## 数据绑定原理

前面已经讲过Vue数据绑定的原理了，现在从源码来看一下数据绑定在Vue中是如何实现的。

首先看一下Vue.js官网介绍响应式原理的这张图。



这张图比较清晰地展示了整个流程，首先通过一次渲染操作触发Data的getter（这里保证只有视图中需要被用到的data才会触发getter）进行依赖收集，这时候其实Watcher与data可以看成一种被绑定的状态（实际上是data的闭包中有一个Deps订阅者，在修改的时候会通知所有的Watcher观察者），在data发生变化的时候会触发它的setter，setter通知Watcher，Watcher进行回调通知组件重新渲染的函数，之后根据diff算法来决定是否发生视图的更新。

Vue在初始化组件数据时，在生命周期的[beforeCreate](https://github.com/vuejs/vue/src/core/instance/init.js#L55)与[created](https://github.com/vuejs/vue/src/core/instance/init.js#L59)钩子函数之间实现了对[data、props、computed、methods、events以及watch](https://github.com/vuejs/vue/blob/dev/src/core/instance/state.js#L43)的处理。

## initData

这里来讲一下[initData](https://github.com/vuejs/vue/blob/dev/src/core/instance/state.js#L107)，可以参考源码instance下的state.js文件，下面所有的中文注释都是我加的，英文注释是尤大加的，请不要忽略英文注释，英文注释都讲到了比较关键或者晦涩难懂的点。

加注释版的vue源码也可以直接通过[传送门](https://github.com/answershuto/learnVue/tree/master/vue-src)查看，这些是我在阅读Vue源码过程中加的注释，持续更新中。

initData主要是初始化data中的数据，将数据进行Observer，监听数据的变化，其他的监视原理一致，这里以data为例。

```javascript

function initData (vm: Component) {

*/\*得到data数据\*/*

  let data = vm.$options.data

  data = vm.\_data = typeof data === 'function'

    ? getData(data, vm)

    : data || {}

*/\*判断是否是对象\*/*

  if (!isPlainObject(data)) {

    data = {}

    process.env.NODE\_ENV !== 'production' && warn(

      'data functions should return an object:\n' +

      'https://vuejs.org/v2/guide/components.html#data-Must-Be-a-Function',

      vm

    )

  }

*// proxy data on instance*

*/\*遍历data对象\*/*

  const keys = Object.keys(data)

  const props = vm.$options.props

  let i = keys.length

*//遍历data中的数据*

  while (i--) {

*/\*保证data中的key不与props中的key重复，props优先，如果有冲突会产生warning\*/*

    if (props && hasOwn(props, keys[i])) {

      process.env.NODE\_ENV !== 'production' && warn(

        `The data property "${keys[i]}" is already declared as a prop. ` +

        `Use prop default value instead.`,

        vm

      )

    } else if (!isReserved(keys[i])) {

*/\*判断是否是保留字段\*/*

*/\*这里是我们前面讲过的代理，将data上面的属性代理到了vm实例上\*/*

      proxy(vm, `\_data`, keys[i])

    }

  }

*/\*Github:https://github.com/answershuto\*/*

*// observe data*

*/\*从这里开始我们要observe了，开始对数据进行绑定，这里有尤大大的注释asRootData，这步作为根数据，下面会进行递归observe进行对深层对象的绑定。\*/*

  observe(data, true */\* asRootData \*/*)

}

```

其实这段代码主要做了两件事，一是将\_data上面的数据代理到vm上，另一件事通过observe将所有数据变成observable。

## proxy

接下来看一下proxy代理。

```javascript

*/\*添加代理\*/*

export function proxy (target: Object, sourceKey: string, key: string) {

  sharedPropertyDefinition.get = function proxyGetter () {

    return this[sourceKey][key]

  }

  sharedPropertyDefinition.set = function proxySetter (val) {

    this[sourceKey][key] = val

  }

  Object.defineProperty(target, key, sharedPropertyDefinition)

}

