**Vue源码分析**

目录

[一、 背景 4](#_Toc65140693)

[二、 桌面应用程序的历史 4](#_Toc65140694)

[三、 Electron的历史 6](#_Toc65140695)

[四、 Electron的入门 9](#_Toc65140696)

[(一) 应用程序结构 9](#_Toc65140697)

[主进程和渲染器进程 9](#_Toc65140698)

[Electron API 11](#_Toc65140699)

[Node.js API 12](#_Toc65140700)

[五、 Electron-vue 13](#_Toc65140701)

[(一) 起步 14](#_Toc65140702)

[(二) 项目结构 16](#_Toc65140703)

[单一的 package.json 设置 16](#_Toc65140704)

[关于 main 进程 16](#_Toc65140705)

[文件树 17](#_Toc65140706)

[渲染进程 18](#_Toc65140707)

[主进程 19](#_Toc65140708)

[Webpack 配置 20](#_Toc65140709)

[(三) 构建electron应用程序 21](#_Toc65140710)

[electron-packager 21](#_Toc65140711)

[electron-builder 21](#_Toc65140712)

[六、 开发中存在的问题 22](#_Toc65140713)

[(一) electron程序显示了文件浏览器 22](#_Toc65140714)

[背景 22](#_Toc65140715)

[分析 22](#_Toc65140716)

[(二) ReferenceError: process is not defined 23](#_Toc65140717)

[背景 23](#_Toc65140718)

[原因分析 23](#_Toc65140719)

[解决方式 23](#_Toc65140720)

[(三) electron 如何打开开发者工具devtools 24](#_Toc65140721)

[背景 24](#_Toc65140722)

[解决方式 24](#_Toc65140723)

[(四) 找不到electron依赖包 25](#_Toc65140724)

[背景 25](#_Toc65140725)

[原因分析 25](#_Toc65140726)

[解决方式 26](#_Toc65140727)

[(五) electron-vue无法改变vuex状态 26](#_Toc65140728)

[背景 26](#_Toc65140729)

[解决方案 26](#_Toc65140730)

[(六) electron-vue中无法使用Element组件 27](#_Toc65140731)

[背景 27](#_Toc65140732)

[解决方案 27](#_Toc65140733)

[(七) Electron无边框窗口中自定义窗口快捷键 28](#_Toc65140734)

[背景 28](#_Toc65140735)

[解决方案 28](#_Toc65140736)

[(八) 监听窗口状态，动态改变窗口最大化图标 29](#_Toc65140737)

[背景 29](#_Toc65140738)

[解决方案 29](#_Toc65140739)

[(九) electron Uncaught TypeError: Cannot read property 'app' of undefined 30](#_Toc65140740)

[背景 30](#_Toc65140741)

[解决方案 31](#_Toc65140742)

[七、 开发成果 31](#_Toc65140743)

摘要

阅读本文需要一定的node基础知识。知道Electron是开发桌面应用程序的技术。

本文主要介绍了桌面应用程序的历史和electron的历史，分析了electron原理和electron-vue项目架构，快速上手了electron-vue项目。

关键词

Electron Electron-vue

正文

# 背景

看的时候明明懂了，但是对某一个现象却说不上原理来。这个老是遇到。我想唯有自己动手理理才能记忆深刻。

# 目录结构

1. .circleci：包含持续集成和持续部署的脚本
2. .github：github中相关配置和说明
3. Benchmarks：性能测试目录
4. dist：vue打包文件
5. examples：例子
6. flow：静态类型检查的声明文件
7. node\_modules：依赖包
8. packages：vue相关第三方包
9. scripts：自动化构建的相关脚本文件和配置
10. src：源码目录
11. test：测试目录
12. types：类型声明文件
13. .babelrc.js：babel配置文件
14. .editorconfig：编辑器的配置文件
15. .eslintignore：eslint检测忽略配置文件
16. .eslintrc.js：eslint配置文件
17. .flowconfig：flow配置文件
18. .gitignore：github提交忽略配置文件
19. BACKERS.md：包含赞助者的文档
20. LICENSE：证书
21. package-lock.json：锁定安装时包的版本号，保证他人用的包一致
22. package.json：项目清单
23. README.md：项目说明
24. yarn-error.log：yarn报错日志
25. yarn.lock：锁定安装包的版本

