

Vue.2x 源码分析之响应式原理

篇幅略长，包含大量图片，阅读时常约20-30分钟。建议在wifi环境下阅读。

原本文章的标题叫《源码解析》，不过后来一想还是以学习的态度写合适一点。在没有彻底掌握源码中每一个代码之前，说“解析”有点标题党了。

我们都知道Vue是一个非常典型的MVVM框架，它的核心功能：

1. 双向数据绑定系统
2. 组件化开发系统

本文我们就聊聊双向数据绑定，不管你是学过或者没学过，我相信看完本文你都会对vue有一个比较简单明确的了解。不过如果哪块有错误，还望指出。

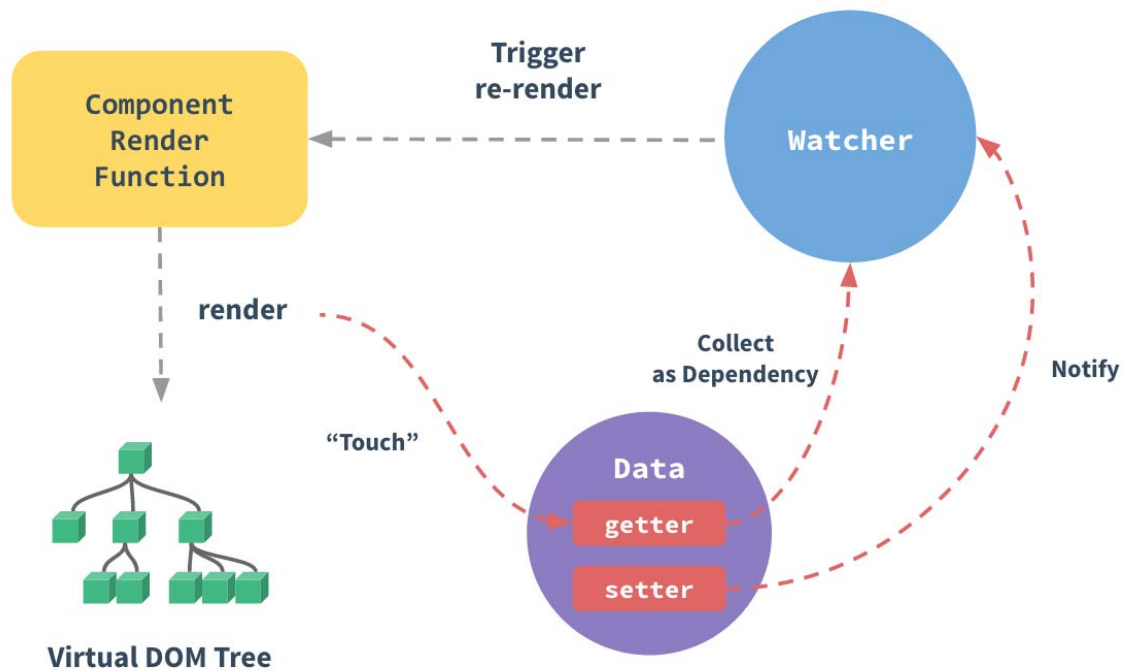
很多朋友说自己读不懂，索性就“不敢”去读。要我说凡事均要去尝试，不尝试永远没有读懂的机会，如果你试着读了，并且坚持了，那你就真的能读懂。

读源码也是有技巧的，我的技巧就是：** 抓住主线，从宏观到微观**。

我们不能一开始就要求自己读懂所有的细节，基本不现实；最好是能找到一条主线，先把大体流程结构摸清楚，再深入到细节，逐项击破，形成对源码整体的认识。

比如，我们都知道Vue中更新数据后会采用virtual DOM（虚拟dom）的方式更新dom。这个时候，如果你不了解virtual DOM，那么听我一句“且不要去研究内部具体实现，因为这会使你丧失主线”，而你仅仅需要知道virtual DOM分为三个步骤：

