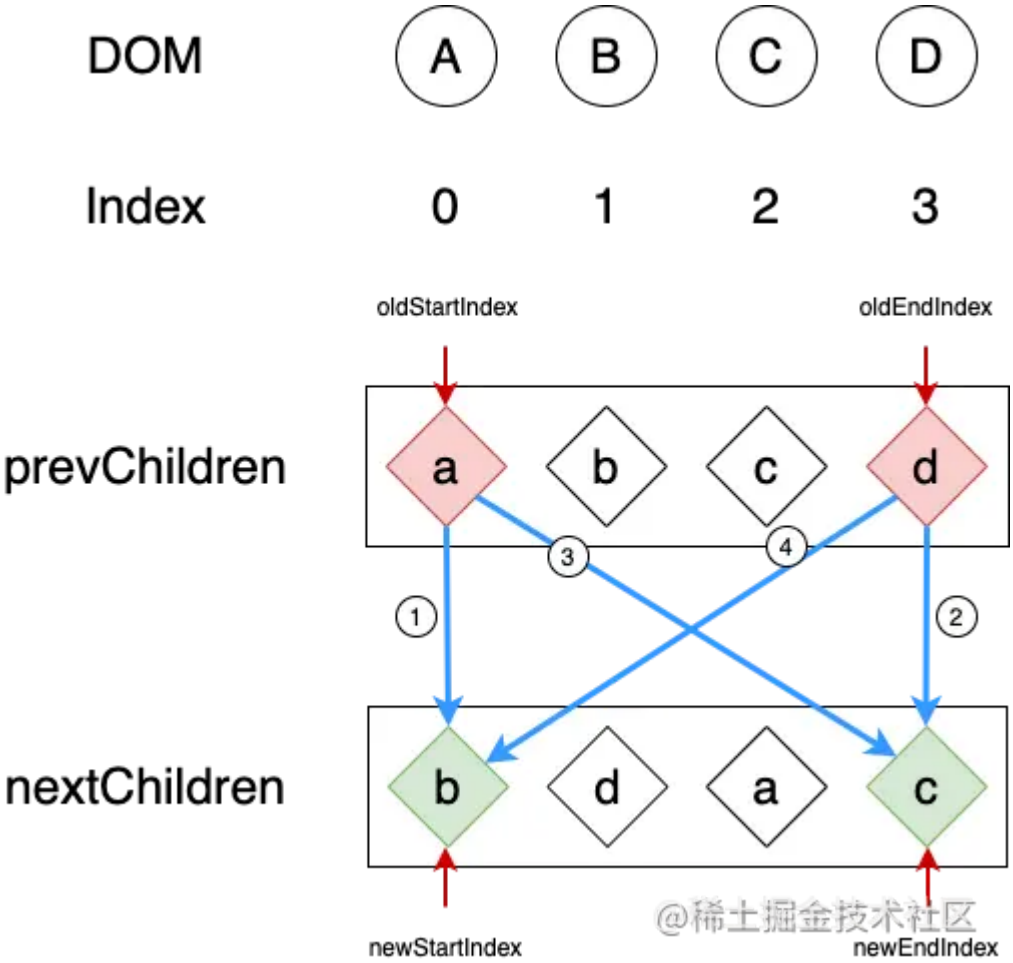
```
          oldStartIndex++
          newEndIndex--
          oldStartNode = prevChildren[oldStartIndex]
          newEndNode = nextChildren[newEndIndex]
        } else if (oldEndNode.key === newStartNode.key) {
         //...
        }
      }
    }
```



