# 组件化——createComponent

#### 我们都知道createElement是将带入的参数转换为对应的VNode，然后将VNode提供给vm.\_update()方法去将VNode转换为真实的DOM然后patch到页面上。在createElement函数中，第一个参数是html中存在的标签，可以创建出对应的VNode，如果第一个参数是组件，依旧可以创建出VNode。这是怎么做到的？

答：首先来明晰一下createElement的使用场景：

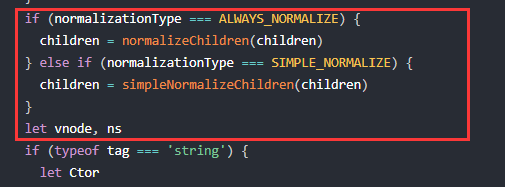


createElement就是为了创建VNode所用的函数，如果想要创建的VNode只表示一个简单的dom节点，那么只需要在第一个参数带入字符串类型的标签名即可。如果希望创建的是一个单纯的文本节点，createElement无法直接提供，但是可以间接地将文本作为子节点，然后让这个父节点是一个Inline元素即可。

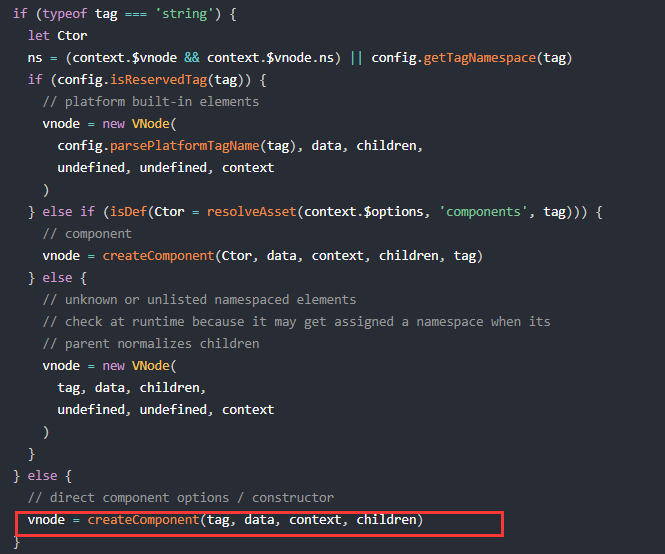
但如果希望VNode表示一个组件，那第一个参数写成这个组件即可。

因此，createElement就不得对第一个参数的类型作出区分，从而作出对简单标签和组件的不同转换方式。

首先，我们得知道无论第一个参数是什么，他的第二个参数和第三个参数可以说都是一样的。尤其第三个参数都是用来描述这个简单标签下，或者这个组件下的子节点（也就是slot中的内容），因此规范化子节点是必须的。



可见，标准化子节点在区分第一个参数值前就已经进行了。

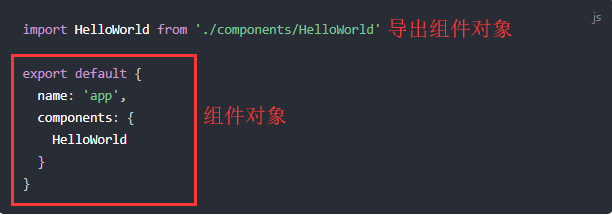


到这里我们已经可以判断createComponent就是封装了所有处理组件逻辑的函数。只要清楚createComponent做了什么我们就知道了如何对组件进行转化成VNode的处理。

## 那createComponent都做了什么？

答：首先我们都知道createComponent将组件对象转换为VNode，想要彻底搞懂Vue接下来的处理思路，我们就不得不弄清组件对象到底是个什么东西？

其实这个东西很简单，就是我们在script中export default出去的对象。或者是Vue.component的第二个参数，他们都是描述组件的对象，都具备组件的特征。

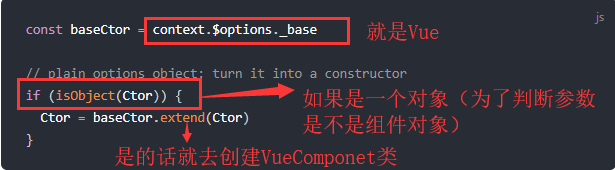


组件对象描述了组件，createElement的第二个参数描述了这个组件的在父组件的‘标签’状态。普通的VNode只有第二个参数所描述的‘标签’状态即可以描述他变成真实dom的状态。而组件可不行，他必须得利用好组件对象描述的内容，走一个类似于Vue实例的初始化过程才可，不然这个组件在使用的时候也顶多只是一个普通的dom而已。

