

## 学习目标

### 学习过程

1. 让一个数据会变
  - 1.1 我首先想到的是，将c放在一个函数里面，当a或者b发生变化的时候，我们调用一下那个函数，让c回炉重造不就可以变化了吗？
2. 让一个对象会变
  - 2.1 拦截到对对象属性的获取与设置
  - 2.2 让对象属性变化
3. 源码阅读
  - 测试代码
  - 跟踪调用栈
  - 自己画的一个流程图
  - talk is cheap, show me the code
    - src\core\instance\index.js
    - src\core\instance\init.js
    - src\core\instance\state.js\initState
    - src\core\instance\state.js\initData
    - src\core\instance\state.js\proxy
    - src\core\observer\index.js\observe
    - src\core\observer\index.js\Observer类
    - src\core\observer\dep.js\Dep类
    - src\core\observer\watcher.js\watcher类

### 学习成果

1. 什么是响应式
  2. vue 怎么知道数据更新？
  3. 什么是 watcher？
  4. 什么是 Dep
  5. 什么是 observer？
  6. vue里面有多少种不同Watcher？
- 面试题：请谈谈你对数据响应式原理的理解

---

## 学习目标

1. 搞清楚什么是响应式
2. vue 怎么知道我们数据更新了
3. 模拟数据响应式
3. 通过阅读 vue2 源码，理解Vue的双向数据绑定原理，可以跟面试官拉扯
4. 什么是 watcher
5. 什么是 Dep

---

## 学习过程

所谓数据响应式，不过是当依赖发生变化的时候，目标（视图）自动更新。

所以，要想理解数据响应式，我们先来尝试一下怎么让数据自动发生变化。

## 1. 让一个数据会变

模拟数据响应式，即当数据的依赖发生变化时，`target` 也发生变化。如：

```
1 | let c = a+b;
```

这里我们把 `c` 称为 `target`，`a, b` 是 `c` 的依赖，因为 `c` 是根据 `a` 和 `b` 得出来的。

那么我们怎么让 `c` 在 `a` 或者 `b` 发生变化时，`c` 也跟着发生变化呢？

### 1.1 我首先想到的是，将 `c` 放在一个函数里面，当 `a` 或者 `b` 发生变化的时候，我们调用一下那个函数，让 `c` 回炉重造不就可以变化了吗？

让我们来测试一下我的想法。

```
1 | "use strict";
2 |
3 | let a = 1
4 | let b = 2
5 | let c = null;
6 |
7 | function fn(){
8 |     c =a+b;
9 | }
10 | console.log('1-',a,b,c)           // 1- 1 2 null
11 | // 明显这里c 为null
12 |
13 | // 调用一下函数，让c得到初始化
14 | fn()
15 |
16 | console.log('2-',a,b,c)           // 2- 1 2 3
17 | // 到了这里c=3
18 |
19 | // 依赖发生变化
20 | a = 9
21 | console.log('3-',a,b,c)           // 3- 9 2 3
22 | // 这里a发生变化，但是c并没有发生变化
23 |
24 | fn()
25 | console.log('4-',a,b,c)           // 4- 9 2 11
26 | // 调用fn之后c=11，发生了变化
```

很明显到了这里，我们手动调用函数 `target` 确实会更新，但是老是手动的话，感觉怪怪的，那有没有什么办法，可以让他自动调用呢？

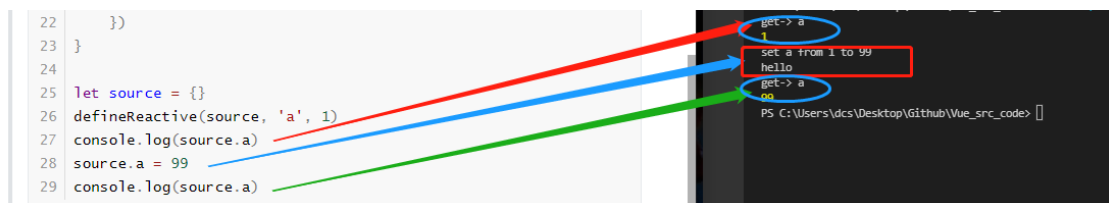
到了这里我们要解决的问题有两个：

1. 怎么知道自动数据发生变化
2. 怎么自动调用一个特定函数

1.2 我想到的办法是红宝书里面看到的  
`Object.defineProperty()`, 利用  
`Object.defineProperty()` 的 `getter` 以及 `setter` 的  
拦截特性, 让我们来测试一下。

```
1  "use strict";
2
3  function say() {
4    console.log('hello')
5  }
6
7  function defineReactive(obj, key, val) {
8    return Object.defineProperty(obj, key, {
9      get() {
10        console.log('get->', key)
11        return val
12      },
13
14      set(newVal) {
15        if (newVal === val) return;
16        console.log(`set ${key} from ${val} to ${newVal}`)
17
18        // 数据发生变化, 我们就调用函数
19        say()
20        val = newVal
21      }
22    })
23  }
24
25  let source = {}
26  defineReactive(source, 'a', 1)
27  console.log(source.a)
28  source.a = 99
29  console.log(source.a)
```

结果:



