## Vue函数构造函数

路径: /src/core/instance/index.js

理解: 传入options配置项, 执行 this.\_\_init(options) 执行组件实例化 并将其挂载到界面中(如果配置了el 选项)

在构造函数下方(Vue框架初始化, 构造器未执行)分别有如下方法:

**intiMixin(Vue)**传入Vue构造器, 方法内部**给Vue原型添加\_\_init**方法

**stateMixin(Vue)** 传入Vue构造器, 方法内部给Vue原型上添加**$data, $props, $set, $delete, $watch**方法

**eventsMixin(Vue)** 传入Vue构造器, 方法内部给Vue原型上添加**$on, $emmit, $off, $once**

**lifecycleMixin(Vue)** 传入Vue构造器, 方法内部给Vue原型上添加**\_update, $forceUpdate, $destory** 生命周期相关钩子函数

**renderMixin(Vue)** 传入Vue构造器, 方法内部给Vue原型上添加**$nextTick, \_render**方法

此外, 方法内部顶部执行了**installRenderHelpers(Vue.prototype)** /src/core/instance/render-helpers/index.js

该方法是给Vue原型添加运行时方法的简写命令, 如\_**o, \_v(createTextVNode), \_e(createEmptyNode)**等.

\_init方法内部

1. 设置\_uid

2. 设置\_isVue = true

3. mergeOptions, 并将返回值赋值给**vm.$options**

4. initProxy 设置vm.\_renderProxy

5. 设置\_self(vm.\_self = vm)

6. 执行initLifeCycle(vm)

1. 将当前组件添加到非abstract父组件的$children中

2. 设置$parent

3. 设置$root

4. 设置$children = []

5. 设置$refs 并赋值为空对象

6. 添加\_watcher, \_inactive, 并赋值为null

7. 添加 \_directInactove. \_isMounted, \_isDestoryed, \_isBeingDestoryed 并赋值为false

7. 执行initEvent(vm)

1. 添加\_events属性, 并赋值为空对象

2. 添加\_hasHookEvent 并赋值为false

3. 将取$options.\_parentListeners, 执行**updateComponentListeners,** 初始化父组件添加在当前组件的方法(**在组件上直**

**接绑定事件 如 <el-button @click="">button</button>).**

8. 执行initRender(vm)

1. 设置\_vnode = null

2. 设置\_staticTrees = null // v-once cached trees

3. 设置$slots

4. 设置$scopeSlots

5. 设置\_c = (a,b,c,d) => createElement(vm, a, b, c, d, false) 模板编译时使用

6. 设置$createElement = (a, b, c, d) => createElement(a, b, c, d, true) render时使用

7. 调用defineReactive, 给当前vm添加**$attrs, $listeners** 响应式属性

9. 执行callHook(vm, 'beforeCreate') 生命周期方法

方法内部有如下判断vm.\_hasHookEvent, 如果为真, 则执行vm.$emit('hook:' + hook).

\_hasHookEvent设置为true的时机?

10. 执行initInjections(vm), 在初始化data, props之前inject.

方法内部执行resolveInject, 将传入的options中的inject配置项统一成一种格式, 并返回result

如果result有值, 调用defineReactive将key及值转化为响应式数据

11. 执行initState(vm)方法

1. 设置\_watchers = []

2. 执行initProps(vm, opts.props)初始化props,

1. 设置\_props ={}

2. 设置\_propKeys = []

3.toggleObserving 在响应式方法中的具体作用????

4. 调用**validateProp**校验PropsData是否否和propsOptions的配置, 并返回数据value

5. 调用**defineReactive**, 将props上的key, value转化为响应式数据

4. 内部调用**proxy**代理**\_props.key** 到组件实例上

3.执行initMethods(vm, opts.methods)初始化methods

1. 循环遍历optios上的methods配置, 检测key是否已经在props上配置,是否是保留key

2. 绑定key对应的方法到vm实例上

4.执行initData(vm)初始化data或 调用observe(vm.\_data = {}, true /\* asRootData \*/) 没有data配置项

1. 添加data = **vm.\_data** = = typeof data === ''function' ? getData(data, vm) : data || {}

2. 遍历data对象

1. 校验key是否已经在methods中配置, 重名时, data中的key会**覆盖**methods中的配置.(非生产环境, 会有提示)

2. 校验key是否已经在props中配置

3. 如果没有再props中配置, 则调用proxy, 代理\_data[key]到vm组件实例上

4. 调用**observe.** 将data转化为**响应式**数据

5. 执行initComputed(vm, opts.computed)初始化computed数据

1. 设置**\_computedWatchers**属性. const watchers = **vm.\_computedWatchers** = Object.create(null)

2. 遍历options.computed,

1. 获取对应key, key对应的userDef, 如果userDef不是方法, 则取其get属性, 得到key对应的**getter**

2. 创建new Watcher, 并返回创建的watcher, 将其添加到**vm.\_computedWatchers**中

3. 检测key是否在vm中,

如果不在vm中, 调用defineComputed,

如果在vm中, 检测是否在$data, props中, 如果在$data, $options.props中配置, 则给出提示

6. 执行initWatch(vm, opts.watch) 初始化watch配置项

调用vm.$watch

12. 执行initProvide(vm)初始化provide

设置vm.\_provide = typeof provide === 'function' ? provide.call(vm) : provide

13. 调用callHook(vm, 'created') 执行created生命周期钩子函数

14. 如果配置了**el**选项, 则执行 **vm.$mount(vm.$options.el)**

**进入**src/platforms/web/entry-runtime-with-compiler.js(**完整版**)

优先级: **render > template > el**

1. 判断是否有render配置项

2. 没有render配置项时, 再**判断是否配置template**, 如果配置了, 则获取模板配置

3.**没有配置template, 再判断是否配置了el选项**, 如果配置了, 调用getOuterHTML(el) 获取配置项

4. 调用**compileToFunctions**, 将template编译为带有**render, staticRenderFns**函数的渲染函数对象

5. 设置**options.render = render, options.staticRenderFns = staticRenderFns**

执行**mount.call(this, el, hydrating)**将组件渲染到页面上.

执行**mountComponent(this, el, hydrating)** src/core/instance/lifecycle.js

1. 设置**vm.$el = el**

2. 检测$options上**是否有render,** 没有则**创建createEmptyVNode并赋值给vm.$options.render**, 并在非生产环境给出警告

3. 执行**callHook(vm, 'beforeMount')**

4. 声明变量**let updateComponent = () => vm.\_update(vm.\_render(), hydrating)**

5. 创建组件级别的watcher.

new Watcher(vm, updateComponent, noop, { // 组件更新前的钩子函数 before () { if (vm.\_isMounted && !vm.\_isDestroyed) { callHook(vm, 'beforeUpdate') } } }, true /\* isRenderWatcher \*/)

6. 判断vm.$vnode === null, (// manually mounted instance, call mounted on self. mounted is called for render-created child components in its inserted hook)

1. 设置**vm.\_isMounted = true**

2. 执行**callHook(vm, 'mounted')**钩子函数