

Table of Contents

Introduction	1.1
Chapter1	1.2
源码解构	1.2.1
入口	1.2.1.1
Vue的定义	1.2.1.2
数据驱动	1.2.1.3
_render	1.2.1.3.1
Virtual Dom	1.2.1.3.2
createElement	1.2.1.3.3
_update	1.2.1.3.4
数据双向绑定	1.2.1.4
响应式对象	1.2.1.4.1
依赖收集Getter	1.2.1.4.2
派发更新Setter	1.2.1.4.3
nextTick	1.2.1.4.4
检测变化	1.2.1.4.5
计算属性和监听属性	1.2.1.4.6
组件更新	1.2.1.4.7

learning-vue

A collection of source code of Vue.js based on gitbook

Chapter2

Vue.js 源码分析

Vue 源码解构

参照vue-2.5.17 版本，解构一下vue的源码结构

源码解构

参照vue-2.5.17 版本，解构一下vue的源码结构

Entry

```

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

```

- compiler

目录包含 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。

- core

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。

- platform

跨平台的 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。

- 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。注意：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。

- 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

- 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。

找入口

Vue.js 构建过程，在 web 应用下，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：

```

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 {
  ...
  return mount.call(this, el, hydrating)
}

...
Vue.compile = compileToFunctions

export default Vue

```

其中有一行，

```
import Vue from './runtime/index'
```

可以看出 Vue 的定义来源于， runtime/index.js.

```

// runtime/index.js

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, isChrome } 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)
}

if(inBrowser) {...}

export default Vue

```

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
```

有两处：

- 第一处 Vue的引入

```
import Vue from './instance/index'
```

- 初始化Vue的全局API

```
initGlobalAPI(Vue)
```

源码解构

参照vue-2.5.17 版本，解构一下vue的源码结构

Vue的定义

```
// src/core/instance/index.js

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'

// new Vue 实例化一个对象
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
```

这里看出，**Vue** 它实际上就是一个用 **Function** 实现的类，我们只能通过 **new Vue** 去实例化它。

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。

```
// core/global-api/index.js
/* @flow */

import config from '../config'
import { initUse } from './use'
import { initMixin } from './mixin'
import { initExtend } from './extend'
import { initAssetRegisters } from './assets'
import { set, del } from '../observer/index'
```

```

import { ASSET_TYPES } from 'shared/constants'
import builtInComponents from '../components/index'

import {
  warn,
  extend,
  nextTick,
  mergeOptions,
  defineReactive
} from '../util/index'

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 // set 方法
  Vue.delete = del // delete 方法
  Vue.nextTick = nextTick // nextTick 方法

  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。

源码解构

参照vue-2.5.17 版本，解构一下vue的源码结构

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。

new Vue() - init

new 在JS中是用于实例化一个对象，而Vue本质就是一个Function

src/core/instance/index.js 中，可以看出：

```

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'

// new Vue 实例化一个对象
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

```

可以看出，new之后，会调用**this._init(options);** ('src/core/instance/init.js'中)

在init.js中可以看出，主线逻辑是很清晰的。

```

export function initMixin (Vue: Class<Component>) {
  // 类型判断，options可以是不存在或 Object 类型;

```

```

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 合并配置
  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) // 初始化生命周期
  initEvents(vm) // 初始化事件中心
  initRender(vm) // 初始化render函数
  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)
  }
}

```

```
// 在初始化的最后，检测到如果有 el 属性，则调用 vm.$mount 方法挂载 vm，挂载的目标就是把模板渲染成最终的 DOM,
if (vm.$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。

见：

```
// expose real self
vm._self = vm
initLifecycle(vm) // 初始化生命周期
initEvents(vm) // 初始化事件中心
initRender(vm) // 初始化render函数
callHook(vm, 'beforeCreate')
initInjections(vm) // resolve injections before data/props
initState(vm)
initProvide(vm) // resolve provide after data/props
callHook(vm, 'created')
```

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。

Vue实例的挂载 - \$mount

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。

- 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。

这里看下Web端的，entry-runtime-with-compiler.js

```
// 这段代码首先缓存了原型上的 $mount 方法，再重新定义该方法，
const mount = Vue.prototype.$mount
Vue.prototype.$mount = function (
  el?: string | Element,
  hydrating?: boolean
): Component {
  el = el && query(el)

  /* istanbul ignore if */
```

```

// 如果是body或者html， 不去挂载el
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
// 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
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, {
      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')
}
}

// 最后，调用原先原型上的 $mount 方法挂载。
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
- 它对 el 做了限制，Vue 不能挂载在 body、html 这样的根节点上
- 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，那么这个过程是 Vue 的一个“在线编译”的过程，它是调用 compileToFunctions 方法实现的，编译过程我们之后会介绍。

- 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 中调用 mountComponent，是在这里定义"import { mountComponent } from 'core/instance/lifecycle'"

```

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)
    /***** 先调用 vm._render 方法先生成虚拟 Node *****/
    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 = () => {
    /***** 最后执行 _update方法 *****/
    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

***** 实例化一个渲染Watcher, 在它的回调函数中会调用 updateComponent 方法 *****/
// Watcher 在这里起到两个作用, 一个是初始化的时候会执行回调函数, 另一个是当 vm 实例中的监测的数据发生变化的时候执行回调函数
new Watcher(vm, updateComponent, noop, null, true /* isRenderWatcher */)
hydrating = false

// manually mounted instance, call mounted on self
// mounted is called for render-created child components in its inserted hook
// 这里注意 vm.$vnode 表示 Vue 实例的父虚拟 Node, 所以它为 Null 则表示当前是根 Vue 的实例
if (vm.$vnode == null) {
    // 已经挂载成功
    vm._isMounted = true
    // 同时执行 mounted 钩子函数。
    callHook(vm, 'mounted')
}
return vm
}
```

mountComponent的核心思路如下：

- 先调用 `vm._render` 方法先生成虚拟 Node
- 再实例化一个渲染Watcher, 在它的回调函数中会调用 `updateComponent` 方法
- 最终调用 `vm._update` 更新 DOM

综上, mountComponent的核心就两个方法: **_render和_update**

vm._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

render.js的结构

```
export function initRender (vm: Component) {...}

export function renderMixin (Vue: Class<Component>) {
    // install runtime convenience helpers
    installRenderHelpers(Vue.prototype)

    Vue.prototype.$nextTick = function (fn: Function) {
        return nextTick(fn, this)
    }

    // _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。
    // 在之前的 mounted 方法的实现中，会把 template 编译成 render 方法，但这个编译过程是非常复杂的
    Vue.prototype._render = function (): VNode {
        const vm: Component = this
        const { render, _parentVnode } = vm.$options

        ...

        // set parent
        vnode.parent = _parentVnode
        return vnode
    }
}
```

_render被挂在原型上：

```
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
}

```

在 Vue 的官方文档中介绍了 render 函数的第一个参数是 createElement

```

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

```

用render相当于：

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

再回到 _render 函数中的 render 方法的调用：

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

vm.\$createElement可以在initRender中看到：

```
import { createElement } from '../vdom/create-element'

export function initRender(vm: Component) {
  vm._vnode = null // the root of the child tree
  vm._staticTrees = null // v-once cached trees
  const options = vm.$options
  const parentVnode = vm.$vnode = options._parentVnode // the placeholder node in parent tree
  const renderContext = parentVnode && parentVnode.context
  vm.$slots = resolveSlots(options._renderChildren, renderContext)
  vm.$scopedSlots = emptyObject
  // bind the createElement fn to this instance
  // so that we get proper render context inside it.
  // args order: tag, data, children, normalizationType, alwaysNormalize
  // internal version is used by render functions compiled from templates
  vm._c = (a, b, c, d) => createElement(vm, a, b, c, d, false)
  // normalization is always applied for the public version, used in
  // user-written render functions.
  vm.$createElement = (a, b, c, d) => createElement(vm, a, b, c, d, true)
}
```

可以看到除了 vm.\$createElement 方法，还有一个 vm._c 方法，它是被模板编译成的 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。

总结

vm._render 最终是通过执行 createElement 方法并返回的是 vnode，它是一个虚拟 Node。

Virtual Dom

`createElement`返回的就是一个VNode，了解VNode需要先看下Virtual Dom

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。

```
/* @flow */

export default class VNode {
  tag: string | void; // 标签
  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.componentInstance = 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
}
}

export const createEmptyVNode = (text: string = '') => {
    const node = new VNode()
    node.text = text
    node.isComment = true
    return node
}

export function createTextVNode (val: string | number) {
    return new VNode(undefined, undefined, undefined, String(val))
}

// optimized shallow clone
// used for static nodes and slot nodes because they may be reused across
// multiple renders, cloning them avoids errors when DOM manipulations rely
// on their elm reference.
export function cloneVNode (vnode: VNode): VNode {
    const cloned = new VNode(
        vnode.tag,
        vnode.data,
        vnode.children,

```

```
vnode.text,  
vnode.elm,  
vnode.context,  
vnode.componentOptions,  
vnode.asyncFactory  
)  
cloned.ns = vnode.ns  
cloned.isStatic = vnode.isStatic  
cloned.key = vnode.key  
cloned.isComment = vnode.isComment  
cloned.fnContext = vnode.fnContext  
cloned.fnOptions = vnode.fnOptions  
cloned.fnScopeId = vnode.fnScopeId  
cloned.isCloned = true  
return cloned  
}
```

总结

其实 VNode 是对真实 DOM 的一种抽象描述，它的核心定义无非就几个关键属性，标签名、数据、子节点、键值等，其它属性都是用来扩展 VNode 的灵活性以及实现一些特殊 feature 的。

由于 VNode 只是用来映射到真实 DOM 的渲染，不需要包含操作 DOM 的方法，因此它是非常轻量和简单的。

Virtual DOM 除了它的数据结构的定义，映射到真实的 DOM 实际上要经历 VNode 的 create、diff、patch 等过程。

createElement

createElement返回的就是一个VNode

它定义在 src/core/vdom/create-element.js 中

方法实际上是对 _createElement 方法的封装，它允许传入的参数更加灵活，在处理这些参数后，调用真正创建 VNode 的函数 _createElement

```
// wrapper function for providing a more flexible interface
// without getting yelled at by flow
// 对 _createElement 方法的封装，它允许传入的参数更加灵活，在处理这些参数后，调用真正创建 VNode
// 的函数 _createElement
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)
}
```