可以看到我们对a的get以及set 都被识别到了, 而且say函数也被成功调用了。

那我们怎么复现 `c = a + b` 这个例子呢? 如下:

```
1  "use strict";
2  function defineReactive(obj, key, val) {
```

```

3   return Object.defineProperty(obj, key, {
4     get() {
5       console.log('get->', key)
6       return val
7     },
8
9     set(newVal) {
10      if (newVal === val) return;
11      console.log(`set ${key} from ${val} to ${newVal}`)
12      // 数据发生变化，我们就调用函数
13      fn()
14      val = newVal
15    }
16  })
17 }
18
19 let source = {}
20 defineReactive(source, 'a', 1)
21 defineReactive(source, 'b', 2)
22
23 let c;
24 function fn(){
25   c = source.a + source.b;
26   console.log('c自动变化成为', c)
27 }
28
29 // 初始化一下c
30 fn()
31
32 source.a = 99
33
34
35

```



我们发现fn虽然被自动调用了，但是c的值依然是3，那应该怎么解决呢？

经过查看发现是set里面的问题（先调用了fn导致val还没来得及发生变化。）

```

1   set(newVal) {
2     if (newVal === val) return;
3     console.log(`set ${key} from ${val} to ${newVal}`)
4     // 数据发生变化，我们就调用函数
5     val = newVal
6     fn()
7   }

```

上面的问题就解决了。

## 2. 让一个对象会变

因为对象操作比较复杂，所以我们先实现对对象操作的拦截，比如对象的获取与设置我都知道。

## 2.1 拦截到对对象属性的获取与设置

新加observe对一个对象的属性遍历进行重新定义（类似于定义一个数据可变）

```
1  "use strict";
2  /**
3   * 这里的defineReactive实际上是一个闭包，
4   * 外面的外面引用着函数内的变量，导致这些临时变量一直存在
5   */
6  function defineReactive(obj, key, val){
7      // 2. observe 避免key的val是一个对象，对象里面的值没有响应式
8      observe(val)
9      // 利用getter setter 去拦截数据
10     Object.defineProperty(obj, key, {
11         get(){
12             console.log('get', key)
13             return val
14         },
15         set(newVal){
16             if( newVal !== val){
17                 console.log(`set ${val} -> ${newVal}`)
18                 val = newVal
19             }
20         }
21     })
22 }
23
24 // 2. 观察一个对象，让这个对象的属性变成响应式
25 function observe(obj){
26     // 希望传入的是一个Object
27
28     if( typeof obj !== 'object' || typeof(obj) == null){
29         return ;
30     }
31     Object.keys(obj).forEach(key=>{
32         defineReactive(obj, key, obj[key])
33     })
34 }
35
36 let o = { a: 1, b: 'hello', c:{age:9}}
37 observe(o)
38
39 o.a
40 o.a = 2
41 o.b
42 o.b = 'world'
```



## 2.2 让对象属性变化

为了简化程序，我们只看一层的对象

```
1  "use strict";
2  const { log } = console;
3
4  let target = null;
5  let data = { a: 1, b:2 }
6  let c, d;
7
8  // 依赖收集,每个Object的key都有一个Dep实例
9  class Dep{
10     constructor(){
11         this.deps = []
12     }
13     depend(){
14         target && !this.deps.includes(target) && this.deps.push(target)
15     }
16     notify() {
17         this.deps.forEach(dep=>dep() )
18     }
19 }
20
21 Object.keys(data).forEach(key=>{
22     let v = data[key]
23     const dep = new Dep()
24
25     Object.defineProperty(data, key, {
26         get(){
27             dep.depend()
28             return v;
29         },
30         set(nv){
31             v = nv
32             dep.notify()
33         }
34     })
35 })
36
37 function watcher(fn) {
38     target = fn
39     target()
40     target = null
41 }
42
43 watcher(()=>{
44     c = data.a + data.b
45 })
46
47 watcher(()=>{
48     d = data.a - data.b
49 })
50
51 log('c=',c)
52 log('d=',d)
53 data.a = 99
```

```
54 log('c=', c)
55 log('d=', d)
56
57 /**
58 c= 3
59 d= -1
60 c= 101
61 d= 97
62 */
```