进入最后一个循环。在第一步旧列表头节点 `oldStartNode` 与新列表头节点 `newStartNode` 位置相同，所以啥也不用做。然后结束循环。

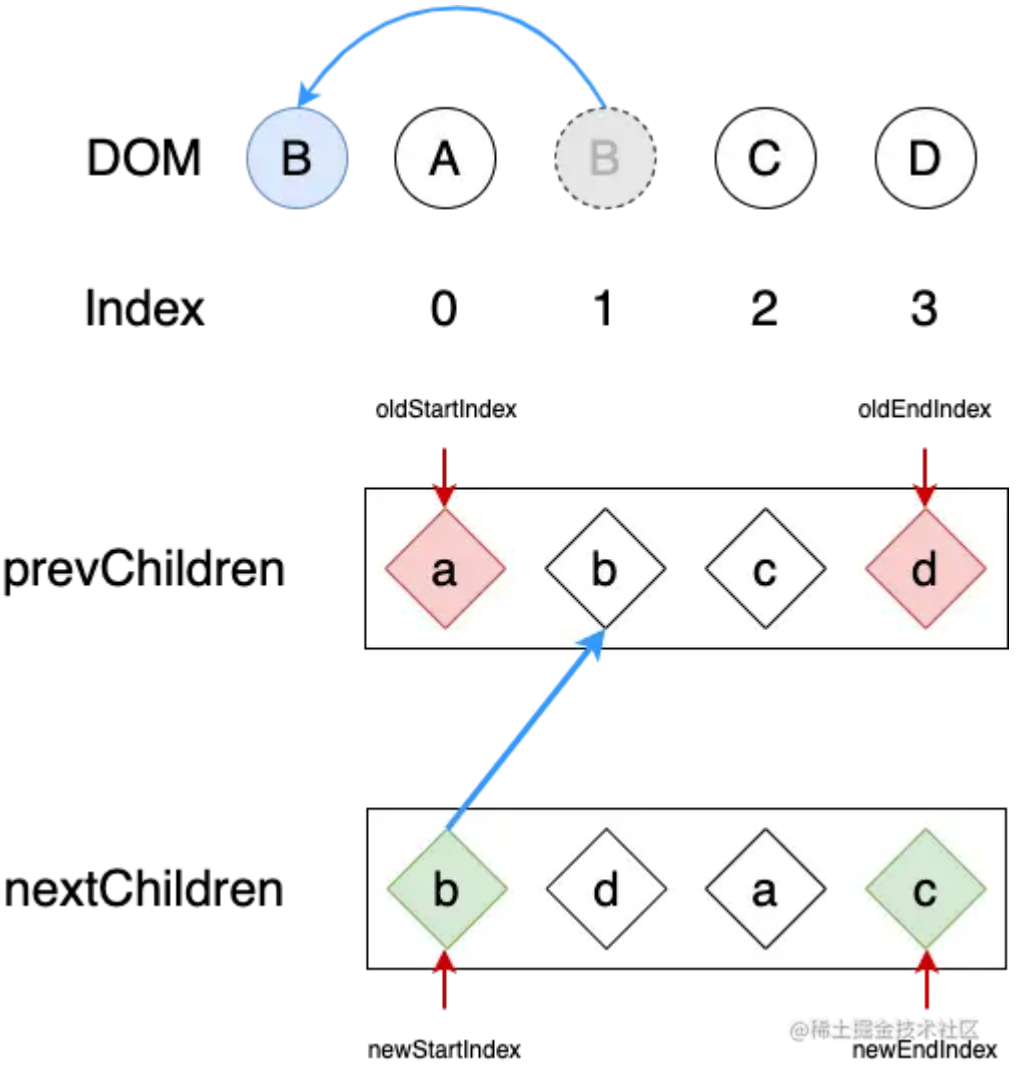
6.2.3.2 非理想情况

上文中有一个特殊情况，当四次对比都没找到复用节点时，我们只能拿新列表的第一个节点去旧列表中找与其key相同的节点。




```
function vue2Diff(prevChildren, nextChildren, parent) {
  //...
  while (oldStartIndex <= oldEndIndex && newStartIndex <= newEndIndex) {
    if (oldStartNode.key === newStartNode.key) {
    //...
    } else if (oldEndNode.key === newEndNode.key) {
    //...
    } else if (oldStartNode.key === newEndNode.key) {
    //...
    } else if (oldEndNode.key === newStartNode.key) {
    //...
    } else {
      // 在旧列表中找到 和新列表头节点key 相同的节点
      let newKey = newStartNode.key,
        oldIndex = prevChildren.findIndex(child => child.key === newKey);

    }
  }
}
```

