# 响应式对象

【前言】之前我们介绍的都是Vue是怎么实现数据渲染和组件化的，主要讲的是初始化的过程，把原始的数据最终映射到DOM中，但并没有涉及到数据变化到DOM变化的部分，而Vue的数据驱动除了数据渲染DOM之外，还有一个很重要的体现就是数据的变更会触发DOM的变化。

接下来我们研究的内容将侧重于响应式原理。

### 问题1：Vue的实现响应式原理的最最核心方法是什么？

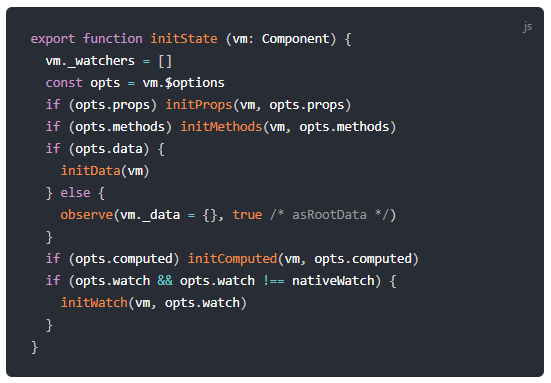
答：要用最简单的话来阐述Vue的响应式原理，那就是Object,defineProperty这个在ES5中定义的方法，这个方法在《JavaScript高级程序设计中》提到，说他是一个定义访问器属性的方法，其中的getter和setter这两种访问器属性就是Vue实现响应式的关键。

回忆一下Object.defineProperty方法，数据来自MDN：

<https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty>

### 问题2：initState是new Vue()初始化的时候所经历的关键步骤，他都做了什么？

initState看上去逻辑很少，但是却做了很多事情，但是大体上可以看出，他就是先拿到了刚刚融合配置后的vm.$options，然后对其中的props、methods、data、compute、watcher醉了初始化操作。



我们先知道初始化的顺序，是props -> methods -> data -> computed -> watcher

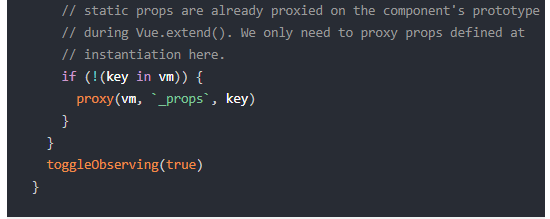
然后我们暂时着重研究props和data的初始化，其他的我们后面再谈：

props最新被初始化的所以，我们先来看看initProps的逻辑，因为initProps的逻辑比较多，但是很简单，initProps做的事情可以总结为一下几步：

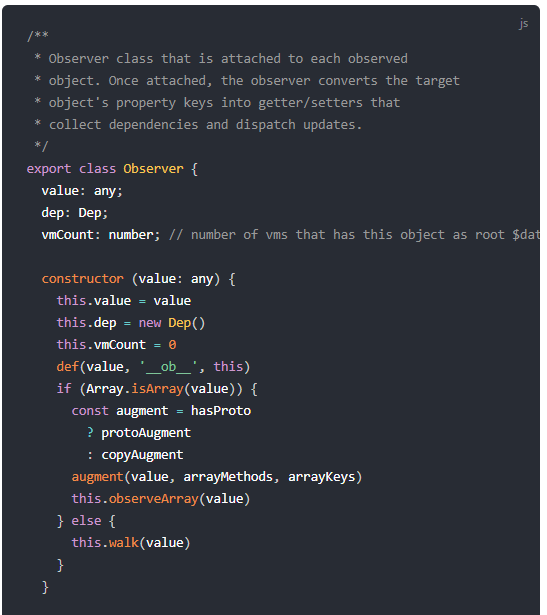
1. 拿到propsOptions，这个propsOPtions就是vm.$option.props，他是作为参数拿过来的。
2. 遍历propsOption，对其中的每一个key对应的值通过definedReactive()变成响应式。
3. 通过proxy方法vm.\_props.xxx的访问代理到vm.xxx上。以实现可以通过vm.xxx去访问vm.\_props.xxx了。（先认为\_props在initProp之前就已经存在值，暂时未找到vm.\_props在什么时候被赋值的额，似乎并不是在合并配置的时候）