_createElement的源码:

```
// _createElement 5 个参数
// context 表示 VNode 的上下文环境，它是 Component 类型；
// tag 表示标签，它可以是一个字符串，也可以是一个 Component；
// 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；
// 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；
// 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。

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 (!__WEEEX__ || !('@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

  // 这里先对 tag 做判断，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 V
  Node,
  if (typeof tag === 'string') {
    let Ctor
    ns = (context.$vnode && context.$vnode.ns) || config.getTagNamespace(tag)
    if (config.isReservedTag(tag)) {
      // platform built-in elements
    }
  }
}

```

```

vnode = new VNode(
  config.parsePlatformTagName(tag), data, children,
  undefined, undefined, context
)
} else if (isDef(Ctor = resolveAsset(context.$options, 'components', tag))) {
  // component
  // 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。
  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
  // 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
  .
  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()
}
}

```

如上：

_createElement两个重要流程，children规范化，VNode创建。

接受5个参数：

- context 表示 VNode 的上下文环境，它是 Component 类型；
- tag 表示标签，它可以是一个字符串，也可以是一个 Component；
- 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
- 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；
- 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。

规范化

normalizeChildren 方法的调用场景有 2 种，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；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

VNode 创建

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/normalize-children.js 中：

- 先对 tag 做判断，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

```
if (typeof tag === 'string') { }
```

- 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。
- 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。

总结

vm._render 是创建了一个 VNode，接下来就是要把这个 VNode 渲染成一个真实的 DOM 并渲染出来，这个过程是通过 vm._update 完成的

_update

_update，用于把VNode渲染成一个真实的DOM并输出，它在src/core/instance/lifecycle.js中。它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。

```
// _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
Vue.prototype._update = function (vnode: VNode, hydrating?: boolean) {
  const vm: Component = this
  if (vm._isMounted) {
    callHook(vm, 'beforeUpdate')
  }
  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.$el, vnode, hydrating, false /* removeOnly */,
      vm.$options._parentElm,
      vm.$options._refElm
    )
    // no need for the ref nodes after initial patch
    // this prevents keeping a detached DOM tree in memory (#5851)
    vm.$options._parentElm = vm.$options._refElm = null
  } else {
    // updates
    vm.$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.
}
```

vm.patch 定义在 src/platforms/web/runtime/index.js 中

```
import { patch } from './patch'

...

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

甚至在 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，也就是从 "./patch" 导入的。

```
/* @flow */

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)

// 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。
// 1. nodeOps 封装了一系列 DOM 操作的方法
// 2. modules 定义了一些模块的钩子函数的实现
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。

- 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

然鹅，createPatchFunction 又是在 src/core/vdom/patch.js 中定义的。

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

function sameVnode (a, b) {
  return (
    a.key === b.key &&
    (
      a.tag === b.tag &&
      a.isComment === b.isComment &&
      isDef(a.data) === isDef(b.data) &&
```

```

        sameInputType(a, b)
    ) || (
        isTrue(a.isAsyncPlaceholder) &&
        a.asyncFactory === b.asyncFactory &&
        isUndef(b.asyncFactory.error)
    )
)
)
}

function sameInputType (a, b) {
    if (a.tag !== 'input') return true
    let i
    const typeA = isDef(i = a.data) && isDef(i = i.attrs) && i.type
    const typeB = isDef(i = b.data) && isDef(i = i.attrs) && i.type
    return typeA === typeB || isTextInputType(typeA) && isTextInputType(typeB)
}

function createKeyToOldIdx (children, beginIdx, endIdx) {
    let i, key
    const map = {}
    for (i = beginIdx; i <= endIdx; ++i) {
        key = children[i].key
        if (isDef(key)) map[key] = i
    }
    return map
}

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]])
            }
        }
    }

    function emptyNodeAt (elm) {
        return new VNode(nodeOps.tagName(elm).toLowerCase(), {}, [], undefined, elm)
    }

    function createRmCb (childElm, listeners) {
        function remove () {
            if (--remove.listeners === 0) {
                removeNode(childElm)
            }
        }
        remove.listeners = listeners
        return remove
    }
}

```

```

        }
    }
    remove.listeners = listeners
    return remove
}

function removeNode (el) {
    const parent = nodeOps.parentNode(el)
    // element may have already been removed due to v-html / v-text
    if (isDef(parent)) {
        nodeOps.removeChild(parent, el)
    }
}

function isUnknownElement (vnode, inVPre) {
    return (
        !inVPre &&
        !vnode.ns &&
        !((
            config.ignoredElements.length &&
            config.ignoredElements.some(ignore => {
                return isRegExp(ignore)
                ? ignore.test(vnode.tag)
                : ignore === vnode.tag
            })
        ) &&
        config.isUnknownElement(vnode.tag)
    )
}

let creatingElmInVPre = 0

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 (__WEEEX__) {
  // in Weex, the default insertion order is parent-first.
  // List items can be optimized to use children-first insertion
  // with append="tree".
  const appendAsTree = isDef(data) && isTrue(data.appendAsTree)
  if (!appendAsTree) {
    if (isDef(data)) {
      invokeCreateHooks(vnode, insertedVnodeQueue)
    }
    insert(parentElm, vnode.elm, refElm)
  }
  createChildren(vnode, children, insertedVnodeQueue)
  if (appendAsTree) {
    if (isDef(data)) {
      invokeCreateHooks(vnode, insertedVnodeQueue)
    }
    insert(parentElm, vnode.elm, refElm)
  }
} 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)
}
}

function createComponent (vnode, insertedVnodeQueue, parentElm, refElm) {
let i = vnode.data
if (isDef(i)) {
    const isReactivated = isDef(vnode.componentInstance) && i.keepAlive
    if (isDef(i = i.hook) && isDef(i = i.init)) {
        i(vnode, false /* hydrating */, parentElm, refElm)
    }
    // after calling the init hook, if the vnode is a child component
    // it should've created a child instance and mounted it. the child
    // component also has set the placeholder vnode's elm.
    // in that case we can just return the element and be done.
    if (isDef(vnode.componentInstance)) {
        initComponent(vnode, insertedVnodeQueue)
        if (isTrue(isReactivated)) {
            reactivateComponent(vnode, insertedVnodeQueue, parentElm, refElm)
        }
        return true
    }
}
}

function initComponent (vnode, insertedVnodeQueue) {
if (isDef(vnode.data.pendingInsert)) {
    insertedVnodeQueue.push.apply(insertedVnodeQueue, vnode.data.pendingInsert)
    vnode.data.pendingInsert = null
}
vnode.elm = vnode.componentInstance.$el
if (isPatchable(vnode)) {
    invokeCreateHooks(vnode, insertedVnodeQueue)
    setScope(vnode)
} else {
    // empty component root.
    // skip all element-related modules except for ref (#3455)
    registerRef(vnode)
    // make sure to invoke the insert hook
    insertedVnodeQueue.push(vnode)
}
}

```

```

        }

    }

    function reactivateComponent (vnode, insertedVnodeQueue, parentElm, refElm) {
        let i
        // hack for #4339: a reactivated component with inner transition
        // does not trigger because the inner node's created hooks are not called
        // again. It's not ideal to involve module-specific logic in here but
        // there doesn't seem to be a better way to do it.
        let innerNode = vnode
        while (innerNode.componentInstance) {
            innerNode = innerNode.componentInstance._vnode
            if (isDef(i = innerNode.data) && isDef(i = i.transition)) {
                for (i = 0; i < cbs.activate.length; ++i) {
                    cbs.activate[i](emptyNode, innerNode)
                }
                insertedVnodeQueue.push(innerNode)
                break
            }
        }
        // unlike a newly created component,
        // a reactivated keep-alive component doesn't insert itself
        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)
            }
        }
    }

    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)))
        }
    }

    function isPatchable (vnode) {

```

```

        while (vnode.componentInstance) {
            vnode = vnode.componentInstance._vnode
        }
        return isDef(vnode.tag)
    }

    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)
        }
    }

    // set scope id attribute for scoped CSS.
    // this is implemented as a special case to avoid the overhead
    // of going through the normal attribute patching process.
    function setScope (vnode) {
        let i
        if (isDef(i = vnode.fnScopeId)) {
            nodeOps.setStyleScope(vnode.elm, i)
        } else {
            let ancestor = vnode
            while (ancestor) {
                if (isDef(i = ancestor.context) && isDef(i = i.$options._scopeId)) {
                    nodeOps.setStyleScope(vnode.elm, i)
                }
                ancestor = ancestor.parent
            }
        }
        // for slot content they should also get the scopeId from the host instance.
        if (isDef(i = activeInstance) &&
            i !== vnode.context &&
            i !== vnode.fnContext &&
            isDef(i = i.$options._scopeId))
        ) {
            nodeOps.setStyleScope(vnode.elm, i)
        }
    }

    function addVnodes (parentElm, refElm, vnodes, startIdx, endIdx, insertedVnodeQueue) {
        for (; startIdx <= endIdx; ++startIdx) {
            createElm(vnodes[startIdx], insertedVnodeQueue, parentElm, refElm, false, vnodes, startIdx)
        }
    }

    function invokeDestroyHook (vnode) {

```

```

let i, j
const data = vnode.data
if (isDef(data)) {
  if (isDef(i = data.hook) && isDef(i = i.destroy)) i(vnode)
  for (i = 0; i < cbs.destroy.length; ++i) cbs.destroy[i](vnode)
}
if (isDef(i = vnode.children)) {
  for (j = 0; j < vnode.children.length; ++j) {
    invokeDestroyHook(vnode.children[j])
  }
}
}

function removeVnodes (parentElm, vnodes, startIdx, endIdx) {
  for (; startIdx <= endIdx; ++startIdx) {
    const ch = vnodes[startIdx]
    if (isDef(ch)) {
      if (isDef(ch.tag)) {
        removeAndInvokeRemoveHook(ch)
        invokeDestroyHook(ch)
      } else { // Text node
        removeNode(ch.elm)
      }
    }
  }
}

function removeAndInvokeRemoveHook (vnode, rm) {
  if (isDef(rm) || isDef(vnode.data)) {
    let i
    const listeners = cbs.remove.length + 1
    if (isDef(rm)) {
      // we have a recursively passed down rm callback
      // increase the listeners count
      rm.listeners += listeners
    } else {
      // directly removing
      rm = createRmCb(vnode.elm, listeners)
    }
    // recursively invoke hooks on child component root node
    if (isDef(i = vnode.componentInstance) && isDef(i = i._vnode) && isDef(i.data)) {
      removeAndInvokeRemoveHook(i, rm)
    }
    for (i = 0; i < cbs.remove.length; ++i) {
      cbs.remove[i](vnode, rm)
    }
    if (isDef(i = vnode.data.hook) && isDef(i = i.remove)) {
      i(vnode, rm)
    } else {
      rm()
    }
  }
}

```

