Vue生命周期

1、new Vue()声明一个vue实例，然后调用this.\_init 方法开始 Vue 的生命周期。

完整的初始化生命周期代码：

initLifecycle(vm) //vm 的生命周期相关变量初始化

initEvents(vm) // vm 的事件监控初始化

initRender(vm) // 模板解析变量初始化

callHook(vm, 'beforeCreate')

initState(vm) //vm 的状态初始化，prop/data/computed/method/watch 都在这里完成初始化，是响应式的关键步！

callHook(vm, 'created')

if (vm.$options.el) {

vm.$mount(vm.$options.el) //模板编译入口

}

initLifecycle 主要是初始化 vm 实例上的一些参数；initEvents 是事件监控的初始化；initRender 是模板解析，值得一提的是在 2.0 的版本中这一块有很大的改动，1.0 的版本中 Vue 使用的是 DocumentFragment 来进行模板解析，而 2.0 中作者采用的 John Resig 的 HTML Parser 将模板解析成可直接执行的 render 函数，这是模板预编译和服务端渲染（SSR）的前提；callHook(vm, ‘beforeCreate’)是执行钩子函数，就是你在 new Vue 实例的时候写的 beforeCreate 方法；initState 是本文的重点，我们下一节会详细讲到；callHook(vm, ‘created’)也是执行钩子函数；最后是是执行 mount 函数。

2、init Events和lifecycle

3、initState

这些初始化都涉及到数据转化为 Observer 对象的过程

export function initState (vm: Component) {

vm.\_watchers = [] //新建一个订阅者列表

const opts = vm.$options

if (opts.props) initProps(vm, opts.props) // 初始化 Props，与 initData 差不多

if (opts.methods) initMethods(vm, opts.methods) // 初始化 Methods，Methods 的初始化比较简单，就是作用域的重新绑定。

if (opts.data) {

initData(vm) // 初始化 Data，响应式关键步

} else {

observe(vm.\_data = {}, true /\* asRootData \*/) //如果没有 data，则观察一个空对象

}

if (opts.computed) initComputed(vm, opts.computed)// 初始化 computed，这部分会涉及 Watcher 类以及依赖收集，computed 其实本身也是一种特殊的 Watcher

if (opts.watch) initWatch(vm, opts.watch)// 初始化 watch，这部分会涉及 Watcher 类以及依赖收集

}

4、initData

这个函数做了以下几件事情：

1）保证 data 为纯对象

2）判断与 props 里的属性是否有重复，有就报错

3）进行数据代理，方便数据读取，代理后我们可以直接使用 vm.key，而不需要 vm.\_data.key

调用 observe 方法，这是响应式的关键步！

所有data都绑定在了vm实例上，类似vm下的全局变量。若有同名的变量或method方法名，会导致变量或函数的覆盖问题。

function initData (vm: Component) {

let data = vm.$options.data

data = vm.\_data = typeof data === 'function'

? data.call(vm)

: data || {}

if (!isPlainObject(data)) {// 保证data必须为纯对象

......

}

// proxy data on instance

const keys = Object.keys(data)

const props = vm.$options.props

let i = keys.length

while (i--) {

if (props && hasOwn(props, keys[i])) {// 是props，则不代理

...... //如果和 props 里面的变量重了，则抛出 Warning

} else {// 否则将属性代理的 vm 上，这样就可以通过 vm.xx 访问到 vm.\_data.xx

proxy(vm, keys[i]) //proxy方法遍历 data 的 key，把 data 上的属性代理到 vm 实例上

}

}

// observe data

observe(data, true /\* asRootData \*/) //关键步！observe(data, this)方法来对 data 做监控

}

5、observe

observe 方法会为传进来的 value 值创建一个 Observer 对象，创建之前会做一些判断

export function observe (value: any, asRootData: ?boolean): Observer | void {

if (!isObject(value)) { //如果不是对象和数组则不监控，直接返回

return

}

let ob: Observer | void

//判断 value 是否已经添加了 \_\_ob\_\_ 属性，并且属性值是 Observer 对象的实例。避免重复引用导致的死循环

if (hasOwn(value, '\_\_ob\_\_') && value.\_\_ob\_\_ instanceof Observer) { //如果是就直接用

ob = value.\_\_ob\_\_

} else if (

observerState.shouldConvert && //只有 root instance props 需要创建 Observer 对象

!isServerRendering() && //不是服务端渲染

(Array.isArray(value) || isPlainObject(value)) && //数组或者普通对象

Object.isExtensible(value) && //可扩展对象

!value.\_isVue // 非 Vue 组件

) {

ob = new Observer(value) //关键步！在 value 满足上述条件的情况下创建一个 Observer 对象

}

if (asRootData && ob) {

ob.vmCount++

}

return ob // 返回一个 Observer 对象

}

observe 方法主要就是判断 value 是否已经是 Observer 对象，如果是直接返回；否则，若干个判断条件成立则将这个对象转化为 Observer 对象。

上面基础知识我们已经讲过 Observer 类，其构造函数主要做了这么几件事：

首先创建了一个 Dep 对象实例；

然后把自身 this 添加到 value 的 \_\_ob\_\_ 属性上；

最后对 value 的类型进行判断，如果是数组则观察数组，否则观察单个元素（要理解这一步是个递归过程，即 value 的元素如果符合条件也需要转化为 Observer 对象）。