```

这里比较好理解，通过proxy函数将data上面的数据代理到vm上，这样就可以用app.text代替app.\_data.text了。

## observe

接下来是[observe](https://github.com/vuejs/vue/blob/dev/src/core/observer/index.js#L106)，这个函数定义在core文件下observer的index.js文件中。

```javascript

*/\*\**

*\* Attempt to create an observer instance for a value,*

*\* returns the new observer if successfully observed,*

*\* or the existing observer if the value already has one.*

*\*/*

*/\**

*尝试创建一个Observer实例（\_\_ob\_\_），如果成功创建Observer实例则返回新的Observer实例，如果已有Observer实例则返回现有的Observer实例。*

*\*/*

export function observe (value: any, asRootData: ?boolean): Observer | void {

*/\*判断是否是一个对象\*/*

  if (!isObject(value)) {

    return

  }

  let ob: Observer | void

*/\*这里用\_\_ob\_\_这个属性来判断是否已经有Observer实例，如果没有Observer实例则会新建一个Observer实例并赋值给\_\_ob\_\_这个属性，如果已有Observer实例则直接返回该Observer实例\*/*

  if (hasOwn(value, '\_\_ob\_\_') && value.\_\_ob\_\_ instanceof Observer) {

    ob = value.\_\_ob\_\_

  } else if (

*/\*这里的判断是为了确保value是单纯的对象，而不是函数或者是Regexp等情况。\*/*

    observerState.shouldConvert &&

    !isServerRendering() &&

    (Array.isArray(value) || isPlainObject(value)) &&

    Object.isExtensible(value) &&

    !value.\_isVue

  ) {

    ob = new Observer(value)

  }

  if (asRootData && ob) {

*/\*如果是根数据则计数，后面Observer中的observe的asRootData非true\*/*

    ob.vmCount++

  }

  return ob

}

```

Vue的响应式数据都会有一个\_\_ob\_\_的属性作为标记，里面存放了该属性的观察器，也就是Observer的实例，防止重复绑定。

## Observer

接下来看一下新建的[Observer](https://github.com/vuejs/vue/blob/dev/src/core/observer/index.js#L34)。Observer的作用就是遍历对象的所有属性将其进行双向绑定。

```javascript

*/\*\**

*\* Observer class that are attached to each observed*

*\* object. Once attached, the observer converts target*

*\* object's property keys into getter/setters that*

*\* collect dependencies and dispatches updates.*

*\*/*

export class  {

  value: any;

  dep: Dep;

  vmCount: number; *// number of vms that has this object as root $data*

  constructor (value: any) {

    this.value = value

    this.dep = new Dep()

    this.vmCount = 0

*/\**

*将Observer实例绑定到data的\_\_ob\_\_属性上面去，之前说过observe的时候会先检测是否已经有\_\_ob\_\_对象存放Observer实例了，def方法定义可以参考https://github.com/vuejs/vue/blob/dev/src/core/util/lang.js#L16*

*\*/*

    def(value, '\_\_ob\_\_', this)

    if (Array.isArray(value)) {

*/\**

*如果是数组，将修改后可以截获响应的数组方法替换掉该数组的原型中的原生方法，达到监听数组数据变化响应的效果。*

*这里如果当前浏览器支持\_\_proto\_\_属性，则直接覆盖当前数组对象原型上的原生数组方法，如果不支持该属性，则直接覆盖数组对象的原型。*

*\*/*

      const augment = hasProto

        ? protoAugment  */\*直接覆盖原型的方法来修改目标对象\*/*

        : copyAugment   */\*定义（覆盖）目标对象或数组的某一个方法\*/*

      augment(value, arrayMethods, arrayKeys)

*/\*Github:https://github.com/answershuto\*/*

*/\*如果是数组则需要遍历数组的每一个成员进行observe\*/*

      this.observeArray(value)

    } else {

*/\*如果是对象则直接walk进行绑定\*/*

      this.walk(value)

    }

  }

*/\*\**

*\* Walk through each property and convert them into*

*\* getter/setters. This method should only be called when*

*\* value type is Object.*

*\*/*

  walk (obj: Object) {

    const keys = Object.keys(obj)

*/\*walk方法会遍历对象的每一个属性进行defineReactive绑定\*/*

    for (let i = 0; i < keys.length; i++) {

      defineReactive(obj, keys[i], obj[keys[i]])

    }

  }

*/\*\**

*\* Observe a list of Array items.*

*\*/*

  observeArray (items: Array<any>) {

*/\*数组需要遍历每一个成员进行observe\*/*

    for (let i = 0, l = items.length; i < l; i++) {

      observe(items[i])

    }

  }

}