找节点的时候其实会有两种情况：一种在旧列表中找到了，另一种情况是没找到。

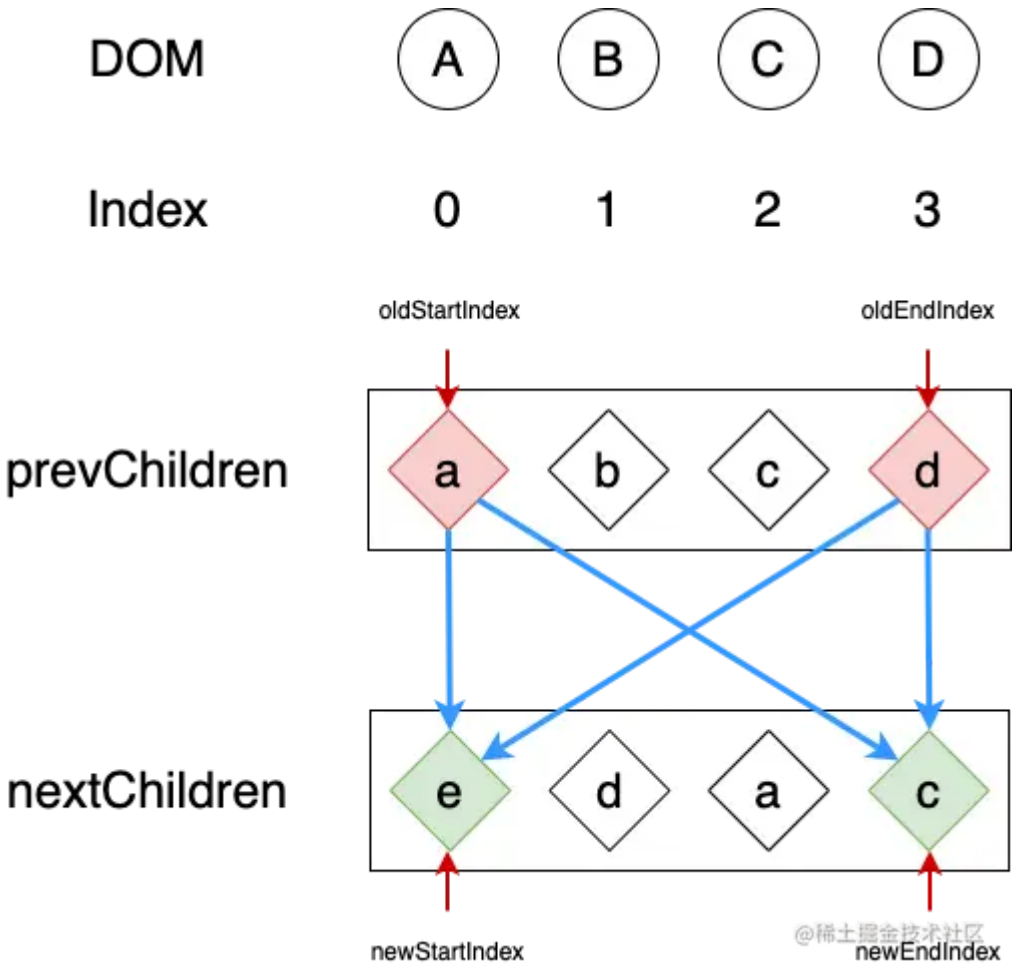


当我们在旧列表中找到对应的VNode，我们只需要将找到的节点的DOM元素，移动到开头就可以了。这里的逻辑其实和第四步的逻辑是一样的，只不过第四步是移动的尾节点，这里是移动找到的节点。DOM移动后，由我们将旧列表中的节点改为undefined，这是至关重要的一步，因为我们已经做了节点的移动了所以我们不需要进行再次的对比了。最后我们将头指针newStartIndex向后移一位。

```
function vue2Diff(prevChildren, nextChildren, parent) {
  //...
  while (oldStartIndex <= oldEndIndex && newStartIndex <= newEndIndex) {
    if (oldStartNode.key === newStartNode.key) {
    //...
    } else if (oldEndNode.key === newEndNode.key) {
    //...
    } else if (oldStartNode.key === newEndNode.key) {
    //...
    } else if (oldEndNode.key === newStartNode.key) {
    //...
    } else {
      // 在旧列表中找到 和新列表头节点key 相同的节点
      let newtKey = newStartNode.key,
        oldIndex = prevChildren.findIndex(child => child.key === newKey);
```

```
        if (oldIndex > -1) {
          let oldNode = prevChildren[oldIndex];
          patch(oldNode, newStartNode, parent)
          parent.insertBefore(oldNode.el, oldStartNode.el)
          prevChildren[oldIndex] = undefined
        }
        newStartNode = nextChildren[++newStartIndex]
      }
    }
  }
```

