

Vue 源码解析 - 响应式原理

课程目标

- Vue.js 的静态成员和实例成员初始化过程
- 首次渲染的过程
- 数据响应式原理

准备工作

Vue 源码的获取

- 项目地址: <https://github.com/vuejs/vue>
- Fork 一份到自己仓库, 克隆到本地, 可以自己写注释提交到 github
- 为什么分析 Vue 2.6
 - 到目前为止 Vue 3.0 的正式版还没有发布
 - 新版本发布后, 现有项目不会升级到 3.0, 2.x 还有很长的一段过渡期
 - 3.0 项目地址: <https://github.com/vuejs/vue-next>

源码目录结构

```
1 | src
2 |   ├── compiler      编译相关
3 |   ├── core          Vue 核心库
4 |   ├── platforms     平台相关代码
5 |   ├── server        SSR, 服务端渲染
6 |   ├── sfc           .vue 文件编译为 js 对象
7 |   └── shared        公共的代码
```

了解 Flow

- 官网: <https://flow.org/>
- JavaScript 的静态类型检查器
- Flow 的静态类型检查错误是通过静态类型推断实现的
 - 文件开头通过 `// @flow` 或者 `/* @flow */` 声明

```
1 | /* @flow */
2 | function square(n: number): number {
3 |   return n * n;
4 | }
5 | square("2"); // Error!
```

调试设置

打包

- 打包工具 Rollup

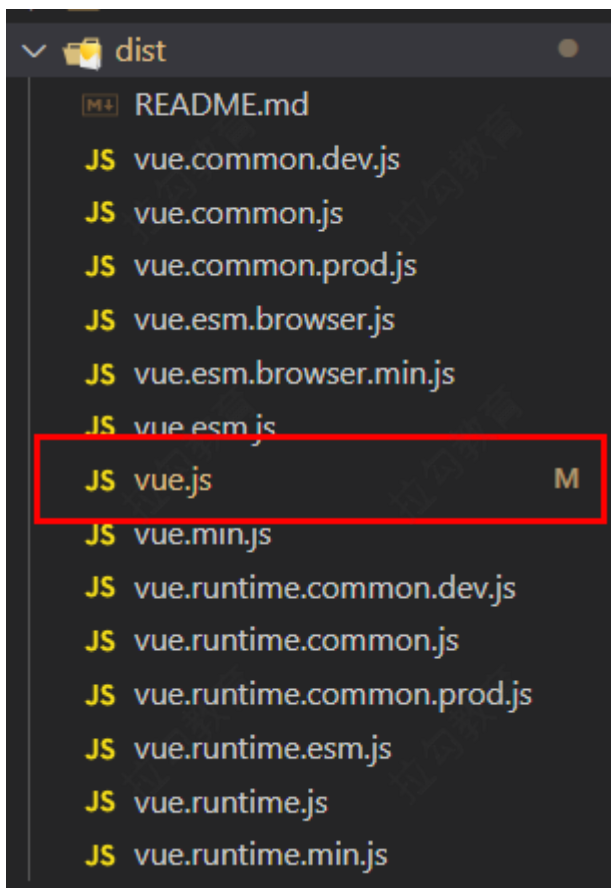
- Vue.js 源码的打包工具使用的是 Rollup, 比 Webpack 轻量
- Webpack 把所有文件当做模块, Rollup 只处理 js 文件更适合在 Vue.js 这样的库中使用
- Rollup 打包不会生成冗余的代码
- 安装依赖

```
1 | npm i
```