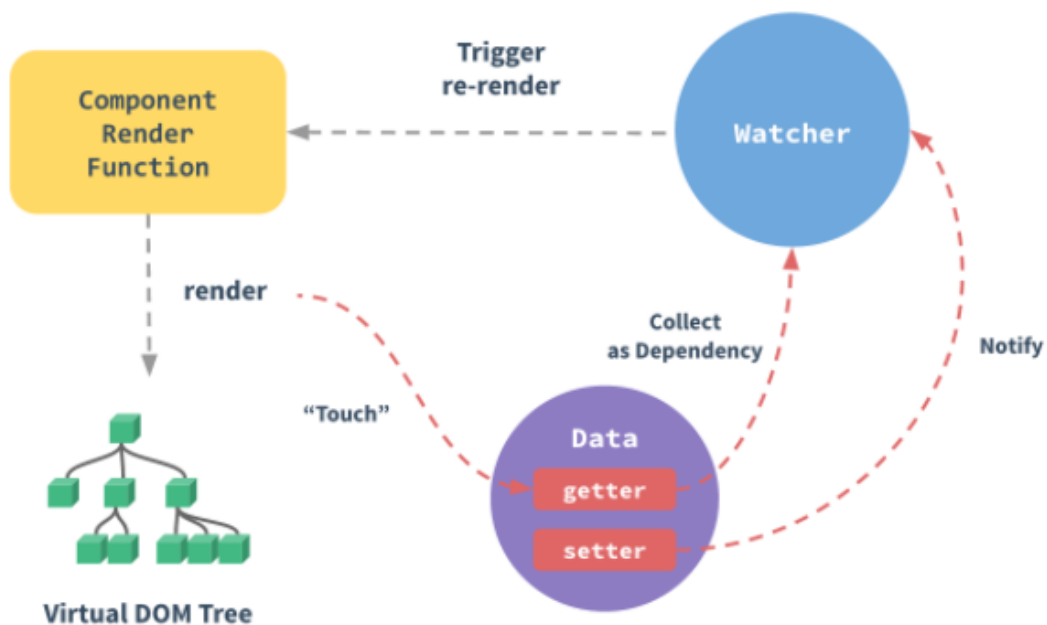
- 简述一下这个过程：

对data对象里面的每一个key利用defineProperty进行数据拦截，在get里面进行Dep依赖收集，在set里面通知数据更新。

依赖收集实则是将watcher实例加入deps队列，当接到通知更新的时候，对队列里面的函数遍历执行，达到数据自动更新的效果。

### 3. 源码阅读

在阅读源码的时候，为了我们方便寻找入口，我们先来看看官网对数据响应式的阐述。



看完官方给的图，我们可以明确知道，`watcher`的粒度是组件，也就是说，每一个组件对应一个`watcher`。

那么`watcher`究竟是什么呢？`Dep`又是什么？`observer`又是做什么用的？下面让我们到源码中去寻找答案吧。

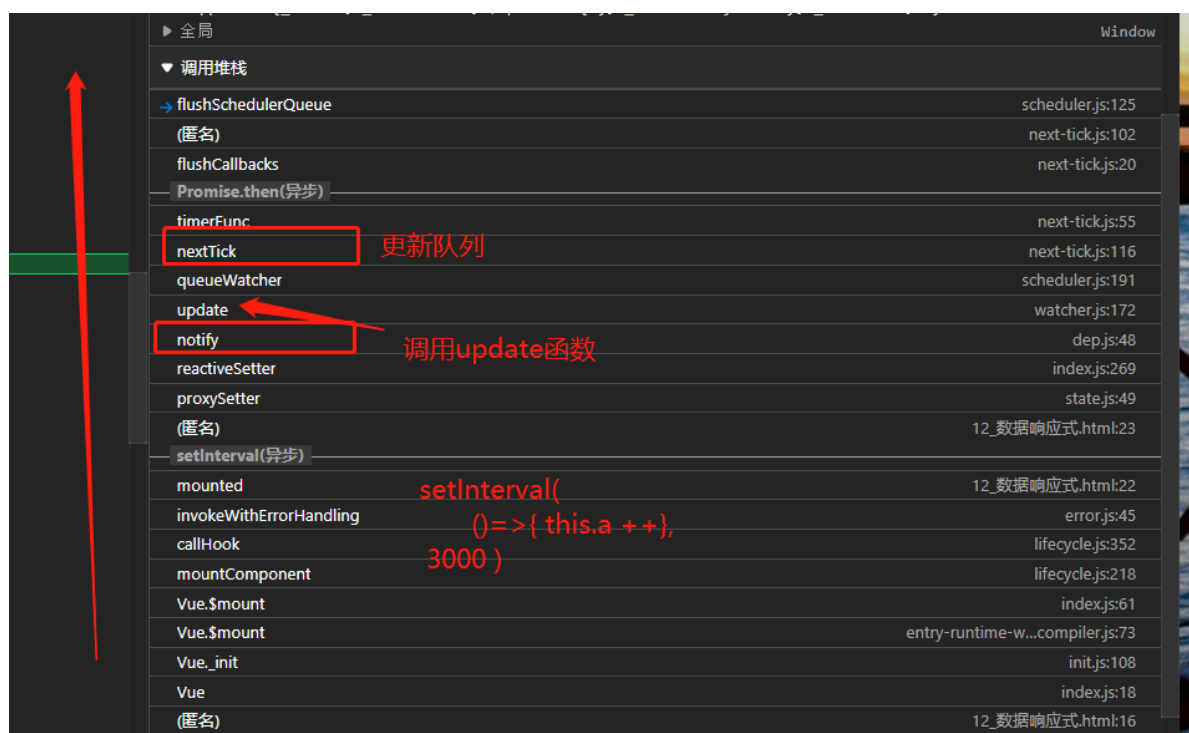
## 测试代码

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Document</title>
8   <script src='../dist/vue.js'></script>
9 </head>
10 <body>
11   <div id="app">
12     {{a}}
13   </div>
14
15   <script>
16     const app = new Vue({
17       el: '#app',
18       data: {
19         a: 1
20       },
21       mounted(){
22         setInterval(()=>{
23           this.a ++
24         }, 3000)
25       }
26     })
27   </script>
28 </body>
29 </html>
30