# src源码目录分析

1. compiler：模板编译。包括AST转化成render函数、预定义指令、解析规则和编译规则等
2. core：核心代码。包括内置组件、全局API、Vue实例化、响应式原理、virtualDom和工具函数等
3. platforms：平台构建。包括web（web端和服务器端渲染、编译和运行）和weex（多端）
4. server：服务器端渲染
5. sfc：解析器。将vue组件转化为javascript对象
6. shared：共享模块

# core核心代码分析

## 全局配置

D:\vue-dev\vue-dev\src\core\config.js

export default ({

  // 自定义合并策略的选项。

  // $flow-disable-line

  optionMergeStrategies: Object.create(null),

  // 取消所有日志警告

  silent: false,

  // 设置false组织vue在启动时生成生产提示

  productionTip: process.env.NODE\_ENV !== 'production',

  // 是否使用devtools

  devtools: process.env.NODE\_ENV !== 'production',

  // 是否记录性能

  // 设置为 true 以在浏览器开发工具的性能/时间线面板中启用对组件初始化、编译、渲染和打补丁的性能追踪。只适用于开发模式和支持 performance.mark API 的浏览器上。

  performance: false,

  // 可获取错误信息和vue实例

  errorHandler: null,

  // 为vue的运行时警告赋予一个自定义处理函数

  warnHandler: null,

  // 须使 Vue 忽略在 Vue 之外的自定义元素 (e.g. 使用了 Web Components APIs)。否则，它会假设你忘记注册全局组件或者拼错了组件名称，从而抛出一个关于 Unknown custom element 的警告。

  ignoredElements: [],

  // 绑定v-on 键位别名

  // $flow-disable-line

  keyCodes: Object.create(null),

  // 检查标签是否被保留，以使其不能注册为

  // 组件。

  // 这是平台相关的，可能会被覆盖。

  isReservedTag: no,

  // 检查一个属性是否被保留，以使其不能作为组件使用

  // 支持。

  //   这是平台相关的，可能会被覆盖。

  isReservedAttr: no,

  // 检查标签是否为未知元素。

  // 与平台相关的。

  isUnknownElement: no,

  // 获取元素的命名空间

  getTagNamespace: noop,

  // 解析特定平台的真正标记名。

  parsePlatformTagName: identity,

  // 检查一个属性是否必须使用属性绑定，例如value

  // 与平台相关的。

  mustUseProp: no,

  // 异步执行更新。

  //   拟由Vue Test Utils使用

  // 如果设置为false，将显著降低性能。

  async: true,

  // 由于遗留原因暴露

  \_lifecycleHooks: LIFECYCLE\_HOOKS

}: Config)

## 全局api

1. components

存在keep-alive组件的目录

1. global-api

该目录包含全局api的注册

### component、directive和filter

assets.js：component、directive和filter的注册。这三者的实现类似。

1. component：注册或获取全局组件
2. directive：注册或获取全局指令
3. filter：注册或获取全局过滤器

// 对component\directive\filter进行申明注册

export function initAssetRegisters (Vue: GlobalAPI) {

  /\*\*

   \* Create asset registration methods.