- 设置 sourcemap
 - package.json 文件中的 dev 脚本中添加参数 --sourcemap

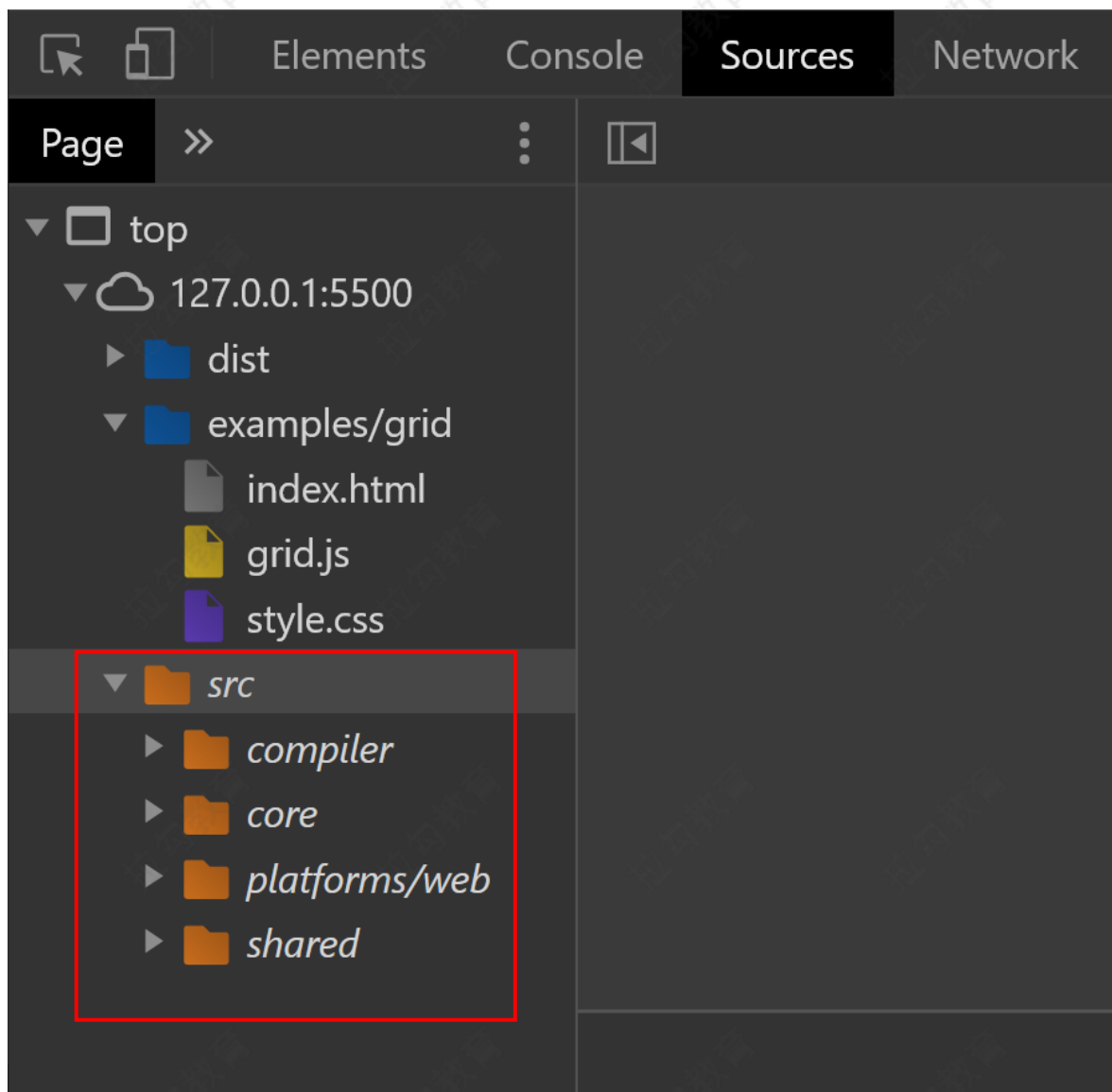
```
1 | "dev": "rollup -w -c scripts/config.js --sourcemap --environment TARGET:web-  
full-dev"
```

- 执行 dev
 - `npm run dev` 执行打包, 用的是 rollup, -w 参数是监听文件的变化, 文件变化自动重新打包
 - 结果:



调试

- examples 的示例中引入的 vue.min.js 改为 vue.js
- 打开 Chrome 的调试工具中的 source



Vue 的不同构建版本

- `npm run build` 重新打包所有文件
- [官方文档 - 对不同构建版本的解释](#)
- `dist\README.md`

	UMD	CommonJS	ES Module
Full	vue.js	vue.common.js	vue.esm.js
Runtime-only	vue.runtime.js	vue.runtime.common.js	vue.runtime.esm.js
Full (production)	vue.min.js		
Runtime-only (production)	vue.runtime.min.js		

术语

- **完整版**：同时包含**编译器**和**运行时的**版本。
- **编译器**：用来将模板字符串编译成为 JavaScript 渲染函数的代码，体积大、效率低。
- **运行时**：用来创建 Vue 实例、渲染并处理虚拟 DOM 等的代码，体积小、效率高。基本上就是除去编译器的代码。

