vue2 响应式

vue实现数据双向绑定主要是：采用数据劫持,结合发布者-订阅者模式的方式，通过 Object.defineProperty() 来劫持各个属性的setter，getter，在数据变动时发布消息给订阅者，触发相应监听回调。当把一个普通 Javascript 对象传给 Vue实例来作为它的 data 选项时，Vue 将遍历它的属性，用 Object.defineProperty() 将它们转为 getter/setter。用户看不到 getter/setter，但是在内部它们让 Vue 追踪依赖，在属性被访问和修改时通知变化。

vue的数据双向绑定 将MVVM作为数据绑定的入口，整合Observer，Compile和Watcher三者，

通过Observer来监听自己的model的数据变化，
通过Compile来解析编译模板指令（vue中是用来解析 {{}}），

最终利用watcher搭起observer和Compile之间的通信桥梁，达到数据变化 ->视图更新；视图交互变化（input）->数据model变更双向绑定效果。

双向数据绑定深入

面试的时候问起vue的原理，大部分的人都会说通过Object.defineProperty修改属性的get，set方法，从而达到数据改变的目的。然而作为vue的MVVM驱动核心，从数据的改变到视图的改变，远远不止这句话就能解释，而是通过Observer，Dep，Watcher，Compile 4个类以及一个CpompileUtil辅助类完成。

Vue 主要通过以下 4 个步骤来实现数据双向绑定的：订阅者 Watcher：Watcher是 Observer 和 Compile 之间通信的桥梁

1.监听器 Observer：对数据对象进行遍历，包括子属性对象的属性，利用 Object.defineProperty() 对属性都加上 setter 和 getter。这样的话，给这个对象的某个值赋值，就会触发 setter，那么就能监听到了数据变化。

2.解析器 Compile：解析 Vue 模板指令，将模板中的变量都替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者，一旦数据有变动，收到通知，调用更新函数进行数据更新。

3.订阅者 Watcher：Watcher 订阅者是 Observer 和 Compile 之间通信的桥梁 ，主要的任务是订阅 Observer 中的属性值变化的消息，当收到属性值变化的消息时，触发解析器 Compile 中对应的更新函数。

4.订阅器 Dep：订阅器采用 发布-订阅 设计模式，用来收集订阅者 Watcher，对监听器 Observer 和 订阅者 Watcher 进行统一管理。

```
//vue中属性变化,首先触发Object.defineProperty中属性的set监听,执行updateComponent方法(异步),通过vm._render()更新vNode(新旧node对比),最后渲染到html中
//observer:观察者
function observer(obj) {
    if (obj && typeof obj === 'object') {
        for (let key in obj) {
            if (!obj.hasOwnProperty(key)) break;
            defineReactive(obj, key, obj[key]);
        }
    }
}
function defineReactive(obj, key, value) {
```

```
    observer(value);
    Object.defineProperty(obj, key, {
        get() {
            return value;
        },
        set(newValue) {
            observer(newValue);
            if (value === newValue) return;
            value = newValue;
        }
    });
}
function $set(data, key, value) {
    defineReactive(data, key, value);
}
```

```
### vue实现数据绑定原理
1、vue使用数据劫持实现数据双向绑定
```text
1. vue.js 采用数据劫持结合发布者-订阅者模式的方式，通过Object.defineProperty()来劫持各个属性的setter，getter
2. 在数据变动时发布消息给订阅者，触发相应的监听回调。
```

2、数据劫持原理

