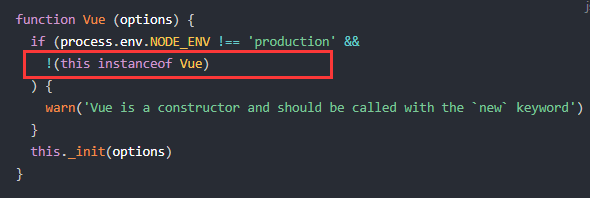
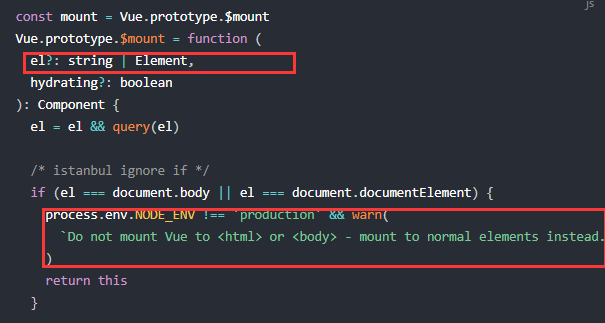
# vue源码分析（一）new Vue都发生了什么？

1. 因为Vue的构造函数有!(this instanceof Vue的这个限制)



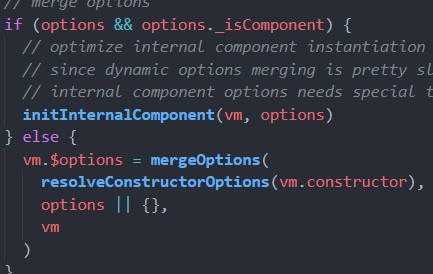
1. 需要注意到 el 可以是字符串也可以是具体的element，并且在开发环境下不能绑定到body或者html中。



1. 很好地判断逻辑，xxx存在然后xxx

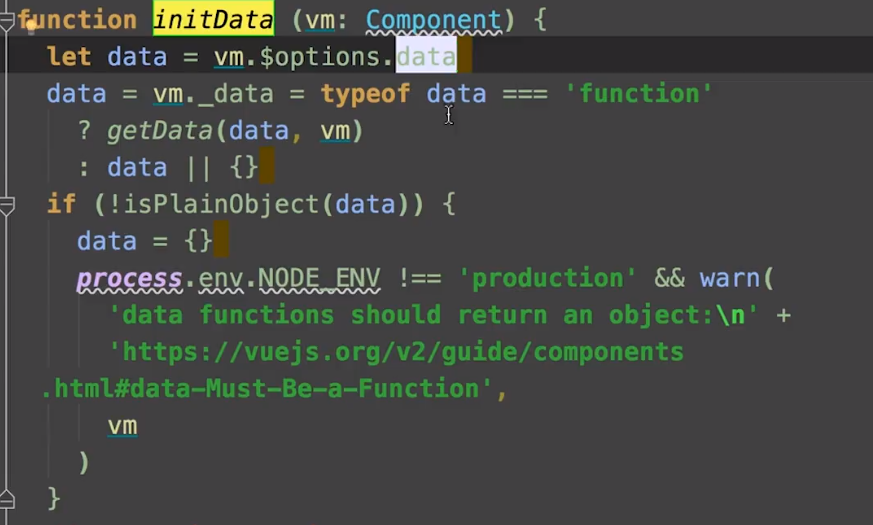


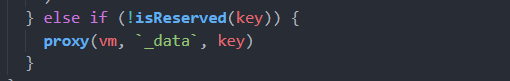
1. 所有实例中的el和template都会转换成render方法，这个编译的过程相当复杂,暂时没讲。另外render函数的参数createElement就是vm.createElement
2. Virtual DOM 除了它的数据结构的定义，映射到真实的 DOM 实际上要经历 VNode 的 create、diff、patch 等过程。那么在 Vue.js 中，VNode 的 create 是通过之前提到的 createElement 方法创建的，我们接下来分析这部分的实现。
3. 补充一点，在Vue构造函数中，除了判断了是否加new操作符外，主要是执行了this.\_init，在\_init中会让我们带入组件中的选项参数和vm.$options相关的属性混合。比如，$el会绑定带vm.$option.el。也就是说vm.$option.el = vm.$el。