   \*/

  ASSET\_TYPES.forEach(type => {

    // 初始化component api

    // 注册全局组件，或者返回构造器

    // Vue.component()

    // 初始化directive api

    // 注册全局指令，或者返回指令

    // Vue.directive()

    // 初始化filter api

    // 注册全局过滤器，或者返回过滤器

    // Vue.filter()

    Vue[type] = function (

      id: string,

      definition: Function | Object

    ): Function | Object | void {

      if (!definition) {

        // 全局配置

        return this.options[type + 's'][id]

      } else {

        /\* istanbul ignore if \*/

        if (process.env.NODE\_ENV !== 'production' && type === 'component') {

          // 校验组件名称

          validateComponentName(id)

        }

        // 如果是component方法

        if (type === 'component' && isPlainObject(definition)) {

          // 自动生成id

          definition.name = definition.name || id

          definition = this.options.\_base.extend(definition)

        }

        // 如果是derective方法

        if (type === 'directive' && typeof definition === 'function') {

          definition = { bind: definition, update: definition }

        }

        this.options[type + 's'][id] = definition

        return definition

      }

    }

  })

}

### extend

extend.js：注册extend全局api，提供注册构造器方法

// 初始化extend api

// 使用基础vue构造器，创建子类

export function initExtend (Vue: GlobalAPI) {

  // 唯一的cid作用：创建的每一个构造器可以在vue原型链上继承和缓存

  Vue.cid = 0

  let cid = 1

  // 类继承

  Vue.extend = function (extendOptions: Object): Function {

    extendOptions = extendOptions || {}

    const Super = this

    const SuperId = Super.cid

    const cachedCtors = extendOptions.\_Ctor || (extendOptions.\_Ctor = {})

    if (cachedCtors[SuperId]) {

      return cachedCtors[SuperId]

    }

    const name = extendOptions.name || Super.options.name

    // 校验组件名称

    if (process.env.NODE\_ENV !== 'production' && name) {

      validateComponentName(name)

    }

    const Sub = function VueComponent (options) {

      this.\_init(options)

    }

    Sub.prototype = Object.create(Super.prototype)

    Sub.prototype.constructor = Sub

    Sub.cid = cid++

    Sub.options = mergeOptions(

      Super.options,

      extendOptions

    )

    Sub['super'] = Super

    //对于props和computed属性，我们定义proxy getters

    //在扩展的原型上扩展时的Vue实例。

    //避免对创建的每个实例调用Object.defineProperty。

    if (Sub.options.props) {

      // 初始化props

      initProps(Sub)

    }

    if (Sub.options.computed) {

      // 初始化computed

      initComputed(Sub)

    }

    // 允许进一步extension/mixin/plugin的使用

    Sub.extend = Super.extend

    Sub.mixin = Super.mixin

    Sub.use = Super.use

    // 创建资产注册，所以扩展类可以有私人资产

    ASSET\_TYPES.forEach(function (type) {

      Sub[type] = Super[type]

    })

    // 使递归self-lookup

    if (name) {

      Sub.options.components[name] = Sub

    }

    //在扩展时引用超选项。

    //稍后在实例化时，我们可以检查Super的选项是否有更新。

    Sub.superOptions = Super.options

    Sub.extendOptions = extendOptions

    Sub.sealedOptions = extend({}, Sub.options)

    // 缓存构造函数

    cachedCtors[SuperId] = Sub

    return Sub

  }

}

### mixin

mixin.js：注册mixin，提供全局混入方法

// 初始化mixin api

// 全局混入，影响每一个实例

export function initMixin (Vue: GlobalAPI) {

  Vue.mixin = function (mixin: Object) {

    this.options = mergeOptions(this.options, mixin)

    return this

  }

}

// 合并两个选项对象到一个新的。

// 在实例化和继承中使用的核心实用程序。

export function mergeOptions (

  parent: Object,

  child: Object,

  vm?: Component

): Object {

  if (process.env.NODE\_ENV !== 'production') {

    checkComponents(child)

  }

  if (typeof child === 'function') {

    child = child.options

  }

  normalizeProps(child, vm)

  normalizeInject(child, vm)

  normalizeDirectives(child)