回到createComponent函数中，我们可以看到createComponent做了很多事情，但是总体可以分为三个过程：**构造子类构造函数、安装组件钩子、实例化vnode**。

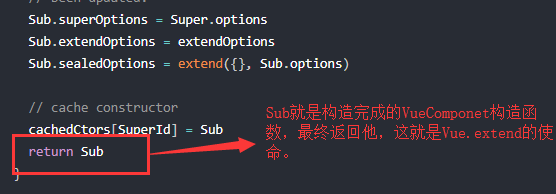
首先为了走一个类似于Vue初始化的流程，他需要将组件对象跑一边Vue实例的构造函数，可Vue的构造函数毕竟和组件的构造函数存在区别，比如一些全局的API等需要和Vue类不同，所以现在我们需要创建一个继承于Vue组件的构造函数VueComponent。

首先先判断带入createComponent的第一个参数Ctor是不是一个对象（组件对象就是export的普通对象嘛）



进入extend后就去创建VueCompnent类

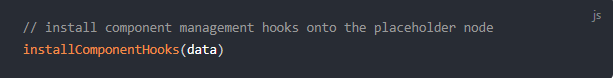




***TIPS: 这里只是返回了Sub（VueComponent），还没有实例化任何东西，之后实例化组建的时候会用到，我们暂且不提。***

目前为止createComponet完成了第一步——构造子类构造函数。

接下来它需要为这个将要创建的组件安装钩子函数，这些钩子函数将会在VNode执行patch过程中被执行，现在我们需要把钩子函数合并在data.hook中待命：

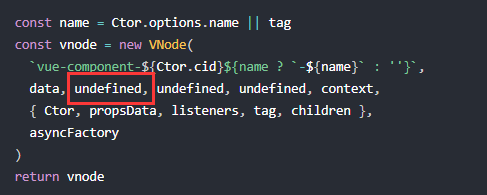


至此，createComponent的第二项使命已经完成。

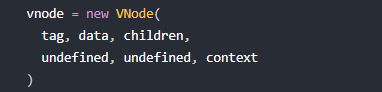
接下来就是最后一步——实例化VNode。具体怎么实例化VNode我们只前已经提及。

**只不过现在需要注意的是，组件的VNode是没有children的！另外，组件的VNode的参数与普通节点的VNode也有很大差距的，这些参数就是Vue区分VNode种类的关键，这些独有的参数也包含了这个组件的构造函数（可以发现在Vue.extend()的时候返回的是一个构造函数，这个返回的构造器都有不通的特征，所以不同的组件都有不同的构造函数，这个构造函数就相当于携带了Vue组件的灵魂，如生命周期.....），换句话说，组件的VNode就是所谓的渲染VNode，他的部分属性携带了他身为组件的特征和灵魂。**

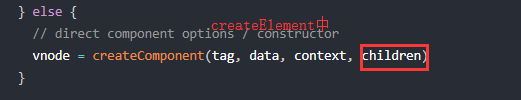
组件的VNode：



普通标签的VNode：



***TIPS：可以注意到其实在createElement调用createComponet的时候其实是带入标准化后的children的，只不过现在没用上这个参数（目前认为这个children就是undefined，因为children是作为createElement的第三个参数，如果带入的第一个参数是组件，那就没有第三个参数。）。***



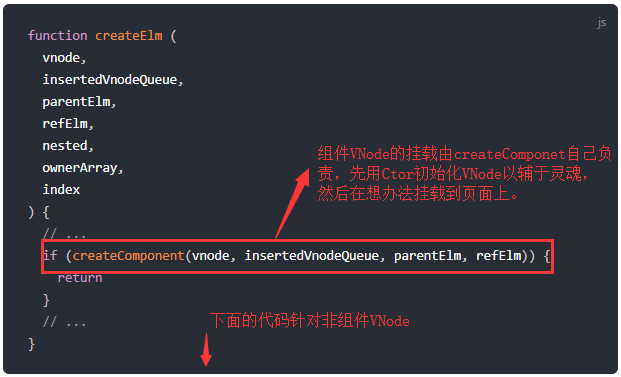
## 那么处理完App组件的初始化之后，在update中的patch函数是怎么处理组件的？