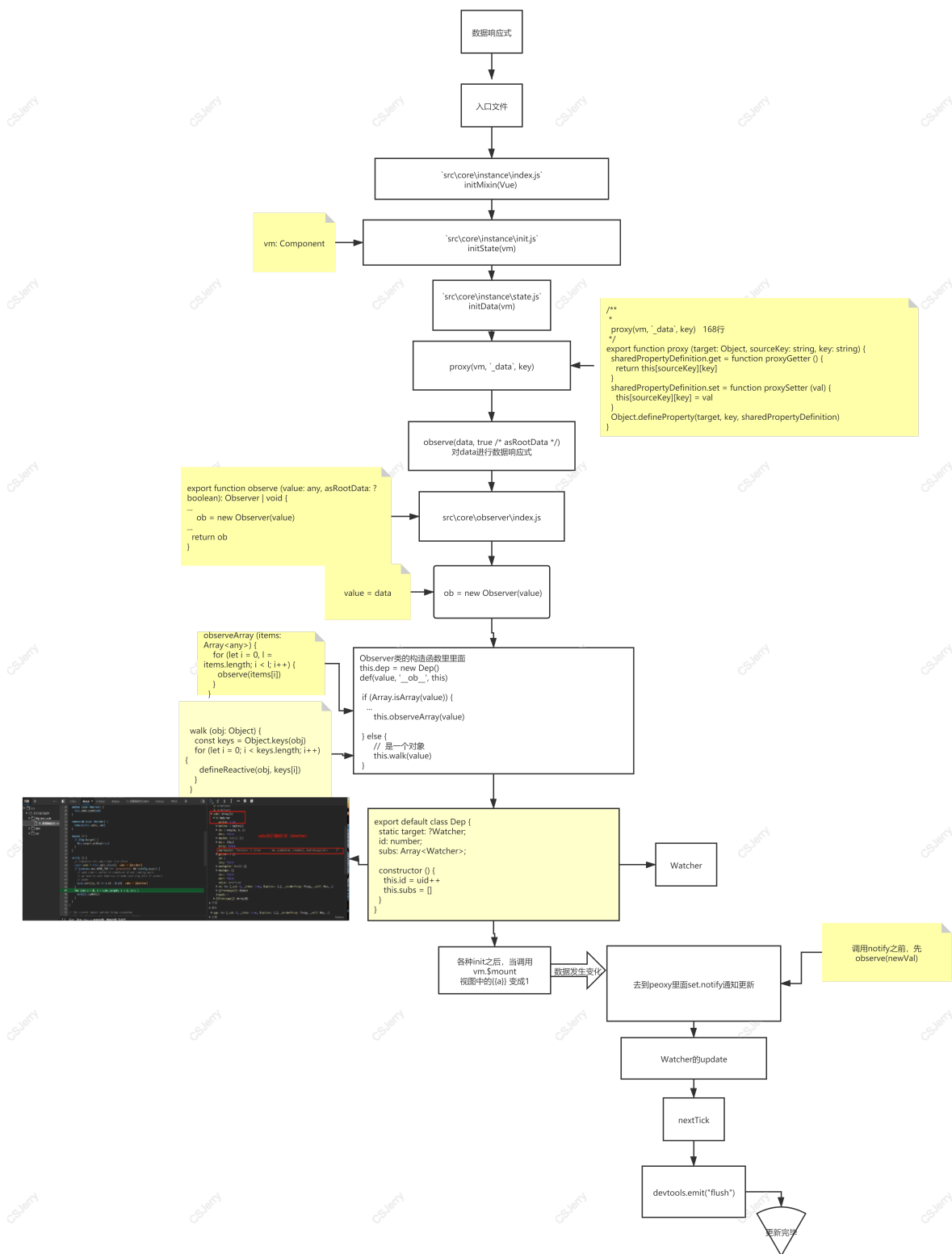
```

## 跟踪调用栈





## 自己画的一个流程图



## talk is cheap, show me the code

特别声明, 为简化流程, 所有的源码展示, 均经过删减

src\core\instance\index.js

一个入口文件

```
1 function Vue (options) {
2   /**
```

```

3      * 初始化
4      */
5      this._init(options)
6  }
7  /**
8   * 以下通过给Vue.prototype挂载的方法，混入其他方法
9   */
10  initMixin(Vue)
11  /**
12   * initMixin
13   通过该方法，给Vue提供__init方法， 初始化生命周期
14   initLifecycle -> initEvents -> initRender
15   -> callHook(vm, 'beforeCreate') -> initInjections
16   -> initState -> initProvide
17   -> callHook(vm, 'created')
18   -> if (vm.$options.el) {
19       vm.$mount(vm.$options.el)
20   }
21  */
22
23  stateMixin(Vue)
24  /**
25  stateMixin
26  $data -> $props -> $set -> $delete -> $watch
27  */
28
29  eventsMixin(Vue)
30  /** eventsMixin
31  $on $once $off $emit
32  */
33
34  lifecycleMixin(Vue)
35  /** lifecycleMixin
36   * _update(), $forceUpdate, $destroy
37   *
38   */
39  renderMixin(Vue)
40  /**
41   * $nextTick, _render, $vnode
42   *
43   */
44
45  export default Vue
46

```

## src\core\instance\init.js

```

/**
 * initMixin
 通过该方法，给Vue提供__init方法， 初始化生命周期
initLifecycle -> initEvents -> initRender
-> callHook(vm, 'beforeCreate') -> initInjections
-> initState -> initProvide

```

```

-> callHook(vm, 'created')

-> if (vm.$options.el) {

  vm.mount(vm.options.el)

}

*/

```

```

1  export function initMixin (Vue: Class<Component>) {
2
3
4    /**
5     * @重要 数据初始化, 响应式
6     * $set $delete $watch
7     * 在reject之后, 初始化数据, 达到去重的效果
8     */
9    initState(vm)
10
11   /**
12    * 调用created钩子函数
13    */
14   callHook(vm, 'created')
15
16   if (vm.$options.el) {
17     vm.$mount(vm.$options.el)
18   }
19 }
20 }

```

## src\core\instance\state.js\initState

对data进行预处理

```

1  export function initState (vm: Component) {
2    vm._watchers = []
3    const opts = vm.$options
4    /**
5     * state的初始化顺序
6     * props -> methods -> data -> computed -> watch
7     */
8    if (opts.props) initProps(vm, opts.props)
9    if (opts.methods) initMethods(vm, opts.methods)
10   if (opts.data) { initData(vm) }
11   if (opts.computed) initComputed(vm, opts.computed)
12 }
13

```

## src\core\instance\state.js\initData

对data进行observe, 对数据的getter,setter拦截

