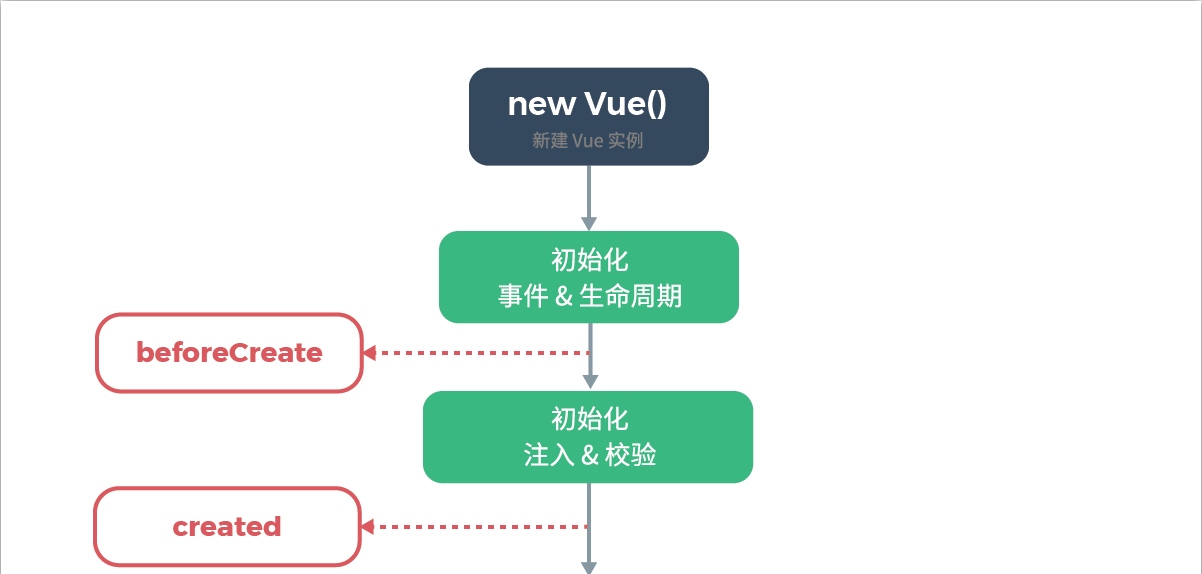
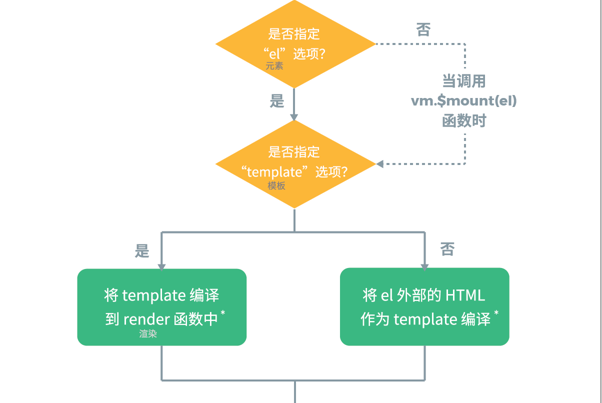
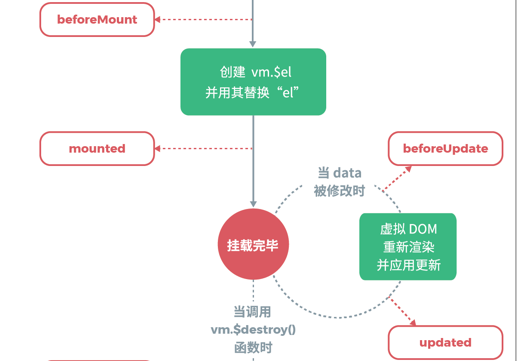
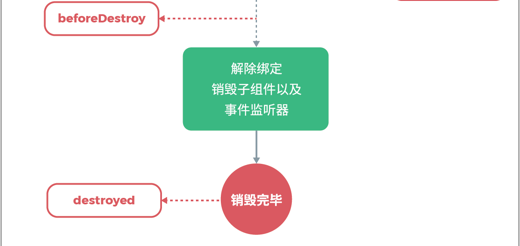
# Vue生命周期