1. createElement(): 用 JavaScript对象(虚拟树) 描述 真实DOM对象(真实树)
2. diff(oldNode, newNode): 对比新旧两个虚拟树的区别，收集差异



上图对于学习过Vue的朋友来说应该不陌生吧，来自Vue 官网 [深入响应式原理](#)，建议先看图一分钟。

为了说明原理，我们会把虚拟dom这块用fragment来代替（这个是1.x版本的实现）。并且只考虑数据为对象的情况。记住今天的主线：搞清楚响应式原理，实现一个简单的MVVM框架。

由一个例子开始：

template：

```
<div id="app">
  <input type="text" v-model="dsx">
  <input type="text" v-model="child.dsx">
  <p>{{getWeChatblog}}</p>
  <button v-on:click="clickBtn">改变child</button>
```

```

        return this.dsx+ this.child.dsx;
    },
    methods: {
        clickBtn: function(e) {
            this.child.dsx='My World !   Dsx'
        }
    }
});

```

我们要解决的问题有：

如何将data中的数据渲染到真实的宿主环境中？

template是如何被编译成真实环境中可用的HTML的？

如何通过“响应式”修改数据？

计算属性getWeChatblog如何和data中的数据绑定的？

带着这些问题开始我们Vues的开发。尽量和Vue代码保持一致。下面是目录结构：

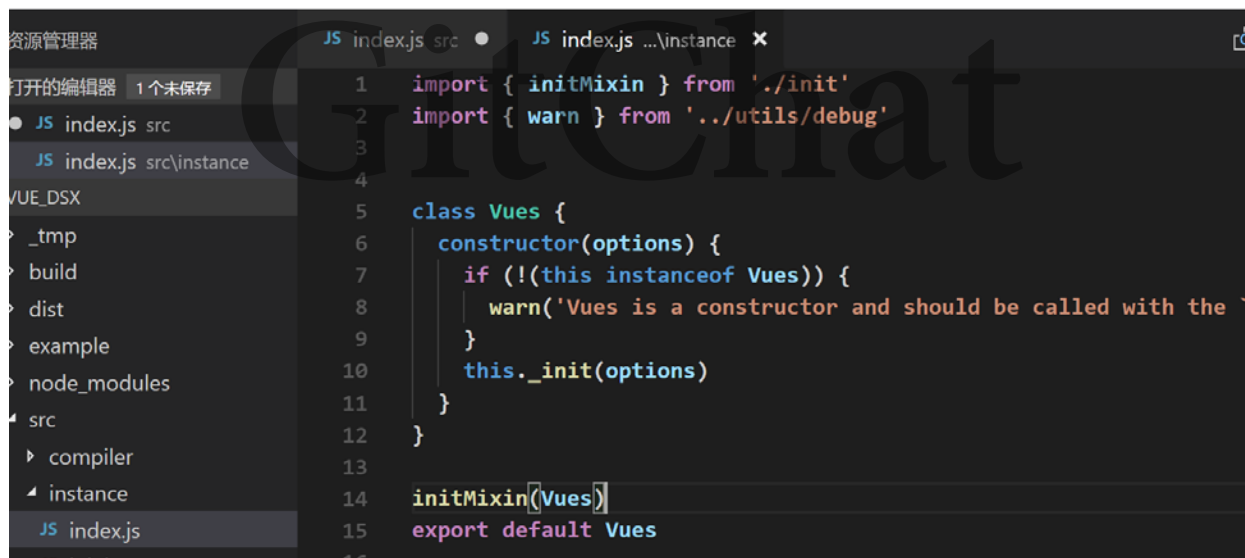


```
(H)
JS index.js  ●
1  import Vues from './instance/index'
2  /**
3   * 静态属性es6没有，需要es7
4   * 因此es6手动绑定
5   */
6  Vues.version = '0.0.1';
7
8  export default Vues;
```

export构造函数Vues，不清楚ES6中module可以可以[点这里](#)

