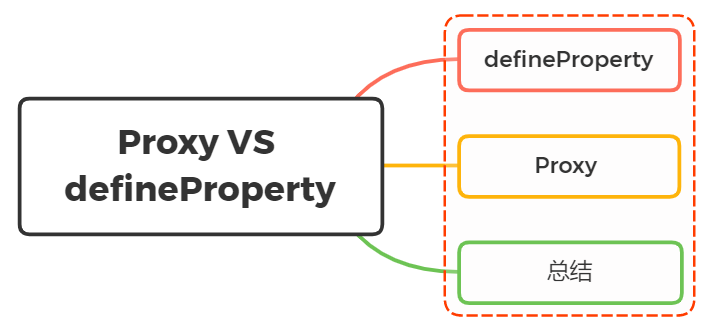
# 面试官：Vue3.0里为什么要用 Proxy API 替代 defineProperty API ？



## 一、Object.defineProperty

定义：Object.defineProperty() 方法会直接在一个对象上定义一个新属性，或者修改一个对象的现有属性，并返回此对象

##### 为什么能实现响应式

通过defineProperty 两个属性，get及set

* get

属性的 getter 函数，当访问该属性时，会调用此函数。执行时不传入任何参数，但是会传入 this 对象（由于继承关系，这里的this并不一定是定义该属性的对象）。该函数的返回值会被用作属性的值

* set

属性的 setter 函数，当属性值被修改时，会调用此函数。该方法接受一个参数（也就是被赋予的新值），会传入赋值时的 this 对象。默认为 undefined

下面通过代码展示：

定义一个响应式函数defineReactive

function update() {  
 app.innerText = obj.foo  
}  
  
function defineReactive(obj, key, val) {  
 Object.defineProperty(obj, key, {  
 get() {  
 console.log(`get ${key}:${val}`);  
 return val  
 },  
 set(newVal) {  
 if (newVal !== val) {  
 val = newVal  
 update()  
 }  
 }  
 })  
}

调用defineReactive，数据发生变化触发update方法，实现数据响应式

const obj = {}  
defineReactive(obj, 'foo', '')  
setTimeout(()=>{  
 obj.foo = new Date().toLocaleTimeString()  
},1000)

在对象存在多个key情况下，需要进行遍历

function observe(obj) {  
 if (typeof obj !== 'object' || obj == null) {  
 return  
 }  
 Object.keys(obj).forEach(key => {  
 defineReactive(obj, key, obj[key])  
 })  
}

如果存在嵌套对象的情况，还需要在defineReactive中进行递归

function defineReactive(obj, key, val) {  
 observe(val)  
 Object.defineProperty(obj, key, {  
 get() {  
 console.log(`get ${key}:${val}`);  
 return val  
 },  
 set(newVal) {  
 if (newVal !== val) {  
 val = newVal  
 update()  
 }  
 }  
 })  
}

当给key赋值为对象的时候，还需要在set属性中进行递归

set(newVal) {  
 if (newVal !== val) {  
 observe(newVal) // 新值是对象的情况  
 notifyUpdate()  
 }  
}

上述例子能够实现对一个对象的基本响应式，但仍然存在诸多问题

现在对一个对象进行删除与添加属性操作，无法劫持到

const obj = {  
 foo: "foo",  
 bar: "bar"  
}  
observe(obj)  
delete obj.foo // no ok  
obj.jar = 'xxx' // no ok

当我们对一个数组进行监听的时候，并不那么好使了

const arrData = [1,2,3,4,5];  
arrData.forEach((val,index)=>{  
 defineProperty(arrData,index,val)  
})  
arrData.push() // no ok  
arrData.pop() // no ok  
arrDate[0] = 99 // ok

