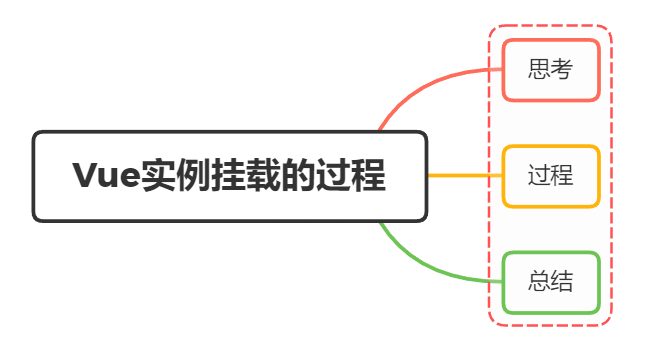
# 面试官：Vue实例挂载的过程



## 一、思考

我们都听过知其然知其所以然这句话

那么不知道大家是否思考过new Vue()这个过程中究竟做了些什么？

过程中是如何完成数据的绑定，又是如何将数据渲染到视图的等等

## 一、分析

首先找到vue的构造函数

源码位置：src.js

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)  
}

options是用户传递过来的配置项，如data、methods等常用的方法

vue构建函数调用\_init方法，但我们发现本文件中并没有此方法，但仔细可以看到文件下方定定义了很多初始化方法

initMixin(Vue); // 定义 \_init  
stateMixin(Vue); // 定义 $set $get $delete $watch 等  
eventsMixin(Vue); // 定义事件 $on $once $off $emit  
lifecycleMixin(Vue);// 定义 \_update $forceUpdate $destroy  
renderMixin(Vue); // 定义 \_render 返回虚拟dom

首先可以看initMixin方法，发现该方法在Vue原型上定义了\_init方法

源码位置：src.js

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  
 // 合并属性，判断初始化的是否是组件，这里合并主要是 mixins 或 extends 的方法  
 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 { // 合并vue属性  
 vm.$options = mergeOptions(  
 resolveConstructorOptions(vm.constructor),  
 options || {},  
 vm  
 )  
 }  
 /\* istanbul ignore else \*/  
 if (process.env.NODE\_ENV !== 'production') {  
 // 初始化proxy拦截器  
 initProxy(vm)  
 } else {  
 vm.\_renderProxy = vm  
 }  
 // expose real self  
 vm.\_self = vm  
 // 初始化组件生命周期标志位  
 initLifecycle(vm)  
 // 初始化组件事件侦听  
 initEvents(vm)  
 // 初始化渲染方法  
 initRender(vm)  
 callHook(vm, 'beforeCreate')  
 // 初始化依赖注入内容，在初始化data、props之前  
 initInjections(vm) // resolve injections before data/props  
 // 初始化props/data/method/watch/methods  
 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 ${vm.\_name} init`, startTag, endTag)  
 }  
 // 挂载元素  
 if (vm.$options.el) {  
 vm.$mount(vm.$options.el)  
 }  
 }

仔细阅读上面的代码，我们得到以下结论：

* 在调用beforeCreate之前，数据初始化并未完成，像data、props这些属性无法访问到
* 到了created的时候，数据已经初始化完成，能够访问data、props这些属性，但这时候并未完成dom的挂载，因此无法访问到dom元素
* 挂载方法是调用vm.$mount方法

initState方法是完成props/data/method/watch/methods的初始化

源码位置：src.js

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

我们和这里主要看初始化data的方法为initData，它与initState在同一文件上

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

仔细阅读上面的代码，我们可以得到以下结论：

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

关于数据响应式在这就不展开详细说明

上文提到挂载方法是调用vm.$mount方法

源码位置：

Vue.prototype.$mount = function (  
 el?: string | Element,  
 hydrating?: boolean  
): Component {  
 // 获取或查询元素  
 el = el && query(el)  
  