```

1  function initData (vm: Component) {
2    let data = vm.$options.data
3    // proxy data on instance
4    /**

```

```

5      * 数据代理
6      */
7      const keys = Object.keys(data)
8      let i = keys.length
9      while (i--) {
10         const key = keys[i]
11         /**
12          * @代理
13          */
14         proxy(vm, `_data`, key)
15     }
16 }
17 /**
18  * @响应式操作
19  */
20 observe(data, true /* asRootData */)
21 }

```

### src\core\instance\state.js\proxy

```

1  const sharedPropertyDefinition = {
2      enumerable: true,
3      configurable: true,
4      get: noop,
5      set: noop
6  }
7
8
9  /**
10   *
11   proxy(vm, `_data`, key)    168行
12   */
13  export function proxy (target: Object, sourceKey: string, key: string) {
14      sharedPropertyDefinition.get = function proxyGetter () {
15          return this[sourceKey][key]
16      }
17      sharedPropertyDefinition.set = function proxySetter (val) {
18          this[sourceKey][key] = val
19      }
20      Object.defineProperty(target, key, sharedPropertyDefinition)
21  }

```

### src\core\observer\index.js\observe

#### 创建观察者实例

```

1  export function observe (value: any, asRootData: ? boolean): Observer | void
2  {
3      if (!isObject(value) || value instanceof VNode) {
4          return
5      }
6      /**
7       * @观察者
8       */

```

```

8   let ob: Observer | void
9   ob = new Observer(value)
10  }
11  if (asRootData && ob) {
12    ob.vmCount++
13  }
14  return ob
15
16 }

```

## src\core\observer\index.js\Observer类

```

1  export class Observer {
2    value: any;
3    dep: Dep;
4    vmCount: number; // number of vms that have this object as root $data
5
6    constructor (value: any) {
7      this.value = value
8
9      /**
10       * @思考为什么在Observer里面声明dep
11       * 创建Dep实例
12       * Object 里面新增或者删除属性
13       * array 中有变更方法
14       */
15      this.dep = new Dep()
16      this.vmCount = 0
17
18      /**
19       * 设置一个一个 __ob__ 属性，引用当前Observer实例
20       */
21
22      /**
23       *
24       export function def (obj: Object, key: string, val: any, enumerable?:
25       boolean) {
26         Object.defineProperty(obj, key, {
27           value: val,
28           enumerable: !!enumerable,
29           writable: true,
30           configurable: true
31         })
32       }
33
34       def(value, '__ob__', this)
35
36       /**
37       * 类型判断
38       */
39       // 数组
40       if (Array.isArray(value)) {
41         if (hasProto) {
42           protoAugment(value, arrayMethods)
43         } else {
44           copyAugment(value, arrayMethods, arrayKeys)
45         }
46       }
47     }
48   }

```

```

45     /**
46      * 如果数组里面的元素还是对象，还需要进行响应式处理
47      */
48     this.observeArray(value)
49
50   } else {
51     // 是一个对象
52     this.walk(value)
53   }
54 }
55
56 /**
57  * walk through all properties and convert them into
58  * getter/setters. This method should only be called when
59  * value type is Object.
60  */
61 walk (obj: Object) {
62   const keys = Object.keys(obj)
63   for (let i = 0; i < keys.length; i++) {
64     defineReactive(obj, keys[i])
65   }
66 }
67
68 /**
69  * Observe a list of Array items.
70  */
71 observeArray (items: Array<any>) {
72   for (let i = 0, l = items.length; i < l; i++) {
73     observe(items[i])
74   }
75 }
76 }

```

## src\core\observer\dep.js\Dep类

依赖收集

subs是一个Watcher队列

```

1  /**
2   * A dep is an observable that can have multiple
3   * directives subscribing to it.
4   * dep是一个可观察对象，可以有多个指令订阅它。
5   */
6  export default class Dep {
7    static target: ?Watcher;
8    id: number;
9    subs: Array<Watcher>;
10
11    constructor () {
12      this.id = uid++
13      this.subs = []
14    }
15
16    addSub (sub: Watcher) {
17      this.subs.push(sub)
18    }

```

```

19
20   removeSub (sub: watcher) {
21     remove(this.subs, sub)
22   }
23
24   depend () {
25     Dep.target && Dep.target.addDep(this)
26   }
27
28   notify () {
29     const subs = this.subs.slice()
30     for (let i = 0, l = subs.length; i < l; i++) {
31       subs[i].update()
32     }
33   }
34 }
35

```

### src\core\observer\watcher.js\watcher类

```

1  /**
2   *观察者解析表达式，收集依赖项，
3   *并在表达式值改变时触发回调。
4   *这用于$watch() api和指令。
5   */
6  export default class Watcher {
7    constructor () {
8      this.vm = vm
9      if (isRenderWatcher) {
10        vm._watcher = this
11      }
12      vm._watchers.push(this)
13
14      this.expression = process.env.NODE_ENV !== 'production'
15        ? expOrFn.toString()
16        : ''
17      // parse expression for getter
18      if (typeof expOrFn === 'function') {
19        this.getter = expOrFn
20      } else {
21        this.getter = parsePath(expOrFn)
22        if (!this.getter) {
23          this.getter = noop
24          process.env.NODE_ENV !== 'production' && warn(
25            `Failed watching path: "${expOrFn}" ` +
26            'watcher only accepts simple dot-delimited paths. ' +
27            'For full control, use a function instead.',
28            vm
29          )
30        }
31      }
32      this.value = this.lazy
33        ? undefined
34        : this.get()
35    }
36
37    /**

```