```

Observer为数据加上响应式属性进行双向绑定。如果是对象则进行深度遍历，为每一个子对象都绑定上方法，如果是数组则为每一个成员都绑定上方法。

如果是修改一个数组的成员，该成员是一个对象，那只需要递归对数组的成员进行双向绑定即可。但这时候出现了一个问题：如果我们进行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](http://v1-cn.vuejs.org/guide/list.html#变异方法)。修改数组原型方法的代码可以参考[observer/array.js](https://github.com/vuejs/vue/blob/dev/src/core/observer/array.js)以及[observer/index.js](https://github.com/vuejs/vue/blob/dev/src/core/observer/index.js#L45)。

```javascript

export class Observer {

  value: any;

  dep: Dep;

  vmCount: number; *// number of vms that has this object as root $data*

  constructor (value: any) {

*//.......*

    if (Array.isArray(value)) {

*/\**

*如果是数组，将修改后可以截获响应的数组方法替换掉该数组的原型中的原生方法，达到监听数组数据变化响应的效果。*

*这里如果当前浏览器支持\_\_proto\_\_属性，则直接覆盖当前数组对象原型上的原生数组方法，如果不支持该属性，则直接覆盖数组对象的原型。*

*\*/*

      const augment = hasProto

        ? protoAugment  */\*直接覆盖原型的方法来修改目标对象\*/*

        : copyAugment   */\*定义（覆盖）目标对象或数组的某一个方法\*/*

      augment(value, arrayMethods, arrayKeys)

*/\*如果是数组则需要遍历数组的每一个成员进行observe\*/*

      this.observeArray(value)

    } else {

*/\*如果是对象则直接walk进行绑定\*/*

      this.walk(value)

    }

  }

}

*/\*\**

*\* Augment an target Object or Array by intercepting*

*\* the prototype chain using \_\_proto\_\_*

*\*/*

*/\*直接覆盖原型的方法来修改目标对象或数组\*/*

function protoAugment (target, src: Object) {

*/\* eslint-disable no-proto \*/*

  target.\_\_proto\_\_ = src

*/\* eslint-enable no-proto \*/*

}

*/\*\**

*\* Augment an target Object or Array by defining*

*\* hidden properties.*

*\*/*

*/\* istanbul ignore next \*/*

*/\*定义（覆盖）目标对象或数组的某一个方法\*/*

function copyAugment (target: Object, src: Object, keys: Array<string>) {

  for (let i = 0, l = keys.length; i < l; i++) {

    const key = keys[i]

    def(target, key, src[key])

  }

}

```

```javascript

*/\**

*\* not type checking this file because flow doesn't play well with*

*\* dynamically accessing methods on Array prototype*

*\*/*

import { def } from '../util/index'

*/\*取得原生数组的原型\*/*

const arrayProto = Array.prototype

*/\*创建一个新的数组对象，修改该对象上的数组的七个方法，防止污染原生数组方法\*/*

export const arrayMethods = Object.create(arrayProto)

*/\*\**

*\* Intercept mutating methods and emit events*

*\*/*

*/\*这里重写了数组的这些方法，在保证不污染原生数组原型的情况下重写数组的这些方法，截获数组的成员发生的变化，执行原生数组操作的同时dep通知关联的所有观察者进行响应式处理\*/*

[

  'push',

  'pop',

  'shift',

  'unshift',

  'splice',

  'sort',

  'reverse'

]

.forEach(function (method) {

*// cache original method*

*/\*将数组的原生方法缓存起来，后面要调用\*/*

  const original = arrayProto[method]

  def(arrayMethods, method, function mutator () {

*// avoid leaking arguments:*

*// http://jsperf.com/closure-with-arguments*

    let i = arguments.length

    const args = new Array(i)

    while (i--) {

      args[i] = arguments[i]

    }

*/\*调用原生的数组方法\*/*

    const result = original.apply(this, args)

*/\*数组新插入的元素需要重新进行observe才能响应式\*/*

    const ob = this.\_\_ob\_\_

    let inserted

    switch (method) {

      case 'push':

        inserted = args

        break

      case 'unshift':

        inserted = args

        break

      case 'splice':

        inserted = args.slice(2)

        break

    }

    if (inserted) ob.observeArray(inserted)

*// notify change*

*/\*dep通知所有注册的观察者进行响应式处理\*/*

    ob.dep.notify()

    return result

  })

})