### vue生命周期

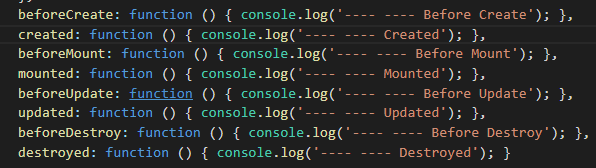








一个 vue对象从创建到销毁，提供了8个不同钩子函数，而这几个钩子函数降vue的生命周期划分了几个部分。



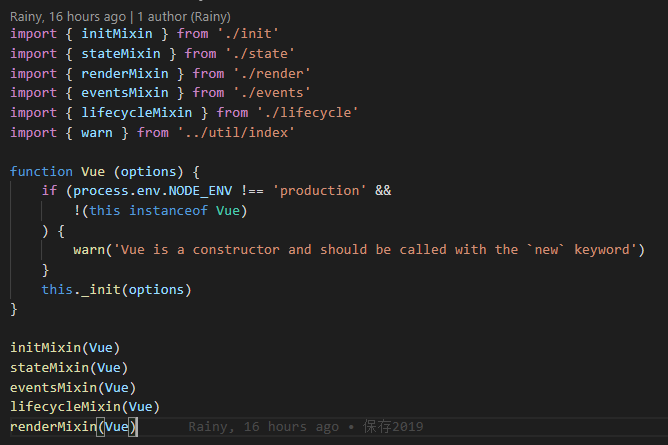
前面两个阶段是初始化vm实例对象，并通过opts添加对应的方法和属性。 实现viewmodel层和数据层。

中间两个阶段是挂载，将视图层和数据层 通过 vm实例关联起来，实现视图和数据的双向绑定。

数据更新的两个钩子会在数据变化的两个阶段调用。

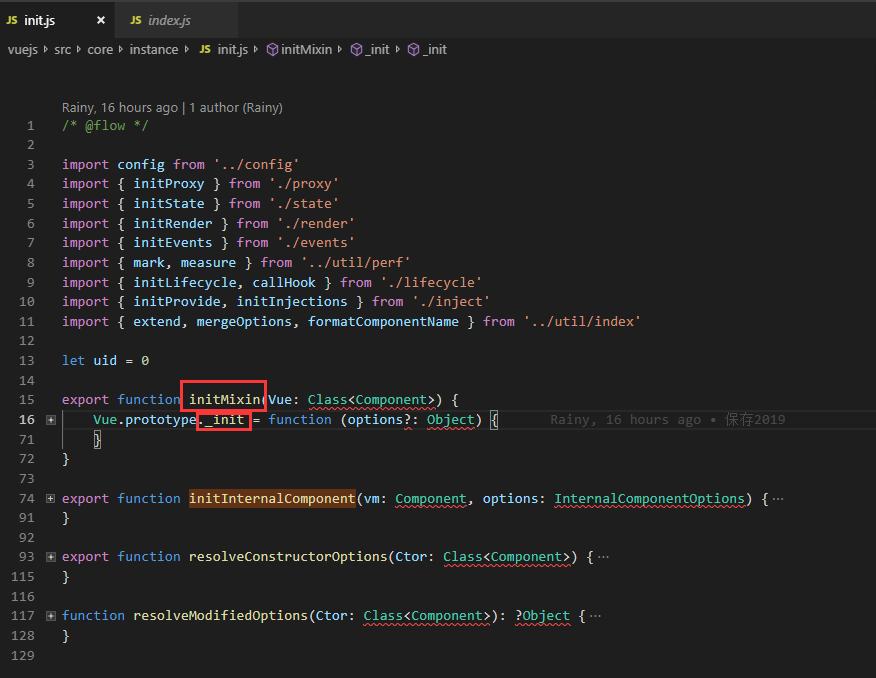
最后两个阶段会在vue实例销毁的时候调用。

### Vue 构造函数



简单分析一下各部分的作用。

InitMIxin (Vue)