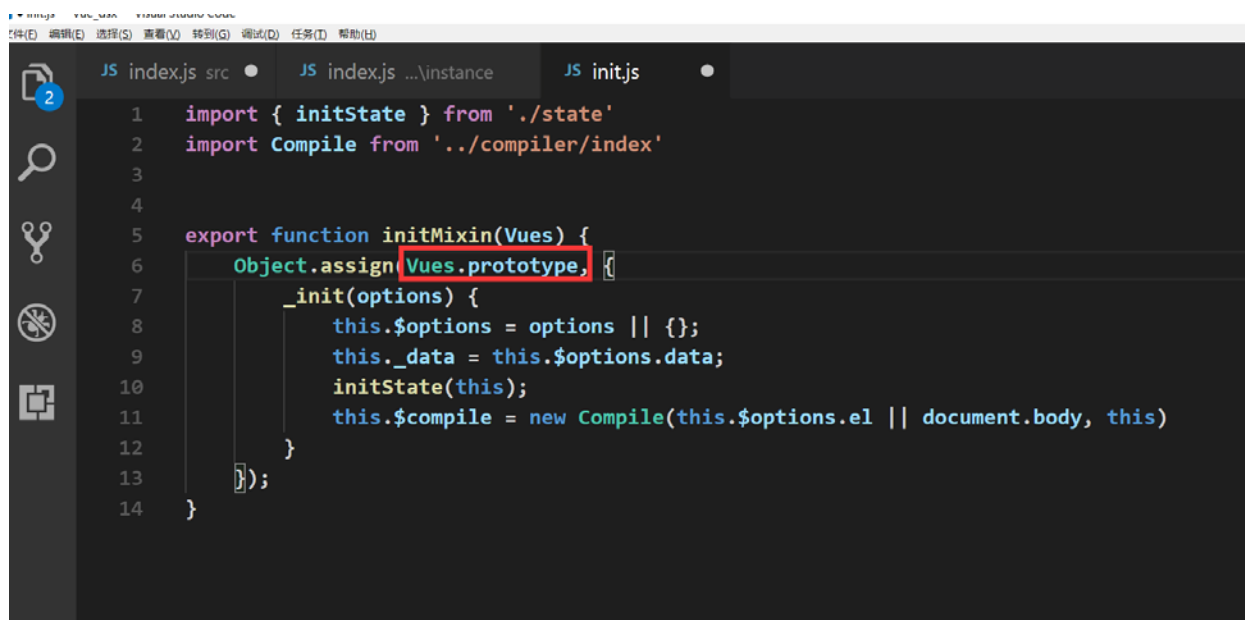
初始化

进入到 ./instance/index.js 就可以看到Vues构造函数



The screenshot shows an IDE with a file explorer on the left and a code editor on the right. The file explorer shows the project structure, including 'src' and 'instance' folders. The code editor displays the 'index.js' file in the 'instance' folder, which contains the Vues constructor and its initialization.