6、defineReactive

defineReactive 是对 Object.defineProperty 方法的包装，结合 observe 方法对数据项进行深入遍历，最终将所有的属性就转化为 getter 和 setter。至此，所有的数据都已经转换为 Observer 对象。即数据的读操作都会触发 getter 函数，写操作都会触发 setter 函数。

export function defineReactive (

obj: Object,

key: string,

val: any,

customSetter?: Function

) {

const dep = new Dep() //每个对象都会有一个 Dep 实例，用来保存依赖 (Watcher 对象)

......

let childOb = observe(val) //结合 observe 函数进行将对象的对象也变成监控对象

// 最重点、基石、核心的部分：通过调用 Object.defineProperty 给 data 的每个属性添加 getter 和 setter 方法。

Object.defineProperty(obj, key, {

enumerable: true,

configurable: true,

get: function reactiveGetter () {

const value = getter ? getter.call(obj) : val

// 依赖收集的重要步骤

if (Dep.target) {//如果存在 Dep.target 这个全局变量不为空，表示是在新建 Watcher 的时候调用的，代码已经保证

dep.depend() // 依赖收集

if (childOb) {

childOb.dep.depend() // 处理好子元素的依赖 watcher

}

if (Array.isArray(value)) { // 如果是数组，进一步处理

dependArray(value)

}

}

return value

},

set: function reactiveSetter (newVal) {

const value = getter ? getter.call(obj) : val

......

childOb = observe(newVal) // 对新数据重新 observe，更新数据的依赖关系

dep.notify() // 通知 dep 进行数据更新，这个方法在前面的 Dep 类中讲过

}

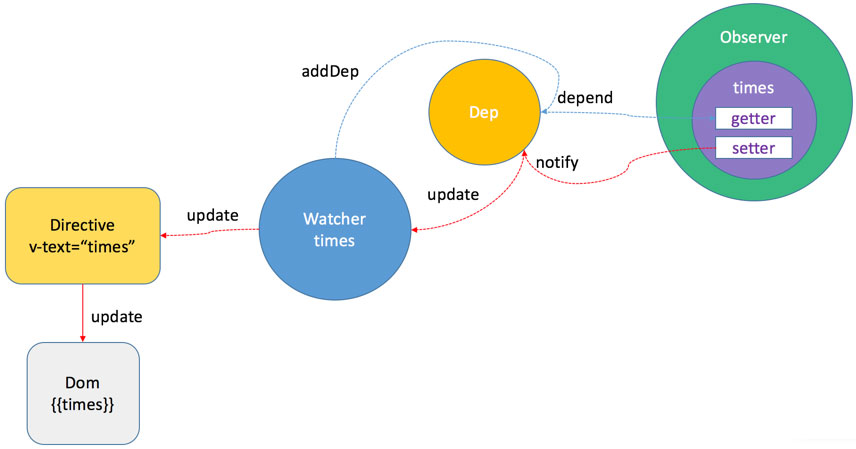
})

}

7、依赖收集

依赖收集是通过属性的 getter 函数完成的，文章一开始讲到的 Observer、Watcher、Dep 都与依赖收集相关。其中 Observer 与 Dep 是一对一的关系， Dep 与 Watcher 是多对多的关系。Dep 则是 Observer 和 Watcher 之间的纽带。依赖收集完成后，当属性变化会执行主题对象（Observer）的 dep.notify 方法，这个方法会遍历订阅者（Watcher）列表向其发送消息，Watcher 会执行 run 方法去更新视图。依赖的关联关系具体可参考下图

其实 observeArray 方法就是对数组进行遍历，递归调用 observe 方法，最终都会走入 walk 方法（代码很简单，就不单独列出来，可以戳链接进入 github 查看）监控单个元素。而 walk 方法就是遍历对象，结合 defineReactive 方法递归将属性转化为 getter 和 setter。



总结依赖关系建立的步骤：

1）模板编译过程中的指令和数据绑定都会生成 Watcher 实例，watch 函数中的对象也会生成 Watcher 实例，在实例化的过程中，会调用 watcher.js 中的 get 函数 touch 这个 Watcher 的表达式或函数涉及的所有属性；

2）touch 开始之前，Watcher 会设置 Dep 的静态属性 Dep.target 指向其自身，然后开始依赖收集；

3）touch 属性的过程中，属性的 getter 函数会被访问；

4）属性 getter 函数中会判断 Dep.target（target 中保存的是第 2 步中设置的 Watcher 实例）是否存在，若存在则将 getter 函数所在的 Observer 实例的 Dep 实例保存到 Watcher 的列表中，并在此 Dep 实例中添加 Watcher 为订阅者；

5）重复上述过程直至 Watcher 的表达式或函数涉及的所有属性均 touch 结束（即表达式或函数中所有的数据的 getter 函数都已被触发），Dep.target 被置为 null，依赖收集完成；

以上就是模板中的指令与数据关联起来的步骤。当数据发生改变后，相应的 setter 函数被触发，然后执行 notify 函数通知订阅者（Watcher）去更新相关视图，也会对新的数据重新 observe，更新相关的依赖关系。

**Vue响应式原理的源码介绍，总结来说就是**：

在生命周期的 initState 方法中将 data、prop、method、computed、watch 中的数据劫持，通过 observe 方法与 defineReactive 方法将相关对象转换为 Observer 对象；

然后在 initRender 方法中解析模板，通过 Watcher 对象，Dep 对象与观察者模式将模板中的指令与对应的数据建立依赖关系，在这个依赖收集的过程中，使用了全局对象 Dep.target ；

最后，当数据发生改变时，触发 Object.defineProperty 方法中的 dep.notify 方法，遍历该数据的依赖列表，执行其 update 方法通知 Watcher 进行视图更新。