

简单的封装看 数据响应式原理

定义发布订阅者系统
包含存放订阅者, 添加订阅者的方法和通知订阅者执行更新方法

```
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
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.anyJack = 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obj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), dep(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 : fuction(oldv, newv){ } }, 调用vm.\$watch 或者 computed的属性监控也会进入\$watcher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
3. obj.a = 100; 会触发 set方法 => dep(a).notify() => w1.update() => w1.callback() => app.innerHTML得到更新
- vue 中改变数据也一样, 触发set

```
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

```
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
  });
});
```

调用 Object.create 方法, 传入 参数 o 结果变成 arrayMethods.__proto__ = o arrayMethods也是一个对象, 通过 arrayMethods.__proto__.push 可以访问到 push方法 这样就 Object.defineProperty(arrayMethods, ...) 重新给这些方法赋值 然后就能监控这些方法的调用

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, 不一定要写get方法

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

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

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wanter更新页面
*/
function protoAugment(target, src: Object) {
  /* eslint-disable no-proto */
  target.__proto__ = src
  /* eslint-enable no-proto */
}
```