给Vue构造函数的原型对象增加了一个\_init();//具体干了什么后面再看。

stateMixin(Vue):



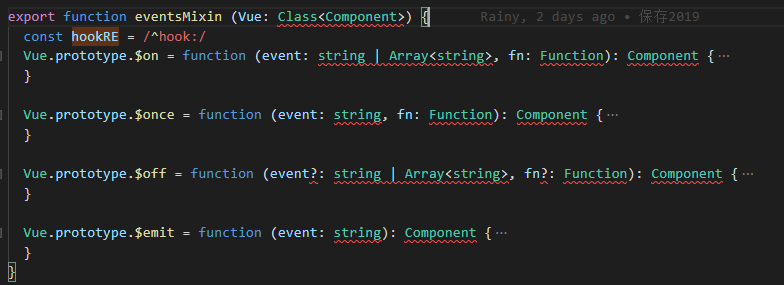
给Vue的原型对象

增加 $data,$props两个属性，通过设置了get分别从实例的\_data,\_props取值。

增加 $set $delete 动态往data上增肌和删除双向绑定监听的数据。

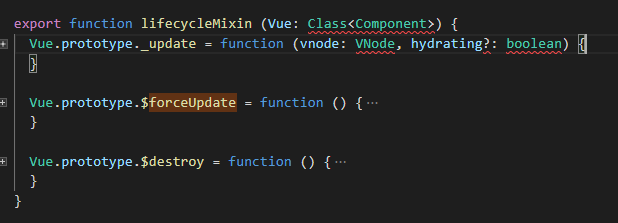
增加 $watch 动态设置监听数据变化的回调。

eventsMixin(Vue):



同样给Vue的原型增加四个方法，$on,$once,$off,$emit,分别对应绑定自顶替事件，触发一次的绑定自定义事件，解除自定义事件绑定，和发送（触发）自定义事件

lifecycleMixin(Vue):

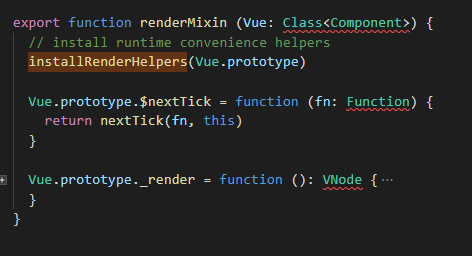


同样也给Vue的原型加了上方法，其中\_update,vue中约定\_开头的方法是私有方法，不是暴露出来的方法我们暂时不看，后面用到看这部分。

$forceUpate:强制按最新的数据更新视图。例：显示一个数据列表，改变数组第一个元素的某个属性，vue可能监听不到变化，官方建议通过$set改变，但是如果我们还是arr[0].X=’X’去改变，之后使用这个函数，强制刷新视图也可以达到效果。并触发beforeUpdate和updated钩子函数。

$destroy:删除当前实例，并触发beforeDestroy和dedestroyed 钩子函数。

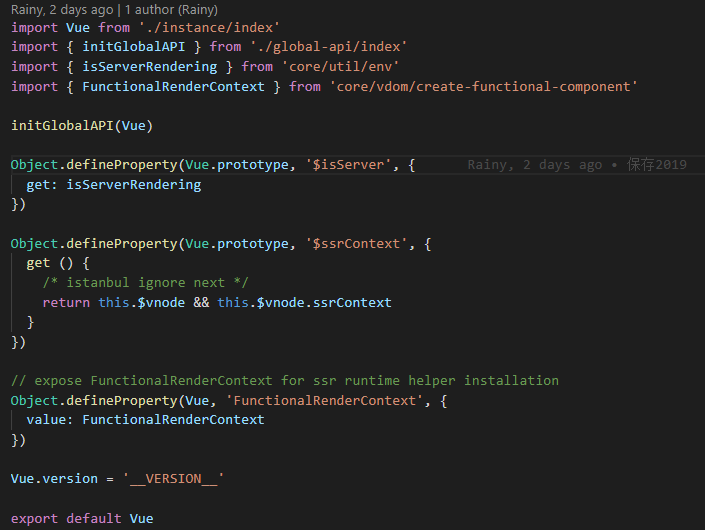
renderMixin(Vue):