```

从数组的原型新建一个Object.create(arrayProto)对象，通过修改此原型可以保证原生数组方法不被污染。如果当前浏览器支持\_\_proto\_\_这个属性的话就可以直接覆盖该属性则使数组对象具有了重写后的数组方法。如果没有该属性的浏览器，则必须通过遍历def所有需要重写的数组方法，这种方法效率较低，所以优先使用第一种。

在保证不污染不覆盖数组原生方法添加监听，主要做了两个操作，第一是通知所有注册的观察者进行响应式处理，第二是如果是添加成员的操作，需要对新成员进行observe。

但是修改了数组的原生方法以后我们还是没法像原生数组一样直接通过数组的下标或者设置length来修改数组，可以通过[Vue.set以及splice方法](https://cn.vuejs.org/v2/guide/list.html#%E6%9B%BF%E6%8D%A2%E6%95%B0%E7%BB%84)。

## Watcher

[Watcher](https://github.com/vuejs/vue/blob/dev/src/core/observer/watcher.js#L24)是一个观察者对象。依赖收集以后Watcher对象会被保存在Deps中，数据变动的时候会由Deps通知Watcher实例，然后由Watcher实例回调cb进行视图的更新。

```javascript

export default class Watcher {

  vm: Component;

  expression: string;

  cb: Function;

  id: number;

  deep: boolean;

  user: boolean;

  lazy: boolean;

  sync: boolean;

  dirty: boolean;

  active: boolean;

  deps: Array<Dep>;

  newDeps: Array<Dep>;

  depIds: ISet;

  newDepIds: ISet;

  getter: Function;

  value: any;

  constructor (

    vm: Component,

    expOrFn: string | Function,

    cb: Function,

    options?: Object

  ) {

    this.vm = vm

*/\*\_watchers存放订阅者实例\*/*

    vm.\_watchers.push(this)

*// options*

    if (options) {

      this.deep = !!options.deep

      this.user = !!options.user

      this.lazy = !!options.lazy

      this.sync = !!options.sync

    } else {

      this.deep = this.user = this.lazy = this.sync = false

    }

    this.cb = cb

    this.id = ++uid *// uid for batching*

    this.active = true

    this.dirty = this.lazy *// for lazy watchers*

    this.deps = []

    this.newDeps = []

    this.depIds = new Set()

    this.newDepIds = new Set()

    this.expression = process.env.NODE\_ENV !== 'production'

      ? expOrFn.toString()

      : ''

*// parse expression for getter*

*/\*把表达式expOrFn解析成getter\*/*

    if (typeof expOrFn === 'function') {

      this.getter = expOrFn

    } else {

      this.getter = parsePath(expOrFn)

      if (!this.getter) {

        this.getter = function () {}

        process.env.NODE\_ENV !== 'production' && warn(

          `Failed watching path: "${expOrFn}" ` +

          'Watcher only accepts simple dot-delimited paths. ' +

          'For full control, use a function instead.',

          vm

        )

      }

    }

    this.value = this.lazy

      ? undefined

      : this.get()

  }

*/\*\**

*\* Evaluate the getter, and re-collect dependencies.*

*\*/*

*/\*获得getter的值并且重新进行依赖收集\*/*

  get () {

*/\*将自身watcher观察者实例设置给Dep.target，用以依赖收集。\*/*

    pushTarget(this)

    let value

    const vm = this.vm

*/\**

*执行了getter操作，看似执行了渲染操作，其实是执行了依赖收集。*

*在将Dep.target设置为自身观察者实例以后，执行getter操作。*

*譬如说现在的的data中可能有a、b、c三个数据，getter渲染需要依赖a跟c，*

*那么在执行getter的时候就会触发a跟c两个数据的getter函数，*

*在getter函数中即可判断Dep.target是否存在然后完成依赖收集，*

*将该观察者对象放入闭包中的Dep的subs中去。*

*\*/*

    if (this.user) {

      try {

        value = this.getter.call(vm, vm)

      } catch (e) {

        handleError(e, vm, `getter for watcher "${this.expression}"`)

      }

    } else {

      value = this.getter.call(vm, vm)

    }

*// "touch" every property so they are all tracked as*

*// dependencies for deep watching*

*/\*如果存在deep，则触发每个深层对象的依赖，追踪其变化\*/*

    if (this.deep) {

*/\*递归每一个对象或者数组，触发它们的getter，使得对象或数组的每一个成员都被依赖收集，形成一个“深（deep）”依赖关系\*/*