- **UMD**: UMD 版本**通用的模块版本**，支持多种模块方式。vue.js 默认文件就是运行时 + 编译器的 UMD 版本
- **CommonJS(cjs)**: CommonJS 版本用来配合老的打包工具比如 [Browserify](#) 或 [webpack 1](#)。
- **ES Module**: 从 2.6 开始 Vue 会提供两个 ES Modules (ESM) 构建文件，为现代打包工具提供的版本。
 - ESM 格式被设计为可以被静态分析，所以打包工具可以利用这一点来进行“tree-shaking”并将用不到的代码排除出最终的包。
 - [ES6 模块与 CommonJS 模块的差异](#)

Runtime + Compiler vs. Runtime-only

```
1 // Compiler
2 // 需要编译器，把 template 转换成 render 函数
3 // const vm = new Vue({
4 //   el: '#app',
5 //   template: '<h1>{{ msg }}</h1>',
6 //   data: {
7 //     msg: 'Hello Vue'
8 //   }
9 // })
10 // Runtime
11 // 不需要编译器
12 const vm = new Vue({
13   el: '#app',
14   render (h) {
15     return h('h1', this.msg)
16   },
17   data: {
18     msg: 'Hello Vue'
19   }
20 })
```

- 推荐使用运行时版本，因为运行时版本相比完整版体积要小大约 30%
- 基于 Vue-CLI 创建的项目默认使用的是 `vue.runtime.esm.js`
 - 通过查看 webpack 的配置文件

```
1 | vue inspect > output.js
```

- **注意**: *.vue 文件中的模板是在构建时预编译的，最终打包后的结果不需要编译器，只需要运行时版本即可

寻找入口文件

- 查看 dist/vue.js 的构建过程

执行构建

```
1 npm run dev
2 # "dev": "rollup -w -c scripts/config.js --sourcemap --environment
  TARGET:web-full-dev"
3 # --environment TARGET:web-full-dev 设置环境变量 TARGET
```

- `script/config.js` 的执行过程
 - 作用：生成 rollup 构建的配置文件
 - 使用环境变量 `TARGET = web-full-dev`

```
1 // 判断环境变量是否有 TARGET
2 // 如果有的话 使用 genConfig() 生成 rollup 配置文件
3 if (process.env.TARGET) {
4   module.exports = genConfig(process.env.TARGET)
5 } else {
6   // 否则获取全部配置
7   exports.getBuild = genConfig
8   exports.getAllBuilds = () => Object.keys(builds).map(genConfig)
9 }
```

- `genConfig(name)`
 - 根据环境变量 `TARGET` 获取配置信息
 - `builds[name]` 获取生成配置的信息

```
1 // Runtime+compiler development build (Browser)
2 'web-full-dev': {
3   entry: resolve('web/entry-runtime-with-compiler.js'),
4   dest: resolve('dist/vue.js'),
5   format: 'umd',
6   env: 'development',
7   alias: { he: './entity-decoder' },
8   banner
9 },
```

- `resolve()`
 - 获取入口和出口文件的绝对路径

```
1 const aliases = require('./alias')
2 const resolve = p => {
3   // 根据路径中的前半部分去alias中找别名
4   const base = p.split('/')[0]
5   if (aliases[base]) {
6     return path.resolve(aliases[base], p.slice(base.length + 1))
7   } else {
8     return path.resolve(__dirname, '../', p)
9   }
10 }
```

结果

- 把 `src/platforms/web/entry-runtime-with-compiler.js` 构建成 `dist/vue.js`，如果设置 `--sourcemap` 会生成 `vue.js.map`
- `src/platform` 文件夹下是 Vue 可以构建成不同平台下使用的库，目前有 `weex` 和 `web`，还有服务端渲染的库

从入口开始

- `src/platform/web/entry-runtime-with-compiler.js`

通过查看源码解决下面问题

- 观察以下代码，通过阅读源码，回答在页面上输出的结果

```
1  const vm = new Vue({
2    el: '#app',
3    template: '<h3>Hello template</h3>',
4    render (h) {
5      return h('h4', 'Hello render')
6    }
7  })
```