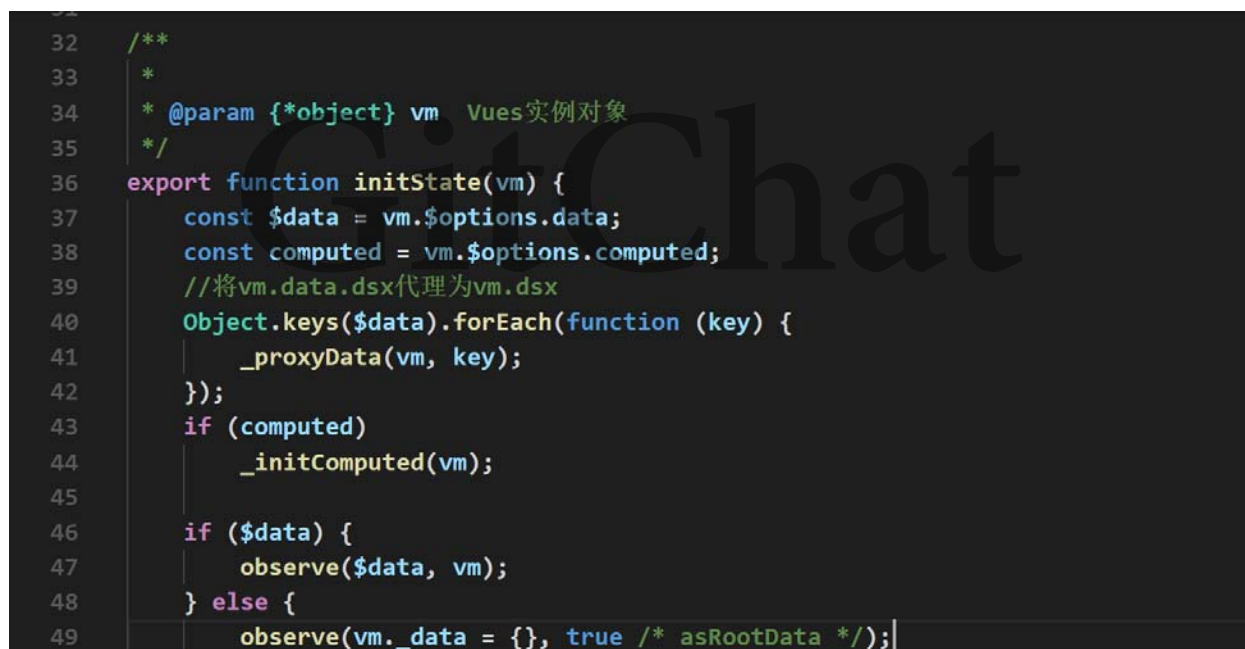
```
JS index.js src  ●  JS index.js ...\instance  x
1  import { initMixin } from './init'
2  import { warn } from '../utils/debug'
3
4
5  class Vues {
6    constructor(options) {
7      if (!(this instanceof Vues)) {
8        warn('Vues is a constructor and should be called with the')
9      }
10     this._init(options)
11   }
12 }
13
14 initMixin(Vues)
15 export default Vues
16
```



```
1 import { initState } from './state'
2 import Compile from '../compiler/index'
3
4
5 export function initMixin(Vues) {
6   Object.assign(Vues.prototype, {
7     _init(options) {
8       this.$options = options || {};
9       this._data = this.$options.data;
10      initState(this);
11      this.$compile = new Compile(this.$options.el || document.body, this)
12    }
13  });
14 }
```

ok, 原来 `_init()` 是在这里定义的, 在Vues的原型上扩展了此方法。Vue也用了这种形式。

在 `_init()` 首先调用了 `initState (this)` :



```
32 /**
33  *
34  * @param {object} vm Vues实例对象
35  */
36 export function initState(vm) {
37   const $data = vm.$options.data;
38   const computed = vm.$options.computed;
39   //将vm.data.dsx代理为vm.dsx
40   Object.keys($data).forEach(function (key) {
41     _proxyData(vm, key);
42   });
43   if (computed)
44     _initComputed(vm);
45
46   if ($data) {
47     observe($data, vm);
48   } else {
49     observe(vm._data = {}, true /* asRootData */);
```

```

    Object.defineProperty(me, key, {
      configurable: false,
      enumerable: true,
      get: function proxyGetter() {
        return me._data[key];
      },
      set: function proxySetter(newVal) {
        me._data[key] = newVal;
      }
    });
  });
};

```

我们初始化计算属性 `computed`，具体就是调用了函数 `_initComputed(vm)`，来看看代码：

```

function _initComputed(vm) {
  var me = vm;
  var computed = me.$options.computed;
  if (typeof computed === 'object') {
    Object.keys(computed).forEach(function (key) {
      Object.defineProperty(me, key, {
        get: typeof computed[key] === 'function' ?
          computed[key] : computed[key].get,
        set: function () { }
      });
    });
  }
}