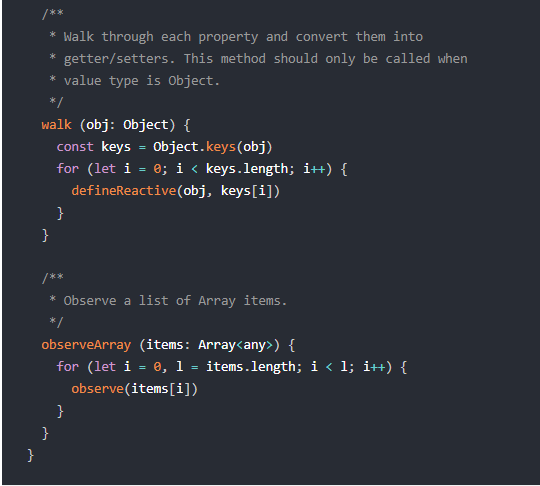
以下为initProps的源代码：





在第（2）步中，他先缓存了propsOption上的key，将其存在数组keys中。然后对propsOptions下的每一个值通过definedReactive方法变成了响应式的，在来看definedReactive之前，我们应该先了解Observer类和observe方法都是什么东西，先来看看Observer类：

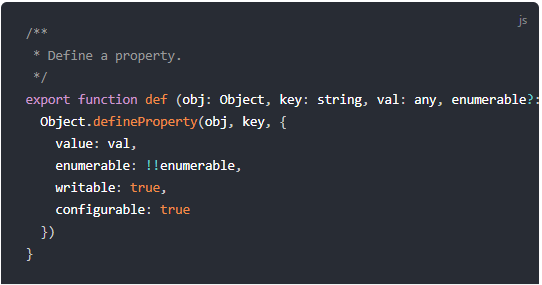




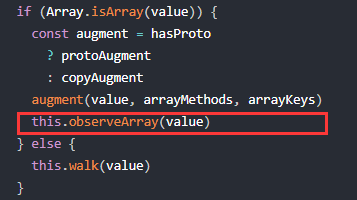
这个Observe方法主要接受三个参数，但是最最主要的也是最常用是第一个参数value，他可以是一个任意类型的值。

Observer的实例的value属性就是参数value，dep属性则是一个实例化的Dep()，然后通过def方法，为这个参数value添加一个\_\_ob\_\_属性，属性的值就是自身，也就是这个根据value值刚刚实例化好的Observe对象。通过这一步，我们就知道问什么我们输出data上的对象类型数据，会多出一个\_\_ob\_\_的属性。

以下就是def的代码，非常的简单：



之后就会根据value值的类型作出判断，对于数组就会调用observeArray方法，否则对春对象调用walk方法。



先不管augment，我们可以看到对于数组类型的value调用了observeArray方法。这个observeArray事实上就是遍历数组，然后对数组的每一项再次调用Observer，这也就是说，如果我们为带入new Observer(value)中的第一个参数是一个数组，那么这个数组中的每一项都会递归的调用observe()方法，**看好是observe()方法，不是我们现讨论的Observer构造函数！**

如果我们带入的是一个对象，那么对于value对象，会对他调用walk方法，这个walk方法事实上就是对value中的每一项都递归调用observe()方法。

**所以最终的结果，如果value是数组，那么Observer的结果就是这个value数组中的每一项（即使那一项还是一个数组）都是被observe方法处理过的。如果带进去的value是一个对象，那么这个对象的每一项都被observe方法处理过的。Observer的实例化全面的覆盖了value中的每一个角落。对于Obsever的实例，他最有价值的地方就是为他的原始数据value添加了\_\_ob\_\_，然后为自身的每个角落都调用了observe方法进行了处理。**

看完了Observer类，我们再来看看observe方法，observe方法的功能就是监测数据的变化：



Observe方法的作用就是非VNode的对象类型数据添加一个Observe（在这个对象上增加一个\_\_ob\_\_，这个\_\_ob\_\_是一个根据这个对象生成的Observer实例），当然如果这个非Vnode对象已经具有了\_\_ob\_\_属性，并且这\_\_ob\_\_属性又是Observer的实例（值得学习的地方，如果是我的话，我可能只会判断是否存在\_\_ob\_\_属性），那么就不必要为当前的这个对象添加Observer，如果没有\_\_ob\_\_，则根据一系列的判断后，调用new Observer去对象添加一个Observer。