```

38     * Evaluate the getter, and re-collect dependencies.
39     */
40     get () {
41         pushTarget(this)
42         ...
43         popTarget()
44         this.cleanupDeps()
45         return value
46     }
47
48     /**
49     * Add a dependency to this directive.
50     */
51     addDep (dep: Dep) {
52         const id = dep.id
53         if (!this.newDepIds.has(id)) {
54             this.newDepIds.add(id)
55             this.newDeps.push(dep)
56             if (!this.depIds.has(id)) {
57                 dep.addSub(this)
58             }
59         }
60     }
61
62     /**
63     * Clean up for dependency collection.
64     */
65     cleanupDeps () {
66         let i = this.deps.length
67         while (i--) {
68             const dep = this.deps[i]
69             if (!this.newDepIds.has(dep.id)) {
70                 dep.removeSub(this)
71             }
72         }
73     }
74
75     /**
76     * Subscriber interface.
77     * Will be called when a dependency changes.
78     */
79     update () {
80         /* istanbul ignore else */
81         if (this.lazy) {
82             this.dirty = true
83         } else if (this.sync) {
84             this.run()
85         } else {
86             queueWatcher(this)
87         }
88     }
89
90     /**
91     * Scheduler job interface.
92     * Will be called by the scheduler.
93     */
94     run () {
95         if (this.active) {

```