 /\* istanbul ignore if \*/  
 // vue 不允许直接挂载到body或页面文档上  
 if (el === document.body || el === document.documentElement) {  
 process.env.NODE\_ENV !== 'production' && warn(  
 `Do not mount Vue to <html> or <body> - mount to normal elements instead.`  
 )  
 return this  
 }  
  
 const options = this.$options  
 // resolve template/el and convert to render function  
 if (!options.render) {  
 let template = options.template  
 // 存在template模板，解析vue模板文件  
 if (template) {  
 if (typeof template === 'string') {  
 if (template.charAt(0) === '#') {  
 template = idToTemplate(template)  
 /\* istanbul ignore if \*/  
 if (process.env.NODE\_ENV !== 'production' && !template) {  
 warn(  
 `Template element not found or is empty: ${options.template}`,  
 this  
 )  
 }  
 }  
 } else if (template.nodeType) {  
 template = template.innerHTML  
 } else {  
 if (process.env.NODE\_ENV !== 'production') {  
 warn('invalid template option:' + template, this)  
 }  
 return this  
 }  
 } else if (el) {  
 // 通过选择器获取元素内容  
 template = getOuterHTML(el)  
 }  
 if (template) {  
 /\* istanbul ignore if \*/  
 if (process.env.NODE\_ENV !== 'production' && config.performance && mark) {  
 mark('compile')  
 }  
 /\*\*  
 \* 1.将temmplate解析ast tree  
 \* 2.将ast tree转换成render语法字符串  
 \* 3.生成render方法  
 \*/  
 const { render, staticRenderFns } = compileToFunctions(template, {  
 outputSourceRange: process.env.NODE\_ENV !== 'production',  
 shouldDecodeNewlines,  
 shouldDecodeNewlinesForHref,  
 delimiters: options.delimiters,  
 comments: options.comments  
 }, this)  
 options.render = render  
 options.staticRenderFns = staticRenderFns  
  
 /\* istanbul ignore if \*/  
 if (process.env.NODE\_ENV !== 'production' && config.performance && mark) {  
 mark('compile end')  
 measure(`vue ${this.\_name} compile`, 'compile', 'compile end')  
 }  
 }  
 }  
 return mount.call(this, el, hydrating)  
}

阅读上面代码，我们能得到以下结论：

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

对template的解析步骤大致分为以下几步：

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

生成render函数，挂载到vm上后，会再次调用mount方法

源码位置：src.js

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

调用mountComponent渲染组件

export function mountComponent (  
 vm: Component,  
 el: ?Element,  
 hydrating?: boolean  
): Component {  
 vm.$el = el  
 // 如果没有获取解析的render函数，则会抛出警告  
 // render是解析模板文件生成的  
 if (!vm.$options.render) {  
 vm.$options.render = createEmptyVNode  
 if (process.env.NODE\_ENV !== 'production') {  
 /\* istanbul ignore if \*/  
 if ((vm.$options.template && vm.$options.template.charAt(0) !== '#') ||  
 vm.$options.el || el) {  
 warn(  
 'You are using the runtime-only build of Vue where the template ' +  
 'compiler is not available. Either pre-compile the templates into ' +  
 'render functions, or use the compiler-included build.',  
 vm  
 )  
 } else {  
 // 没有获取到vue的模板文件  
 warn(  
 'Failed to mount component: template or render function not defined.',  
 vm  
 )  
 }  
 }  
 }  
 // 执行beforeMount钩子  
 callHook(vm, 'beforeMount')  
  
 let updateComponent  
 /\* istanbul ignore if \*/  
 if (process.env.NODE\_ENV !== 'production' && config.performance && mark) {  
 updateComponent = () => {  
 const name = vm.\_name  
 const id = vm.\_uid  
 const startTag = `vue-perf-start:${id}`  
 const endTag = `vue-perf-end:${id}`  
  
 mark(startTag)  
 const vnode = vm.\_render()  
 mark(endTag)  
 measure(`vue ${name} render`, startTag, endTag)  
  