```

我们可以看出我们已经两次用到同一个方法——`Object.defineProperty()`，这就是**Vue实现响应式数据的利器之一**。举个栗子来说明。

```

var a = {}
Object.defineProperty(a, "b", {
  set: function(newVal){
    console.log("你赋值给我，我的新值是" + newVal);
  }
});

```

observe

这是我们第一个重点，observe很明显我们会用到**观察者**模式，事实上Vue也是这么干的。我们看一下 observe() 做了什么？

```
44
45  /**
46   *
47   * @param {*object} value data对象
48   * @param {*object} vm vues实例
49   */
50  export function observe(value, vm) {
51    if (!value || typeof value !== 'object') {
52      return;
53    }
54    return new Observer(value);
55  };
56
57
```

就是做了类型判断，之后就直接实例化 Observer，参数为data对象。

官网的[Reactivity in Depth](#)上有这么句话：

When you pass a plain JavaScript object to a Vue instance as its data option, Vue.js will walk through all of its properties and convert them to getter/setters

The getter/setters are invisible to the user, but under the hood they enable Vue.js to perform dependency-tracking and change-notification when properties are accessed or modified

observe使data变成“发布者”，watcher是订阅者，订阅data的变化。那如何使data变为“发布者”呢？当然是我们的利器— Object.defineProperty()，将数据对象data的属性转换为访问器属性。看看我们的代码：

```
JS index.js src • JS index.js ...\instance JS init.js JS state.js JS index.js ...\observe x
1
2 class Observer {
3   constructor(data) {
4     this.data = data;
5     this.walk(data);
6   }
7
8   walk(data) {
9     Object.keys(data).forEach((key) => {
10      this.convert(key, data[key]);
11    });
12  }
13  convert(key, val) {
14    this.defineReactive(this.data, key, val);
15  }
16
17  defineReactive(data, key, val) {
18    var dep = new Dep();
19    var childObj = observe(val);
20  }
```

我们遍历data对象的所有可配置属性，最终调用了 `defineReactive()` 函数。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 set 和 get。

先看看 `defineReactive()` 的我们是怎么实现的：

```
defineReactive(data, key, val) {
  var dep = new Dep();
  var childObj = observe(val);
  Object.defineProperty(data, key, {
    enumerable: true, // 可枚举
    configurable: false, // 不能再define
    get: function () {
      if (Dep.target) {
        dep.depend();
      }
      return val;
    },
    set: function (newVal) {
```


pop、push等操作的时候，push进去的对象根本没有进行过双向绑定，更别说pop了，那么我们如何监听数组的这些变化呢？

Vue.js提供的方法是重写push、pop、shift、unshift、splice、sort、reverse这七个数组方法。修改数组原型方法的代码可以参考[observer/array.js](#)以及[observer/index.js](#)。

还有就是利用 `vue.set()` ,借用官方的API

`Vue.set(target, key, value)`

- 参数：

- `{Object | Array} target`
- `{string | number} key`
- `{any} value`

- 返回值：设置的值。

- 用法：

设置对象的属性。如果对象是响应式的，确保属性被创建后也是响应式的，同时触发视图更新。这个方法主要用于避开 Vue 不能检测属性被添加的限制。



注意对象不能是 Vue 实例，或者 Vue 实例的根数据对象。

get的方法主要用来进行**依赖收集**，就是添加订阅者。所以我们只要在最开始进行一次render，那么所有被渲染所依赖的data中的数据就会被getter收集到Dep的subs中去。

set方法会在对象被修改的时候触发（不存在添加属性的情况，添加属性请用Vue.set），这时候set会通知闭包中的Dep，Dep中有一些订阅了这个对象改变的Watcher观察者对象，Dep会通知Watcher对象更新视图。