- 阅读源码记录

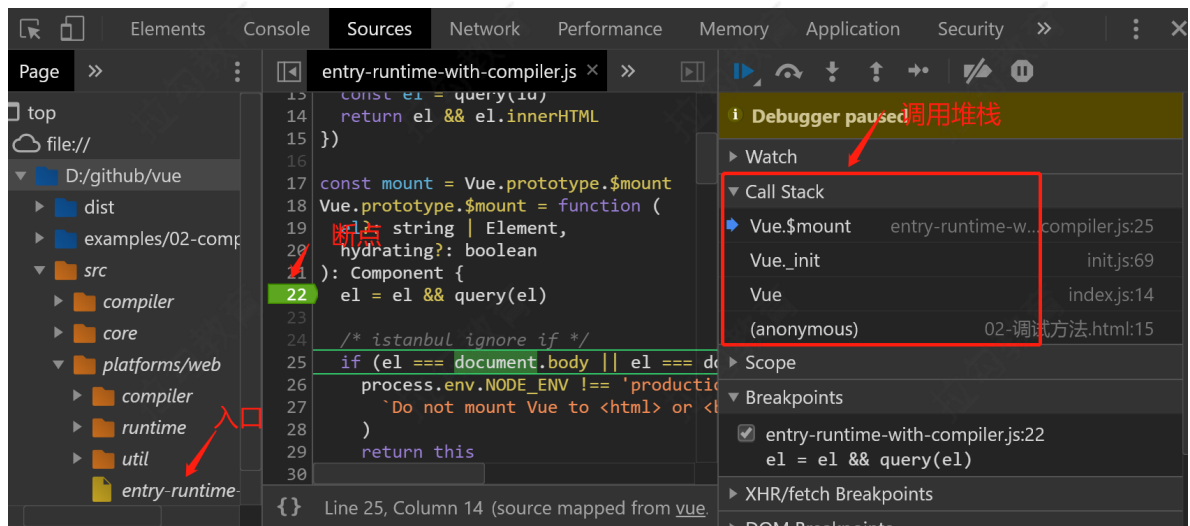
- el 不能是 body 或者 html 标签
- 如果没有 render，把 template 转换成 render 函数
- 如果有 render 方法，直接调用 mount 挂载 DOM

```
1  // 1. el 不能是 body 或者 html
2  if (el === document.body || el === document.documentElement) {
3    process.env.NODE_ENV !== 'production' && warn(
4      `Do not mount Vue to <html> or <body> - mount to normal elements
5      instead.`
6    )
7    return this
8  }
9  const options = this.$options
10 if (!options.render) {
11   // 2. 把 template/el 转换成 render 函数
12   .....
13 }
14 // 3. 调用 mount 方法，挂载 DOM
15 return mount.call(this, el, hydrating)
```

- 调试代码

- 调试的方法

```
1  const vm = new Vue({
2    el: '#app',
3    template: '<h3>Hello template</h3>',
4    render (h) {
5      return h('h4', 'Hello render')
6    }
7  })
```



Vue 的构造函数在哪？

Vue 实例的成员/Vue 的静态成员从哪里来的？

Vue 的构造函数在哪里

- src/platform/web/entry-runtime-with-compiler.js 中引用了 './runtime/index'
- src/platform/web/runtime/index.js
 - 设置 Vue.config
 - 设置平台相关的指令和组件
 - 指令 v-model、v-show
 - 组件 transition、transition-group
 - 设置平台相关的 __patch__ 方法（打补丁方法，对比新旧的 VNode）
 - **设置 \$mount 方法，挂载 DOM**

```
1 // install platform runtime directives & components
2 extend(Vue.options.directives, platformDirectives)
3 extend(Vue.options.components, platformComponents)
4
5 // install platform patch function
6 Vue.prototype.__patch__ = inBrowser ? patch : noop
7
8 // public mount method
9 Vue.prototype.$mount = function (
10   el?: string | Element,
11   hydrating?: boolean
12 ): Component {
13   el = el && inBrowser ? query(el) : undefined
14   return mountComponent(this, el, hydrating)
15 }
```

- src/platform/web/runtime/index.js 中引用了 'core/index'
- src/core/index.js
 - 定义了 Vue 的静态方法
 - initGlobalAPI(Vue)
- src/core/index.js 中引用了 './instance/index'
- src/core/instance/index.js
 - 定义了 Vue 的构造函数