```

    } else {
      removeNode(vnode.elm)
    }
}

function updateChildren (parentElm, oldCh, newCh, insertedVnodeQueue, removeOnly) {
  let oldStartIdx = 0
  let newStartIdx = 0
  let oldEndIdx = oldCh.length - 1
  let oldStartVnode = oldCh[0]
  let oldEndVnode = oldCh[oldEndIdx]
  let newEndIdx = newCh.length - 1
  let newStartVnode = newCh[0]
  let newEndVnode = newCh[newEndIdx]
  let oldKeyToIdx, idxInOld, vnodeToMove, refElm

  // removeOnly is a special flag used only by <transition-group>
  // to ensure removed elements stay in correct relative positions
  // during leaving transitions
  const canMove = !removeOnly

  if (process.env.NODE_ENV !== 'production') {
    checkDuplicateKeys(newCh)
  }

  while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
    if (isUndef(oldStartVnode)) {
      oldStartVnode = oldCh[++oldStartIdx] // Vnode has been moved left
    } else if (isUndef(oldEndVnode)) {
      oldEndVnode = oldCh[--oldEndIdx]
    } else if (sameVnode(oldStartVnode, newStartVnode)) {
      patchVnode(oldStartVnode, newStartVnode, insertedVnodeQueue)
      oldStartVnode = oldCh[++oldStartIdx]
      newStartVnode = newCh[++newStartIdx]
    } else if (sameVnode(oldEndVnode, newEndVnode)) {
      patchVnode(oldEndVnode, newEndVnode, insertedVnodeQueue)
      oldEndVnode = oldCh[--oldEndIdx]
      newEndVnode = newCh[--newEndIdx]
    } else if (sameVnode(oldStartVnode, newEndVnode)) { // Vnode moved right
      patchVnode(oldStartVnode, newEndVnode, insertedVnodeQueue)
      canMove && nodeOps.insertBefore(parentElm, oldStartVnode.elm, nodeOps.nextSibling(
        oldEndVnode.elm))
      oldStartVnode = oldCh[++oldStartIdx]
      newEndVnode = newCh[--newEndIdx]
    } else if (sameVnode(oldEndVnode, newStartVnode)) { // Vnode moved left
      patchVnode(oldEndVnode, newStartVnode, insertedVnodeQueue)
      canMove && nodeOps.insertBefore(parentElm, oldEndVnode.elm, oldStartVnode.elm)
      oldEndVnode = oldCh[--oldEndIdx]
      newStartVnode = newCh[++newStartIdx]
    } else {
      if (isUndef(oldKeyToIdx)) oldKeyToIdx = createKeyToOldIdx(oldCh, oldStartIdx, oldE

```

```

    ndIdx)
      idxInOld = isDef(newStartVnode.key)
        ? oldKeyToIdx[newStartVnode.key]
        : findIdxInOld(newStartVnode, oldCh, oldStartIdx, oldEndIdx)
      if (isUndef(idxInOld)) { // New element
        createElm(newStartVnode, insertedVnodeQueue, parentElm, oldStartVnode.elm, false
, newCh, newStartIdx)
      } else {
        vnodeToMove = oldCh[idxInOld]
        if (sameVnode(vnodeToMove, newStartVnode)) {
          patchVnode(vnodeToMove, newStartVnode, insertedVnodeQueue)
          oldCh[idxInOld] = undefined
          canMove && nodeOps.insertBefore(parentElm, vnodeToMove.elm, oldStartVnode.elm)
        } else {
          // same key but different element. treat as new element
          createElm(newStartVnode, insertedVnodeQueue, parentElm, oldStartVnode.elm, fal
se, newCh, newStartIdx)
        }
      }
      newStartVnode = newCh[++newStartIdx]
    }
  }
  if (oldStartIdx > oldEndIdx) {
    refElm = isUndef(newCh[newEndIdx + 1]) ? null : newCh[newEndIdx + 1].elm
    addVnodes(parentElm, refElm, newCh, newStartIdx, newEndIdx, insertedVnodeQueue)
  } else if (newStartIdx > newEndIdx) {
    removeVnodes(parentElm, oldCh, oldStartIdx, oldEndIdx)
  }
}

function checkDuplicateKeys (children) {
  const seenKeys = {}
  for (let i = 0; i < children.length; i++) {
    const vnode = children[i]
    const key = vnode.key
    if (isDef(key)) {
      if (seenKeys[key]) {
        warn(
          `Duplicate keys detected: '${key}'. This may cause an update error.`,
          vnode.context
        )
      } else {
        seenKeys[key] = true
      }
    }
  }
}

function findIdxInOld (node, oldCh, start, end) {
  for (let i = start; i < end; i++) {
    const c = oldCh[i]
  }
}

```

```

        if (isDef(c) && sameVnode(node, c)) return i
    }
}

function patchVnode (oldVnode, vnode, insertedVnodeQueue, removeOnly) {
  if (oldVnode === vnode) {
    return
  }

  const elm = vnode.elm = oldVnode.elm

  if (isTrue(oldVnode.isAsyncPlaceholder)) {
    if (isDef(vnode.asyncFactory.resolved)) {
      hydrate(oldVnode.elm, vnode, insertedVnodeQueue)
    } else {
      vnode.isAsyncPlaceholder = true
    }
    return
  }

  // reuse element for static trees.
  // note we only do this if the vnode is cloned -
  // if the new node is not cloned it means the render functions have been
  // reset by the hot-reload-api and we need to do a proper re-render.
  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)
  }
  if (isUndef(vnode.text)) {
    if (isDef(oldCh) && isDef(ch)) {
      if (oldCh !== ch) updateChildren(elm, oldCh, ch, insertedVnodeQueue, removeOnly)
    } else if (isDef(ch)) {
      if (isDef(oldVnode.text)) nodeOps.setTextContent(elm, '')
      addVnodes(elm, null, ch, 0, ch.length - 1, insertedVnodeQueue)
    }
  }
}

```

```

    } else if (isDef(oldCh)) {
      removeVnodes(elm, oldCh, 0, oldCh.length - 1)
    } else if (isDef(oldVnode.text)) {
      nodeOps.setTextContent(elm, '')
    }
  } 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)
  }
}

function invokeInsertHook (vnode, queue, initial) {
  // delay insert hooks for component root nodes, invoke them after the
  // element is really inserted
  if (isTrue(initial) && isDef(vnode.parent)) {
    vnode.parent.data.pendingInsert = queue
  } else {
    for (let i = 0; i < queue.length; ++i) {
      queue[i].data.hook.insert(queue[i])
    }
  }
}

let hydrationBailed = false
// list of modules that can skip create hook during hydration because they
// are already rendered on the client or has no need for initialization
// Note: style is excluded because it relies on initial clone for future
// deep updates (#7063).
const isRenderedModule = makeMap('attrs,class,staticClass,staticStyle,key')

// Note: this is a browser-only function so we can assume elms are DOM nodes.
function hydrate (elm, vnode, insertedVnodeQueue, inVPre) {
  let i
  const { tag, data, children } = vnode
  inVPre = inVPre || (data && data.pre)
  vnode.elm = elm

  if (isTrue(vnode.isComment) && isDef(vnode.asyncFactory)) {
    vnode.isAsyncPlaceholder = true
    return true
  }
  // assert node match
  if (process.env.NODE_ENV !== 'production') {
    if (!assertNodeMatch(elm, vnode, inVPre)) {
      return false
    }
  }
  if (isDef(data)) {
    if (isDef(i = data.hook) && isDef(i = i.init)) i(vnode, true /* hydrating */)
  }
}

```

```

        if (isDef(i = vnode.componentInstance)) {
          // child component. it should have hydrated its own tree.
          initComponent(vnode, insertedVnodeQueue)
          return true
        }
      }
      if (isDef(tag)) {
        if (isDef(children)) {
          // empty element, allow client to pick up and populate children
          if (!elm.childNodes()) {
            createChildren(vnode, children, insertedVnodeQueue)
          } else {
            // v-html and domProps: innerHTML
            if (isDef(i = data) && isDef(i = i.domProps) && isDef(i = i.innerHTML)) {
              if (i !== elm.innerHTML) {
                /* istanbul ignore if */
                if (process.env.NODE_ENV !== 'production' &&
                  typeof console !== 'undefined' &&
                  !hydrationBailed
                ) {
                  hydrationBailed = true
                  console.warn('Parent: ', elm)
                  console.warn('server innerHTML: ', i)
                  console.warn('client innerHTML: ', elm.innerHTML)
                }
              }
              return false
            }
          }
        } else {
          // iterate and compare children lists
          let childrenMatch = true
          let childNode = elm.firstChild
          for (let i = 0; i < children.length; i++) {
            if (!childNode || !hydrate(childNode, children[i], insertedVnodeQueue, inVPr
e)) {
              childrenMatch = false
              break
            }
            childNode = childNode.nextSibling
          }
          // if childNode is not null, it means the actual childNodes list is
          // longer than the virtual children list.
          if (!childrenMatch || childNode) {
            /* istanbul ignore if */
            if (process.env.NODE_ENV !== 'production' &&
              typeof console !== 'undefined' &&
              !hydrationBailed
            ) {
              hydrationBailed = true
              console.warn('Parent: ', elm)
              console.warn('Mismatching childNodes vs. VNodes: ', elm.childNodes, childr
en)
            }
          }
        }
      }
    }
  }
}

```

```

        }
        return false
    }
}
}

if (isDef(data)) {
    let fullInvoke = false
    for (const key in data) {
        if (!isRenderedModule(key)) {
            fullInvoke = true
            invokeCreateHooks(vnode, insertedVnodeQueue)
            break
        }
    }
    if (!fullInvoke && data['class']) {
        // ensure collecting deps for deep class bindings for future updates
        traverse(data['class'])
    }
}
} else if (elm.data !== vnode.text) {
    elm.data = vnode.text
}
return true
}

function assertNodeMatch (node, vnode, inVPre) {
    if (isDef(vnode.tag)) {
        return vnode.tag.indexOf('vue-component') === 0 || (
            !isUnknownElement(vnode, inVPre) &&
            vnode.tag.toLowerCase() === (node.tagName && node.tagName.toLowerCase())
        )
    } else {
        return node.nodeType === (vnode.isComment ? 8 : 3)
    }
}