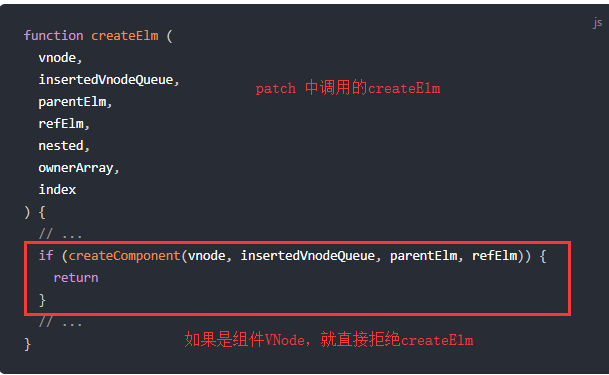
答：首先我们需要明确updata调用的patch是在接收到VNode之后负责将VNode描述的结构挂载到页面上，上一步我们已经通过createElement把组件的VNode得到，上问题说过，组件的VNode携带了他的灵魂，其中有一些属性包含了这个组件的构造函数，现在我们要挂载它，**首先要做的必然就是用构造函数赋予这个组件灵魂，然后在渲染到页面上（Vue实例也是先走一个赋予灵魂的过程，如果初始化生命周期，事件中心...然后在挂载到页面上的）**。

\_update处理非组件的时候，主要调用了createElm和insert两个方法，createElm方法可以深入优先遍历vnode.children的tag，然后根据render中的tag创建出相应的DOM，最后通过insert方法插入到真正的DOM容器中。但是如果是VNode是组件类型的，他没有原生DOM标签（是一个vue-componet-...的标签），肯定不能通过nodeOps封装的一系列dom方法去把他插入在DOM中。组件是一个复杂的结构，他需要具有Vue实例的系列能力，在插入dom中还要有组件的灵魂，也就是说现在需要执行他的构造函数去赋予灵魂，这就需要一个全新的方法去接管，也就是createComponet方法。

回想上一个问题中，我们说\_createElement函数判断出带入参数中的是一个组件类型，所以他创建了不同于普通DOM节点的VNode，这个VNode具有普通DOM节点VNode的不具有的特征，因此update函数作为将VNode处理成真正DOM并挂载在页面上的角色，他就不得不分清这个由\_createElement函数转换而来的VNode到底是表示组件的还是普通节点的。

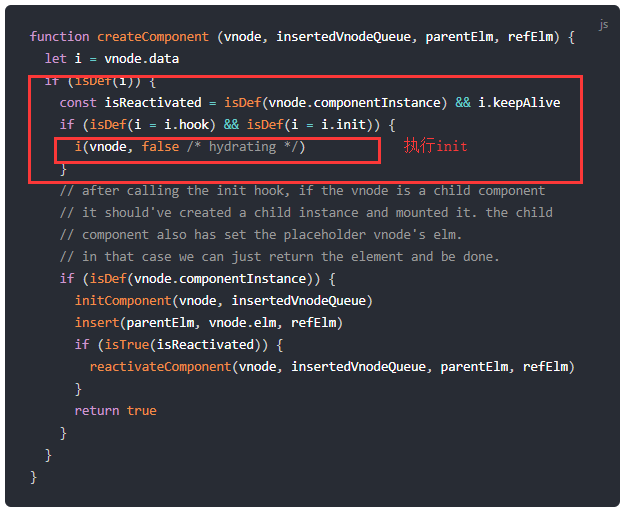
辨别手段也就是createComponent方法。通过返回一个布尔值的方式来决定是否劫持createElm的继续执行。