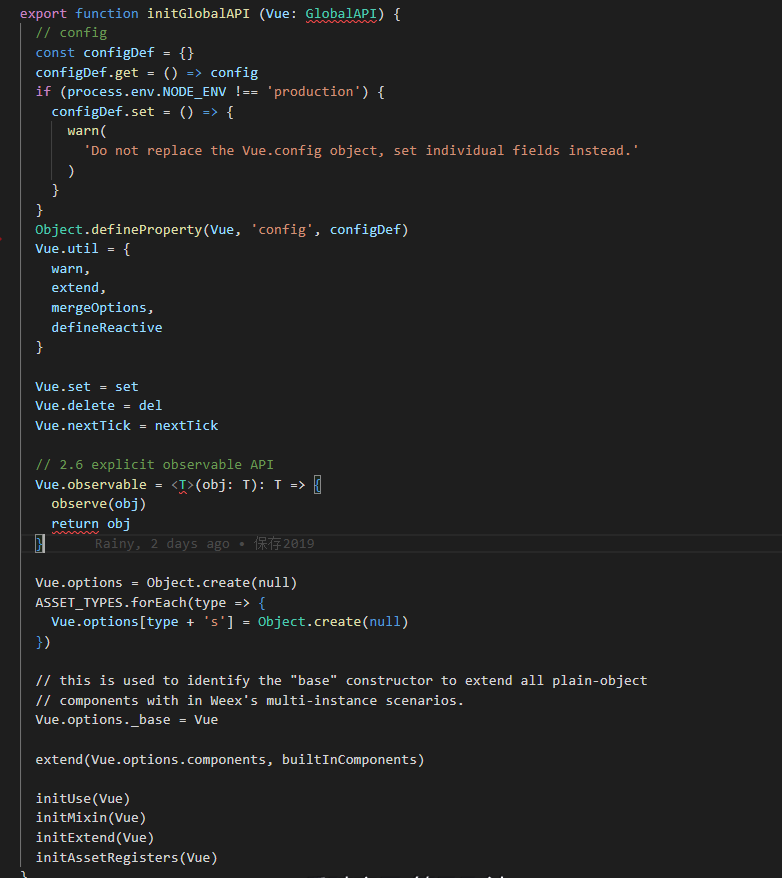


\_render 和 installRenderHelpers函数都是向Vue原型增加的内部方法。

$nextTick:将回调延迟到下次 DOM 更新循环之后执行，如果有异步操作，这个函数会在所有同步操作之后 异步之前（Updated之后）调用。

最后 将Vue exprot出去的时候，





initGlobalAPI(Vue):给Vue构造函数上增加了一些静态属性和静态方法。

Config: 一些配置（dev or product）

Set，del,设置（删除u）对应属性值并监听变化。

Nextick，observable，

初始化 options

并将属性components,directives,filters 初始化为null，

然后将options .components 默认加上 KeepAlive

并将options .\_base 设置为Vue.

给Vue加上静态方法 use，mixin，extend，component,directive,filter.

$isServer:判断代码是否运行在服务器上；

$ssrContext:

### vue实例化。

现在我们按照官方提供的生命周期图示看看从vue实例创建到销毁中间都做了那些事情。

首先是通过new来调用Vue函数 到beforeCreated钩子被调用这中间，

初始化事件和生命周期。

Vue构造函数：

function Vue (options) {

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

!(this instanceof Vue)