      traverse(value)

    }

*/\*将观察者实例从target栈中取出并设置给Dep.target\*/*

    popTarget()

    this.cleanupDeps()

    return value

  }

*/\*\**

*\* Add a dependency to this directive.*

*\*/*

*/\*添加一个依赖关系到Deps集合中\*/*

  addDep (dep: Dep) {

    const id = dep.id

    if (!this.newDepIds.has(id)) {

      this.newDepIds.add(id)

      this.newDeps.push(dep)

      if (!this.depIds.has(id)) {

        dep.addSub(this)

      }

    }

  }

*/\*\**

*\* Clean up for dependency collection.*

*\*/*

*/\*清理依赖收集\*/*

  cleanupDeps () {

*/\*移除所有观察者对象\*/*

    let i = this.deps.length

    while (i--) {

      const dep = this.deps[i]

      if (!this.newDepIds.has(dep.id)) {

        dep.removeSub(this)

      }

    }

    let tmp = this.depIds

    this.depIds = this.newDepIds

    this.newDepIds = tmp

    this.newDepIds.clear()

    tmp = this.deps

    this.deps = this.newDeps

    this.newDeps = tmp

    this.newDeps.length = 0

  }

*/\*\**

*\* Subscriber interface.*

*\* Will be called when a dependency changes.*

*\*/*

*/\**

*调度者接口，当依赖发生改变的时候进行回调。*

*\*/*

  update () {

*/\* istanbul ignore else \*/*

    if (this.lazy) {

      this.dirty = true

    } else if (this.sync) {

*/\*同步则执行run直接渲染视图\*/*

      this.run()

    } else {

*/\*异步推送到观察者队列中，由调度者调用。\*/*

      queueWatcher(this)

    }

  }

*/\*\**

*\* Scheduler job interface.*

*\* Will be called by the scheduler.*

*\*/*

*/\**

*调度者工作接口，将被调度者回调。*

*\*/*

  run () {

    if (this.active) {

      const value = this.get()

      if (

        value !== this.value ||

*// Deep watchers and watchers on Object/Arrays should fire even*

*// when the value is the same, because the value may*

*// have mutated.*

*/\**

*即便值相同，拥有Deep属性的观察者以及在对象／数组上的观察者应该被触发更新，因为它们的值可能发生改变。*

*\*/*

        isObject(value) ||

        this.deep

      ) {

*// set new value*

        const oldValue = this.value

*/\*设置新的值\*/*

        this.value = value

*/\*触发回调渲染视图\*/*

        if (this.user) {

          try {

            this.cb.call(this.vm, value, oldValue)

          } catch (e) {

            handleError(e, this.vm, `callback for watcher "${this.expression}"`)

          }

        } else {

          this.cb.call(this.vm, value, oldValue)

        }

      }

    }

  }

*/\*\**

*\* Evaluate the value of the watcher.*

*\* This only gets called for lazy watchers.*

*\*/*

*/\*获取观察者的值\*/*

  evaluate () {

    this.value = this.get()

    this.dirty = false

  }

*/\*\**

*\* Depend on all deps collected by this watcher.*

*\*/*

*/\*收集该watcher的所有deps依赖\*/*

  depend () {

    let i = this.deps.length

    while (i--) {

      this.deps[i].depend()

    }

  }

*/\*\**

*\* Remove self from all dependencies' subscriber list.*

*\*/*

*/\*将自身从所有依赖收集订阅列表删除\*/*

  teardown () {

    if (this.active) {

*// remove self from vm's watcher list*

*// this is a somewhat expensive operation so we skip it*

*// if the vm is being destroyed.*

*/\*从vm实例的观察者列表中将自身移除，由于该操作比较耗费资源，所以如果vm实例正在被销毁则跳过该步骤。\*/*

      if (!this.vm.\_isBeingDestroyed) {

        remove(this.vm.\_watchers, this)

      }

      let i = this.deps.length

      while (i--) {

        this.deps[i].removeSub(this)

      }

      this.active = false

    }

  }

}

