vue2的响应式原理

vue2对引用类型进行观察，对于基本类型不观察

vue对象类型的劫持

在初始化数据的方法内部调用响应式方法`observe`

```
import {observe} from './observer/index.js'
function initData(vm){
    let data = vm.$options.data;
    data = vm._data = typeof data === 'function' ? data.call(vm) : data;
    observe(data);
}
```

对于响应式方法`observe`:

1. 首先判断是对象类型的才去监控 `typeof data !== 'object' && data !== null`
2. 提供一个响应式叠类耦合多种响应式的方法 `class Observer {}`,返回这个实例对象

```
export function observe(data) {
    if(typeof data !== 'object' || data == null){
        return;
    }
    return new Observer(data);
}
```

3. 对于Observer类内部：使用 defineReactive 重新定义属性
4. 提供重新定义属性方法walk
    - 1. 获取对象的key，提供defineReactive方法，对对象的每一项一次调用define方法
    - 2. 核心逻辑在defineReactive方法中，使用 OBject.defineProperty的get/set方法，并且这个方法是挂载在Vue的util上面，即可以通过`Vue.util.defineReactive`使用
    - 3. 由于对象是多层的嵌套，对象嵌套对象，那就需要多次监测，用到递归方法，所以在`defineReactive`最上层进行调用`observe`方法，如果是对象就会再次监测不是就会`return`，同时修改data的属性值为一个新对象时(即`vm._data.a = { b: 1 }`)也需要在set方法内部再次进行劫持，再次调用`observe`方法

```
class Observer { // 观测值
    constructor(value){
        this.walk(value);
    }
    walk(data){ // 让对象上的所有属性依次进行观测
        let keys = Object.keys(data);
        for(let i = 0; i < keys.length; i++){
            let key = keys[i];
```

```
            let value = data[key];
            defineReactive(data,key,value);
        }
    }
}
function defineReactive(data,key,value){
    observe(value);
    Object.defineProperty(data,key,{
        get(){
            return value
        },
        set(newValue){
            if(newValue == value) return;
            observe(newValue);
            value = newValue
        }
    })
}
```

> 面试题：在defineReactive的set方法内部，不直接value[key]赋值
>
> 答案：会陷入死循环

数组的劫持

出于性能方面的考虑主要是拦截可以改变数组的方法进行操作，重写数组的方法（即切片编程）

1. 首先将要监测的value值的原型绑定到重写的数组方法的对象上面，通过 `value.__proto__ = arrayMethods;`
2. 拿到数组的原型上的方法`let oldArrayProtoMethods = Array.prototype;`，`arrayMethods`是继承数组的原型上的方法，copy一份， 然后进行重写 `export let arrayMethods = Object.create(oldArrayProtoMethods);` 然后调用数组的方法的时候，如果重写了，走重写的方法的逻辑，如果没有重写则是沿着原型链向上查找
3. 为了修改数组内的对象也可以被监听到，那么对于数组中的对象进行观测提供 `observeArray`方法，该方法主要是对遍历每一个数组的元素执行`observe`方法即可
4. 为了给数组增加一个对象类型的元素的时候也可以被监听到的，所以在重写数组方法的时候就需要做特殊的处理，提供一个变量`inserted`保存要插入的元素，重写的方法`push、unshift、splice`都是可能产生插入元素的行为操作，所以对于这些方法的重写要再次观测，即`inserted`就调用`observeArray`再次观测
5. 但是`observeArray`并不是任何文件内部都可以拿到的，所以需要特殊处理,`observe`类当中对`value`通过`Object.defineProperty`方法添加`__ob__`属性，给所有响应式数据增加标识，并且可以在响应式上获取Observer实例上的方法，同时用来判断一个对象是否被观测过了，这个属性是不能被枚举不能被循环出来的，指向value，此时通过`this.__ob__.observeArray`就可以拿到

> 面试题：为什么不可以直接使用value.**ob** = this?
>
> 答案：因为value是一个被观测得响应式的值，如果这样赋值就会触发Object.defineProperty方法的set方法，此时无限递归会陷入死循环

```
import { arrayMethods } from './array';
class Observer { // 观测值
    constructor(value){
        Object.defineProperty(value,'__ob__',{
            enumerable:false,
            configurable:false,
            value:this
        });
        if(Array.isArray(value)){
            // 通过原型链找到重写的方法
            value.__proto__ = arrayMethods; // 重写数组原型方法
            this.observeArray(value);
        } else {
            this.walk(value);
        }
    }
    observeArray(value){
        for(let i = 0 ; i < value.length ;i ++){
            observe(value[i]);
        }
    }
}
```

```
// 拿到数组的原型上的方法
let oldArrayProtoMethods = Array.prototype;
// 继承数组的原型上的方法，copy一份，然后进行重写
export let arrayMethods = Object.create(oldArrayProtoMethods);

// 然后调用数组的方法的时候，如果重写了，走重写的方法的逻辑，如果没有重写则是沿着原型链向上查找

let methods = [
    'push',
    'pop',
    'shift',
    'unshift',
    'reverse',
    'sort',
    'splice'
];
methods.forEach(method => {
    arrayMethods[method] = function (...args) { // this就是observer里面的value
        const result = oldArrayProtoMethods[method].apply(this, args);
        const ob = this.__ob__;
        let inserted; // 保存要插入的item
        switch (method) {
            case 'push':
            case 'unshift':
                inserted = args;
                break;
```

```
            case 'splice':
                inserted = args.slice(2)
            default:
                break;
        }
        if (inserted) ob.observeArray(inserted); // 对新增的每一项进行观测
        return result
    }
})
```

数据代理

当用户去vm上面取值的时候，将属性的取值代理到vm._data 上，使用户使用起来更方便

```
function proxy(vm,source,key){
    Object.defineProperty(vm,key,{
        get(){
            return vm[source][key];
        },
        set(newValue){
            vm[source][key] = newValue;
        }
    });
}
function initData(vm){
    let data = vm.$options.data;
    data = vm._data = typeof data === 'function' ? data.call(vm) : data;
    for(let key in data){ // 将_data上的属性全部代理给vm实例
        proxy(vm,'_data',key)
    }
    observe(data);
}
```