简单的封装看 数据响应式原理
为什么需要依赖追踪, 因为通过追踪可以知道哪个组件需要更新, 不像react那样, 需要再componentShouldUpdate判断是否需要更新, 引入immutable这些库
因为vue数据监控写的比较复杂, 这里从一个精简看下数据追踪流程

实际上就是利用发布订阅者系统

vue中还有一个地方使用这个系统, 事件的emit on 可以看看代码

包含存放订阅者, 添加订阅者的方法
通知所有Watcher对象更新视图

```
function Dep(){
    this.subs = [];

    this.addSub = function (watcher) {
        this.subs.push(watcher);
    }

    this.notify = function(){
        this.subs.forEach(function(watcher){
            watcher.update();
        });
    }
}
```

生成订阅者

vue中的watcher就是各种组件, computed 或者 watcher 方法

update方法对应一些页面的更新操作

```
function Watcher(fn){
    this.update = function(){
        Dep.target = this;
        this.callback();
        Dep.target = null;
    }
    this.anyback = fn;
    this.update();
}
```

```
var obj = {
    a: 1,
    b: {
        b1: 33
    },
    c: 3
}

Object.keys(obj).forEach(function(key){
    new Observer(obj, key, obj[key])
});

new Watcher(function(){
    document.querySelector("#app").innerHTML = obj.a + obj.b.b1;
})

obj.a = 100; //进入ojb属性a的set方法
```

1. 遍历属性, 执行 new Observer, Observer执行了4次,
会生成4的依赖 dep(a), dep(b), dep(b.b1), deb(c)
由于闭包的关系, 这4个dep一直保存在内存中

2.执行 new Watcher, 会执行一次callback, 也就是执行了 obj.a + obj.b.b1;
会触发三次get方法 => 依次 dep(a).add(w1), dep(b).add(w1), dep(b.b1).add(w1)

这个过程也是寻找依赖的过程, 在vue中有两个方法会触发表达式收集依赖,

第一是 $watcher方法 处理 watch( ) 里面的监控的属性
如 watch(a : fucntion(oldv, newv){ }),
调用vm.$watch 的属性监控也会进入$wathcer方法

第二是 _render()方法 会创建节点 从而执行表达式 => 触发get方法

vnode = render.call(vm._renderProxy, vm.$createElement)

3. obj.a = 100; 会触发 set方法 => dep(a).notify() =>w1.update() =>w1.callback() => app的innerHTML得到更新

vue 中改变数据也一样, 触发set

这里的一个关键点是巧妙的利用了闭包的特性, 稍微解释一下:

key 为a 的时候, new出来的dep (取个别名 dep(a))

dep(a) 这个活动变量 分别被 get 和 set函数引用;

而get和 set又被 Object这个全局对象引用, 所以形成了闭包,

defineReactive 执行完后, dep(a)
包括 obj, key, value 这些活动变量都没有释放...

闭包比较特殊, get和set中的作用域已经确定, 不像普通函数执行的时候才确定,

所以get和set保存的都是父级函数defineReactive 的变量dep(a)

这就使得, 访问this.a, 设置值 this.a=1 的时候保证触发的这两个函数

的dep是同一个 dep, 即dep(a); dep(a) 存放依赖 a属性的 订阅者.

如果不用闭包也可以实现, 外面定义一个deps ={ },

key 为 a的时候, 在get 函数 dep = deps.data.a = new Dep()

在set函数 deps.data.a.notify()

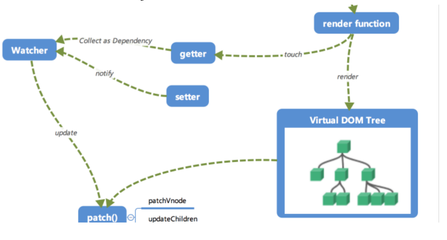
```
function Observer(obj, key, value){

    var dep = new Dep();
    if (Object.prototype.toString.call(value) == '[object Object]') {
        Object.keys(value).forEach(function(key){
            new Observer(value,key,value[key])
        })
    };

    Object.defineProperty(obj, key, {
        enumerable: true,
        configurable: true,
        get: function(){
            if (Dep.target) {
                dep.addSub(Dep.target);
            };
            return value;
        },
        set: function(newVal){
            value = newVal;
            dep.notify();
        }
    })
}
```

数据监控系统

在obj取值和设置值的时候进行拦截



vue中监控数组

调用 Object.create 方法, 实际上是复制了一份 arrayProto

避免去监听不必要的属性, 比如 concat join slice这些操作后不会引起数组本身变化的函数

通过 arrayMethods.__proto__.push 可以访问到 push方法
这样就 Object.definedProperty(arrayMethods, ...) 重新给这些方法赋值
然后就能监控这些方法的调用

```
var arrayProto = Array.prototype;
var arrayMethods = Object.create(arrayProto);

var methodsToPatch = ['push','pop','shift','unshift','splice','sort','reverse'];


methodsToPatch.forEach(function (method) {
  // cache original method
  var original = arrayProto[method];
  def(arrayMethods, method, function mutator () {
    var args = [], len = arguments.length;
    while ( len-- ) args[ len ] = arguments[ len ];

    var result = original.apply(this, args);
    var ob = this.__ob__;
    var inserted;
    switch (method) {
      case 'push':
      case 'unshift':
        inserted = args;
        break
      case 'splice':
        inserted = args.slice(2);
        break
    }
    if (inserted) { ob.observeArray(inserted); }
    // notify change
    ob.dep.notify();
    return result
  });
})
```

def调用的是右边这个方法, 具体是下面这个列子
Object.defineProperty(arrayM, push, {
value: mutator
});
只要调用arrM的 push方法, 就会触发 mutator, 不一定是要写get方法

先执行原生的方法 获取最终结果, 这里的this 就是指向 arrayMethods

因为 siplce (0,2,a,b) 这里是要取得新插入的数组值 a b
有新增的值, 继续监听
然后通知watchers

```
/**
 * Define a property.
 */
function def (obj, key, val, enumerable) {
  Object.defineProperty(obj, key, {
    value: val,
    enumerable: !!enumerable,
    writable: true,
    configurable: true
  });
}
```

this.arr.push(3)

会执行这个方法

```
/**
 * Augment an target Object or Array by intercepting
 * the prototype chain using __proto__
 利用 __proto__ 改写原有的原型链
     相当于
 arrA.__proto__ = arrayMethods

 这样的话 , 用户使用 arrA.push('haha');
 就会沿着 __proto__找到 arrayMethods 属性 __proto__ 的push方法, 因为这个方法被检测了, 所以会触发
 arrayMethods 的 push属性的 Mutator函数 , 最终触发 notify(), 通知wahter更新页面
 */
function protoAugment(target, src: Object) {
    /* eslint-disable no-proto */
    target.__proto__ = src
    /* eslint-enable no-proto */
}
```