```
1、实现一个数据监听器Observer，能够对数据对象的所有属性进行监听，如有变动可拿到最新值并通知订阅者

2、实现一个指令解析器Compile，对每个元素节点的指令进行扫描和解析，根据指令模板替换数据，以及绑定相应的更新函数

3、实现一个Watcher，作为连接Observer和Compile的桥梁，能够订阅并收到每个属性变动的通知，执行指令绑定的相应回调函数，从而更新视图

4、mvvm入口函数，整合以上三者
```

https://img2018.cnblogs.com/blog/1080958/201905/1080958-20190509151007317-380306546.png

第一步：实现Observer

1. 我们知道可以利用Obeject.defineProperty()来监听属性变动
2. 那么将需要observe的数据对象进行递归遍历，包括子属性对象的属性，都加上 setter和getter
3. 这样的话，给这个对象的某个值赋值，就会触发setter，那么就能监听到了数据变化。

```
//Observer 给对象添加setter、getter属性

var data = {name: 'kindeng'};
observe(data);
data.name = 'dmq'; // 哈哈哈，监听到值变化了 kindeng --> dmq

function observe(data) {
    if (!data || typeof data !== 'object') {
        return;
```
```

```
    }
    // 取出所有属性遍历
    Object.keys(data).forEach(function(key) {
        defineReactive(data, key, data[key]);
    });
};

function defineReactive(data, key, val) {
    observe(val); // 监听子属性
    Object.defineProperty(data, key, {
        enumerable: true, // 可枚举
        configurable: false, // 不能再define
        get: function() {
            return val;
        },
        set: function(newVal) {
            console.log('哈哈哈，监听到值变化了 ', val, ' --> ', newVal);
            val = newVal;
        }
    });
}
```

4. 这样我们已经可以监听每个数据的变化了，那么监听到变化之后就是怎么通知订阅者了，所以接下来我们需要实现一个消息订阅器

5. 很简单维护一个数组，用来收集订阅者，数据变动触发notify，再调用订阅者的update方法

数据变化触发notify调用订阅者的update方法

```
// ... 省略
function defineReactive(data, key, val) {
    var dep = new Dep();
    observe(val); // 监听子属性

    Object.defineProperty(data, key, {
        // ... 省略
        set: function(newVal) {
            if (val === newVal) return;
            console.log('哈哈哈，监听到值变化了 ', val, ' --> ', newVal);
            val = newVal;
            dep.notify(); // 通知所有订阅者
        }
    });
}

function Dep() {
    this.subs = [];
}
Dep.prototype = {
    addSub: function(sub) {
        this.subs.push(sub);
    },
    notify: function() {
        this.subs.forEach(function(sub) {
            sub.update();
        });
    }
```

```
};
```

https://www.cnblogs.com/aeipyuan/p/12726202.html

vue3实现

主要分为两大步，设置响应式对象和依赖收集(发布订阅):

1.设置响应式对象
首先创建Proxy，传入将要监听的对象，然后通过handler设置对象的监听,通过get等函数的形参对数据进行劫持处理，然后创建两个WeakMap实例toProxy,toRow来记录当前对象的代理状态，防止重复代理，在set函数中，通过判断属性的类别（新增属性/修改属性）来减少不必要的操作。

```
/* ----------------响应式对象---------------- */
function reactive(target) {
    /* 创建响应式对象 */
    return createReactiveObject(target);
}
/* 防止重复设置代理(target,observer) */
let toProxy = new WeakMap();
/* 防止重复被代理(observer,target) */
let toRow = new WeakMap();
/* 设置响应监听 */
function createReactiveObject(target) {
    /* 非对象或被代理过则直接返回 */
    if (!isObject(target) || toRow.has(target)) return target;
    /* 已经有代理则直接返回 */
    let proxy = toProxy.get(target);
    if (proxy) {
        return proxy;
    }
    /* 监听 */
    let handler = {
        get(target, key) {
            console.log(`get---key(${key})`);
            let res = Reflect.get(target, key);
            /* 添加追踪 */
            track(target, key);
            /* 如果是对象则继续往下设置响应 */
            return isObject(res) ? reactive(res) : res;
        },/* 获取属性 */
        set(target, key, val, receiver) {
            console.log(`set---key(${key})`);
            /* 判断是否为新增属性 */
            let hasKey = hasOwn(target, key);
            /* 存储旧值用于比对 */
            let oldVal = target[key];
            let res = Reflect.set(target, key, val, receiver);
            if (!hasKey) {
                console.log(`新增属性---key(${key})`);
                /* 调用追踪器,绑定新增属性 */
                track(target, key);
                /* 调用触发器,更改视图 */
                trigger(target, key);
            } else if (val !== oldVal) {
                console.log(`修改属性---key(${key})`);
```

```
                trigger(target, key);
            }
            return res;
        },/* 修改属性 */
        deleteProperty(target, key) {
            console.log(`delete---key(${key})`);
            let res = Reflect.deleteProperty(target, key);
            return res;
        }/* 删除属性 */
    }
    /* 创建代理 */
    let observer = new Proxy(target, handler);
    /* 记录与target的联系 */
    toProxy.set(target, observer);
    toRow.set(observer, target);
    return observer;
}
```