) {

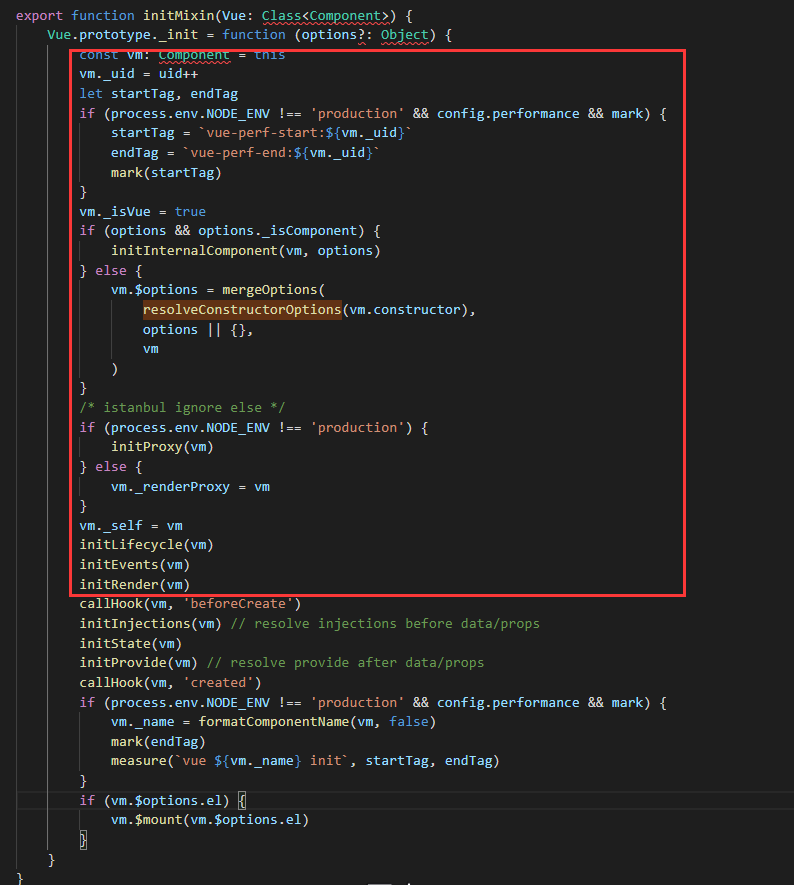
warn('Vue is a constructor and should be called with the `new` keyword')

}

this.\_init(options)

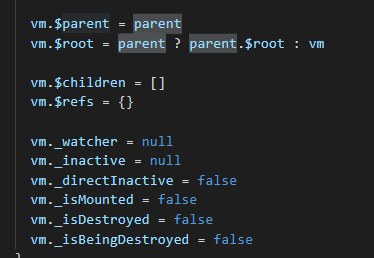
}

在new Vue(opts) 这个过程中，会首先调用\_init方法，并将接受的参数传入进去。



在\_init中，除了参数和环境的判断之后，执行了三个initLifeCyle(vm),initEvent(vm),initRender(vm),按照字面理解，分别是初始化生命周期，初始化事件和初始化渲染器。

initLifeCyle(vm)：



初始化生命周期相关的参数。

$parent 指定已创建的实例之父实例，在两者之间建立父子关系。子实例可以用 this.访问父实例，子实例被推入父实例的children 数组中。（这里的父组件是指第一次非抽象父组件 ，抽象组件例如 keeplive transition 会继续往上寻找）

$root 当前组件树的根 Vue 实例。如果当前实例没有父实例，此实例将会是其自己。

$children 当前实例的直接子组件。需要注意 $children 并不保证顺序，也不是响应式的。

$refs 一个对象，持有已注册过 ref 的所有子组件。

\_watcher 组件实例相应的 watcher 实例对象。

\_inactive 表示keep-alive中组件状态，如被激活，该值为false,反之为true。

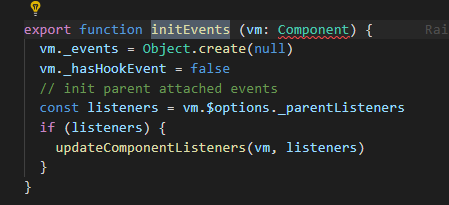
\_directInactive 也是表示keep-alive中组件状态的属性。