```

96     const value = this.get()
97     if (
98         value !== this.value ||
99         // Deep watchers and watchers on Object/Arrays should fire even
100         // when the value is the same, because the value may
101         // have mutated.
102         isObject(value) ||
103         this.deep
104     ) {
105         // set new value
106         const oldValue = this.value
107         this.value = value
108         if (this.user) {
109             const info = `callback for watcher "${this.expression}"`
110             invokeWithErrorHandling(this.cb, this.vm, [value, oldValue],
111 this.vm, info)
112         } else {
113             this.cb.call(this.vm, value, oldValue)
114         }
115     }
116 }
117
118 /**
119  * Evaluate the value of the watcher.
120  * This only gets called for lazy watchers.
121  */
122 evaluate () {
123     this.value = this.get()
124     this.dirty = false
125 }
126
127 /**
128  * Depend on all deps collected by this watcher.
129  */
130 depend () {
131     let i = this.deps.length
132     while (i--) {
133         this.deps[i].depend()
134     }
135 }
136
137 /**
138  * Remove self from all dependencies' subscriber list.
139  */
140 teardown () {
141     if (this.active) {
142         if (!this.vm._isBeingDestroyed) {
143             remove(this.vm._watchers, this)
144         }
145         let i = this.deps.length
146         while (i--) {
147             this.deps[i].removeSub(this)
148         }
149         this.active = false
150     }
151 }
152 }

```

# 学习成果

声明，以下所说的data如果没有特别声明，都是指，定义组件时预定义的data对象。

## 1. 什么是响应式

拿  $c=a+b$  说明，当a或者b发生变化的时候，c也会跟着发生变化，这就是数据响应式的本质。

## 2. Vue怎么知道数据更新？

Vue2.x对用户定义的data，利用 `Object.defineProperty` 进行重定义然后才挂载到组件上，当组件获取或者更新数据，会触发getter或setter,也就让组件知道了用户对数据进行了“操作”。

## 3. 什么是watcher？

在Vue里面一个组件对应着一个watcher，我们称之为“订阅者”。

它的作用是：订阅数据的变化并执行相应操作（例如更新视图 `update`）。

## 4. 什么是Dep

组件data对象的每一个key都对应者一个Dep实例。

Dep是一个可观察类，当他被实例化之后，Watcher就可以订阅它。

Vue会在Dep里面进行依赖收集。

Dep有一个类属性，`target`，用来存放Watcher订阅者，他是依赖收集的关键。

1. 当data的属性被get之后，会调用 `dep.depend()`。
2. 而在 `dep.depend` 函数中，如果存在 `Dep.target`，则会通知与之对应的Watcher添加依赖
3. 当data的属性key被set（也就是更新的时候），调用与之对应 `dep.notify()`，`notify`会调用所有订阅它的Watcher进行update更新

## 5. 什么是Observer？

当组件调用 `observe(data)` 的时候，创建Observer实例，他会将data利用 `Object.defineProperty` 重新定义然后挂载到组件上。

我们称之为“观察者”，因为他通过观察data然后将data转化成为一个数据响应式的data，有人开玩笑的说，从此data经过了“社会主义的洗礼”，已经是一个成熟的社会主义接班人。值得注意的是每一个Observer实例，也有一个唯一的Dep实例与之对应。

## 6. vue里面有多少种不同Watcher？

因为Watcher是用来更新的，所以我们可以想一下Vue里面有多少个场景需要进行数据更新。

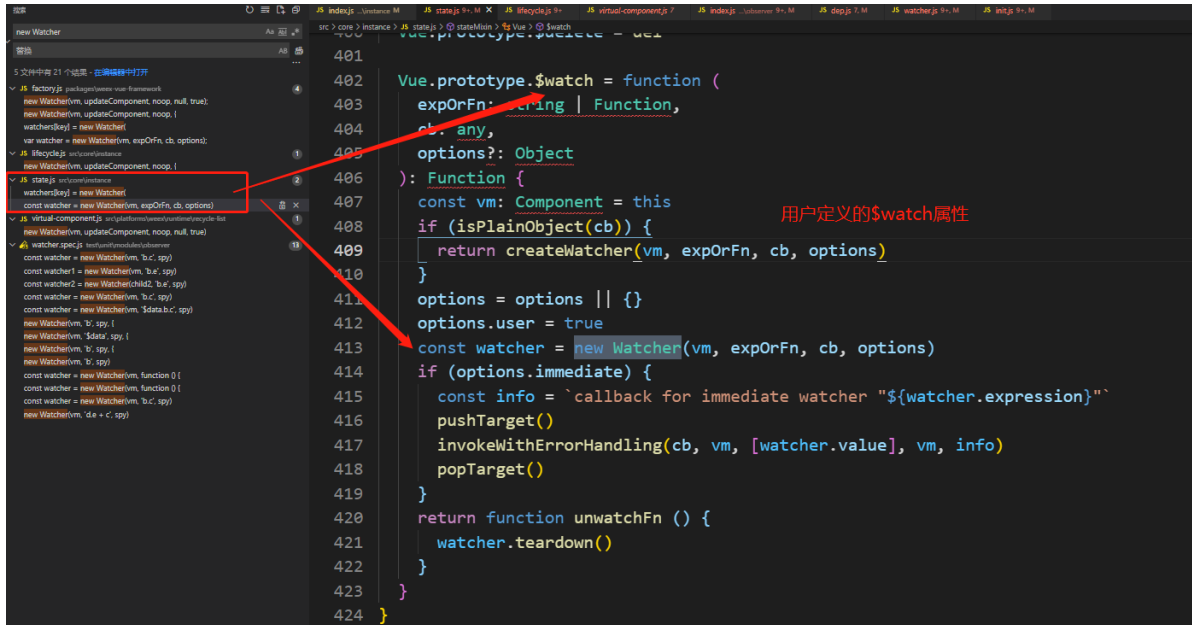
比如：

- 数据变 → 使用数据的视图变
- 数据变 → 使用数据的计算属性变 → 使用计算属性的视图变

- 数据变 → 开发者主动注册的watch回调函数执行

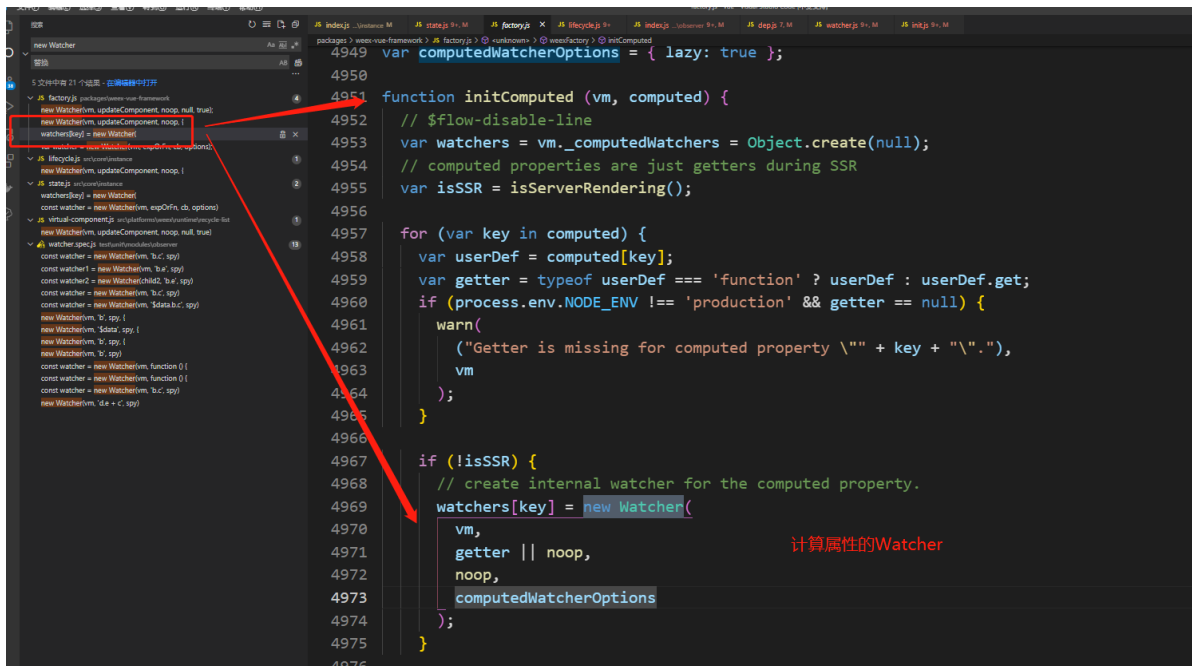
三个场景，对应三种watcher：

- 组件的Watcher,即render-watcher
- 用户定义的计算属性对应的computed-watcher
- 用户定义的监听属性对应的Watcher（watch-api或watch属性）