// 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__。
return function patch (oldVnode, vnode, hydrating, removeOnly, parentElm, refElm) {
    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, parentElm, refElm)
} 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__`。

为了实现 `patch`，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。因为前面介绍过，`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。而不同平台的

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来区别，这里用到了一个函数柯里化的技巧，通过 createPatchFunction 把差异化参数提前固化，这样不用每次调用 patch 的时候都传递 nodeOps 和 modules 了，这种编程技巧也非常值得学习。

patch 方法，它接收 4 个参数，

```
return function patch (oldVnode, vnode, hydrating, removeOnly, parentElm, refElm) { }
```

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

例如，例子1-1：

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

在_update的时候，patch方法就是

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

- 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 "，vm.\$el 的赋值是在之前 mountComponent 函数做的
- vnode 对应的是调用 render 函数的返回值
- hydrating 在非服务端渲染情况下为 false
- removeOnly 为 false

patch里的关键方法

如例子1-1，由于我们传入的 oldVnode 实际上是一个 DOM container，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

```

if (isUndef(oldVnode)) {
  // empty mount (likely as component), create new root element
  isInitialPatch = true
  createElm(vnode, insertedVnodeQueue, parentElm, refElm)
} 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 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。
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)
    }
}
}
}

```

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。

```

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
        )
      }
    }
  }

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

  /* istanbul ignore if */
  if (__WEEEX__) {
    // in Weex, the default insertion order is parent-first.
    // List items can be optimized to use children-first insertion
    // with append="tree".
    const appendAsTree = isDef(data) && isTrue(data.appendAsTree)
    if (!appendAsTree) {
      if (isDef(data)) {

```

```

        invokeCreateHooks(vnode, insertedVnodeQueue)
    }
    insert(parentElm, vnode.elm, refElm)
}
createChildren(vnode, children, insertedVnodeQueue)
if (appendAsTree) {
    if (isDef(data)) {
        invokeCreateHooks(vnode, insertedVnodeQueue)
    }
    insert(parentElm, vnode.elm, refElm)
}
} else {
    // createChildren 方法去创建子元素
    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)
}
}
}

```

其中：

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。

```

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，

```

// createChildren 方法去创建子元素
createChildren(vnode, children, insertedVnodeQueue)

// createChildren 方法去创建子元素
// 实际上是遍历子虚拟节点，递归调用 createElement，这是一种常用的深度优先的遍历算法，
// 这里要注意的一点是在遍历过程中会把 vnode.elm 作为父容器的 DOM 节点占位符传入。

```

```

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)))
  }
}

```

最后，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)
}
insert(parentElm, vnode.elm, refElm)

```

insert方法定义在patch.js中

```

// 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 v
node 树节点的插入顺序是先子后父
function insert (parent, elm, ref) {
  if (isDef(parent)) {
    if (isDef(ref)) {
      if (ref.parentNode === parent) {
        // 在辅助方法，runtime/node-ops.js中，其实就是调用原生 DOM 的 API 进行 DOM 操作
        nodeOps.insertBefore(parent, elm, ref)
      }
    } else {
      nodeOps.appendChild(parent, elm)
    }
  }
}

```

nodeOps在platforms中的node-ops中定义：

```

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。

总结

自此，整个Vue的渲染成DOM的过程就完成了，可以归纳为：

new Vue -> init方法 -> \$mount -> compile -> render函数 -> vnode -> patch方法 -> DOM

源码解构

参照vue-2.5.17 版本，解构一下vue的源码结构

双向绑定原理

响应式原理

响应式对象

整个Vue的响应式原理，都和ES5的一个属性有关，就是Object.defineProperty

Object.defineProperty

会直接在一个对象上定义一个新属性，或者修改一个对象的现有属性，并返回这个对象。

```
Object.defineProperty(obj, prop, descriptor)
```

- obj是要在其上定义属性的对象
- prop是要定义或修改的属性名称
- descriptor是将被定义或修改的属性描述符

比较核心的是 descriptor，它有很多可选键值，configurable, enumerable, value, writable, get, set

这里比较关心 get和set，get 是一个给属性提供的 getter 方法，当我们访问了该属性的时候会触发 getter 方法；set 是一个给属性提供的 setter 方法，当我们对该属性做修改的时候会触发 setter 方法。

initState

在_init的时候，会执行initState(vm)方法，这个方法的内容定义在src/core/instance/state.js

```
// initState 方法主要是对 props、methods、data、computed 和 wathcer 等属性做了初始化操作。
export function initState (vm: Component) {
  vm._watchers = []
  const opts = vm.$options
  if (opts.props) initProps(vm, opts.props) // 初始化 props
  if (opts.methods) initMethods(vm, opts.methods) // 初始化 methods
  if (opts.data) {
    initData(vm) // 初始化Data
  } else {
    observe(vm._data = {}, true /* asRootData */)
  }
  if (opts.computed) initComputed(vm, opts.computed) // 初始化computed
  if (opts.watch && opts.watch !== nativeWatch) {
    initWatch(vm, opts.watch) // 初始化watch
  }
}
```

这里两个重点initProps和initData。

initProps

props 的初始化主要过程，就是遍历定义的 props 配置。

遍历的过程中，做了两件事，

- 一个是调用 `defineReactive` 方法把每个 prop 对应的值变成响应式，可以通过 `vm._props.xxx` 访问到定义 props 中对应的属性。
 - 一个是通过 proxy 把 `vm._props.xx` 的访问代理到 `vm.xxx` 上

```
function initProps (vm: Component, propsOptions: Object) {
  const propsData = vm.$options.propsData || {}
  const props = vm._props = {}
  // cache prop keys so that future props updates can iterate using Array
  // instead of dynamic object key enumeration.
  const keys = vm.$options._propKeys = []
  const isRoot = !vm.$parent
  // root instance props should be converted
  if (!isRoot) {
    toggleObserving(false)
  }
  for (const key in propsOptions) {
    keys.push(key)
    const value = validateProp(key, propsOptions, propsData, vm)
    /* istanbul ignore else */
    if (process.env.NODE_ENV !== 'production') {
      const hyphenatedKey = hyphenate(key)
      if (isReservedAttribute(hyphenatedKey) ||
          config.isReservedAttr(hyphenatedKey)) {
        warn(
          `${hyphenatedKey}` + ' is a reserved attribute and cannot be used as component prop'
        )
      }
    }
    defineReactive(props, key, value, () => {
      if (vm.$parent && !isUpdatingChildComponent) {
        warn(
          `Avoid mutating a prop directly since the value will be ` +
          `overwritten whenever the parent component re-renders. ` +
          `Instead, use a data or computed property based on the prop's ` +
          `value. Prop being mutated: "${key}"`,
        )
      }
    })
  }
} else {
  defineReactive(props, key, value)
}
// static props are already proxied on the component's prototype
// during Vue.extend(). We only need to proxy props defined at
// instantiation here.
if (!(key in vm)) {
  proxy(vm, `props`, key)
```

```

        }
    }
    toggleObserving(true)
}

```

initData

```

function initData (vm: Component) {
    let data = vm.$options.data
    data = vm._data = typeof data === 'function'
        ? getData(data, vm)
        : data || {}
    if (!isPlainObject(data)) {
        data = {}
        process.env.NODE_ENV !== 'production' && warn(
            'data functions should return an object:\n' +
            'https://vuejs.org/v2/guide/components.html#data-Must-Be-a-Function',
            vm
        )
    }
    // proxy data on instance
    const keys = Object.keys(data)
    const props = vm.$options.props
    const methods = vm.$options.methods
    let i = keys.length
    while (i--) {
        const key = keys[i]
        if (process.env.NODE_ENV !== 'production') {
            if (methods && hasOwn(methods, key)) {
                warn(
                    `Method "${key}" has already been defined as a data property.`,
                    vm
                )
            }
        }
        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)) {
            proxy(vm, `_data`, key)
        }
    }
    // observe data
    observe(data, true /* asRootData */)
}

```

data的初始化也做两件事,

- 一个是对定义data函数返回对象的遍历，通过proxy把每个值vm._data.xxx代理到vm.xxx上
- 另一个是调用observe方法，观测data的变化，把data变成响应式，可以通过 vm._data.xxx 访问到定义 data 返回函数中对应的属性

proxy

proxy 方法的实现很简单，通过 Object.defineProperty 把 target[sourceKey][key] 的读写变成了对 target[key] 的读写。

```
const sharedPropertyDefinition = {
  enumerable: true,
  configurable: true,
  get: noop,
  set: noop
}

export function proxy (target: Object, sourceKey: string, key: string) {
  sharedPropertyDefinition.get = function proxyGetter () {
    return this[sourceKey][key]
  }
  sharedPropertyDefinition.set = function proxySetter (val) {
    this[sourceKey][key] = val
  }
  Object.defineProperty(target, key, sharedPropertyDefinition)
}
```

observe

observe 的功能就是用来监测数据的变化，它的定义在 src/core/observer/index.js 中：

```
/**
 * Attempt to create an observer instance for a value,
 * returns the new observer if successfully observed,
 * or the existing observer if the value already has one.
 */
// observe 方法的作用就是给非 VNode 的对象类型数据添加一个 Observer，如果已经添加过则直接返回，否则在满足一定条件下去实例化一个 Observer 对象实例。
export function observe (value: any, asRootData: ?boolean): Observer | void {
  if (!isObject(value) || value instanceof VNode) {
    return
  }
  let ob: Observer | void
  if (hasOwn(value, '__ob__') && value.__ob__ instanceof Observer) {
    ob = value.__ob__
  } else if (
    shouldObserve &&
    !isServerRendering() &&
```

```
(Array.isArray(value) || isPlainObject(value)) &&
Object.isExtensible(value) &&
!value._isVue
) {
ob = new Observer(value)
}
if (asRootData && ob) {
ob.vmCount++
}
return ob
}
```

Observer 是一个类，它的作用是给对象的属性添加 getter 和 setter，用于依赖收集和派发更新。