如果在旧列表中没有找到复用节点，就直接创建一个新的节点放到最前面就可以了，然后后移头指针 `newStartIndex` 。



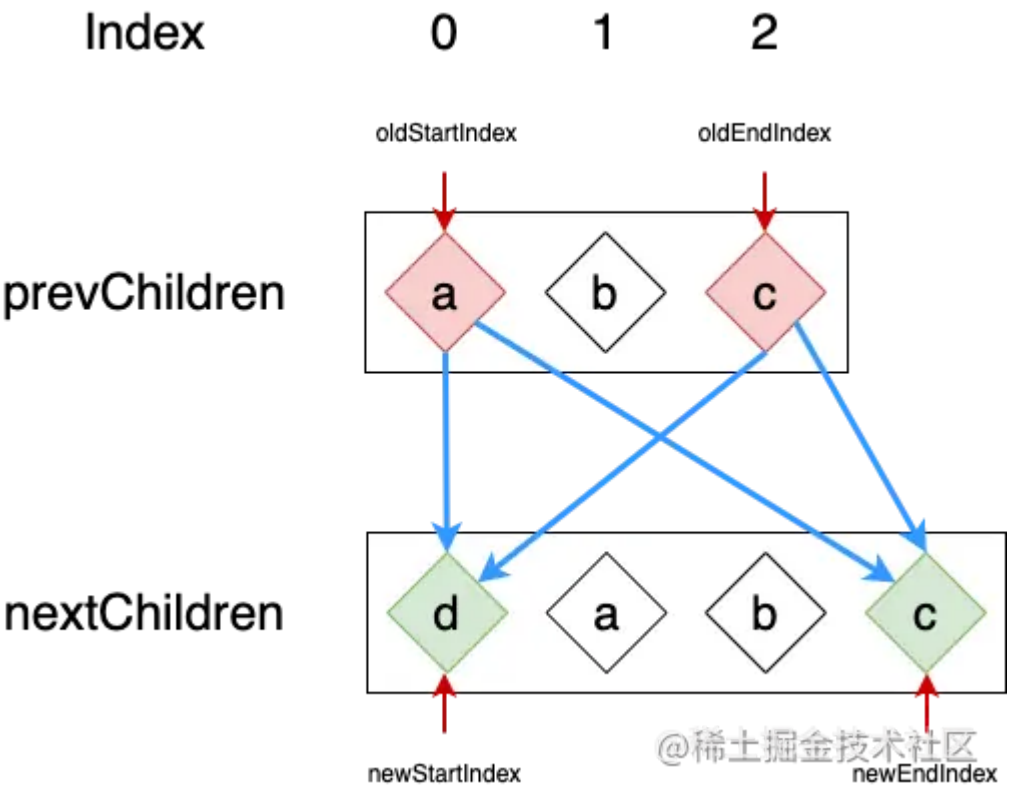

```
function vue2Diff(prevChildren, nextChildren, parent) {
  //...
  while (oldStartIndex <= oldEndIndex && newStartIndex <= newEndIndex) {
    if (oldStartNode.key === newStartNode.key) {
    //...
    } else if (oldEndNode.key === newEndNode.key) {
    //...
    } else if (oldStartNode.key === newEndNode.key) {
```

```
    //...
    } else if (oldEndNode.key === newStartNode.key) {
    //...
    } else {
      // 在旧列表中找到 和新列表头节点key 相同的节点
      let newtKey = newStartNode.key,
        oldIndex = prevChildren.findIndex(child => child.key === newKey);

      if (oldIndex > -1) {
        let oldNode = prevChildren[oldIndex];
        patch(oldNode, newStartNode, parent)
        parent.insertBefore(oldNode.el, oldStartNode.el)
        prevChildren[oldIndex] = undefined
      } else {
              mount(newStartNode, parent, oldStartNode.el)
      }
      newStartNode = nextChildren[++newStartIndex]
    }
  }
}
```