**也就是说，对于非VNode类型的对象，经过了new Observer()后，他自身被添加了\_\_ob\_\_，然后他子元素毫无遗留的都被添加了属于自己的\_\_ob\_\_，如果是一个数组， 则这个数组本身会添加一个\_\_ob\_\_，然后他的子元素的每一项都被添加了\_\_ob\_\_。**

再知道什么Observer和observe之后，我们就清晰了何为为一个对象添加一个Observer。

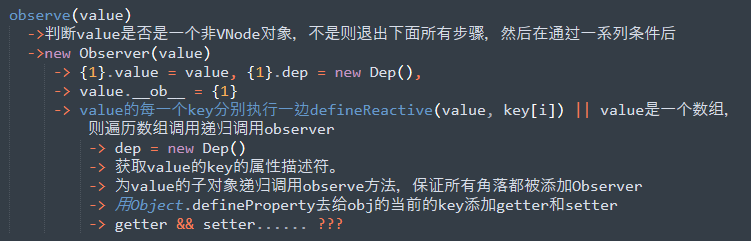
然后我们回到defineReactive方法中：

defindReactive的功能就是定义一个响应式对象，他接受五个参数，主要是obj，key，他的核心作用是给对象动态添加getter和setter，他的行为大致可以分为以下几步：

1. 创建dep实例
2. 获取obj的属性描述符
3. 然后对obj的子对象递归调用observe方法，这样就保证了无论obj是一个多复杂的对象，他的每一个角落都变成了响应式的对象，同时也给这个obj本身添加setter和getter。如果obj不是对象，那就直接添加setter和getter

**让他的每一个角落变成响应式的对象的目的就是，当访问或者修改obj的一个嵌套较深的属性时候，也能触发getter和setter。**

看上去比较复杂，我们对目前探索到的内容抽象出一个逻辑图：

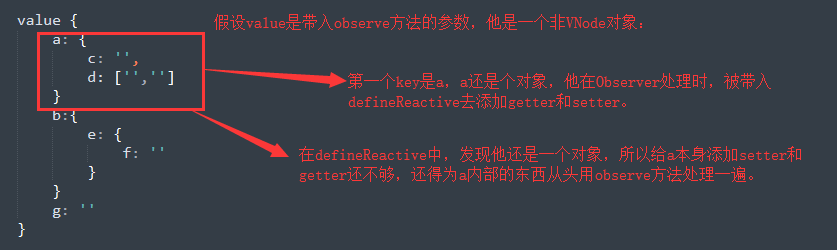


可以总结出来observe的目标就是将一个非VNode对添加一个Observer，同时获取这个Observer实例并作为返回值返回。

而new Observer()则是创建一个Dep实例，然后让这个非VNode对象的\_\_ob\_\_属性指向这个创建好的Dep实例（这个过程就相当于为非VNode对象添加一个Observer），然后让Dep实例的value属性指向这个非VNode对象，最终以这个Dep作为返回值返回给observe方法。在new Observer的过程中还未这个非VNode对象进行遍历，如果他是一个对象那么就对他的每一key都用defineReactive添加setter和getter，这样就做到了让这个非VNode对象是一个响应式的。如果是一个数组就遍历数组，为其中的每一项重新走一遍observe（因为数组可能是对象的组合，我们希望的是让数组中的每一个对象都是响应式的，包括这个数组本身，以此达到修改任何角落都会触发这个数组的getter或者setter)。

至于给这个非VNode对象添加getter和setter的真正方法是defineReactive。

**因此我们可以看到，observe方法的终极目标就是为一个非VNode对象中任何一个角落都添加一个getter和setter。而且无论是从defineReative还是Observer出发都有机会走到observe方法。**



好了，再知道observe、Observer、defineReactive之后，我们言归主线，来看看initProps和initData，这会看一眼就知道initProps和initData都做了什么。不再赘述....