//在子选项上应用extends和mixins，

//但仅当它是一个原始选项对象而不是

//另一个合并选项调用的结果。

//只有合并的选项具有\_base属性。

  if (!child.\_base) {

    if (child.extends) {

      parent = mergeOptions(parent, child.extends, vm)

    }

    if (child.mixins) {

      for (let i = 0, l = child.mixins.length; i < l; i++) {

        parent = mergeOptions(parent, child.mixins[i], vm)

      }

    }

  }

  // 初始化选项

  const options = {}

  let key

  // 遍历父组件，全部合并

  for (key in parent) {

    mergeField(key)

  }

  // 遍历子组件，如果父中没有，那就合并

  for (key in child) {

    if (!hasOwn(parent, key)) {

      mergeField(key)

    }

  }

  function mergeField (key) {

    const strat = strats[key] || defaultStrat

    options[key] = strat(parent[key], child[key], vm, key)

  }

  return options

}

### use

use.js：注册use，提供安装插件方法

// 初始化use api

// 安装vue插件

export function initUse (Vue: GlobalAPI) {

  Vue.use = function (plugin: Function | Object) {

    const installedPlugins = (this.\_installedPlugins || (this.\_installedPlugins = []))

    if (installedPlugins.indexOf(plugin) > -1) {

      return this

    }

    // additional parameters

    const args = toArray(arguments, 1)

    args.unshift(this)

    // 插件是个对象，必须提供install方法

    // 插件是函数，直接作为install方法

    if (typeof plugin.install === 'function') {

      plugin.install.apply(plugin, args)

    } else if (typeof plugin === 'function') {

      plugin.apply(null, args)

    }

    installedPlugins.push(plugin)

    return this

  }

}

### set

index.js：抛出所有的全局api，其中可以从

import { set, del } from '../observer/index'

import { observe } from 'core/observer/index'

发现set、del和observe都在observer目录下，在实现响应式的同时注册全局api。

set：

一般有两种场景更改数据不触发试图更新：

1. 数据初始化后，给响应式对象设置新属性
2. 给数组添加新值

// 在对象上设置一个属性

// 增加一个新的属性和如果属性不存在则触发更改提示

export function set (target: Array<any> | Object, key: any, val: any): any {

  if (process.env.NODE\_ENV !== 'production' &&

    (isUndef(target) || isPrimitive(target))

  ) {

    // 警告：不能在undefined、null和原始类型上设置属性

    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)) {

    // 警告：避免向Vue实例或其根$data添加响应性属性，在运行时-在数据选项中声明它。

    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

}

### delete

删除对象属性，避免删除后不会更新视图

由于vue2中object.defineProperty()的缺陷，导致删除对象属性，无法触发视图更新，所以vue引进了删除delete api触发更新。

D:\vue-dev\vue-dev\src\core\observer\index.js

/\*\*

 \* @description 如果有需要删除一个属性并且触发更改

 \* @param {\*} target 目标对象

 \* @param {\*} key 某个属性

 \* @returns

 \*/

export function del(target: Array < any > | Object, key: any) {

  if (process.env.NODE\_ENV !== 'production' &&

    (isUndef(target) || isPrimitive(target))

  ) {

    // 警告：不能在undefined、null和原始类型上删除属性

    warn(`Cannot delete reactive property on undefined, null, or primitive value: ${(target: any)}`)

  }

  if (Array.isArray(target) && isValidArrayIndex(key)) {

    target.splice(key, 1)

    return

  }

  const ob = (target: any).\_\_ob\_\_

  if (target.\_isVue || (ob && ob.vmCount)) {

    // 警告：避免向Vue实例或其根$data删除响应性属性，在运行时-在数据选项中声明它。

    process.env.NODE\_ENV !== 'production' && warn(

      'Avoid deleting properties on a Vue instance or its root $data ' +

      '- just set it to null.'