```
/**
 * Observer class that is attached to each observed
 * object. Once attached, the observer converts the target
 * object's property keys into getter/setters that
 * collect dependencies and dispatch updates.
 */
export class Observer {
value: any;
dep: Dep;
vmCount: number; // number of vms that has this object as root $data

constructor (value: any) {
this.value = value
this.dep = new Dep()
this.vmCount = 0
def(value, '__ob__', this)
if (Array.isArray(value)) {
const augment = hasProto
? protoAugment
: copyAugment
augment(value, arrayMethods, arrayKeys)
this.observeArray(value)
} else {
this.walk(value)
}
}

/**
 * Walk through each property and convert them into
 * getter/setters. This method should only be called when
 * value type is Object.
 */
walk (obj: Object) {
const keys = Object.keys(obj)
for (let i = 0; i < keys.length; i++) {
defineReactive(obj, keys[i])
}}
```

```

        }
    }

    /**
     * Observe a list of Array items.
     */
    observeArray (items: Array<any>) {
        for (let i = 0, l = items.length; i < l; i++) {
            observe(items[i])
        }
    }
}

```

Observer 的构造函数逻辑很简单，首先实例化 Dep 对象，这块稍后会介绍，接着通过执行 def 函数把自身实例添加到数据对象 value 的 `_ob_` 属性上，def 的定义在 `src/core/util/lang.js` 中：

def 函数是一个非常简单的 `Object.defineProperty` 的封装，这就是为什么我在开发中输出 `data` 上对象类型的数据，会发现该对象多了一个 `_ob_` 的属性

defineReactive

`defineReactive` 的功能就是定义一个响应式对象，给对象动态添加 `getter` 和 `setter`，它的定义在 `src/core/observer/index.js` 中。

```

/**
 * Define a reactive property on an Object.
 */
export function defineReactive (
    obj: Object,
    key: string,
    val: any,
    customSetter?: ?Function,
    shallow?: boolean
) {
    const dep = new Dep()

    const property = Object.getOwnPropertyDescriptor(obj, key)
    if (property && property.configurable === false) {
        return
    }

    // cater for pre-defined getter/setters
    const getter = property && property.get
    if (!getter && arguments.length === 2) {
        val = obj[key]
    }
    const setter = property && property.set

    let childOb = !shallow && observe(val)
    Object.defineProperty(obj, key, {

```

```

enumerable: true,
configurable: true,
get: function reactiveGetter () {
  const value = getter ? getter.call(obj) : val
  if (Dep.target) {
    dep.depend()
    if (childOb) {
      childOb.dep.depend()
      if (Array.isArray(value)) {
        dependArray(value)
      }
    }
  }
  return value
},
set: function reactiveSetter (newVal) {
  const value = getter ? getter.call(obj) : val
  /* eslint-disable no-self-compare */
  if (newVal === value || (newVal !== newVal && value !== value)) {
    return
  }
  /* eslint-enable no-self-compare */
  if (process.env.NODE_ENV !== 'production' && customSetter) {
    customSetter()
  }
  if (setter) {
    setter.call(obj, newVal)
  } else {
    val = newVal
  }
  childOb = !shallow && observe(newVal)
  dep.notify()
}
})
}
}

```

defineReactive 函数最开始初始化 Dep 对象的实例，接着拿到 obj 的属性描述符，然后对子对象递归调用 observe 方法，这样就保证了无论 obj 的结构多复杂，它的所有子属性也能变成响应式的对象，这样我们访问或修改 obj 中一个嵌套较深的属性，也能触发 getter 和 setter。

Object.defineProperty 去给 obj 的属性 key 添加 getter 和 setter。而关于 getter 和 setter 的具体实现

总结

核心就是利用 Object.defineProperty 给数据添加了 getter 和 setter

getter 用于依赖收集，setter 用于派发更新

依赖收集

依赖收集

defineReactive里有getter的逻辑。

getter

```
/*
 * Define a reactive property on an Object.
 */
export function defineReactive (
  obj: Object,
  key: string,
  val: any,
  customSetter?: ?Function,
  shallow?: boolean
) {
  // new Dep的实例
  const dep = new Dep()

  const property = Object.getOwnPropertyDescriptor(obj, key)
  if (property && property.configurable === false) {
    return
  }

  // cater for pre-defined getter/setters
  const getter = property && property.get
  if (!getter && arguments.length === 2) {
    val = obj[key]
  }
  const setter = property && property.set

  let childOb = !shallow && observe(val)
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter () {
      const value = getter ? getter.call(obj) : val
      if (Dep.target) {
        // 另一个是在 get 函数中通过 dep.depend 做依赖收集
        dep.depend()
        // 判断childOb
        if (childOb) {
          childOb.dep.depend()
          if (Array.isArray(value)) {
            dependArray(value)
          }
        }
      }
    }
  })
}
```

```

        }
    }
    return value
},
set: function reactiveSetter (newVal) {
    const value = getter ? getter.call(obj) : val
    /* eslint-disable no-self-compare */
    if (newVal === value || (newVal !== newVal && value !== value)) {
        return
    }
    /* eslint-enable no-self-compare */
    if (process.env.NODE_ENV !== 'production' && customSetter) {
        customSetter()
    }
    if (setter) {
        setter.call(obj, newVal)
    } else {
        val = newVal
    }
    childOb = !shallow && observe(newVal)
    dep.notify()
}
})
}

```

Dep

依赖收集的核心，定义在instance/observer/dep.js中

```

/* @flow */

import type Watcher from './watcher'
import { remove } from '../util/index'

let uid = 0

/**
 * A dep is an observable that can have multiple
 * directives subscribing to it.
 */
// A Class 一个类
// 这里需要特别注意的是它有一个静态属性 target，这是一个全局唯一 Watcher，这是一个非常巧妙的设计，
// 因为在同一时间只能有一个全局的 Watcher 被计算，另外它的自身属性 subs 也是 Watcher 的数组。
export default class Dep {
    static target: ?Watcher; //Dep 实际上就是对 Watcher 的一种管理，Dep 脱离 Watcher 单独存在是
    没有意义的
    id: number;
    subs: Array<Watcher>;
}

```

```

constructor () {
  this.id = uid++
  this.subs = []
}

addSub (sub: Watcher) {
  this.subs.push(sub)
}

removeSub (sub: Watcher) {
  remove(this.subs, sub)
}

depend () {
  if (Dep.target) {
    Dep.target.addDep(this)
  }
}

notify () {
  // stabilize the subscriber list first
  const subs = this.subs.slice()
  for (let i = 0, l = subs.length; i < l; i++) {
    subs[i].update()
  }
}

// the current target watcher being evaluated.
// this is globally unique because there could be only one
// watcher being evaluated at any time.
Dep.target = null
const targetStack = []

export function pushTarget (_target: ?Watcher) {
  if (Dep.target) targetStack.push(Dep.target)
  Dep.target = _target
}

export function popTarget () {
  Dep.target = targetStack.pop()
}

```

Watcher

Watcher 是一个 Class，在它的构造函数中，定义了一些和 Dep 相关的属性

源码在 instance/observer/watcher.js 中

```
this.deps = []
```

```
this.newDeps = []
this.depIds = new Set()
this.newDepIds = new Set()
```

过程

Vue的mount过程，之前介绍过，通过mountComponent函数进行挂载。

```
updateComponent = () => {
  vm._update(vm._render(), hydrating)
}

new Watcher(vm, updateComponent, noop, {
  before () {
    if (vm._isMounted) {
      callHook(vm, 'beforeUpdate')
    }
  }
}, true /* isRenderWatcher */)
```

- 进入Watcher，先执行get方法，也就会执行pushTarget(this)

```
constructor (
  vm: Component,
  expOrFn: string | Function,
  cb: Function,
  options?: ?Object,
  isRenderWatcher?: boolean
) {
  this.vm = vm
  if (isRenderWatcher) {
    vm._watcher = this
  }
  vm._watchers.push(this)
  // options
  if (options) {
    this.deep = !!options.deep
    this.user = !!options.user
    this.lazy = !!options.lazy
    this.sync = !!options.sync
  } else {
    this.deep = this.user = this.lazy = this.sync = false
  }
  this.cb = cb
  this.id = ++uid // uid for batching
  this.active = true
  this.dirty = this.lazy // for lazy watchers
  this.deps = []
  this.newDeps = []
  this.depIds = new Set()
  this.newDepIds = new Set()
```

```

this.expression = process.env.NODE_ENV !== 'production'
  ? expOrFn.toString()
  : ''
// parse expression for getter
if (typeof expOrFn === 'function') {
  this.getter = expOrFn
} else {
  this.getter = parsePath(expOrFn)
  if (!this.getter) {
    this.getter = function () {}
  }
  process.env.NODE_ENV !== 'production' && warn(
    `Failed watching path: "${expOrFn}" ` +
    'Watcher only accepts simple dot-delimited paths. ' +
    'For full control, use a function instead.',
    vm
  )
}
this.value = this.lazy
  ? undefined
  : this.get() // this.get()
}

```

```

/**
 * Evaluate the getter, and re-collect dependencies.
 */
get () {
  pushTarget(this)
  let value
  const vm = this.vm
  try {
    value = this.getter.call(vm, vm) // this.getter 对应就是 updateComponent 函数， 实际执行 vm._update(vm._render(), hydrating)
  } catch (e) {
    if (this.user) {
      handleError(e, vm, `getter for watcher "${this.expression}"`)
    } else {
      throw e
    }
  } finally {
    // "touch" every property so they are all tracked as
    // dependencies for deep watching
    if (this.deep) {
      traverse(value)
    }
    popTarget()
    this.cleanupDeps()
  }
  return value
}

```

`pushTarget`在`dep.js`中定义，实际上就是把`Dep.target`赋值为当前的渲染`watcher`并压栈（为了恢复用）。

```
// 实际上就是把 Dep.target 赋值为当前的渲染 watcher 并压栈（为了恢复用）。
export function pushTarget (_target: ?Watcher) {
  if (Dep.target) targetStack.push(Dep.target)
  Dep.target = _target
}
```

接着执行

```
value = this.getter.call(vm, vm)
```

实际的对应就是`updateComponent`函数，实际执行`vm._update(vm._render(), hydrating)`

它会先执行`vm._render()`方法，因为之前分析过这个方法会生成渲染`VNode`，并且在这个过程中会对`vm`上的数据访问，这个时候就触发了数据对象的`getter`。

那么每个对象值的`getter`都持有一个`dep`，在触发`getter`的时候会调用`dep.depend()`方法，也就会执行`Dep.target.addDep(this)`。