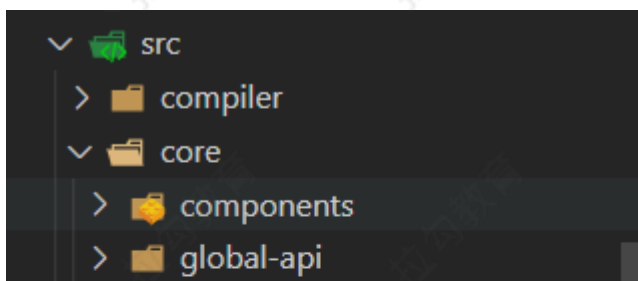

```

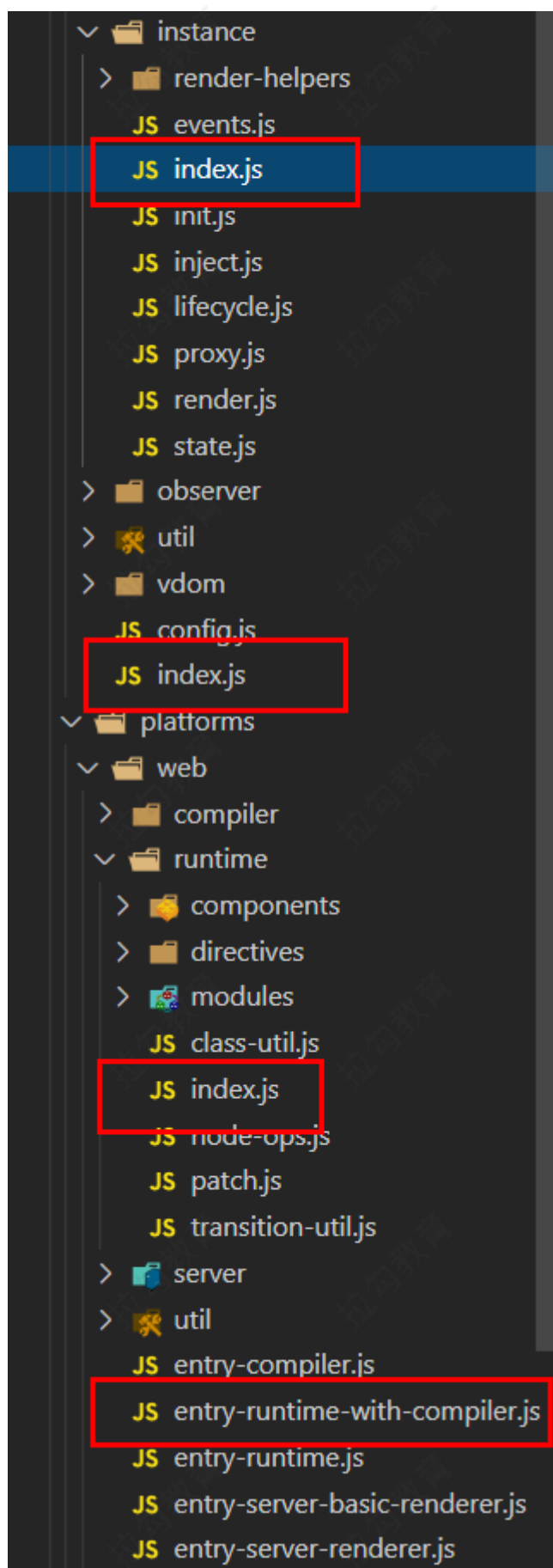
1 function Vue (options) {
2   if (process.env.NODE_ENV !== 'production' &&
3     !(this instanceof Vue)
4   ) {
5     warn('Vue is a constructor and should be called with the `new`
keyword')
6   }
7   // 调用 _init() 方法
8   this._init(options)
9 }
10 // 注册 vm 的 _init() 方法, 初始化 vm
11 initMixin(Vue)
12 // 注册 vm 的 $data/$props/$set/$delete/$watch
13 stateMixin(Vue)
14 // 初始化事件相关方法
15 // $on/$once/$off/$emit
16 eventsMixin(Vue)
17 // 初始化生命周期相关的混入方法
18 // _update/$forceUpdate/$destroy
19 lifecycleMixin(Vue)
20 // 混入 render
21 // $nextTick/_render
22 renderMixin(Vue)

```

四个导出 Vue 的模块

- src/**platforms/web**/entry-runtime-with-compiler.js
 - web 平台相关的入口
 - 重写了平台相关的 \$mount() 方法
 - 注册了 Vue.compile() 方法, 传递一个 HTML 字符串返回 render 函数
- src/**platforms/web**/runtime/index.js
 - web 平台相关
 - 注册和平台相关的全局指令: v-model、v-show
 - 注册和平台相关的全局组件: v-transition、v-transition-group
 - 全局方法:
 - __patch__: 把虚拟 DOM 转换成真实 DOM
 - \$mount: 挂载方法
- src/**core**/index.js
 - 与平台无关
 - 设置了 Vue 的静态方法, initGlobalAPI(Vue)
- src/**core**/instance/index.js
 - 与平台无关
 - 定义了构造函数, 调用了 this._init(options) 方法
 - 给 Vue 中混入了常用的实例成员