\_isMounted 当前实例是否完成挂载(对应生命周期图示中的mounted)。

\_isDestroyed 当前实例是否已经被销毁(对应生命周期图示中的destroyed)。

\_isBeingDestroyed 当前实例是否正在被销毁,还没有销毁完成(介于生命周期图示中deforeDestroy和destroyed之间)。

InitEvents(vm):



初始化事件参数。

\_event: 储存调用时@绑定的事件和当前实例使用on绑定的事件(before周期前不存在)。

\_hasHookEvent: 是否有@hook：created 这种方式调用生命周期钩子函数。

InitRender(Vm):

给实例附加渲染相关的属性和方法:

\_vnode:

\_staticTresss:

$slots: 静态插槽

$scopedSlots:作用域插槽

\_c:私有方法创建vnode，同时提供一个标准方法$createElement()供外部条调用。通过definedReactive 定义实例的$attr 和 $listeners

$attr：储存调用时，在组件上定义的 并且没有在props里面定义的属性。（不包括原生dom的属性），并将prop里面的属性绑定实例，未赋值。

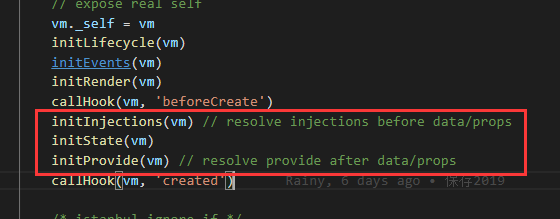
$listeners： 存储调用时通过@监听的事件，（\_events 还有实例通过on绑定的事件，并且一个事件名称可以多个事件回调，但是@方式只能一个。 @绑定的时间还可以通过on的方式增加回调）

完成了初始化事件（初始化\_events,$listeners）和初始化生命周期(ismounted...这些参数)。

触发beforeCreate钩子函数。



开始初始化注入&校验

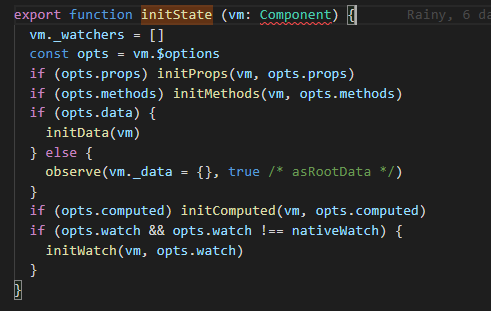


InitInjections(vm):



判断有没有inject参数，如果有，就该字段获取挂在vm实例上，provide 和 inject 绑定并不是可响应的。这是刻意为之的。然而，如果你传入了一个可监听的对象，那么其对象的属性还是可响应的。

InitState(vm):



initProps: 验证props传值的type，并且将vm上对应的之代理到props里面，设置对应值的数据监听。

InitMethods:根据methods在vm上添加对应的方法，并指向相同的方法，并绑定this指向当前实例，

InitData:根据data在vm上添加对应的属性，并将get，和set代理到data中的数据上，设置对应值的数据监听

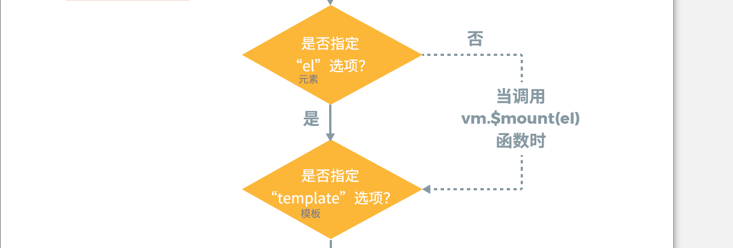
InitComputed:给每个值设置监听，并且当对呀属性调加到实例上，get代理到computed商对应的函数的返回值。