```
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424 }

Vue.prototype.$watch = function (
  expOrFn: string | Function,
  cb: any,
  options?: Object
): Function {
  const vm: Component = this
  if (isPlainObject(cb)) {
    return createWatcher(vm, expOrFn, cb, options)
  }
  options = options || {}
  options.user = true
  const watcher = new Watcher(vm, expOrFn, cb, options)
  if (options.immediate) {
    const info = `callback for immediate watcher "${watcher.expression}"`
    pushTarget()
    invokeWithErrorHandling(cb, vm, [watcher.value], vm, info)
    popTarget()
  }
  return function unwatchFn () {
    watcher.teardown()
  }
}
```

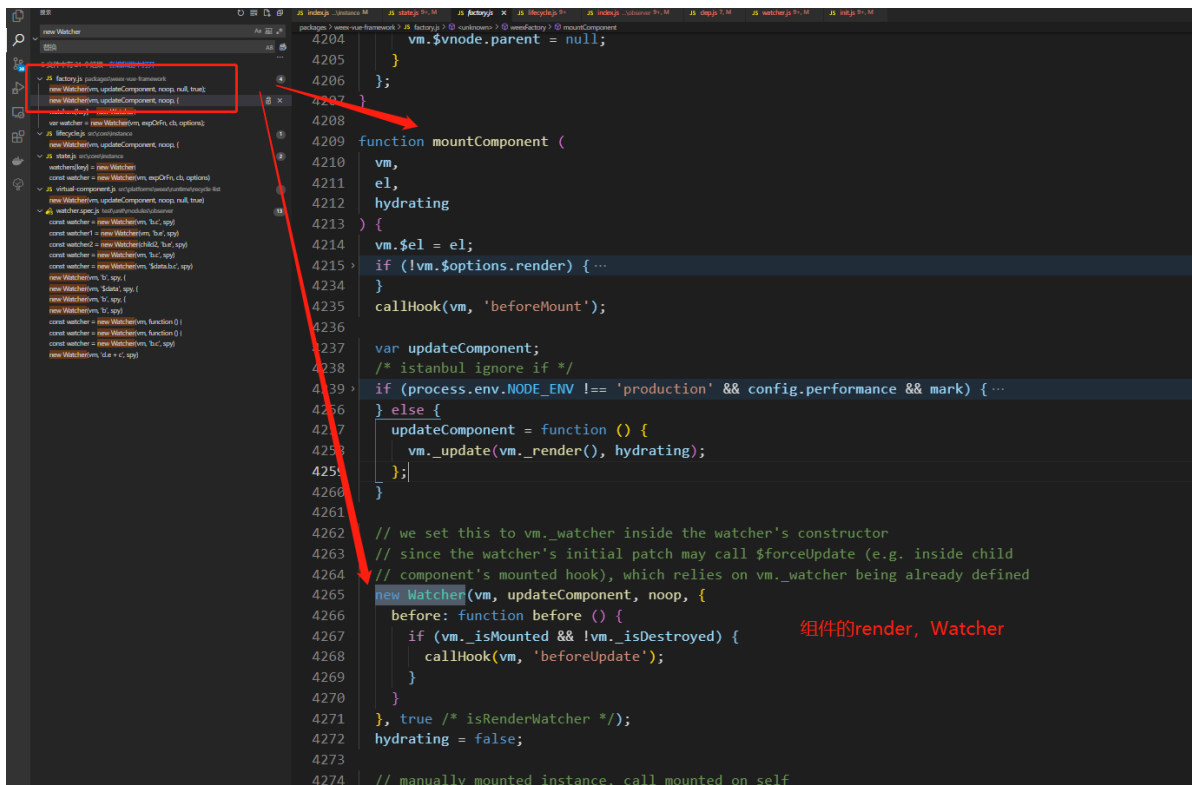


```
4949
4950
4951
4952
4953
4954
4955
4956
4957
4958
4959
4960
4961
4962
4963
4964
4965
4966
4967
4968
4969
4970
4971
4972
4973
4974
4975
4976 }

var computedWatcherOptions = { lazy: true };

function initComputed (vm, computed) {
  // $flow-disable-line
  var watchers = vm._computedWatchers = Object.create(null);
  // computed properties are just getters during SSR
  var isSSR = isServerRendering();

  for (var key in computed) {
    var userDef = computed[key];
    var getter = typeof userDef === 'function' ? userDef : userDef.get;
    if (process.env.NODE_ENV !== 'production' && getter == null) {
      warn(
        ("Getter is missing for computed property \"" + key + "\"."),
        vm
      );
    }
    if (!isSSR) {
      // create internal watcher for the computed property.
      watchers[key] = new Watcher(
        vm,
        getter || noop,
        noop,
        computedWatcherOptions
      );
    }
  }
}
```



## 面试题： 请谈谈你对数据响应式原理的理解

首先我们可以看一下  $c=a+b$ ，在这条等式里面， $c$ 是我们的目标数据， $a$ 和 $b$ 都是 $c$ 的依赖，当 $a$ 或者 $b$ 发生变化的时候， $c$ 会自动进行更新，就是我所理解的数据响应式。

而在Vue里面跟数据响应式有关的主要有Dep, Watcher,以及Observer三个类。

在我们创建组件的时候，vue会对用户预定义的数据进行observe创建一个Observer实例，将data对象利用 `Object.defineProperty()` 的getter以及setter进行重定义，使后面对data所做的所有操作，均可被组件察觉。

然后我们获取Watcher实例的时候，会先调用Watcher里面的get函数，这个get函数，会将当前触发的依赖push到targetStack里面，然后触发这个依赖也就是我们前面定义的 `obj.key` 的getter，在这个getter里面，如果Dep.target不为空则调用depend进行依赖收集。

当我们再次更改数据的时候，触发的obj.key.setter会调用与之对应Dep实例的notify函数，notify遍历订阅者队列，调用所有订阅了这个key依赖的Watcher的update函数进行数据更新。从而达到数据响应的目的。

请问面试官我的理解是否有错？