Vue 的初始化

src/core/global-api/index.js

- 初始化 Vue 的静态方法

```

1 // 注册 vue 的静态属性/方法
2 initGlobalAPI(Vue)
3
4 // src/core/global-api/index.js
5 // 初始化 Vue.config 对象
6 Object.defineProperty(Vue, 'config', configDef)
7
8 // exposed util methods.
9 // NOTE: these are not considered part of the public API - avoid relying on
10 // them unless you are aware of the risk.
11 // 这些工具方法不视作全局API的一部分，除非你已经意识到某些风险，否则不要去依赖他们
12 Vue.util = {
13   warn,
14   extend,
15   mergeOptions,
16   defineReactive
17 }
18 // 静态方法 set/delete/nextTick
19 Vue.set = set
20 Vue.delete = del
21 Vue.nextTick = nextTick
22
23 // 2.6 explicit observable API
24 // 让一个对象可响应
25 Vue.observable = <T>(obj: T): T => {
26   observe(obj)
27   return obj
28 }
29 // 初始化 Vue.options 对象，并给其扩展
30 // components/directives/filters/_base
31 Vue.options = Object.create(null)
32 ASSET_TYPES.forEach(type => {
33   Vue.options[type + 's'] = Object.create(null)
34 })
35
36 // this is used to identify the "base" constructor to extend all plain-
37 // object
38 // components with in weex's multi-instance scenarios.
39 Vue.options._base = Vue
40
41 // 设置 keep-alive 组件
42 extend(Vue.options.components, builtInComponents)
43
44 // 注册 vue.use() 用来注册插件
45 initUse(Vue)
46 // 注册 vue.mixin() 实现混入
47 initMixin(Vue)
48 // 注册 vue.extend() 基于传入的 options 返回一个组件的构造函数
49 initExtend(Vue)
50 // 注册 vue.directive()、vue.component()、vue.filter()
51 initAssetRegisters(Vue)

```

src/core/instance/index.js

- 定义 Vue 的构造函数
- 初始化 Vue 的实例成员

```

1 // 此处不用 class 的原因是因为方便，后续给 Vue 实例混入实例成员
2 function Vue (options) {
3   if (process.env.NODE_ENV !== 'production' &&
4     !(this instanceof Vue)
5   ) {
6     warn('Vue is a constructor and should be called with the `new`
keyword')
7   }
8   this._init(options)
9 }
10 // 注册 vm 的 _init() 方法，初始化 vm
11 initMixin(Vue)
12 // 注册 vm 的 $data/$props/$set/$delete/$watch
13 stateMixin(Vue)
14 // 初始化事件相关方法
15 // $on/$once/$off/$emit
16 eventsMixin(Vue)
17 // 初始化生命周期相关的混入方法
18 // _update/$forceUpdate/$destroy
19 lifecycleMixin(Vue)
20 // 混入 render
21 // $nextTick/_render
22 renderMixin(Vue)

```

- initMixin(Vue)
 - 初始化 _init() 方法

```

1 // src\core\instance\init.js
2 export function initMixin (Vue: Class<Component>) {
3   // 给 vue 实例增加 _init() 方法
4   // 合并 options / 初始化操作
5   Vue.prototype._init = function (options?: Object) {
6     // a flag to avoid this being observed
7     // 如果是 vue 实例不需要被 observe
8     vm._isVue = true
9     // merge options
10    // 合并 options
11    if (options && options._isComponent) {
12      // optimize internal component instantiation
13      // since dynamic options merging is pretty slow, and none of the
14      // internal component options needs special treatment.
15      initInternalComponent(vm, options)
16    } else {
17      vm.$options = mergeOptions(
18        resolveConstructorOptions(vm.constructor),
19        options || {},
20        vm
21      )
22    }
23    /* istanbul ignore else */
24    if (process.env.NODE_ENV !== 'production') {
25      initProxy(vm)
26    } else {
27      vm._renderProxy = vm
28    }
29    // expose real self