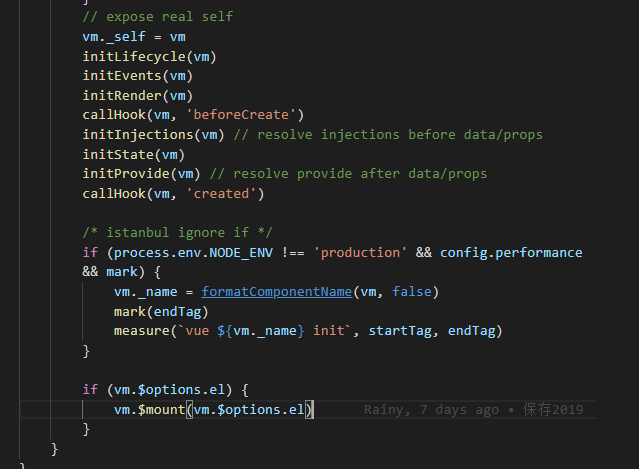
InitWatch:给watch对应值的变化添加一个回调函数。

这阶段已经将数据的双向绑定准备好，并且将所有数据的初始值注入，并监听。后面改变这些被监听的值，都会触发回调。即上面的初始化注入和校验。整个数据和视图未绑定，因为还没有视图层。（后面如果直接替换整个数组或者对象 然后改变其中的属性，视图不跟变化，就是因为第一次初始化的时间监听的那个对象被整体替换了，第一次监听的对象被整体替换了，而新增设置的那个对象和属性没有被监听，也就是对应变化的时候没有回调函数给他调用，所以变化触发不了视图层变化。所以需要使用$set）.



然后调用created钩子函数，到这里代表vue对象已经初始化完成了。



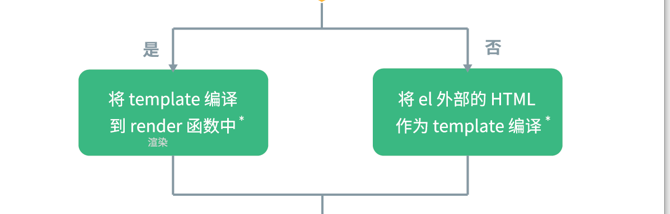


数据层的初始化和vue实例初始化到这里都完成，准备将视图层和数据层关联起来的动作mount。

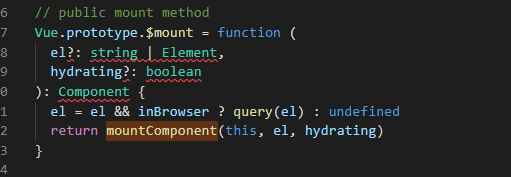
判断是否el参数，如果有，就直接盗用Vm.$mount(vm.$options.el)进行挂载，结束。

如果没有el参数，就等手动自己使用Vm.$mount进行挂载。

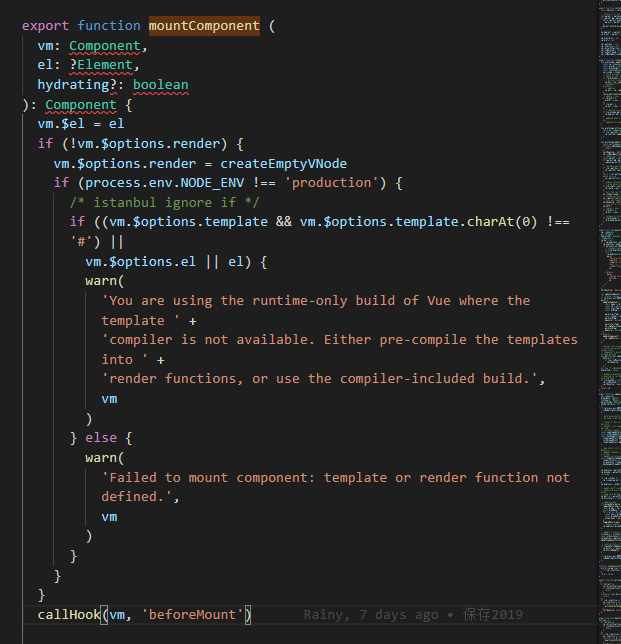
将vm挂载的过程，也是就讲视图和数据通过vm实例进行双向绑定的过程：



1. 判断是否有template 参数，如果有就讲tempale编译到render函数中。
2. 如果没有，就获取el参数 的outhtml内容 作为template的编译到render参数中。