    )

    return

  }

  if (!hasOwn(target, key)) {

    return

  }

  // 删除属性

  delete target[key]

  if (!ob) {

    return

  }

  // notify()触发试图更新

  ob.dep.notify()

}

申明后绑定在vue原型上

D:\vue-dev\vue-dev\src\core\instance\state.js

 Vue.prototype.$delete = del

### observable

作用：使一个对象可响应。Vue使用他来处理data函数返回的对象。

返回的对象可以用于函数渲染和计算属性，并且修改可以触发试图更新。

state.js：初始化数据，最后劫持数据

// 初始化数据

function initData (vm: Component) {

  let data = vm.$options.data

  // data是函数，获取数据

  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

  // data数据唯一性：判断是否存在于props和methods，有警告

  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

  // 劫持数据

  observe(data, true /\* asRootData \*/)

}

index.js：observe方法一系列判断后，实例化一个观察者对象

// 尝试为一个值创建一个观察者实例，

// 如果成功观察，返回新的观察者，

// 或现有的观察者(如果值已经有一个)。

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

}

这里定义了Observer观察者类

// 观察者类附加到每个被观察对象。

// 一旦连接，观察者转换目标，对象的属性键到getter/setter

// 收集依赖项和调度更新。

export class Observer {

  value: any;

  dep: Dep;

  // 使用该对象作为root $data的虚拟数量

  vmCount: number;

  constructor (value: any) {

    this.value = value

    // 订阅模式

    this.dep = new Dep()

    this.vmCount = 0

    def(value, '\_\_ob\_\_', this)

    if (Array.isArray(value)) {

      if (hasProto) {

        protoAugment(value, arrayMethods)

      } else {

        copyAugment(value, arrayMethods, arrayKeys)

      }

      this.observeArray(value)

    } else {

      this.walk(value)

    }

  }

// 遍历所有属性并将它们转换为getter / setter响应式对象。

//    只有在以下情况下才应该调用此方法：值类型为Object。

  walk (obj: Object) {

    const keys = Object.keys(obj)

    for (let i = 0; i < keys.length; i++) {

      defineReactive(obj, keys[i])

    }

  }

// 观察数组列表

  observeArray (items: Array<any>) {

    for (let i = 0, l = items.length; i < l; i++) {

      observe(items[i])

    }

  }

}

defineReactive：定义响应式对象

// 在对象上定义响应式属性

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

  }

  // 迎合预定义的getter/setter

  const getter = property && property.get

  const setter = property && property.set

  if ((!getter || setter) && arguments.length === 2) {

    val = obj[key]

  }

  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()

      }

      // #7981: for accessor properties without setter

      if (getter && !setter) return

      if (setter) {

        setter.call(obj, newVal)

      } else {

        val = newVal

      }

      childOb = !shallow && observe(newVal)

      dep.notify()

    }

  })

}

1. util

### nextTick

next-tick.js：定义nextTick方法，适用于dom更新后立即执行延迟回调。

原理：因为浏览器没有实现nextTick，所以vue中用Promise、MutationObserver、setImmediate和setTimeout去执行flushCallbacks方法将其插入microTask微任务中，目的是在调用栈执行完毕后去执行flushCallbacks方法。

/\* @flow \*/

/\* globals MutationObserver \*/

import { noop } from 'shared/util'

import { handleError } from './error'

import { isIE, isIOS, isNative } from './env'

export let isUsingMicroTask = false

const callbacks = []

let pending = false

// dom更新完成了，将异步函数意义执行

function flushCallbacks () {

  pending = false

  const copies = callbacks.slice(0)

  callbacks.length = 0

  for (let i = 0; i < copies.length; i++) {

    copies[i]()