1. 为什么在data中定义的数据可以再任何选项中通过this.xxx访问的到？

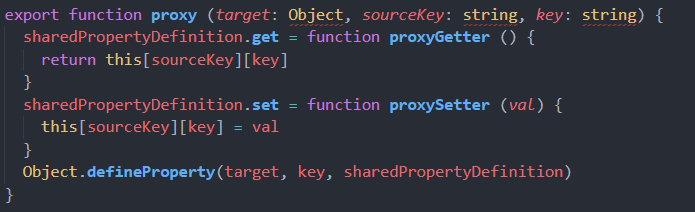
这时候还要回到Vue的构造函数，this.\_init()的执行事实上有分为了好几个init，比如initprops，initcomputed，当然也包括initData，这些init是为当前实例vm做的。vue能够做到通过在选项内通过this来访问data就是在initData的时候做了手脚。





因为Vue在初始化实例的时候会将选项中的内容混合到vm.$options上。因此data被绑定到vm.$options.data上。在initData的时候，先让vm.$options.data = data。然后用又将data赋值给vm.\_data，这样我们要访问的this.xxx的xxx就成为了vm.\_data的一个key。在initData临近最后的时候对要访问的xxx设置了一个代理。

proxy函数的定义如下：



最关键的是最后一步defineProperty。

这里的target就是vm实例。通过defineProperty为传来的key进行访问监听，key就是我们要访问的xxx，当我们set或者get这个传来的key的时候，就自动return当前实例的vm[‘\_data’][key]

举个例子：

我们要访问this.name。

那么在initData的时候先data = vm.$options.data。然后this.\_data = data.

然后当访问this.name时，name就会作为proxy中的第三个参数,target是当前vm实例，而定值字符串`\_data`是第二个参数。

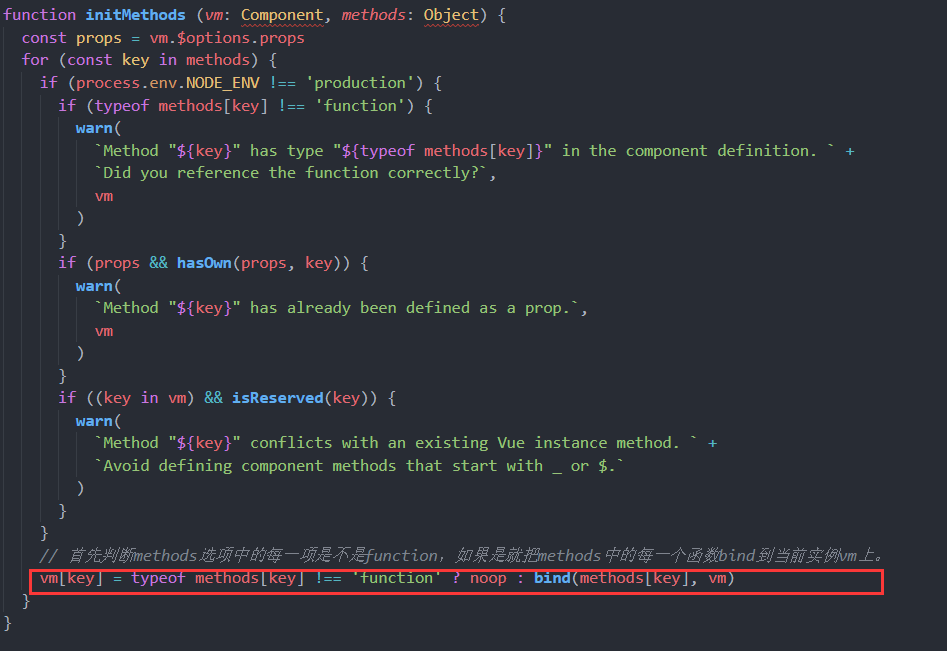
proxy的执行过程中，会使用Object.defineProperty对vm.xxx进行get和set监听。此时假如我们访问this,xxx，就会触发get函数，然后返回this[‘\_data’][xxx]。这样就实现了通过this就可以访问到data中的数据。

其实可以通过this访问props和methods中定义的内容也是类似的原理。他们都在init的最后调用了这个proxy。

那这也是props和methods等为什么不能重名的原因，因为他们都是共享了同一个proxy方法，一旦重名，Object.defineProperty就会监听vm下的同一个键名。导致后者被覆盖。这并不是我们想要的。

1. 为什么methods中的this都默认指向当前vue实例？

因为在new Vue的时候执行了initState，在initState中除了实现了上文提到的data、props等可以通过this.xxx访问到的功能外，还为methods中的每一个方法都通过bind绑定了当前vm实例。