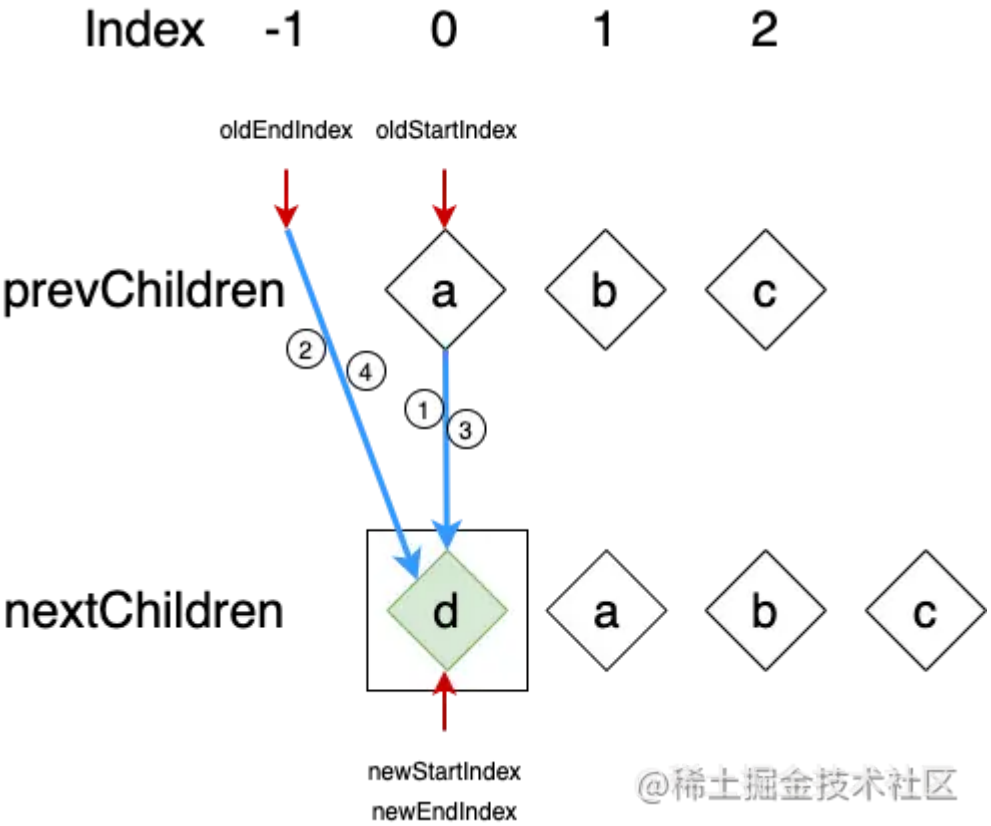
最后当旧列表遍历到undefind时就跳过当前节点。

```
function vue2Diff(prevChildren, nextChildren, parent) {
  //...
  while (oldStartIndex <= oldEndIndex && newStartIndex <= newEndIndex) {
    if (oldStartNode === undefind) {
            oldStartNode = prevChildren[++oldStartIndex]
    } else if (oldEndNode === undefind) {
            oldEndNode = prevChildren[--oldEndIndex]
    } else if (oldStartNode.key === newStartNode.key) {
    //...
    } else if (oldEndNode.key === newEndNode.key) {
    //...
    } else if (oldStartNode.key === newEndNode.key) {
    //...
    } else if (oldEndNode.key === newStartNode.key) {
    //...
    } else {
    // ...
    }
  }
}
```

6.2.3.3 添加节点



针对上述例子，几次循环都是尾节点相同，尾指针一直向前移动，直到循环结束；



此时 `oldEndIndex` 以及小于了 `oldStartIndex` ，但是新列表中还有剩余的节点，我们只需要将剩余的节点依次插入到 `oldStartNode` 的DOM之前就可以了。为什么是插入 `oldStartNode` 之前呢？原因是剩余的节点在新列表的位置是位于 `oldStartNode` 之前的，如果剩余节点是在 `oldStartNode` 之后， `oldStartNode` 就会先行对比

```
function vue2Diff(prevChildren, nextChildren, parent) {
  //...
  while (oldStartIndex <= oldEndIndex && newStartIndex <= newEndIndex) {
  // ...
  }
  if (oldEndIndex < oldStartIndex) {
    for (let i = newStartIndex; i <= newEndIndex; i++) {
      mount(nextChildren[i], parent, prevStartNode.el)
    }
  }
}
```