我们用一个简单的栗子来说明观察者模式：

```
//一个发布者 publisher
let pub = {
```

```

    }
    notify() {
      this.subs.forEach((sub) => {
        sub.update();
      })
    }
  }
}

```

```

//发布者发布消息，主题对象执行notify方法，触发订阅者执行update方法
let dep = new Dep();
pub.publish();//1,2,3

```

那应用到我们这里就是：每个data属性值在defineReactive函数监听处理的时候，添加一个主题对象，当data属性发生改变,通过set函数去通知所有的观察者们，那么如何添加观察者们呢，就是在compleie函数编译template时，通过初始化value值，触发set函数，在set函数中为主题对象添加观察者。有点难理解？直接看代码就明白了。ok，我们继续。看看我们的 Dep：

```

var uid = 0;
export class Dep {
  constructor() {
    //唯一id标识
    this.id = uid++;
    this.subs = [];
  }
  addSub(sub) {
    this.subs.push(sub);
  }
  depend() {
    Dep.target.addDep(this);
  }
  removeSub(sub) {
    var index = this.subs.indexOf(sub);
    if (index !== -1) {
      this.subs.splice(index, 1);
    }
  }
}

```

Watcher

如何实现一个Watcher，通过上面的分析我可以确定得要一个 `update()` 方法。见下图：

```
JS watcher.js ●
1  import { Dep } from './index'
2
3  export class Watcher {
4    constructor(vm, expOrFn, cb) {
5      this.cb = cb;
6      this.vm = vm;
7      this.expOrFn = expOrFn;
8      this.depIds = {};
9      //要判断是表达式还是function
10     if (typeof expOrFn === 'function') {
11       this.getter = expOrFn;
12     } else {
13       this.getter = this.parseGetter(expOrFn);
14     }
15     this.value = this.get();
16   }
17
18   update() {
19     this.run();
20   }
```

很关键的一个地方就是 `this.value = this.get()`，这个就是我们之前说的最开始要进行一次render，我们看 `get()` 实现：

```
get() {
  Dep.target = this;
  var value = this.getter.call(this.vm, this.vm);
  Dep.target = null;
  return value;
}
```

这个最关键了，主要做了以下几件事：

```

update() {
  this.run();
}
run() {
  var value = this.get();
  var oldVal = this.value;
  if (value !== oldVal) {
    this.value = value;
    this.cb.call(this.vm, value, oldVal);
  }
}

```

有一个值得注意的地方，`this.cb.call(this.vm, value, oldVal)`；这个cb是什么？没错就是我们在编译template的时候为每一个指令绑定的更新dom的函数。

最后对watcher做一个总结：

1. 每次调用run()的时候会触发相应属性的get
2. **get**里面会触发 `dep.depend()`，继而触发这里的 `addDep`
3. 假如相应属性的dep.id已经在当前watcher的depIds里，说明不是一个新的属性，仅仅是改变了其值而已。则不需要将当前watcher添加到该属性的dep里
4. 假如相应属性是新的属性，则将当前watcher添加到新属性的dep里，如通过 `vm.child = {name: 'a'}` 改变了 `child.name` 的值，`child.name` 就是个新属性。则需要将当前watcher(`child.name`)加入到新的 `child.name` 的dep里，因为此时 `child.name` 是个新值，**之前的 set，dep 都已经失效**，如果不把 watcher 加入到新的 `child.name` 的dep中。通过 `child.name = xxx` 赋值的时候，对应的 watcher 就收不到通知，等于失效了。
5. 每个子属性的watcher在添加到子属性的dep的同时，也会添加到父属性的dep。
6. 监听子属性的同时监听父属性的变更，这样，父属性改变时，子属性的watcher也能收到通知进行update。这一步是在 `this.get()` --> `this.getVMVal()` 里面

```
this.$compile = new Compile(this.$options.el || document.body,
this)
```

new Compile ,看看做了什么？

```
98
99 class Compile {
100   constructor(el, vm) {
101     this.$vm = vm;
102     this.$el = this.isElementNode(el) ? el : document.querySelector(el);
103
104     if (this.$el) {
105       //把template转化为fragment
106       this.$fragment = this.node2Fragment(this.$el);
107       //解析指令
108       this.init();
109       //把真正dom添加到el中
110       this.$el.appendChild(this.$fragment);
111     }
112   }
113 }
```