```

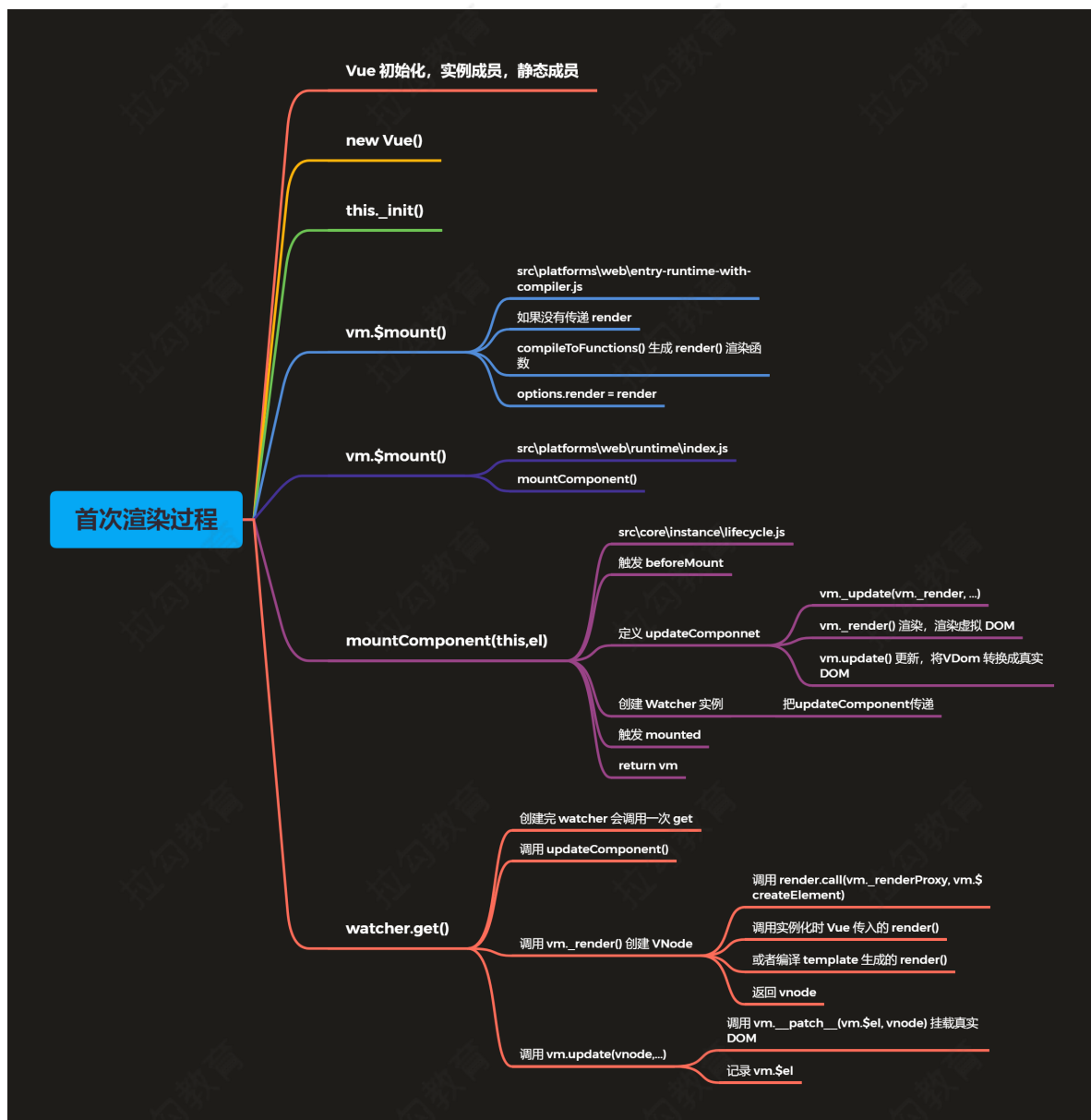
```

30     vm._self = vm
31     // vm 的生命周期相关变量初始化
32     // $children/$parent/$root/$refs
33     initLifecycle(vm)
34     // vm 的事件监听初始化，父组件绑定在当前组件上的事件
35     initEvents(vm)
36     // vm 的编译render初始化
37     // $slots/$scopedSlots/_c/$createElement/$attrs/$listeners
38     initRender(vm)
39     // beforeCreate 生命钩子的回调
40     callHook(vm, 'beforeCreate')
41     // 把 inject 的成员注入到 vm 上
42     initInjections(vm) // resolve injections before data/props
43     // 初始化状态 vm 的 _props/methods/_data/computed/watch
44     initState(vm)
45     // 初始化 provide
46     initProvide(vm) // resolve provide after data/props
47     // created 生命钩子的回调
48     callHook(vm, 'created')
49
50     /* istanbul ignore if */
51     if (process.env.NODE_ENV !== 'production' && config.performance &&
mark) {
52         vm._name = formatComponentName(vm, false)
53         mark(endTag)
54         measure(`vue ${vm._name} init`, startTag, endTag)
55     }
56     // 如果没有提供 el, 调用 $mount() 挂载
57     if (vm.$options.el) {
58         vm.$mount(vm.$options.el)
59     }
60 }
61 }

