重复造轮子：从0开始实现Vue数据绑定

2017-08-02 | 2210

随着前端模块化、工程化的进行，Vue, React, Angular等框架越来越流行，MVC（MVVM）的设计模式也越来深入人心。这类框架将开发者从繁琐的dom操作中解放出来，推动了开发者去了解和使用抽象程度更高的领域。包括但不限于数据结构，设计模式，数据流，抽象数据类型，抽象过程等。

那么这类框架是如何实现数据驱动的呢？ 以Vue为例。

1 Object.defineProperty()

Vue使用了ES5的`Object.defineProperty()`实现数据双向绑定

> 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，并返回这个对象。

语法：

```
/**
 *   obj: 要在其上定义属性的对象。
 *   prop: 要定义或修改的属性的名称。
 *   descriptor: 将被定义或修改的属性的描述符。
*/

Object.defineProperty(obj, prop, descriptor)
```

数据描述符(descriptor)和存取描述符均具有以下可选键值：

- `configurable`
  当且仅当该属性的 configurable 为 true 时，该属性描述符才能够被改变，同时该属性也能从对应的

对象上被删除。默认为 false。

- `enumerable`
  当且仅当该属性的 enumerable 为 true 时，该属性才能够出现在对象的枚举属性中。默认为 false。
  数据描述符同时具有以下可选键值：

- `value`
  该属性对应的值。可以是任何有效的 JavaScript 值（数值，对象，函数等）。默认为 undefined。

- `writable`
  当且仅当该属性的 writable 为 true 时，该属性才能被赋值运算符改变。默认为 false。
  存取描述符同时具有以下可选键值：

- `get`
  一个给属性提供 getter 的方法，如果没有 getter 则为 undefined。该方法返回值被用作属性值。默认为 undefined。

- `set`
  一个给属性提供 setter 的方法，如果没有 setter 则为 undefined。该方法将接受唯一参数，并将该参数的新值分配给该属性。默认为 undefined。

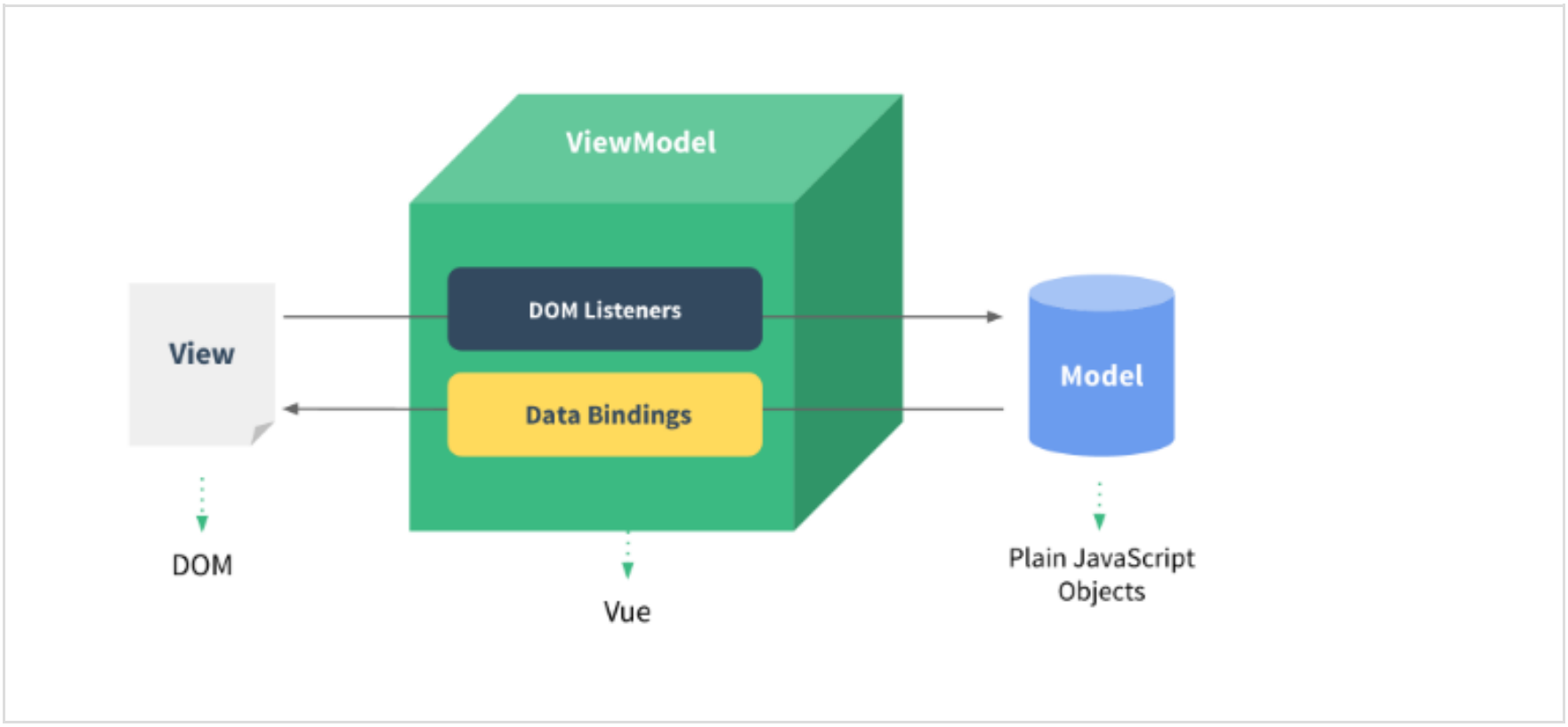
```
function Archiver() {
  var temperature = null;
  var archive = [];

  Object.defineProperty(this, 'temperature', {
    get: function() {
      console.log('get!');
      return temperature;
    },
    set: function(value) {
      console.log('set:', value);
      temperature = value;
      archive.push({ val: temperature });
    }
  });

  this.getArchive = function() { return archive; };
}
```

```
19
20 var arc = new Archiver();
21 arc.temperature; // 'get!'
22 arc.temperature = 11; // 'set:11'
23 arc.temperature = 13; // 'set:13'
24 arc.getArchive(); // [{ val: 11 }, { val: 13 }]
```