注释说的很明白，就不做解释。看看如何转化 fragment：

```
node2Fragment(el) {
  var fragment = document.createDocumentFragment(),
      child;
  // 将原生节点拷贝到fragment
  while (child = el.firstChild) {
    fragment.appendChild(child);
  }
  return fragment;
}
```

就是遍历子节点添加到 fragment 。

```
125
126   init() {
127     this.compileElement(this.$fragment);
```

接下来我们就会对 fragment 节点包括子节点遍历，判断其节点类型，然后调用对应的解析函数解析其中的指令。

我们只看一个 compile：

```
compile(node) {
  var nodeAttrs = node.attributes, me = this;
  [].slice.call(nodeAttrs).forEach((attr) =>{
    var attrName = attr.name;
    //判断是不是内置的指令, v-model, v-click, v-html...
    if (me.isDirective(attrName)) {
      //获取指令要绑定的data中的属性
      var exp = attr.value;
      //将v-model去掉v-
      var dir = attrName.substring(2);
      //判断指令类型 事件||普通
      if (me.isEventDirective(dir)) { // 事件指令
        //调用内置的方法处理
        compileUtil.eventHandler(node, me.$vm, exp, dir);
      } else { // 普通指令
        //调用内置的方法处理
        compileUtil[dir] && compileUtil[dir](node, me.$vm,
exp);
      }
      //最后移除指令属性
      node.removeAttribute(attrName);
    }
  });
}
```

内置的指令处理方法：

```
25
26 // 指令处理集合
27 var compileUtil = {
28   text: function (node, vm, exp) {
29     this.bind(node, vm, exp, 'text');
30   }
31 }
```

```

bind: function (node, vm, exp, dir) {
  var updaterFn = updater[dir + 'Updater'];
  updaterFn && updaterFn(node, this._getVMVal(vm, exp));
  new Watcher(vm, exp, function (value, oldValue) {
    updaterFn && updaterFn(node, value, oldValue);
  });
}

```

1. 获取对应指令的更新方法，并执行
2. new Watcher，在回调函数执行 updaterFn

对于第一条，在执行 updaterFn 的时候会调用 this._getVMVal(vm, exp)：

```

_getVMVal: function (vm, exp) {
  var val = vm;
  exp = exp.split('.');
  exp.forEach(function (k) {
    val = val[k];
  });
  return val;
}

```

很简单，就是获取对应的数据返回。我们看一个text类型的更新函数：

```

textUpdater: function (node, value) {
  node.textContent = typeof value !== 'undefined' ? '' : value;
}

```

这个也是没毛病吧？ok。

第二条，new Watcher，在回调函数执行 updaterFn。还记得之前讲Watcher时提过一句：

```

this.cb = cb;

```

```

JS watcher.js
1  import { Dep } from './index'
2
3  export class Watcher {
4      constructor(vm, expOrFn, cb) {
5          this.cb = cb;
6          this.vm = vm;
7          this.expOrFn = expOrFn;
8          this.depIds = {};
9          //要判断是表达式还是function
10         if (typeof expOrFn === 'function') {
11             this.getter = expOrFn;
12         } else {
13             this.getter = this.parseGetter(expOrFn);
14         }
15         this.value = this.get();
16     }
17
18     update() {
19         this.run();
20     }

```

哈哈，明白了吧。那这个cb啥时执行呢？

当我们修改了数据就会触发对应的 set ，然后就会调用 dep.notify(); ，通知订阅者，再调用订阅者的 update() 方法，update() 方法就会调用 this.run() ，run() 就会执行下面这一句：

```
this.cb.call(this.vm, value, oldVal);
```