```
/*
 * Add a dependency to this directive.
 */
addDep (dep: Dep) {
  const id = dep.id
  if (!this.newDepIds.has(id)) {
    this.newDepIds.add(id)
    this.newDeps.push(dep)
    if (!this.depIds.has(id)) {
      dep.addSub(this)
    }
  }
}
```

这时候会做一些逻辑判断（保证同一数据不会被添加多次）后执行`dep.addSub(this)`，那么就会执行`this.subs.push(sub)`，也就是说把当前的`watcher`订阅到这个数据持有的`dep`的`subs`中，这个目的是为后续数据变化时候能通知到哪些`subs`做准备。

所以在`vm._render()`过程中，会触发所有数据的`getter`，这样实际上已经完成了一个依赖收集的过程。

```
if (this.deep) {
  traverse(value)
}
```

递归访问value， 触发所有子项的getter

```
popTarget()  
  
Dep.target = targetStack.pop()
```

实际上就是把 Dep.target 恢复成上一个状态，因为当前 vm 的数据依赖收集已经完成，那么对应的渲染Dep.target 也需要改变。

```
this.cleanupDeps()
```

总结

收集依赖的目的是为了当这些响应式数据发送变化，触发它们的 setter 的时候，能知道应该通知哪些订阅者去做相应的逻辑处理，我们把这个过程叫派发更新，其实 Watcher 和 Dep 就是一个非常经典的观察者设计模式的实现

依赖收集

依赖收集

依赖收集之后就是为了派发更新，当数据修改的时候，可以对相关的依赖派发更新。

setter

依然看defineReactive这个方法

```
set: function reactiveSetter (newVal) {
  const value = getter ? getter.call(obj) : val
  /* eslint-disable no-self-compare */
  if (newVal === value || (newVal !== newVal && value !== value)) {
    return
  }
  /* eslint-enable no-self-compare */
  if (process.env.NODE_ENV !== 'production' && customSetter) {
    customSetter()
  }
  if (setter) {
    setter.call(obj, newVal)
  } else {
    val = newVal
  }
  // childOb = !shallow && observe(newVal), 如果 shallow 为 false 的情况，会对新设置的值变成
  // 一个响应式对象;
  childOb = !shallow && observe(newVal)
  // dep.notify(), 通知所有的订阅者,
  dep.notify()
}
```



过程分析

同样，也是dep的一个实例方法。

```
/**
 * A dep is an observable that can have multiple
 * directives subscribing to it.
 */
// A Class 一个类
// 这里需要特别注意的是它有一个静态属性 target，这是一个全局唯一 Watcher，这是一个非常巧妙的设计，
// 因为在同一时间只能有一个全局的 Watcher 被计算，另外它的自身属性 subs 也是 Watcher 的数组。
export default class Dep {
  static target: ?Watcher; //Dep 实际上就是对 Watcher 的一种管理，Dep 脱离 Watcher 单独存在是
  没有意义的
```

```

id: number;
subs: Array<Watcher>

constructor () {
  this.id = uid++
  this.subs = []
}

addSub (sub: Watcher) {
  this.subs.push(sub)
}

removeSub (sub: Watcher) {
  remove(this.subs, sub)
}

depend () {
  if (Dep.target) {
    Dep.target.addDep(this)
  }
}

notify () {
  // stabilize the subscriber list first
  // 遍历所有的 subs, 也就是 Watcher 的实例数组, 然后调用每一个 watcher 的 update 方法,
  const subs = this.subs.slice()
  for (let i = 0, l = subs.length; i < l; i++) {
    subs[i].update()
  }
}
}

```

update 定义在 watcher 中。

```

class Watcher {
  // lazy 和 sync 等状态的分析我会之后抽一小节详细介绍, 在一般组件数据更新的场景, 会走到最后一个 queueWatcher(this) 的逻辑,
  update () {
    /* istanbul ignore else */
    if (this.lazy) {
      this.dirty = true
    } else if (this.sync) {
      this.run()
    } else {
      queueWatcher(this)
    }
  }
}

```

除了computed和sync的状态，其他会走到queueWatcher(this)的逻辑。

queueWatcher定义在src/core/observer/scheduler.js中

```
/*
 * Push a watcher into the watcher queue.
 * Jobs with duplicate IDs will be skipped unless it's
 * pushed when the queue is being flushed.
 */
export function queueWatcher (watcher: Watcher) {
  const id = watcher.id
  if (has[id] == null) {
    has[id] = true
    if (!flushing) {
      queue.push(watcher)
    } else {
      // if already flushing, splice the watcher based on its id
      // if already past its id, it will be run next immediately.
      let i = queue.length - 1
      while (i > index && queue[i].id > watcher.id) {
        i--
      }
      queue.splice(i + 1, 0, watcher)
    }
    // queue the flush
    if (!waiting) {
      waiting = true
      nextTick(flushSchedulerQueue)
    }
  }
}
```

queueWatcher就是一个队列的概念，这是Vue的一个特点。

它并不会每次数据改变都触发 **watcher** 的回调，而是把这些 **watcher** 先添加到一个队列里，然后在 **nextTick** 后执行 **flushSchedulerQueue**。

flushSchedulerQueue的实现，也在这个文件中。

```
/*
 * Flush both queues and run the watchers.
 */
function flushSchedulerQueue () {
  flushing = true
  let watcher, id

  // Sort queue before flush.
  // This ensures that:
  // 1. Components are updated from parent to child. (because parent is always
  //     created before the child)
}
```

```

// 2. A component's user watchers are run before its render watcher (because
//     user watchers are created before the render watcher)
// 3. If a component is destroyed during a parent component's watcher run,
//     its watchers can be skipped.
queue.sort((a, b) => a.id - b.id)

// do not cache length because more watchers might be pushed
// as we run existing watchers
for (index = 0; index < queue.length; index++) {
  watcher = queue[index]
  id = watcher.id
  has[id] = null
  watcher.run()
  // in dev build, check and stop circular updates.
  if (process.env.NODE_ENV !== 'production' && has[id] != null) {
    circular[id] = (circular[id] || 0) + 1
    if (circular[id] > MAX_UPDATE_COUNT) {
      warn(
        'You may have an infinite update loop ' +
        watcher.user
        ? `in watcher with expression "${watcher.expression}"` +
        : `in a component render function.`
      ),
      watcher.vm
    )
    break
  }
}
}

// keep copies of post queues before resetting state
const activatedQueue = activatedChildren.slice()
const updatedQueue = queue.slice()

resetSchedulerState()

// call component updated and activated hooks
callActivatedHooks(activatedQueue)
callUpdatedHooks(updatedQueue)

// devtool hook
/* istanbul ignore if */
if (devtools && config.devtools) {
  devtools.emit('flush')
}

function callUpdatedHooks (queue) {
  let i = queue.length
  while (i--) {
    const watcher = queue[i]

```

```

const vm = watcher.vm
if (vm._watcher === watcher && vm._isMounted) {
  callHook(vm, 'updated')
}
}
}

```

这里的代码，梳理如下：

- 队列排序

```
queue.sort((a, b) => a.id - b.id) 对队列做了从小到大的排序
```

* 组件的更新由父到子；因为父组件的创建过程是先于子的，所以 `watcher` 的创建也是先父后子，执行顺序也应该保持先父后子。
* 用户的自定义 `watcher` 要优先于渲染 `watcher` 执行；因为用户自定义 `watcher` 是在渲染 `watcher` 之前创建的。
* 如果一个组件在父组件的 `watcher` 执行期间被销毁，那么它对应的 `watcher` 执行都可以被跳过，所以父组件的 `watcher` 应该先执行。

- 队列遍历

在对 `queue` 排序后，接着就是要对它做遍历，拿到对应的 `watcher`，执行 `watcher.run()`。

- 状态恢复

这个过程就是执行 `resetSchedulerState` 函数

```

/**
 * Reset the scheduler's state.
 */
function resetSchedulerState () {
  index = queue.length = activatedChildren.length = 0
  has = {}
  if (process.env.NODE_ENV !== 'production') {
    circular = {}
  }
  waiting = flushing = false
}

```

逻辑非常简单，就是把这些控制流程状态的一些变量恢复到初始值，把 `watcher` 队列清空。

之后走 `watcher.run()`

```

/**
 * Scheduler job interface.
 * Will be called by the scheduler.
*/

```

```

run () {
  if (this.active) {
    const value = this.get() // 获取当前值
    if (
      value !== this.value ||
      // Deep watchers and watchers on Object/Arrays should fire even
      // when the value is the same, because the value may
      // have mutated.
      isObject(value) ||
      this.deep
    ) {
      // set new value
      const oldValue = this.value
      this.value = value
      if (this.user) {
        try {
          this.cb.call(this.vm, value, oldValue)
        } catch (e) {
          handleError(e, this.vm, `callback for watcher "${this.expression}"`)
        }
      } else {
        this.cb.call(this.vm, value, oldValue)
      }
    }
  }
}

```

通过 this.get() 得到它当前的值，然后做判断，如果满足新旧值不等、新值是对象类型、deep 模式任何一个条件，则执行 watcher 的回调，

所以这就是当我们去修改组件相关的响应式数据的时候，会触发组件重新渲染的原因，接着就会重新执行 patch 的过程，但它和首次渲染有所不同

总结

所以Vue的数据修改派发更新的过程，实际上就是当数据发生变化的时候，触发 setter 逻辑，把在依赖过程中订阅的所有观察者，也就是 watcher，都触发它们的 update 过程，这个过程又利用了队列做了进一步优化，在 nextTick 后执行所有 watcher 的 run，最后执行它们的回调函数。

nextTick 原理

nextTick 可以让我们在下次 DOM 更新循环结束之后执行延迟回调，用于获得更新后的 DOM。

源码在src/core/util/next-tick.js中

用法

```
Vue.nextTick([callback, context]);
```

摘自官网：

```
// 修改数据
vm.msg = 'Hello'
// DOM 还没有更新
Vue.nextTick(function () {
  // DOM 更新了
})