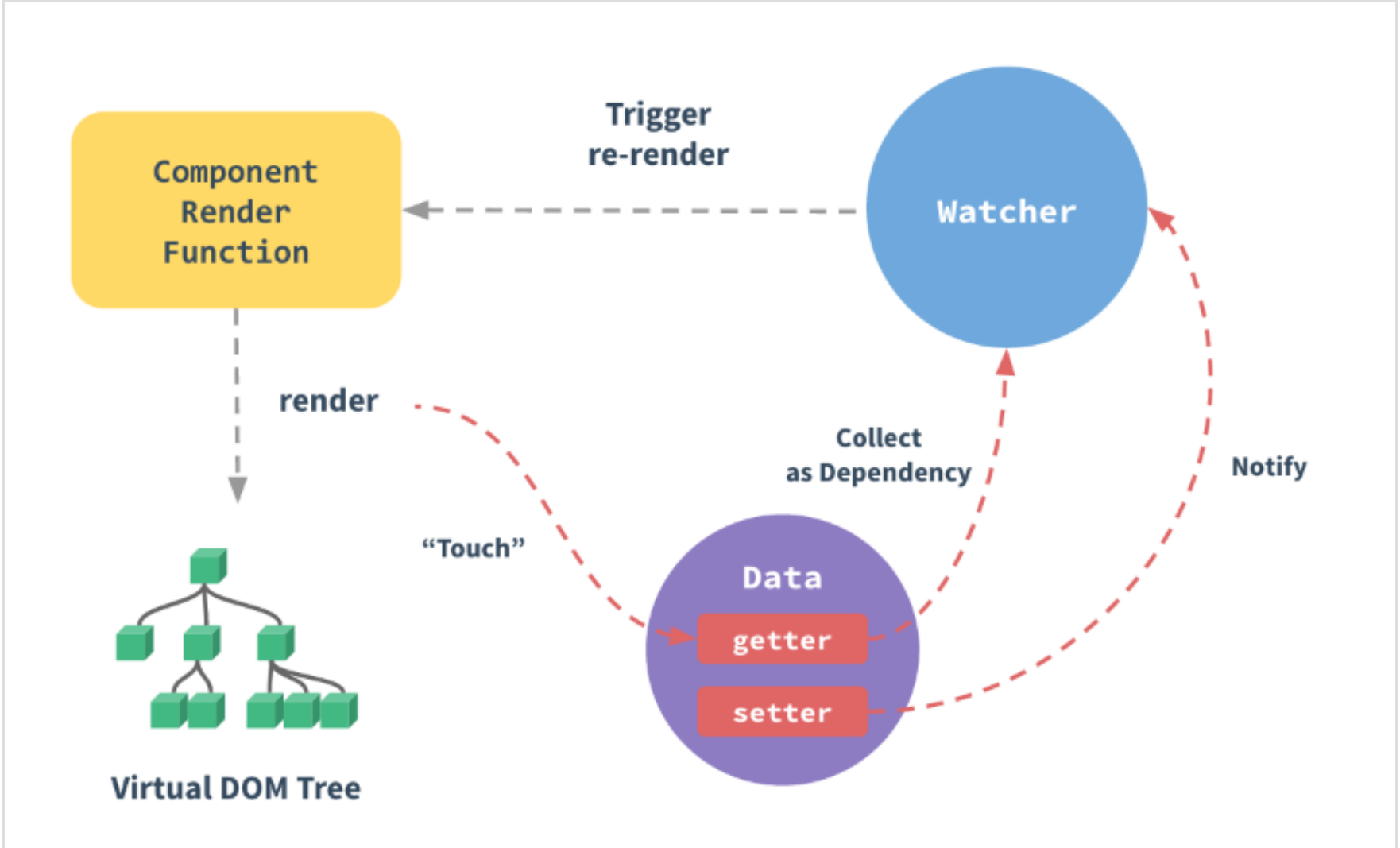
我们通过 `Object.defineProperty` 可以监听到对数据的访问以及修改，从而执行相应的方法。