6.2.3.4 移除节点

当新列表的`newEndIndex`小于`newStartIndex`时，我们将旧列表剩余的节点删除即可。这里我们需要注意，旧列表的undefind。在第二小节中我们提到过，当头尾节点都不相同时，我们会去旧列表中找新列表的第一个节点，移动完DOM节点后，将旧列表的那个节点改为undefind。所以我们在最后的删除时，需要注意这些undefind，遇到的话跳过当前循环即可。

```
function vue2Diff(prevChildren, nextChildren, parent) {
  //...
  while (oldStartIndex <= oldEndIndex && newStartIndex <= newEndIndex) {
  // ...
  }
  if (oldEndIndex < oldStartIndex) {
    for (let i = newStartIndex; i <= newEndIndex; i++) {
      mount(nextChildren[i], parent, prevStartNode.el)
    }
  } else if (newEndIndex < newStartIndex) {
    for (let i = oldStartIndex; i <= oldEndIndex; i++) {
      if (prevChildren[i]) {
        partent.removeChild(prevChildren[i].el)
      }
    }
  }
}
```

6.2.3.5 总结

```
function vue2diff(prevChildren, nextChildren, parent) {
  let oldStartIndex = 0,
    newStartIndex = 0,
```

```
    oldStartIndex = prevChildren.length - 1,
    newStartIndex = nextChildren.length - 1,
    oldStartNode = prevChildren[oldStartIndex],
    oldEndNode = prevChildren[oldStartIndex],
    newStartNode = nextChildren[newStartIndex],
    newEndNode = nextChildren[newStartIndex];
  while (oldStartIndex <= oldStartIndex && newStartIndex <= newStartIndex) {
    if (oldStartNode === undefined) {
      oldStartNode = prevChildren[++oldStartIndex]
    } else if (oldEndNode === undefined) {
      oldEndNode = prevChildren[--oldStartIndex]
    } else if (oldStartNode.key === newStartNode.key) {
      patch(oldStartNode, newStartNode, parent)

      oldStartIndex++
      newStartIndex++
      oldStartNode = prevChildren[oldStartIndex]
      newStartNode = nextChildren[newStartIndex]
    } else if (oldEndNode.key === newEndNode.key) {
      patch(oldEndNode, newEndNode, parent)

      oldStartIndex--
      newStartIndex--
      oldEndNode = prevChildren[oldStartIndex]
      newEndNode = nextChildren[newStartIndex]
    } else if (oldStartNode.key === newEndNode.key) {
      patch(oldStartNode, newEndNode, parent)
      parent.insertBefore(oldStartNode.el, oldEndNode.el.nextSibling)
      oldStartIndex++
      newStartIndex--
      oldStartNode = prevChildren[oldStartIndex]
      newEndNode = nextChildren[newStartIndex]
    } else if (oldEndNode.key === newStartNode.key) {
      patch(oldEndNode, newStartNode, parent)
      parent.insertBefore(oldEndNode.el, oldStartNode.el)
      oldStartIndex--
      newStartIndex++
      oldEndNode = prevChildren[oldStartIndex]
      newStartNode = nextChildren[newStartIndex]
    } else {
      let newKey = newStartNode.key,
        oldIndex = prevChildren.findIndex(child => child && (child.key ===
newKey));
      if (oldIndex === -1) {
        mount(newStartNode, parent, oldStartNode.el)
      } else {
        let prevNode = prevChildren[oldIndex]
```

```
            patch(prevNode, newStartNode, parent)
            parent.insertBefore(prevNode.el, oldStartNode.el)
            prevChildren[oldIndex] = undefined
          }
          newStartIndex++
          newStartNode = nextChildren[newStartIndex]
        }
      }
      if (newStartIndex > newStartIndex) {
        while (oldStartIndex <= oldStartIndex) {
          if (!prevChildren[oldStartIndex]) {
            oldStartIndex++
            continue
          }
          parent.removeChild(prevChildren[oldStartIndex++].el)
        }
      } else if (oldStartIndex > oldStartIndex) {
        while (newStartIndex <= newStartIndex) {
          mount(nextChildren[newStartIndex++], parent, oldStartNode.el)
        }
      }
    }
```

6.3 缺点

Vue2 是全量 Diff（当数据发生变化，它就会新生成一个DOM树，并和之前的DOM树进行比较，找到不同的节点然后更新），如果层级很深，很消耗内存；