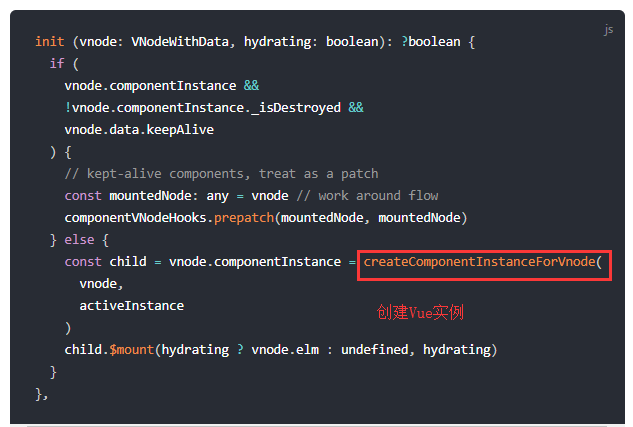


既然拒绝了createElm，那createComponent就得具有能代替createElm处理组件VNode的能力。

createComponent一上来就执行了组件的init钩子函数去初始化这个组件VNode，也就是说在此时他要让这个组件VNode具有Vue的灵魂。这个init钩子函数在上一个问题中谈到的createComponet方法中已经安装到了VNode.data.hook中（在生成组件VNode的时候就已经安装好了钩子）。



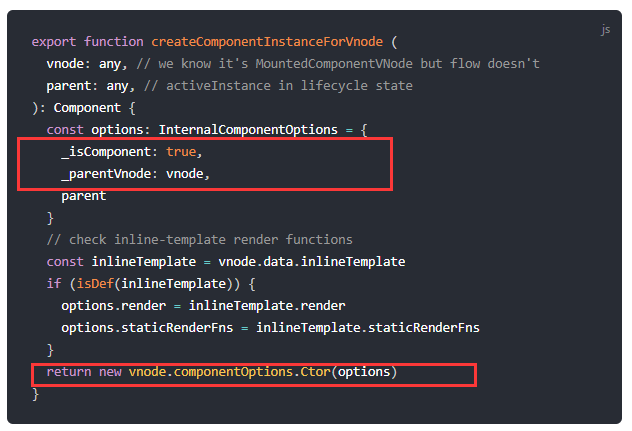
然后执行init，init钩子函数中包含了createComponentInstanceForVNode方法，这个方法的核心就是调用了这个组件的构造函数：



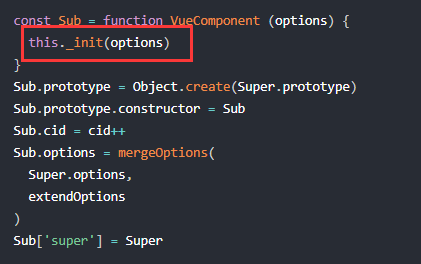
在执行构造函数前，他首先给这个组件VNode的option添加了两个标志位，因为要执行Vue实例的vm.init方法，有一些逻辑不适合组件初始化的时候执行（比如组件没有$el），所以用标志位来选择执行哪些逻辑：

分别是\_isComponent和\_parentVnode。

\_isComponent是设置为true，表示这个VNode是一个组件。然后提供了父Vnode给\_parentVnode。



最后调用上一个问题中提到的Vue.extend方法返回的构造函数（组件构造函数）来构造这个组件实例。所以我们回到Vue.extend方法返回的sub中。



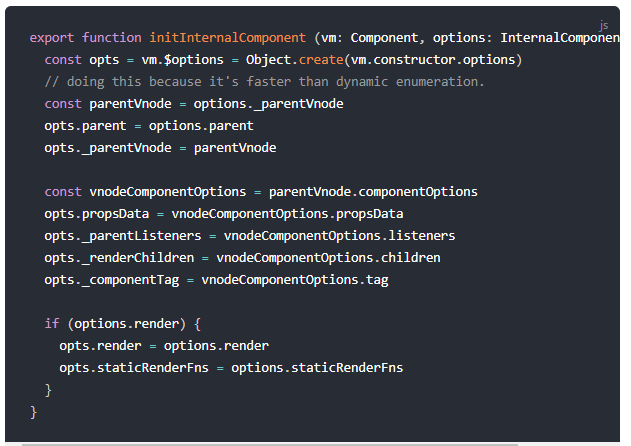
VueComponent继承于 Vue，因此，在这里他会跑一遍Vue.init()。带入的参数options也就相当于是.vue文件中export default的对象，也就是相当于执行了初始化Vue的过程：



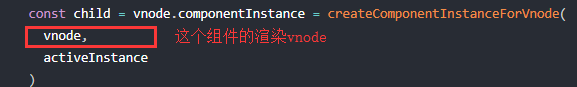
回想Vue的初始化过程，其中包含一个合并配置的过程，在这里，因为是组件，所以和根实例还是存在区别的（比如组件涉及到parentVnode....）。因此Vue.init()方法特对此作出逻辑分割：