2 数据绑定

熟悉MVC（MVVM）框架的同学都知道，数据驱动是这类框架最大的特点。在vuejs中，所谓的数据驱动就是当数据发生变化的时候，用户界面发生相应的变化，开发者不需要手动的去修改dom。



vuejs是通过在实现一个观察者来实现的数据驱动。



首先，vuejs在实例化的过程中，会对遍历传给实例化对象选项中的data 选项，遍历其所有属性并使用 Object.defineProperty 把这些属性全部转为 getter/setter。

同时每一个实例对象都有一个watcher实例对象，他会在模板编译的过程中,用getter去访问data的属性，watcher此时就会把用到的data属性记为依赖，这样就建立了视图与数据之间的联系。当之后我们渲染视图的数据依赖发生改变（即数据的setter被调用）的时候，watcher会对比前后两个的数值是否发生变化，然后确定是否通知视图进行重新渲染。

这样就实现了所谓的数据对于视图的驱动。

接下来我们一步步实现一个简版的Vue.js

2.1 数据驱动

首先我们需要一个Vue类，接收一个参数，声明式的将数据渲染为 DOM。

```
// vue.js
import Observer, {observe} from './Observer' // 监听数据变化的方法（后面实现）
import Watcher from './Watcher' // 观察者实例 （后面实现）
// vue 实例，接收一个 option(Object) 参数
export default class Vue{
```

```
    constructor(options = {}){
        // 简化了$options的处理
        this.$options = options
        // 简化了对data的处理
        let data = this._data = this.$options.data
        // 遍历data，将所有data最外层属性代理到Vue实例上
        // this.key 就能访问到 data 对象中的数据
        Object.keys(data).forEach(key => this._proxy(key))
        // 监听数据
        observe(data)
        // 渲染DOM
        this._randerDom()
    }
    _randerDom (val) {
      // TODO 渲染dom
      console.log('更新了dom', this._data)
    }
    // 对外暴露调用订阅者的接口，内部主要在指令中使用订阅者
    $watch(expOrFn, cb){
        // 当监听的value发生变化时， 促发 cb() 方法
        new Watcher(this, expOrFn, cb)
    }
    _proxy(key){
        // 把这data属性全部转为 getter/setter。
        Object.defineProperty(this, key, {
            configurable: true,
            enumerable: true,
            get: () => this._data[key],
            set: (val) => {
                this._data[key] = val
            }
        })
    }
}
```

2.2 监听数据变化

我们需要一个Observer类，在调用 `observe` 方法的时候会实例化一个Observer，将所有的data属性添加set&get方法

```
// Observer.js
export default class Observer{
    constructor(value){
        this.value = value
        this.walk(value)
    }
```

```
    walk(value){
        // 遍历传入的data，将所有data的属性添加set&get
        Object.keys(value).forEach(key => this.convert(key, value[key]))
    }
    convert(key, val){
        // 添加set&get方法
        defineReactive(this.value, key, val)
    }
}
export function observe(value){
    // 当值不存在，或者不是复杂数据类型时，不再需要继续深入监听
    if(!value || typeof value !== 'object'){
        return
    }
    return new Observer(value)
}
```

给data属性添加set&get方法的实现

```
// Observer.js
import Dep from 'Dep'
// Dep用于订阅者的存储和收集，将在后面实现

export function defineReactive(obj, key, val){
    var dep = new Dep()
    // 给传入的data内部对象递归的调用observe，来实现深度监听
    // Vue.js 里需要显示的声明 deep 属性为true
    var chlidOb = observe(val)

    Object.defineProperty(obj, key, {
        enumerable: true, // 可枚举
        configurable: true, // 可修改
        get: ()=> {
            console.log('get value')
            // Watcher实例在实例化过程中，会为Dep添加一个target属性，在读取data中的某个
            // 如果Dep类存在target属性，将订阅者添加到dep实例的subs数组中
            // 此处的问题是：并不是每次Dep.target有值时都需要添加到订阅者管理员中去管理，
            if(Dep.target){
                dep.addSub(Dep.target)
            }
            return val
        },
        set: (newVal) => {
            console.log('new value seted')
            if(val === newVal) return
            val = newVal
            // 对新值进行监听
```

```
            chlidOb = observe(newVal)
            // 通知所有订阅者，数值被改变了
            dep.notify()
        }
    })
}
```