// 作为一个 Promise 使用 (2.1.0 起新增, 详见接下来的提示)
Vue.nextTick()
  .then(function () {
    // DOM 更新了
  })
}
```

JS运行机制

JS两特点：

- 单线程语言
- 基于事件循环 Event Loop

详情见事件循环。

简单的说：

- 所有同步任务，都在主线程上，形成一个执行栈（execution context stack）
- 主线程之外，还存在一个“任务队列”（task queue）。只要异步任务有了运行结果，就在“任务队列”之中放置一个事件。
- 一旦“执行栈”中的所有同步任务执行完毕，系统就会读取“任务队列”，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。
- 主线程重复上面三个

主线程的执行过程就是一个 tick，而所有的异步结果都是通过“任务队列”来调度被调度。消息队列中存放的是一个个的任务（task）。规范中规定 task 分为两大类，分别是 macro task 和 micro task，并且每个 macro task 结束后，都要清空所有的 micro task。

执行时顺序为：

```

for (macroTask of macroTaskQueue) {
    // 1. Handle current MACRO-TASK
    handleMacroTask();

    // 2. Handle all MICRO-TASK
    for (microTask of microTaskQueue) {
        handleMicroTask(microTask);
    }
}

```

常见的 macro task 有 setTimeout、MessageChannel、postMessage、setImmediate； 常见的 micro task 有 MutationObserver 和 Promise.then。

Vue中next-tick.js的实现

```

/* @flow */
/* globals MessageChannel */

import { noop } from 'shared/util'
import { handleError } from './error'
import { isIOS, isNative } from './env'

const callbacks = []
let pending = false

function flushCallbacks () {
    pending = false
    const copies = callbacks.slice(0)
    callbacks.length = 0
    for (let i = 0; i < copies.length; i++) {
        copies[i]()
    }
}

// Here we have async deferring wrappers using both microtasks and (macro) tasks.
// In < 2.4 we used microtasks everywhere, but there are some scenarios where
// microtasks have too high a priority and fire in between supposedly
// sequential events (e.g. #4521, #6690) or even between bubbling of the same
// event (#6566). However, using (macro) tasks everywhere also has subtle problems
// when state is changed right before repaint (e.g. #6813, out-in transitions).
// Here we use microtask by default, but expose a way to force (macro) task when
// needed (e.g. in event handlers attached by v-on).
let microTimerFunc
let macroTimerFunc
let useMacroTask = false

// Determine (macro) task defer implementation.

```

```

// Technically setImmediate should be the ideal choice, but it's only available
// in IE. The only polyfill that consistently queues the callback after all DOM
// events triggered in the same loop is by using MessageChannel.
/* istanbul ignore if */
if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
  macroTimerFunc = () => {
    setImmediate(flushCallbacks)
  }
} else if (typeof MessageChannel !== 'undefined' &&
  isNative(MessageChannel) ||
  // PhantomJS
  MessageChannel.toString() === '[object MessageChannelConstructor]')
) {
  const channel = new MessageChannel()
  const port = channel.port2
  channel.port1.onmessage = flushCallbacks
  macroTimerFunc = () => {
    port.postMessage(1)
  }
} else {
  /* istanbul ignore next */
  macroTimerFunc = () => {
    setTimeout(flushCallbacks, 0)
  }
}

// Determine microtask defer implementation.
/* istanbul ignore next, $flow-disable-line */
if (typeof Promise !== 'undefined' && isNative(Promise)) {
  const p = Promise.resolve()
  microTimerFunc = () => {
    p.then(flushCallbacks)
    // in problematic UIWebViews, Promise.then doesn't completely break, but
    // it can get stuck in a weird state where callbacks are pushed into the
    // microtask queue but the queue isn't being flushed, until the browser
    // needs to do some other work, e.g. handle a timer. Therefore we can
    // "force" the microtask queue to be flushed by adding an empty timer.
    if (isIOS) setTimeout(noop)
  }
} else {
  // fallback to macro
  microTimerFunc = macroTimerFunc
}

/***
 * Wrap a function so that if any code inside triggers state change,
 * the changes are queued using a (macro) task instead of a microtask.
 */
export function withMacroTask (fn: Function): Function {
  return fn._withTask || (fn._withTask = function () {
    useMacroTask = true
  })
}

```

```

        const res = fn.apply(null, arguments)
        useMacroTask = false
        return res
    })
}

export function nextTick (cb?: Function, ctx?: Object) {
    let _resolve
    callbacks.push(() => {
        if (cb) {
            try {
                cb.call(ctx)
            } catch (e) {
                handleError(e, ctx, 'nextTick')
            }
        } else if (_resolve) {
            _resolve(ctx)
        }
    })
    if (!pending) {
        pending = true
        if (useMacroTask) {
            macroTimerFunc()
        } else {
            microTimerFunc()
        }
    }
    // $flow-disable-line
    if (!cb && typeof Promise !== 'undefined') {
        return new Promise(resolve => {
            _resolve = resolve
        })
    }
}

```

其中，`microTimerFunc`和`macroTimerFunc`分别对应`macroTask`和`microTask`

```

let microTimerFunc
let macroTimerFunc

```

然后就是常规判断：

对于`macroTask`:

```

// 优先检测是否支持原生 setImmediate，这是一个高版本 IE 和 Edge 才支持的特性
// 不支持的话再去检测是否支持原生的 MessageChannel
// 如果也不支持的话就会降级为 setTimeout 0
if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
    macroTimerFunc = () => {

```

```

        setImmediate(flushCallbacks)
    }
} else if (typeof MessageChannel !== 'undefined' && (
    isNative(MessageChannel) ||
    // PhantomJS
    MessageChannel.toString() === '[object MessageChannelConstructor]'
)) {
    const channel = new MessageChannel()
    const port = channel.port2
    channel.port1.onmessage = flushCallbacks
    macroTimerFunc = () => {
        port.postMessage(1)
    }
} else {
    /* istanbul ignore next */
    macroTimerFunc = () => {
        setTimeout(flushCallbacks, 0)
    }
}
}

```

对于microTask:

```

// Determine microtask defer implementation.
/* istanbul ignore next, $flow-disable-line */
// 而对于 micro task 的实现，则检测浏览器是否原生支持 Promise，不支持的话直接指向 macro task 的
// 实现。
if (typeof Promise !== 'undefined' && isNative(Promise)) {
    const p = Promise.resolve()
    microTimerFunc = () => {
        p.then(flushCallbacks)
        // in problematic UIWebViews, Promise.then doesn't completely break, but
        // it can get stuck in a weird state where callbacks are pushed into the
        // microtask queue but the queue isn't being flushed, until the browser
        // needs to do some other work, e.g. handle a timer. Therefore we can
        // "force" the microtask queue to be flushed by adding an empty timer.
        if (isIOS) setTimeout(noop)
    }
} else {
    // fallback to macro
    microTimerFunc = macroTimerFunc
}

```

函数的实现：

next-tick.js中有两个函数withMacroTask和nextTick

- 先看nextTick

```

export function nextTick (cb?: Function, ctx?: Object) {
    let _resolve

```

```

    callbacks.push(() => {
      if (cb) {
        try {
          cb.call(ctx)
        } catch (e) {
          handleError(e, ctx, 'nextTick')
        }
      } else if (_resolve) {
        _resolve(ctx)
      }
    })
    if (!pending) {
      pending = true
      if (useMacroTask) {
        macroTimerFunc()
      } else {
        microTimerFunc()
      }
    }
    // $flow-disable-line
    if (!cb && typeof Promise !== 'undefined') {
      return new Promise(resolve => {
        _resolve = resolve
      })
    }
  }
}

```

例如执行 `nextTick(flushSchedulerQueue)`。逻辑也很简单，把传入的回调函数 `cb` 压入 `callbacks` 数组，最后一次性地根据 `useMacroTask` 条件执行 `macroTimerFunc` 或者是 `microTimerFunc`，而它们都会在下一个 tick 执行 `flushCallbacks`，`flushCallbacks` 的逻辑非常简单，对 `callbacks` 遍历，然后执行相应的回调函数。

```

function flushCallbacks () {
  pending = false
  const copies = callbacks.slice(0)
  callbacks.length = 0
  for (let i = 0; i < copies.length; i++) {
    copies[i]()
  }
}

```

这里使用 `callbacks` 而不是直接在 `nextTick` 中执行回调函数的原因是保证在同一个 tick 内多次执行 `nextTick`，不会开启多个异步任务，而把这些异步任务都压成一个同步任务，在下一个 tick 执行完毕。

- `withMacroTask`

另一个方法 `withMacroTask` 函数，它是对函数做一层包装，确保函数执行过程中对数据任意的修改，触发变化执行 `nextTick` 的时候强制走 `macroTimerFunc`

总结

数据的变化到 DOM 的重新渲染是一个异步过程，发生在下一个 tick。这就是我们平时在开发的过程中，比如从服务端接口去获取数据的时候，数据做了修改，如果我们的某些方法去依赖了数据修改后的 DOM 变化，我们就必须在 nextTick 后执行。

```
// 修改数据
vm.msg = 'Hello'
// DOM 还没有更新
Vue.nextTick(function () {
    // DOM 更新了
})

// 作为一个 Promise 使用 (2.1.0 起新增，详见接下来的提示)
Vue.nextTick().then(function () {
    // DOM 更新了
})
```

Vue.js 提供了 2 种调用 nextTick 的方式，一种是全局 API Vue.nextTick，一种是实例上的方法 vm.\$nextTick，无论我们使用哪一种，最后都是调用 next-tick.js 中实现的 nextTick 方法。

检测变化的注意事项

对象添加属性

对于使用 `Object.defineProperty` 实现响应式的对象，当我们去给这个对象添加一个新的属性的时候，是不能够触发它的 `setter` 的。

```
let vm = new Vue({
  data: {
    a: 1
  }
})

vm.b = 2; // b就是非响应式的
```

那么如何添加新属性，`Vue`中有一个全局的API，就是`set`

```
Vue.set = set
```

这个 `set` 方法的定义在 `src/core/observer/index.js` 中

```
/** 
 * Set a property on an object. Adds the new property and
 * triggers change notification if the property doesn't
 * already exist.
 */
export function set (target: Array<any> | Object, key: any, val: any): any {
  if (process.env.NODE_ENV !== 'production' &&
    (isUndef(target) || isPrimitive(target)))
  ) {
    warn(`Cannot set reactive property on undefined, null, or primitive value: ${target: any}`)
  }
  if (Array.isArray(target) && isValidArrayIndex(key)) {
    target.length = Math.max(target.length, key)
    target.splice(key, 1, val)
    return val
  }
  if (key in target && !(key in Object.prototype)) {
    target[key] = val
    return val
  }
  const ob = (target: any).__ob__
  if (target._isVue || (ob && ob.vmCount)) {
    process.env.NODE_ENV !== 'production' && warn(
      'Avoid adding reactive properties to a Vue instance or its root $data ' +
      'at runtime - declare it upfront in the data option.'
    )
  }
}
```