  }

}

//这里我们有使用微任务的异步延迟包装器。

//在2.5中，我们使用(宏)任务(与微任务结合)。

//然而，它有微妙的问题，当状态更改之前，重绘

//(例如#6813，输出-输出转换)。

//另外，在事件处理程序中使用(宏)任务会导致一些奇怪的行为

//不能被回避的(例如#7109，#7153，#7546，#7834，#8109)。

//所以我们现在在任何地方都使用微任务。

//这个权衡的一个主要缺点是有一些场景

//在这种情况下，微任务的优先级太高，按理应该是介于两者之间

//连续事件(例如#4521，#6690，它们有变通方法)

//甚至在冒泡相同的事件(#6566)之间。

let timerFunc

// nextTick行为利用微任务队列，它可以被访问

//通过native Promise。

// 然后或MutationObserver。

// MutationObserver有更广泛的支持，但是它有严重的bug

// UIWebView在iOS中的>= 9.3.3当触摸事件处理程序触发。

//它触发几次后完全停止工作…

//所以,如果本地promise是可用的，我们将使用它:

/\* istanbul ignore next, $flow-disable-line \*/

if (typeof Promise !== 'undefined' && isNative(Promise)) {

  const p = Promise.resolve()

  timerFunc = () => {

    p.then(flushCallbacks)

//在有问题的UIWebViews中，Promise并没有完全崩溃，但是

//它可能会陷入一个奇怪的状态，回调被推入

//微任务队列，但队列不会被刷新，直到浏览器

//需要做一些其他的工作，例如处理一个定时器。因此,我们可以

//通过添加空计时器强制刷新微任务队列。

    if (isIOS) setTimeout(noop)

  }

  isUsingMicroTask = true

} else if (!isIE && typeof MutationObserver !== 'undefined' && (

  isNative(MutationObserver) ||

  // PhantomJS and iOS 7.x

  MutationObserver.toString() === '[object MutationObserverConstructor]'

)) {

//在native Promise不可用的地方使用MutationObserver，

  // e.g. PhantomJS, iOS7, Android 4.4

  // (#6466 MutationObserver is unreliable in IE11)

  let counter = 1

  const observer = new MutationObserver(flushCallbacks)

  const textNode = document.createTextNode(String(counter))

  observer.observe(textNode, {

    characterData: true

  })

  timerFunc = () => {

    counter = (counter + 1) % 2

    textNode.data = String(counter)

  }

  isUsingMicroTask = true

} else if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {

//回退到setimmediate。

//技术上，它利用(宏)任务队列，

//但是它仍然是一个比setTimeout更好的选择。

  timerFunc = () => {

    setImmediate(flushCallbacks)

  }

} else {

  // 回调 setTimeout.

  // 使用setTimeout立即执行函数组

  timerFunc = () => {

    setTimeout(flushCallbacks, 0)

  }

}

// nextTick解析

// cb：dom更新后触发的函数，

// ctx：上下文参数

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

  let \_resolve

  // 如果函数存在插入callbacks数组

  callbacks.push(() => {

    if (cb) {

      try {

        cb.call(ctx)

      } catch (e) {

        handleError(e, ctx, 'nextTick')

      }

    } else if (\_resolve) {

      \_resolve(ctx)

    }

  })

  // 不需要等待则进入timerFunc()

  if (!pending) {

    pending = true

    timerFunc()

  }

  // $flow-disable-line

  if (!cb && typeof Promise !== 'undefined') {

    return new Promise(resolve => {

      \_resolve = resolve

    })

  }

}

### compile

作用：将一个模板字符串编译成render函数。

没找到

## 选项/数据

### data

vue实例的数据对象。Vue在初始化的时候observer会对其进行劫持，将所有对象遍历，使用object.defineProperty()为每一个对象加入getter和setter，并且为其建立一个watcher，观察变化。  
具体实现如下：

#### new Vue(options)=>this.\_init(options)

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)

}

#### this.\_init(options)=>initState(vm)

export function initMixin(Vue: Class < Component > ) {

  Vue.prototype.\_init = function (options ? : Object) {

   ……

    // 初始化state数据

    initState(vm)