2.3 管理订阅者

对订阅者进行收集、存储和通知

```
// Dep.js
export default class Dep{
    constructor(){
        this.subs = [] // 订阅者队列
    }
    addSub(sub){
        this.subs.push(sub) // 添加订阅者
    }
    notify(){
        // 通知所有的订阅者(Watcher)，触发订阅者的相应逻辑处理
        this.subs.forEach((sub) => sub.update())
    }
}
```

2.4 订阅者

此时已经完成了对数据的监听，我们需要订阅者来接收更新事件，执行数据更之后的逻辑。

- 每个订阅者都是对某条数据的订阅
- 订阅者维护着每一次更新之前的数据，将其和更新之后的数据进行对比，如果发生了变化，则执行相应的业务逻辑，并更新订阅者中维护的数据的值

```
// Watcher.js
import Dep from './Dep'
export default class Watcher{
    constructor(vm, expOrFn, cb){
        this.vm = vm // 被订阅的数据一定来自于当前Vue实例
        this.cb = cb // 当数据更新时需要执行的回调函数
        this.expOrFn = expOrFn // 被监听的数据（表达式或函数）
        this.val = this.get() // 维护更新之前的数据
    }
    // 对外暴露的接口，用于在订阅的数据被更新时，由订阅者管理员(Dep)调用
```

```
    update(){
        this.vm._randerDom() // 检测的数据变动后，更新dom （后面实现）
        this.run()
    }
    run(){
        const val = this.get()
        if(val !== this.val){
            this.val = val;
            this.cb.call(this.vm)
        }
    }
    get(){
        // 当前订阅者(Watcher)读取被订阅数据的最新更新后的值时，通知订阅者管理员收集当前订
        Dep.target = this
        const val = this.vm._data[this.expOrFn]
        // 置空，用于下一个Watcher使用
        Dep.target = null
        return val;
    }
}
```

2.5 Have a Try

首先实例化Vue并赋值给变量`dome`，获取data数据时，会触发get方法，打印出`get value`,修改data时，会触发set方法，打印出`new value seted`，当set的值与旧的值不同时，通知订阅者执行相应的事件。

```
import Vue from './Vue';
let demo = new Vue({
    data: {
        'a': {
            'ab': {
                'c': 'C'
            }
        },
        'b': {
            'bb': 'BB'
        },
        'c': 'C'
    }
});
// 监听c的变化
demo.$watch('c', () => console.log('c is changed'))
// get value
demo.c = 'CCC'
// 更新dom
```

```
// new value seted
// get value
// c is changed
demo.c = 'DDD'
// 更新dom
// new value seted
// get value
// c is changed
demo.a
// get value
demo.a.ab = {
    'd': 'D'
}
// 更新dom
// get value
// get value
// new value seted
console.log(demo.a.ab)
// get value
// get value
// {get d: (), set d: ()}
demo.a.ab.d = 'DD'
// 更新dom
// get value
// get value
// new value seted
console.log(demo.a.ab);
// get value
// get value
// {get d: (), set d: ()}
```

3 模版渲染

Vuejs模版的解析实现较复杂，暂不在这里赘述， 暂时使用ES6的模版字符串代替，便于理解。

3.1 渲染dom

将模版解析后挂载到dom元素（el）上

```
// vue.js
export default class Vue{
    constructor(options = {}){
        // 重复已省略...

        // 获取dom节点
```

```
        this.$el = document.querySelector(options.el)
    }
    _randerDom () {
      // 解析字符串模版
      if (this.$el && this.$options && this.$options.template) {
        this.$el.innerHTML = this.$options.template(this._data)
      }
    }
    // 重复已省略...
}
```

3.2 运行

```
// index.js
import Vue from './Vue';
let demo = new Vue({
    el: '#app',
    // 这里简化了模版的处理
    template (data) {
        return `
        <h1>${data.title}</h1>
        <h2>作者: <strong>${data.author.name}</strong></h2>
        <p>${data.info}</p>
        <p>${data.date}</p>`
    },
    data: {
        'title': 'Hello Vue',
        'info': ` 重复造轮子: 从0开始实现Vue数据绑定`,
        'author': {
            name: 'Shellming'
        },
        'date': new Date()
    }
});
setInterval(() => {
    demo.date = new Date()
}, 1000)
```

结果:

javascript # vue

‹ Vue2.4组件间通信新姿势 JSONP原理及简单实现 ›