```

    )
    return val
}
if (!ob) {
  target[key] = val
  return val
}
defineReactive(ob.value, key, val)
ob.dep.notify()
return val
}

```

set 方法接收 3个参数:

- target 可能是数组或者是普通对象,
- key 代表的是数组的下标或者是对象的键值,
- val 代表添加的值。

首先判断如果 target 是数组且 key 是一个合法的下标, 则之前通过 splice 去添加进数组然后返回, 这里的 splice 其实已经不仅仅是原生数组的 splice 了, 稍后我会详细介绍数组的逻辑。

接着又判断 key 已经存在于 target 中, 则直接赋值返回, 因为这样的变化是可以观测到了。

接着再获取到 target.ob 并赋值给 ob, 之前分析过它是在 Observer 的构造函数执行的时候初始化的, 表示 Observer 的一个实例, 如果它不存在, 则说明 target 不是一个响应式的对象, 则直接赋值并返回。

最后通过 defineReactive(ob.value, key, val) 把新添加的属性变成响应式对象, 然后再通过 ob.dep.notify() 手动的触发依赖通知

defineReactive的时候, 有这样一段:

```

Object.defineProperty(obj, key, {
  enumerable: true,
  configurable: true,
  get: function reactiveGetter () {
    const value = getter ? getter.call(obj) : val
    if (Dep.target) {
      dep.depend()
      if (childOb) {
        childOb.dep.depend()
        if (Array.isArray(value)) {
          dependArray(value)
        }
      }
    }
    return value
  },
})

```

判断了 childOb，并调用了 childOb.dep.depend() 收集了依赖，这就是为什么执行 Vue.set 的时候通过 ob.dep.notify() 能够通知到 watcher，从而让添加新的属性到对象也可以检测到变化。

Array数组

我们都知道Vue中数组有些情况是无法双向绑定更新数据的，需要手动触发。都有什么情况呢？

- 当你利用索引直接设置一个项时，例如：vm.items[indexOfItem] = newValue
- 当你修改数组的长度时，例如：vm.items.length = newLength

对于第一种情况，可以使用：Vue.set(example1.items, indexOfItem, newValue);

而对于第二种情况，可以使用 vm.items.splice(newLength)。

这里vm.items.splice(newLength)为啥可以把添加的数组对象变成响应式对象。

在通过 observe 方法去观察对象的时候会实例化 Observer，在它的构造函数中是专门对数组做了处理，它的定义在 src/core/observer/index.js 中。

```
export class Observer {
  constructor (value: any) {
    this.value = value
    this.dep = new Dep()
    this.vmCount = 0
    def(value, '__ob__', this)
    if (Array.isArray(value)) {
      const augment = hasProto
        ? protoAugment
        : copyAugment
      augment(value, arrayMethods, arrayKeys)
      this.observeArray(value)
    } else {
      // ...
    }
  }
}
```

，首先获取 augment，这里的 hasProto 实际上就是判断对象中是否存在 **proto**，如果存在则 augment 指向 protoAugment，否则指向 copyAugment

```
/**
 * Augment an target Object or Array by intercepting
 * the prototype chain using __proto__
 */
function protoAugment (target, src: Object, keys: any) {
  /* eslint-disable no-proto */
  target.__proto__ = src
  /* eslint-enable no-proto */
}
```

```
/** 
 * Augment an target Object or Array by defining
 * hidden properties.
 */
/* istanbul ignore next */
function copyAugment (target: Object, src: Object, keys: Array<string>) {
  for (let i = 0, l = keys.length; i < l; i++) {
    const key = keys[i]
    def(target, key, src[key])
  }
}
```

protoAugment 方法是直接把 target.proto 原型直接修改为 src, 而 copyAugment 方法是遍历 keys, 通过 def, 也就是 Object.defineProperty 去定义它自身的属性值。对于大部分现代浏览器都会走到 protoAugment, 那么它实际上就把 value 的原型指向了 arrayMethods, arrayMethods 的定义在 src/core/observer/array.js 中

```
import { def } from '../util/index'

const arrayProto = Array.prototype
export const arrayMethods = Object.create(arrayProto)

const methodsToPatch = [
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]

/** 
 * Intercept mutating methods and emit events
 */
methodsToPatch.forEach(function (method) {
  // cache original method
  const original = arrayProto[method]
  def(arrayMethods, method, function mutator (...args) {
    const result = original.apply(this, args)
    const ob = this.__ob__
    let inserted
    switch (method) {
      case 'push':
      case 'unshift':
        inserted = args
        break
      case 'splice':
        inserted = args.slice(2)
    }
    if (inserted) {
      ob.observeArray(inserted)
      if (method === 'push' || method === 'unshift') {
        ob.push(inserted)
      } else {
        ob.splice(...args)
      }
    }
    return result
  })
})
```

```
        break
    }
    if (inserted) ob.observeArray(inserted)
    // notify change
    ob.dep.notify()
    return result
})
})
```

arrayMethods 首先继承了 Array，然后对数组中所有能改变数组自身的方法，如 push、pop 等这些方法进行重写。

重写后的办法会先执行它们本身原有的逻辑，并对能增加数组长度的 3 个方法 push、unshift、splice 方法做了判断，获取到插入的值，然后把新添加的值变成一个响应式对象，并且再调用 ob.dep.notify() 手动触发依赖通知，这就很好地解释了之前的示例中调用 vm.items.splice(newLength) 方法可以检测到变化。

总结

Vue.del和Vue.set大同小异。

计算属性和监听属性

Vue的组件对象支持computed和watch两种属性。

computed

计算属性的初始化是发生在 Vue 实例初始化阶段的 initState 函数中

执行了 if (opts.computed) initComputed(vm, opts.computed), initComputed 的定义在 src/core/instance/state.js 中

```
// initState 方法主要是对 props、methods、data、computed 和 wathcer 等属性做了初始化操作。
export function initState (vm: Component) {
  vm._watchers = []
  const opts = vm.$options
  if (opts.props) initProps(vm, opts.props) // 初始化 props
  if (opts.methods) initMethods(vm, opts.methods) // 初始化 methods
  if (opts.data) {
    initData(vm) // 初始化Data
  } else {
    observe(vm._data = {}, true /* asRootData */)
  }
  if (opts.computed) initComputed(vm, opts.computed) //初始化computed
  if (opts.watch && opts.watch !== nativeWatch) {
    initWatch(vm, opts.watch) // 初始化watch
  }
}
```

initComputed源码在同一文件内：

```
function initComputed (vm: Component, computed: Object) {
  // $flow-disable-line
  const watchers = vm._computedWatchers = Object.create(null)
  // computed properties are just getters during SSR
  const isSSR = isServerRendering()

  for (const key in computed) {
    const userDef = computed[key]
    const getter = typeof userDef === 'function' ? userDef : userDef.get
    if (process.env.NODE_ENV !== 'production' && getter == null) {
      warn(
        `Getter is missing for computed property "${key}".`,
        vm
      )
    }
    if (!isSSR) {
      // create internal watcher for the computed property.
    }
  }
}
```

```
watchers[key] = new Watcher(
  vm,
  getter || noop,
  noop,
  computedWatcherOptions
)
}

// component-defined computed properties are already defined on the
// component prototype. We only need to define computed properties defined
// at instantiation here.
if (!(key in vm)) {
  defineComputed(vm, key, userDef)
} else if (process.env.NODE_ENV !== 'production') {
  if (key in vm.$data) {
    warn(`The computed property "${key}" is already defined in data.`, vm)
  } else if (vm.$options.props && key in vm.$options.props) {
    warn(`The computed property "${key}" is already defined as a prop.`, vm)
  }
}
}
```

watch

总结

组件更新

组件的更新还是调用了 `vm._update` 方法

判断

通过 `sameVNode(oldVnode, vnode)` 判断它们是否是相同的 VNode 来决定走不同的更新逻辑

```
function sameVnode (a, b) {
  return (
    a.key === b.key && (
      (
        a.tag === b.tag &&
        a.isComment === b.isComment &&
        isDef(a.data) === isDef(b.data) &&
        sameInputType(a, b)
      ) || (
        isTrue(a.isAsyncPlaceholder) &&
        a.asyncFactory === b.asyncFactory &&
        isUndef(b.asyncFactory.error)
      )
    )
  )
}
```

- 如果两个 vnode 的 key 不相等，则是不同的；
- 否则继续判断对于同步组件，则判断 `isComment`、`data`、`input` 类型等是否相同，对于异步组件，则判断 `asyncFactory` 是否相同。

新旧节点不同

分三步： 创建新的节点 => 更新父的占位符节点 => 删除旧节点

新旧节点相同

调用 `patchVNode` 方法

`patchVnode` 的作用就是把新的 vnode patch 到旧的 vnode 上，这里我们只关注关键的核心逻辑

可以拆成四个步骤：

- 执行 `prepatch` 钩子函数
- 执行 `update` 钩子函数
- 完成 `patch` 过程

如果 vnode 是个文本节点且新旧文本不相同，则直接替换文本内容。如果不是文本节点，则判断它们的子节点，并分了几种情况处理：

1. oldCh 与 ch 都存在且不相同时，使用 updateChildren 函数来更新子节点，这个后面重点讲。
2. 如果只有 ch 存在，表示旧节点不需要了。如果旧的节点是文本节点则先将节点的文本清除，然后通过 addVnodes 将 ch 批量插入到新节点 elm 下。
3. 如果只有 oldCh 存在，表示更新的是空节点，则需要将旧的节点通过 removeVnodes 全部清除。
4. 当只有旧节点是文本节点的时候，则清除其节点文本内容。
5. 执行 postpatch 钩子函数

总结

组件更新的过程核心就是新旧 vnode diff，对新旧节点相同以及不同的情况分别做不同的处理。

新旧节点不同的更新流程是创建新节点->更新父占位符节点->删除旧节点；

而新旧节点相同的更新流程是去获取它们的 children，根据不同情况做不同的更新逻辑。最复杂的情况是新旧节点相同且它们都存在子节点，那么会执行 updateChildren 逻辑