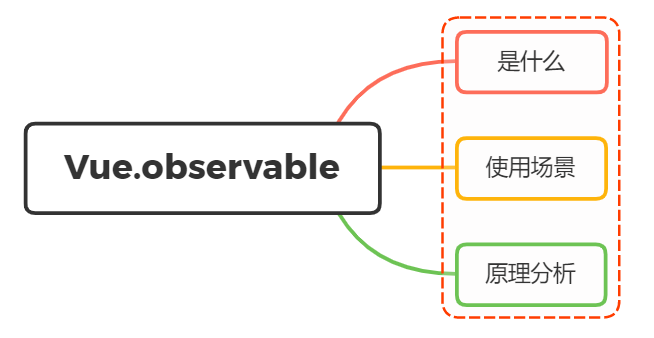
# 面试官：Vue.observable你有了解过吗？说说看



## 一、Observable 是什么

Observable 翻译过来我们可以理解成**可观察的**

我们先来看一下其在Vue中的定义

Vue.observable，让一个对象变成响应式数据。Vue 内部会用它来处理 data 函数返回的对象

返回的对象可以直接用于渲染函数和计算属性内，并且会在发生变更时触发相应的更新。也可以作为最小化的跨组件状态存储器

Vue.observable({ count : 1})

其作用等同于

new vue({ count : 1})

在 Vue 2.x 中，被传入的对象会直接被 Vue.observable 变更，它和被返回的对象是同一个对象

在 Vue 3.x 中，则会返回一个可响应的代理，而对源对象直接进行变更仍然是不可响应的

## 二、使用场景

在非父子组件通信时，可以使用通常的bus或者使用vuex，但是实现的功能不是太复杂，而使用上面两个又有点繁琐。这时，observable就是一个很好的选择

创建一个js文件

// 引入vue  
import Vue from 'vue  
// 创建state对象，使用observable让state对象可响应  
export let state = Vue.observable({  
 name: '张三',  
 'age': 38  
})  
// 创建对应的方法  
export let mutations = {  
 changeName(name) {  
 state.name = name  
 },  
 setAge(age) {  
 state.age = age  
 }  
}

在.vue文件中直接使用即可

<template>  
 <div>  
 姓名：{{ name }}  
 年龄：{{ age }}  
 <button @click="changeName('李四')">改变姓名</button>  
 <button @click="setAge(18)">改变年龄</button>  
 </div>  
</template>  
import { state, mutations } from '@/store  
export default {  
 // 在计算属性中拿到值  
 computed: {  
 name() {  
 return state.name  
 },  
 age() {  
 return state.age  
 }  
 },  
 // 调用mutations里面的方法，更新数据  
 methods: {  
 changeName: mutations.changeName,  
 setAge: mutations.setAge  
 }  
}

## 三、原理分析

源码位置：src.js

export function observe (value: any, asRootData: ?boolean): Observer | void {  
 if (!isObject(value) || value instanceof VNode) {  
 return  
 }  
 let ob: Observer | void  
 // 判断是否存在\_\_ob\_\_响应式属性  
 if (hasOwn(value, '\_\_ob\_\_') && value.\_\_ob\_\_ instanceof Observer) {  
 ob = value.\_\_ob\_\_  
 } else if (  
 shouldObserve &&  
 !isServerRendering() &&  
 (Array.isArray(value) || isPlainObject(value)) &&  
 Object.isExtensible(value) &&  
 !value.\_isVue  
 ) {  
 // 实例化Observer响应式对象  
 ob = new Observer(value)  
 }  
 if (asRootData && ob) {  
 ob.vmCount++  
 }  
 return ob  
}

Observer类

export class Observer {  
 value: any;  
 dep: Dep;  
 vmCount: number; // number of vms that have this object as root $data  
  
 constructor (value: any) {  
 this.value = value  
 this.dep = new Dep()  
 this.vmCount = 0  
 def(value, '\_\_ob\_\_', this)  
 if (Array.isArray(value)) {  
 if (hasProto) {  
 protoAugment(value, arrayMethods)  
 } else {  
 copyAugment(value, arrayMethods, arrayKeys)  
 }  
 this.observeArray(value)  
 } else {  
 // 实例化对象是一个对象，进入walk方法  
 this.walk(value)  
 }  
}

walk函数

walk (obj: Object) {  
 const keys = Object.keys(obj)  
 // 遍历key，通过defineReactive创建响应式对象  
 for (let i = 0; i < keys.length; i++) {  
 defineReactive(obj, keys[i])  
 }  
}

defineReactive方法

export function defineReactive (  
 obj: Object,  
 key: string,  
 val: any,  
 customSetter?: ?Function,  
 shallow?: boolean  
) {  
 const dep = new Dep()  
  
 const property = Object.getOwnPropertyDescriptor(obj, key)  
 if (property && property.configurable === false) {  
 return  
 }  
  
 // cater for pre-defined getter/setters  
 const getter = property && property.get  
 const setter = property && property.set  
 if ((!getter || setter) && arguments.length === 2) {  
 val = obj[key]  
 }  
  
 let childOb = !shallow && observe(val)  
 // 接下来调用Object.defineProperty()给对象定义响应式属性  
 Object.defineProperty(obj, key, {  
 enumerable: true,  
 configurable: true,  
 get: function reactiveGetter () {  
 const value = getter ? getter.call(obj) : val  
 if (Dep.target) {  
 dep.depend()  
 if (childOb) {  
 childOb.dep.depend()  
 if (Array.isArray(value)) {  
 dependArray(value)  
 }  
 }  
 }  
 return value  
 },  
 set: function reactiveSetter (newVal) {  
 const value = getter ? getter.call(obj) : val  
 /\* eslint-disable no-self-compare \*/  
 if (newVal === value || (newVal !== newVal && value !== value)) {  
 return  
 }  
 /\* eslint-enable no-self-compare \*/  
 if (process.env.NODE\_ENV !== 'production' && customSetter) {  
 customSetter()  
 }  
 // #7981: for accessor properties without setter  
 if (getter && !setter) return  
 if (setter) {  
 setter.call(obj, newVal)  
 } else {  
 val = newVal  
 }  
 childOb = !shallow && observe(newVal)  
 // 对观察者watchers进行通知,state就成了全局响应式对象  
 dep.notify()  
 }  
 })  
}

## 参考文献

* https://blog.csdn.net/qq\_32682301/article/details/105419673
* https://wbbyouzi.com/archives/343