**问题：为什么data()要是返回一个对象？**

1. **在Vue初始化的最后都做了什么？**

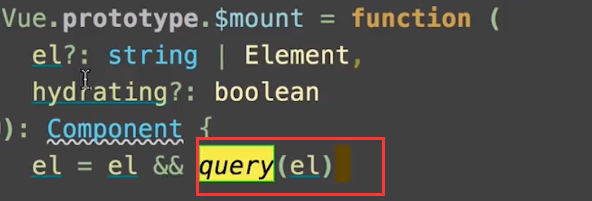
答：Vue初始化就干了这几件事，首先判断是否有new，然后执行\_init方法，\_init方法执行之初先融合配置选项，\_init方法又分成了好多个子方法，分别用来初始化生命周期，事件中心，初始化data、props、computed、methods、watcher等。

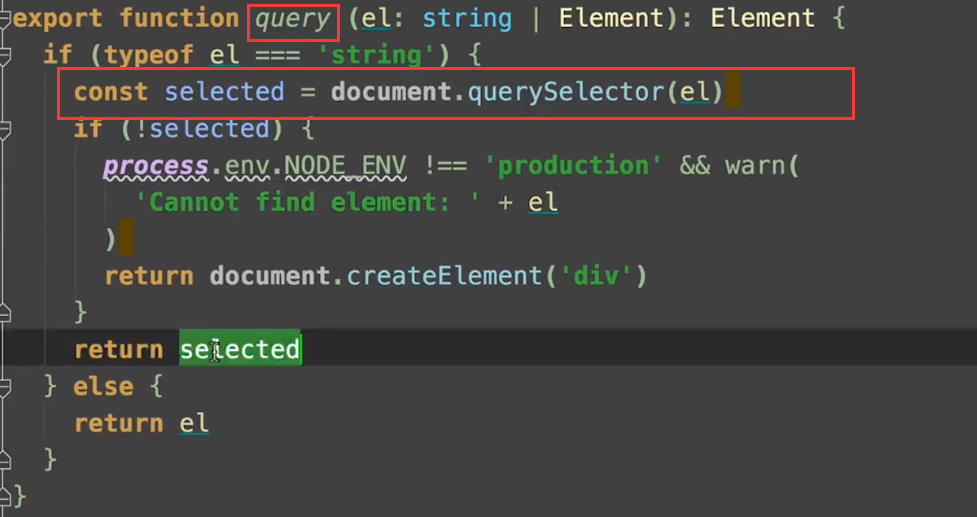
最后一步正是检测有没有vm.$options.el属性，如果有就调用vm.$mount挂载到制定的dom上。

1. **$mount如何挂载到DOM上的？**

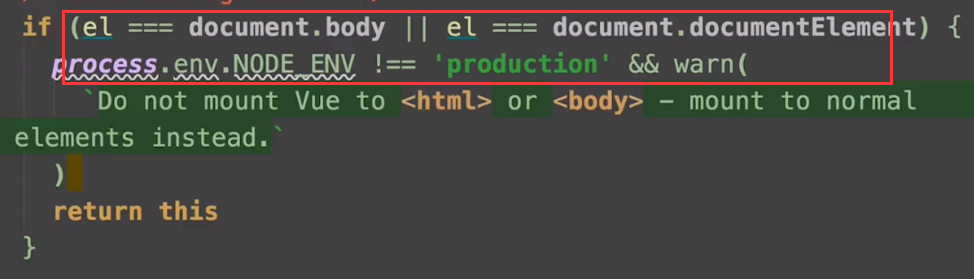
答：$mount函数首先会判断vm实例上是否定义了render方法，如果没有就会把el或者template字符串转换为render方法。

$mount方法支持传入2个参数，第一个是el，它表示挂载的元素，可以是是字符串也可以是dom对象。如果是字符串就转换为dom对象。这一步实际是调用了query方法，query方法的实现很简单，就是调用了document.querySelector根据el后的字符串选择器选择到dom。这就是实现了string -> dom的过程





然后$mount方法对el的目标做了一个判断，不能绑定到body或者html节点上。



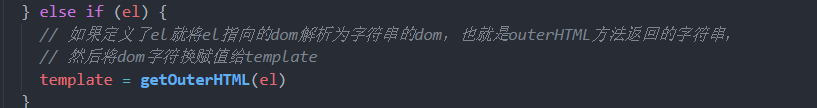
然后又判断了是否设置了render方法。如果没有设置render方法就会继续判断是否设置了template或者el。

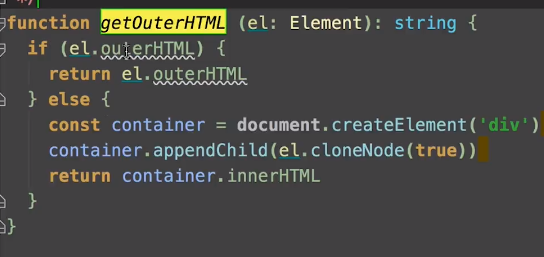
如果设置了template就进行一系列的处理，处理的方式主要使用了idToComplate方法。至于idToComplate方法，我们之后再谈。