再看看 mountComponent 函数。



MountComponent开始到beforeMount钩子：

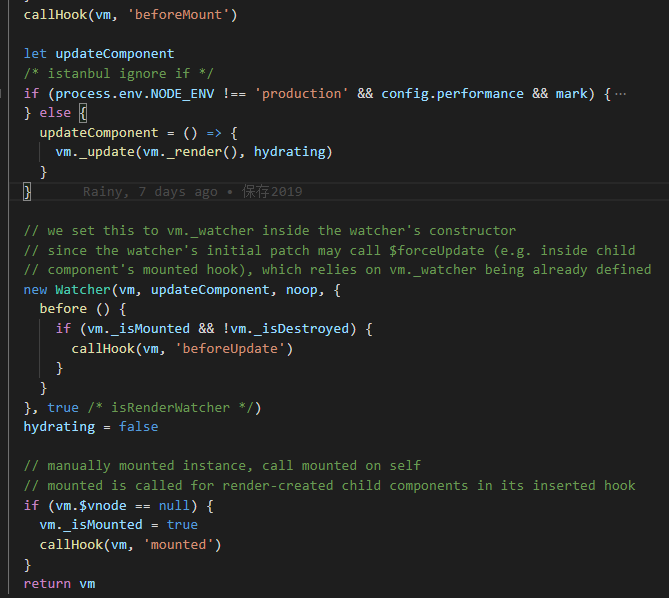
首先将el的获取到的dom挂载实例的$el属性上。

判断有没传入的render函数，没有就创建一个孔的vnode。

触发回调 beforeMount。



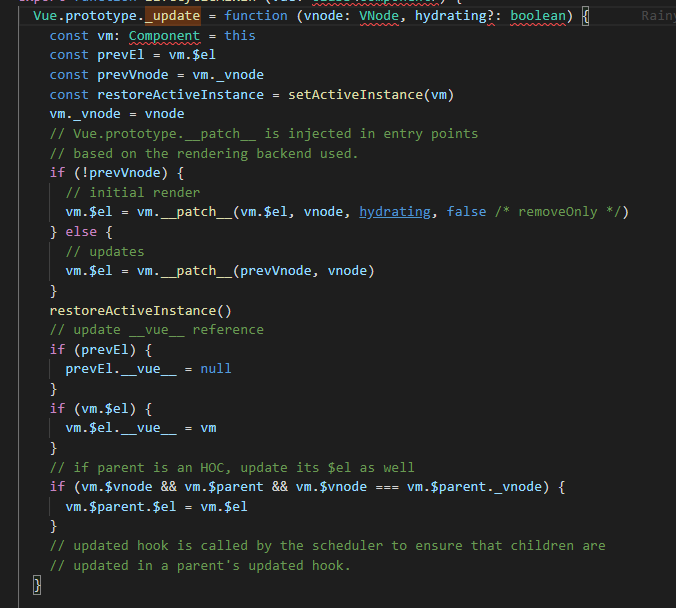
BeforeMount ===> mount:



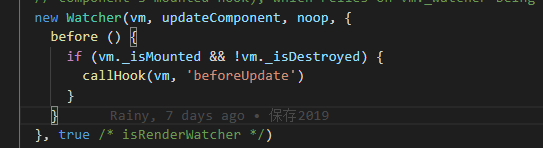
申明一个变量updateComponent 等于一个函数，函数内部调用vm.\_update(vm.\_render(), hydrating)。

vm.\_render():返回当前实例的vnode对象。

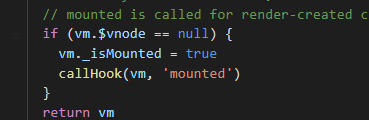
vm.\_update()：将实例渲染成真是dom。



Vm.\_\_patch\_\_ ，将虚拟dom渲染成实际的方法，而且在不同的平台上各有不同。



设置渲染监听，并将updateComponent 函数传入，用来更新视图，并设置数据更新的钩子函数。



挂载完成 设置标识属性和调用回调。