  ……

    /\* istanbul ignore if \*/

    if (process.env.NODE\_ENV !== 'production' && config.performance && mark) {

      vm.\_name = formatComponentName(vm, false)

      mark(endTag)

      measure(`vue ${vm.\_name} init`, startTag, endTag)

    }

    if (vm.$options.el) {

      vm.$mount(vm.$options.el)

    }

  }

}

#### initState(vm)=>initData(vm)

// 初始化状态

export function initState(vm: Component) {

  vm.\_watchers = []

  const opts = vm.$options

……

  // 初始化数据

  if (opts.data) {

    // 在initData中会有observe()方法去劫持数据

    initData(vm)

  } else {

    // 选项中没有定义data的话，vue默认劫持空对象

    observe(vm.\_data = {}, true /\* asRootData \*/ )

  }

  // 初始化computed

  if (opts.computed) initComputed(vm, opts.computed)

  // 初始化watch

  if (opts.watch && opts.watch !== nativeWatch) {

    initWatch(vm, opts.watch)

  }

}

#### initData()=>observe(data,true)

// 初始化数据

function initData(vm: Component) {

  let data = vm.$options.data

  // data是函数，获取数据

  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

  // data数据唯一性：判断是否存在于props和methods，有警告

  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

  // 劫持数据

  observe(data, true /\* asRootData \*/ )

}

observe后面在全局api中可以查看。

data从new Vue()进入到被数据劫持的流程就是上述4个步骤。

### props

父组件往子组件传入的数据，可以是对象或者数组。对象支持高级配置，可以类型校验type、默认值default、必填required、校验函数validator四个配置。

具体实现：

Props和data前两步骤是一样的，后面不同而已

#### InitState(vm)=>initProps(vm)

// 初始化状态

export function initState(vm: Component) {

  vm.\_watchers = []

  const opts = vm.$options

  // 初始化props

  if (opts.props) initProps(vm, opts.props)

  ……

}

initProps(vm,propsOptions)=>toggleObserving(true)

/\*\*

 \* @description 初始化props

 \* @param {\*} vm 组件实例

 \* @param {\*} propsOptions props选项

 \*/

function initProps(vm: Component, propsOptions: Object) {

  const propsData = vm.$options.propsData || {}

  const props = vm.\_props = {}

  //缓存prop键，以便未来的props更新可以使用数组迭代

  //代替动态对象键枚举。

  const keys = vm.$options.\_propKeys = []

  const isRoot = !vm.$parent

  // 根实例props应该被转换

  if (!isRoot) {

    toggleObserving(false)

  }

  // 遍历props

  for (const key in propsOptions) {

    // 传入keys数组

    keys.push(key)

    const value = validateProp(key, propsOptions, propsData, vm)

    /\* istanbul ignore else \*/

    if (process.env.NODE\_ENV !== 'production') {

      // 开发环境

      const hyphenatedKey = hyphenate(key)

      // 用连接符连接的key是一个保留属性，不能用作组件prop

      if (isReservedAttribute(hyphenatedKey) ||

        config.isReservedAttr(hyphenatedKey)) {

        warn(

          `"${hyphenatedKey}" is a reserved attribute and cannot be used as component prop.`,

          vm

        )

      }

      // props响应式化

      defineReactive(props, key, value, () => {

        if (!isRoot && !isUpdatingChildComponent) {

          // 避免直接改变一个prop，因为每当父组件重新渲染时，他的值将是被覆盖。

          // 相反，可以基于prop的值来用一个data或computed属性来更改。

          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}"`,

            vm

          )

        }

      })

    } else {

      // 生产环境下

      defineReactive(props, key, value)

    }

    // 静态props已经在组件的原型上通过Vue.extend()代理了。

    // 我们只需要通过代理props定义实例化

    if (!(key in vm)) {

      proxy(vm, `\_props`, key)

    }

  }

  toggleObserving(true)

}

总结