2.收集依赖(发布订阅)

vue3中主要通过嵌套的方式实现，原理如下图：
https://img-blog.csdnimg.cn/20200406192057372.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3dlaXhpbl80MjA2MDg5Ng==,size_16,color_FFFFFF,t_70

每次向effect函数传入一个fun----console.log(person.name)后，会先执行一遍run函数，将effect推入栈中，然后执行fun，在执行fun的过程中,会读取person对象，进而触发get函数

```
/* 事件栈 */
let effectStack = [];
/* ----effect函数---- */
function effect(fun) {
    /* 将fun压入栈 */
    let effect = createReactiveEffect(fun);
    /* 初始化执行一次 */
    effect();//实际上是运行run
}
function createReactiveEffect(fun) {
    /* 创建响应式effect */
    let effect = function () {
        return run(effect, fun);
    }
    return effect;
}
function run(effect, fun) {
    /* 防止报错导致栈内元素无法弹出 */
    try {
        effectStack.push(effect);
        fun();
    } finally {
        effectStack.pop();
    }
}
```

get函数调用追踪器track并传入(person,name)，在track中先获取栈顶元素，也就是刚刚触发的fun，假设当前targetsMap是空的，那么此时将会创建一个新的映射target->new Map(),此时depsMap必然也要创建一个新的映射，把key映射到new Set(),然后向key对应的deps中放入effect，此时，name和fun函数之间的绑定已经实现，执行完后effectStack将会把fun函数弹出，防止越堆越多。

```
/* 目标Map */
let targetsMap = new WeakMap();
/* ----追踪器---- */
function track(target, key) {
    /* 获取触发track的事件 */
    let effect = effectStack[effectStack.length - 1];
    if (effect) {
        /* 获取以target作为标识的depsMap */
        let depsMap = targetsMap.get(target);
        if (!depsMap) {
            /* 如果不存在就创建一个新Map */
            targetsMap.set(target, depsMap = new Map());
        }
        /* 获取以key为标识的deps */
        let deps = depsMap.get(key);
        if (!deps) {
            depsMap.set(key, deps = new Set());
        }
        /* 向deps中加入事件 */
        if (!deps.has(effect)) {
            deps.add(effect);
        }
    }
}
```

接下来是触发的过程，当每次进行类似person.name='lisi'这样的改值操作时，就会触发响应的set函数，set函数对比属性的新旧值后调用trigger函数将(person,name)传入，trigger根据两个传入值结合targetsMap->depsMap->deps的顺序找到name对应的事件数组，然后执行所有事件达到响应更新的目的，至此，简化版的vue3响应机制就实现了。

```
/* ----触发器---- */
function trigger(target, key) {
    /* 获取depsMap */
    let depsMap = targetsMap.get(target);
    if (depsMap) {
        /* 获取deps */
        let deps = depsMap.get(key);
        /* 执行deps数组中所有的事件 */
        deps.forEach(effect => {
            effect();
        });
    }
}
```