可以看到数据的api无法劫持到，从而无法实现数据响应式，

所以在Vue2中，增加了set、delete API，并且对数组api方法进行一个重写

还有一个问题则是，如果存在深层的嵌套对象关系，需要深层的进行监听，造成了性能的极大问题

### 小结

* 检测不到对象属性的添加和删除
* 数组API方法无法监听到
* 需要对每个属性进行遍历监听，如果嵌套对象，需要深层监听，造成性能问题

## 二、proxy

Proxy的监听是针对一个对象的，那么对这个对象的所有操作会进入监听操作，这就完全可以代理所有属性了

在ES6系列中，我们详细讲解过Proxy的使用，就不再述说了

下面通过代码进行展示：

定义一个响应式方法reactive

function reactive(obj) {  
 if (typeof obj !== 'object' && obj != null) {  
 return obj  
 }  
 // Proxy相当于在对象外层加拦截  
 const observed = new Proxy(obj, {  
 get(target, key, receiver) {  
 const res = Reflect.get(target, key, receiver)  
 console.log(`获取${key}:${res}`)  
 return res  
 },  
 set(target, key, value, receiver) {  
 const res = Reflect.set(target, key, value, receiver)  
 console.log(`设置${key}:${value}`)  
 return res  
 },  
 deleteProperty(target, key) {  
 const res = Reflect.deleteProperty(target, key)  
 console.log(`删除${key}:${res}`)  
 return res  
 }  
 })  
 return observed  
}

测试一下简单数据的操作，发现都能劫持

const state = reactive({  
 foo: 'foo'  
})  
// 1.获取  
state.foo // ok  
// 2.设置已存在属性  
state.foo = 'fooooooo' // ok  
// 3.设置不存在属性  
state.dong = 'dong' // ok  
// 4.删除属性  
delete state.dong // ok

再测试嵌套对象情况，这时候发现就不那么 OK 了

const state = reactive({  
 bar: { a: 1 }  
})  
  
// 设置嵌套对象属性  
state.bar.a = 10 // no ok

如果要解决，需要在get之上再进行一层代理

function reactive(obj) {  
 if (typeof obj !== 'object' && obj != null) {  
 return obj  
 }  
 // Proxy相当于在对象外层加拦截  
 const observed = new Proxy(obj, {  
 get(target, key, receiver) {  
 const res = Reflect.get(target, key, receiver)  
 console.log(`获取${key}:${res}`)  
 return isObject(res) ? reactive(res) : res  
 },  
 return observed  
}

## 三、总结

Object.defineProperty只能遍历对象属性进行劫持

function observe(obj) {  
 if (typeof obj !== 'object' || obj == null) {  
 return  
 }  
 Object.keys(obj).forEach(key => {  
 defineReactive(obj, key, obj[key])  
 })  
}

Proxy直接可以劫持整个对象，并返回一个新对象，我们可以只操作新的对象达到响应式目的

function reactive(obj) {  
 if (typeof obj !== 'object' && obj != null) {  
 return obj  
 }  
 // Proxy相当于在对象外层加拦截  
 const observed = new Proxy(obj, {  
 get(target, key, receiver) {  
 const res = Reflect.get(target, key, receiver)  
 console.log(`获取${key}:${res}`)  
 return res  
 },  
 set(target, key, value, receiver) {  
 const res = Reflect.set(target, key, value, receiver)  
 console.log(`设置${key}:${value}`)  
 return res  
 },  
 deleteProperty(target, key) {  
 const res = Reflect.deleteProperty(target, key)  
 console.log(`删除${key}:${res}`)  
 return res  
 }  
 })  
 return observed  
}

Proxy可以直接监听数组的变化（push、shift、splice）

const obj = [1,2,3]  
const proxtObj = reactive(obj)  
obj.psuh(4) // ok

Proxy有多达13种拦截方法,不限于apply、ownKeys、deleteProperty、has等等，这是Object.defineProperty不具备的

正因为defineProperty自身的缺陷，导致Vue2在实现响应式过程需要实现其他的方法辅助（如重写数组方法、增加额外set、delete方法）

// 数组重写  
const originalProto = Array.prototype  
const arrayProto = Object.create(originalProto)  
['push', 'pop', 'shift', 'unshift', 'splice', 'reverse', 'sort'].forEach(method => {  
 arrayProto[method] = function () {  
 originalProto[method].apply(this.arguments)  
 dep.notice()  
 }  
});  
  
// set、delete  
Vue.set(obj,'bar','newbar')  
Vue.delete(obj),'bar')

Proxy 不兼容IE，也没有 polyfill, defineProperty 能支持到IE9

### 参考文献

* https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global\_Objects/Object/defineProperty