```

## Dep

来看看[Dep](https://github.com/vuejs/vue/blob/dev/src/core/observer/dep.js#L12)类。其实Dep就是一个发布者，可以订阅多个观察者，依赖收集之后Deps中会存在一个或多个Watcher对象，在数据变更的时候通知所有的Watcher。

```javascript

*/\*\**

*\* A dep is an observable that can have multiple*

*\* directives subscribing to it.*

*\*/*

export default class Dep {

  static target: ?Watcher;

  id: number;

  subs: Array<Watcher>;

  constructor () {

    this.id = uid++

    this.subs = []

  }

*/\*添加一个观察者对象\*/*

  addSub (sub: Watcher) {

    this.subs.push(sub)

  }

*/\*移除一个观察者对象\*/*

  removeSub (sub: Watcher) {

    remove(this.subs, sub)

  }

*/\*依赖收集，当存在Dep.target的时候添加观察者对象\*/*

  depend () {

    if (Dep.target) {

      Dep.target.addDep(this)

    }

  }

*/\*通知所有订阅者\*/*

  notify () {

*// stabilize the subscriber list first*

    const subs = this.subs.slice()

    for (let i = 0, l = subs.length; i < l; i++) {

      subs[i].update()

    }

  }

}

*// the current target watcher being evaluated.*

*// this is globally unique because there could be only one*

*// watcher being evaluated at any time.*

Dep.target = null

*/\*依赖收集完需要将Dep.target设为null，防止后面重复添加依赖。\*/*

```

## defineReactive

接下来是[defineReactive](https://github.com/vuejs/vue/blob/dev/src/core/observer/index.js#L131)。defineReactive的作用是通过Object.defineProperty为数据定义上getter\setter方法，进行依赖收集后闭包中的Deps会存放Watcher对象。触发setter改变数据的时候会通知Deps订阅者通知所有的Watcher观察者对象进行试图的更新。

```javascript

*/\*\**

*\* Define a reactive property on an Object.*

*\*/*

export function defineReactive (

  obj: Object,

  key: string,

  val: any,

  customSetter?: Function

) {

*/\*在闭包中定义一个dep对象\*/*

  const dep = new Dep()

  const property = Object.getOwnPropertyDescriptor(obj, key)

  if (property && property.configurable === false) {

    return

  }

*/\*如果之前该对象已经预设了getter以及setter函数则将其取出来，新定义的getter/setter中会将其执行，保证不会覆盖之前已经定义的getter/setter。\*/*

*// cater for pre-defined getter/setters*

  const getter = property && property.get

  const setter = property && property.set

*/\*对象的子对象递归进行observe并返回子节点的Observer对象\*/*

  let childOb = observe(val)

  Object.defineProperty(obj, key, {

    enumerable: true,

    configurable: true,

    get: function reactiveGetter () {

*/\*如果原本对象拥有getter方法则执行\*/*

      const value = getter ? getter.call(obj) : val

      if (Dep.target) {

*/\*进行依赖收集\*/*

        dep.depend()

        if (childOb) {

*/\*子对象进行依赖收集，其实就是将同一个watcher观察者实例放进了两个depend中，一个是正在本身闭包中的depend，另一个是子元素的depend\*/*

          childOb.dep.depend()

        }

        if (Array.isArray(value)) {

*/\*是数组则需要对每一个成员都进行依赖收集，如果数组的成员还是数组，则递归。\*/*

          dependArray(value)

        }

      }

      return value

    },

    set: function reactiveSetter (newVal) {

*/\*通过getter方法获取当前值，与新值进行比较，一致则不需要执行下面的操作\*/*

      const value = getter ? getter.call(obj) : val

*/\* eslint-disable no-self-compare \*/*

      if (newVal === value || (newVal !== newVal && value !== value)) {

        return

      }

*/\* eslint-enable no-self-compare \*/*

      if (process.env.NODE\_ENV !== 'production' && customSetter) {

        customSetter()

      }

      if (setter) {

*/\*如果原本对象拥有setter方法则执行setter\*/*

        setter.call(obj, newVal)

      } else {

        val = newVal

      }

*/\*新的值需要重新进行observe，保证数据响应式\*/*

      childOb = observe(newVal)

*/\*dep对象通知所有的观察者\*/*

      dep.notify()

    }

  })

}

```

现在再来看这张图是不是更清晰了呢？

![](https://cn.vuejs.org/images/data.png)