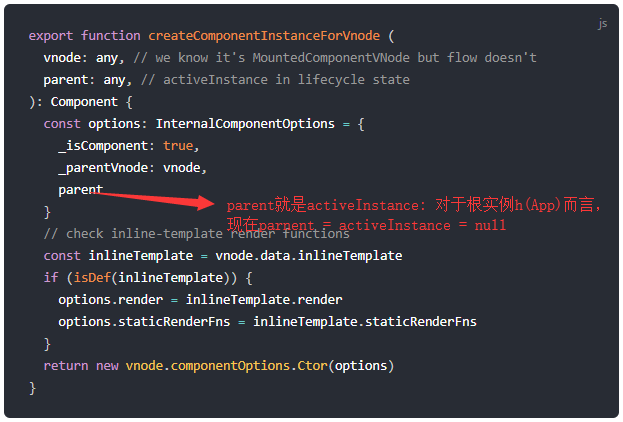


在createComponetInstanceForVnode中，因为为option添加了标志位\_isComponent所以这里执行initInternalComponent方法。



在这个方法中，我们只需要知道opts.parent = options.parent、opts.\_parentVnode = parentVnode，它们是把之前我们通过createComponentInstanceForVnode函数传入的几个参数合并到内部的选项$options里了，也就是说





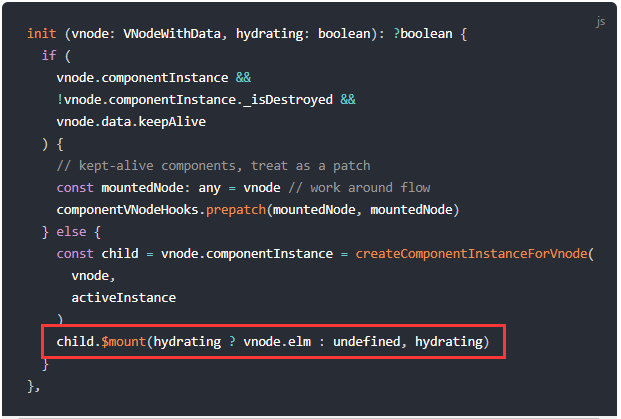
然后值得注意的是在最后挂载组件的时候，在vm.init中并没有和挂载Vue实例一样执行$mount方法，组件的$mount方法虽然和根实例的$mount是同一个，但是带入的参数不同，组件的参数是undefined而不是$el.

因为Vue组件和实例不同，并不是需要挂载到指定的DOM上，所以没有$el。



既然$mount不服务组件，那组件就得有自己特有的挂载方式，并且自己接管自己的$mount。

组件挂载的时机应该是在通过VueComponent构造完后执行（回想一下Vue实例也是在构造后接下来执行$mount的），所以回到组件的init钩子：



观察$mount，第一个参数相关与服务端渲染，我们考虑的是客户端，所以这里执行的是$mount相当于:child.$mount(undefined, false)。具体的逻辑同实例的$mount。值得注意的是原来$el的参数位置带入的是undefined，然后就和实例一样执行\_\_patch\_\_的逻辑即可实现深度优先的挂载。

我们知道在$mount中，通过\_update去挂载组件，在\_update中有几处关键的地方：

首先 vm.\_vnode = vnode 的逻辑，这个 vnode是通过 vm.\_render() 返回的组件渲染 VNode，

也就是说我们之前渲染出的组件VNode就是组件渲染VNode。

在这里，我们还需要知道vm.\_vnode和vm.$vnode的关系就是一种父子关系，用代码表达就是vm.\_vnode.parent === vm.$vnode。**至于细节暂时没研究明白，不过这里我们只需要知道js是单线程的，在深度递归patch的过程中父实例的追踪会丢失，但vue需要知道他的上下文Vue实例是什么，并把它作为子组件的父Vue实例。在挂载的某个过程中，parent.$children都存入了子节点，子节点也能通过$parent去获取。所以只要知道我们挂载的时候总是能获取到父子关系即可。**

**现在我们回过头来关注\_\_patch\_\_的过程，虽然执行的逻辑和之前讨论的根实例的\_\_patch\_\_一模一样，但是因为没有$el所以会有一点不同的细节，在这里重新走一个组件深度遍历\_\_patch\_\_的过程。**

**由于这个过程非常复杂，我们会根据vue-cli的初始化场景从头重新理顺一下整个过程：**