如果没有设置template，设置了el，那么就会通过getOuterHtml方法获取el这个dom对象的html结构，他是一个字符串（这之前已经通过query将el所指向的元素转换成了dom然后又赋值给el了），之后会把这个字符串再次赋值给template。



这个getOuterHTML其实就是原生outerHTML的一个polyfill。最终返回的是一个字符串（因为outerhtml和innerhtml都是返回的字符串）。



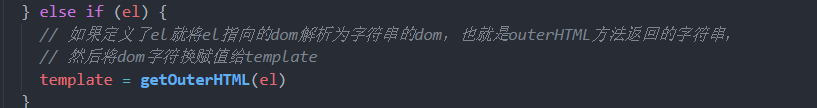


关于outerHTML可以参考博文：

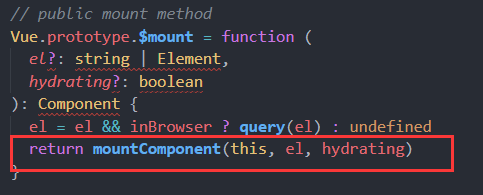
<https://www.cnblogs.com/hanwater/archive/2009/05/12/1454577.html>

在拿到getOuterHTML返回的dom结构字符串后，$mount方法就会对这个dom结构字符串进行编译，主要使用了compileToFunctions

方法编译成一个render函数(template这个dom字符串具体怎么变成render函数的原理之后详解)



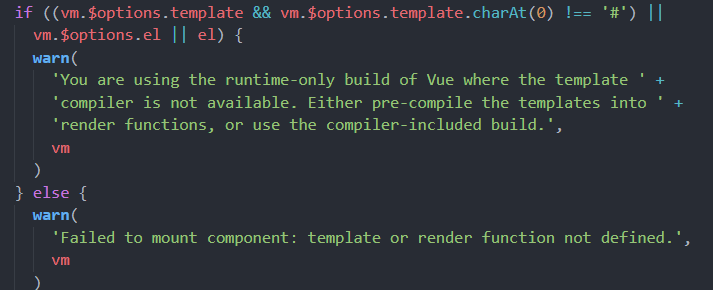
在拿到render函数后，会经过mountComponent这个方法，会通过这个方法，将render函数渲染成真正的dom然后添加观察者(watcher)，之后就会挂载到页面上去(详细后讲)。



1. **mountComponent是如何将render函数转换成dom的 ?**

**答：这个函数非常的严谨，他首先判断了当前这个vm的vm.$options.render是否存在。**

**如果不存在，说明你可能写了el或者template但是并没有经过compiler的编译（没安装compiler版本的Vue）。所以会造成没有vm.$options.render属性，这是就会抛出一些错误的警告。**

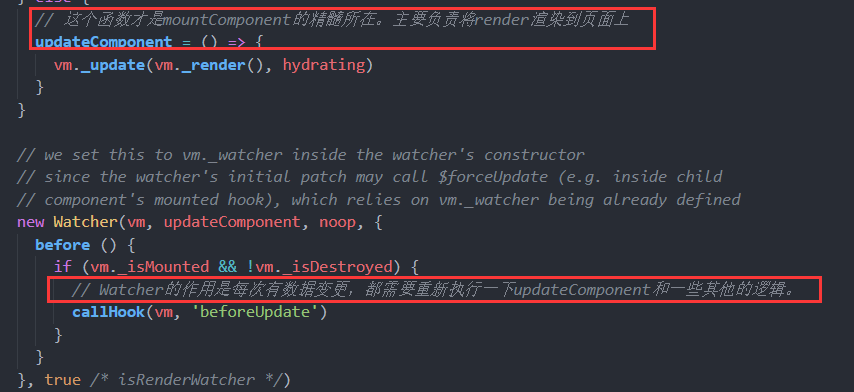


**需要注意的是，在上一步中我们通过dom字符串获得了render，然后让vm.$options.render = render了。**





其实mountComponet非常的关键，他的核心逻辑就是初始化了一个watcher，然后每当组件发生更新的时候都会调用updateComponent方法，在updateComponent方法中，主要调用了vm.\_render去生成虚拟dom。watcher起到两个作用，一个是初始化的时候执行updateComponent方法，一个是在检测数据发生变化的时候执行updateComponet方法。详细的内容我们会在以后讲到。mountComponent中核心的逻辑是这个：



TIP：注意，$mount再多处都有定义，之前的$mount方法是对runtime + compiler模式的一种扩充。