最后补充，input 的 v-model 双向数据绑定，其实就是监听了 input 事件，还是贴代码：

```

model: function (node, vm, exp) {
    this.bind(node, vm, exp, 'model');
    var me = this,
        val = this._getVMVal(vm, exp);
    node.addEventListener('input', function (e) {
        var newValue = e.target.value;

```

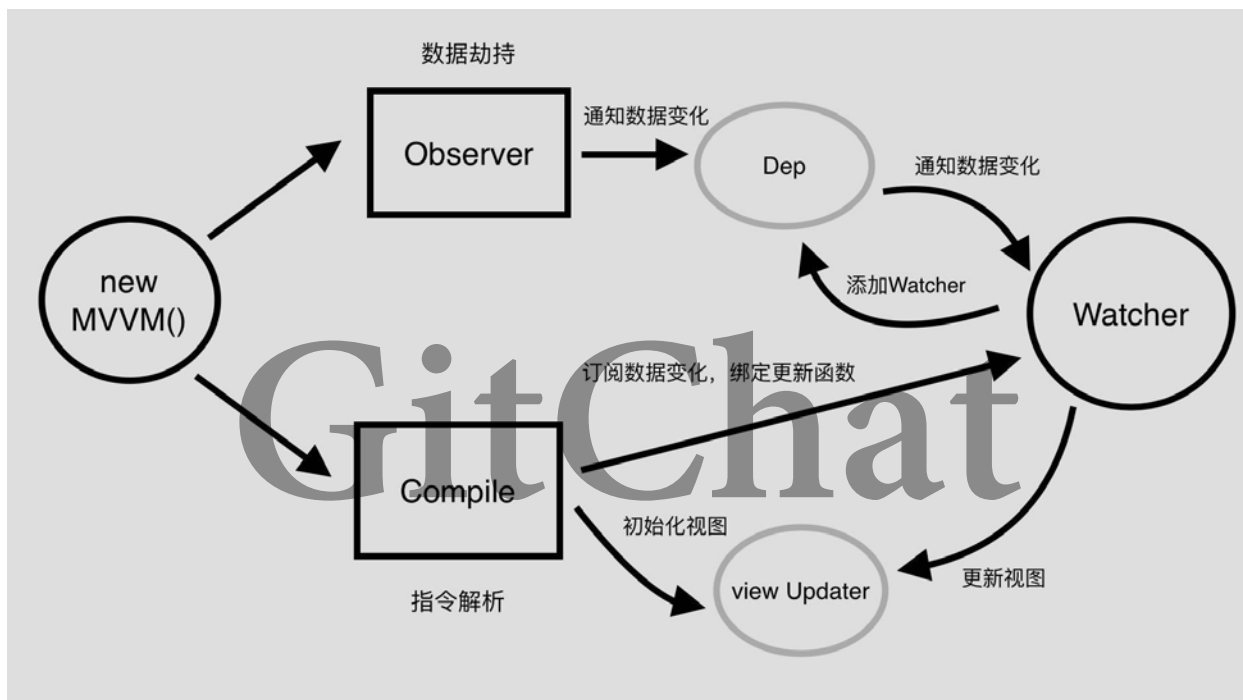


```

exp = exp.split('.');
exp.forEach(function (k, i) {
  // 非最后一个key, 更新val的值
  if (i < exp.length - 1) {
    val = val[k];
  } else {
    val[k] = value;
  }
});
}

```

到这我们就完成了一个缩减版的Vue，当然Vue功能远远不止这些。我们这只是凤毛麟角。学会这个对于你看真正的Vue源码帮助绝对很大，因为逻辑都是相似的。最后再看看下图，回味一下整个过程。



篇幅比较长，有时还很罗嗦，当然我只是想更清楚的讲解，真怕漏掉那个难点。

此次分享的所有源码均上传 [git](#)，[传送门](#)。