 mark(startTag)  
 vm.\_update(vnode, hydrating)  
 mark(endTag)  
 measure(`vue ${name} patch`, startTag, endTag)  
 }  
 } else {  
 // 定义更新函数  
 updateComponent = () => {  
 // 实际调⽤是在lifeCycleMixin中定义的\_update和renderMixin中定义的\_render  
 vm.\_update(vm.\_render(), hydrating)  
 }  
 }  
 // we set this to vm.\_watcher inside the watcher's constructor  
 // since the watcher's initial patch may call $forceUpdate (e.g. inside child  
 // component's mounted hook), which relies on vm.\_watcher being already defined  
 // 监听当前组件状态，当有数据变化时，更新组件  
 new Watcher(vm, updateComponent, noop, {  
 before () {  
 if (vm.\_isMounted && !vm.\_isDestroyed) {  
 // 数据更新引发的组件更新  
 callHook(vm, 'beforeUpdate')  
 }  
 }  
 }, true /\* isRenderWatcher \*/)  
 hydrating = false  
  
 // manually mounted instance, call mounted on self  
 // mounted is called for render-created child components in its inserted hook  
 if (vm.$vnode == null) {  
 vm.\_isMounted = true  
 callHook(vm, 'mounted')  
 }  
 return vm  
}

阅读上面代码，我们得到以下结论：

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

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

render的作用主要是生成vnode

源码位置：src.js

// 定义vue 原型上的render方法  
Vue.prototype.\_render = function (): VNode {  
 const vm: Component = this  
 // render函数来自于组件的option  
 const { render, \_parentVnode } = vm.$options  
  
 if (\_parentVnode) {  
 vm.$scopedSlots = normalizeScopedSlots(  
 \_parentVnode.data.scopedSlots,  
 vm.$slots,  
 vm.$scopedSlots  
 )  
 }  
  
 // set parent vnode. this allows render functions to have access  
 // to the data on the placeholder node.  
 vm.$vnode = \_parentVnode  
 // render self  
 let vnode  
 try {  
 // There's no need to maintain a stack because all render fns are called  
 // separately from one another. Nested component's render fns are called  
 // when parent component is patched.  
 currentRenderingInstance = vm  
 // 调用render方法，自己的独特的render方法， 传入createElement参数，生成vNode  
 vnode = render.call(vm.\_renderProxy, vm.$createElement)  
 } catch (e) {  
 handleError(e, vm, `render`)  
 // return error render result,  
 // or previous vnode to prevent render error causing blank component  
 /\* istanbul ignore else \*/  
 if (process.env.NODE\_ENV !== 'production' && vm.$options.renderError) {  
 try {  
 vnode = vm.$options.renderError.call(vm.\_renderProxy, vm.$createElement, e)  
 } catch (e) {  
 handleError(e, vm, `renderError`)  
 vnode = vm.\_vnode  
 }  
 } else {  
 vnode = vm.\_vnode  
 }  
 } finally {  
 currentRenderingInstance = null  
 }  
 // if the returned array contains only a single node, allow it  
 if (Array.isArray(vnode) && vnode.length === 1) {  
 vnode = vnode[0]  
 }  
 // return empty vnode in case the render function errored out  
 if (!(vnode instanceof VNode)) {  
 if (process.env.NODE\_ENV !== 'production' && Array.isArray(vnode)) {  
 warn(  
 'Multiple root nodes returned from render function. Render function ' +  
 'should return a single root node.',  
 vm  
 )  
 }  
 vnode = createEmptyVNode()  
 }  
 // set parent  
 vnode.parent = \_parentVnode  
 return vnode  
}

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

源码位置：src.js

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

## 三、结论

* new Vue的时候调用会调用\_init方法
* 定义 $set、$get 、$delete、$watch 等方法
* 定义 $on、$off、$emit、$off等事件
* 定义 \_update、$forceUpdate、$destroy生命周期
* 调用$mount进行页面的挂载
* 挂载的时候主要是通过mountComponent方法
* 定义updateComponent更新函数
* 执行render生成虚拟DOM
* \_update将虚拟DOM生成真实DOM结构，并且渲染到页面中

## 参考文献

* https://www.cnblogs.com/gerry2019/p/12001661.html
* https://github.com/vuejs/vue/tree/dev/src/core/instance
* https://vue3js.cn