```

首次渲染过程

- Vue 初始化完毕，开始真正的执行
- 调用 new Vue() 之前，已经初始化完毕
- 通过调试代码，记录首次渲染过程



数据响应式原理

通过查看源码解决下面问题

- `vm.msg = { count: 0 }`, 重新给属性赋值, 是否是响应式的?
- `vm.arr[0] = 4`, 给数组元素赋值, 视图是否会更新
- `vm.arr.length = 0`, 修改数组的 length, 视图是否会更新
- `vm.arr.push(4)`, 视图是否会更新

响应式处理的入口

整个响应式处理的过程是比较复杂的, 下面我们先从

- `src\core\instance\init.js`
 - `initState(vm)` vm 状态的初始化
 - 初始化了 `_data`、`_props`、`methods` 等
- `src\core\instance\state.js`

```

1 // 数据的初始化
2 if (opts.data) {
3   initData(vm)
4 } else {
5   observe(vm._data = {}, true /* asRootData */)
6 }

```

- initData(vm) vm 数据的初始化

```

1 function initData (vm: Component) {
2   let data = vm.$options.data
3   // 初始化 _data, 组件中 data 是函数, 调用函数返回结果
4   // 否则直接返回 data
5   data = vm._data = typeof data === 'function'
6     ? getData(data, vm)
7     : data || {}
8   .....
9
10  // proxy data on instance
11  // 获取 data 中的所有属性
12  const keys = Object.keys(data)
13  // 获取 props / methods
14  const props = vm.$options.props
15  const methods = vm.$options.methods
16  let i = keys.length
17  // 判断 data 上的成员是否和 props/methods 重名
18  .....
19
20  // observe data
21  // 数据的响应式处理
22  observe(data, true /* asRootData */)
23 }

```

- src\core\observer\index.js
 - observe(value, asRootData)
 - 负责为每一个 Object 类型的 value 创建一个 observer 实例

```

1 export function observe (value: any, asRootData: ?boolean): Observer | void
2 {
3   // 判断 value 是否是对象
4   if (!isObject(value) || value instanceof VNode) {
5     return
6   }
7   let ob: Observer | void
8   // 如果 value 有 __ob__(observer对象) 属性 结束
9   if (hasOwn(value, '__ob__') && value.__ob__ instanceof Observer) {
10     ob = value.__ob__
11   } else if (
12     shouldObserve &&
13     !isServerRendering() &&
14     (Array.isArray(value) || isPlainObject(value)) &&
15     Object.isExtensible(value) &&
16     !value._isVue
17   ) {
18     // 创建一个 Observer 对象
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

```

18     ob = new Observer(value)
19   }
20   if (asRootData && ob) {
21     ob.vmCount++
22   }
23   return ob
24 }

```

Observer

- src\core\observer\index.js
 - 对对象做响应化处理
 - 对数组做响应化处理

```

1  export class Observer {
2    // 观测对象
3    value: any;
4    // 依赖对象
5    dep: Dep;
6    // 实例计数器
7    vmCount: number; // number of vms that have this object as root $data
8
9    constructor (value: any) {
10     this.value = value
11     this.dep = new Dep()
12     // 初始化实例的 vmCount 为0
13     this.vmCount = 0
14     // 将实例挂载到观测对象的 __ob__ 属性，设置为不可枚举
15     def(value, '__ob__', this)
16     if (Array.isArray(value)) {
17       // 数组的响应式处理
18       if (hasProto) {
19         protoAugment(value, arrayMethods)
20       } else {
21         copyAugment(value, arrayMethods, arrayKeys)
22       }
23       // 为数组中的每一个对象创建一个 observer 实例
24       this.observeArray(value)
25     } else {
26       // 对象的响应化处理
27       // 遍历对象中的每一个属性，转换成 setter/getter
28       this.walk(value)
29     }
30   }
31
32   /**
33    * walk through all properties and convert them into
34    * getter/setters. This method should only be called when
35    * value type is object.
36    */
37   walk (obj: Object) {
38     // 获取观察对象的每一个属性
39     const keys = Object.keys(obj)
40     // 遍历每一个属性，设置为响应式数据
41     for (let i = 0; i < keys.length; i++) {
42       defineReactive(obj, keys[i])

```



```

43     }
44   }
45
46   /**
47    * Observe a list of Array items.
48    */
49   observeArray (items: Array<any>) {
50     for (let i = 0, l = items.length; i < l; i++) {
51       observe(items[i])
52     }
53   }
54 }

```

- walk(obj)
 - 遍历 obj 的所有属性，为每一个属性调用 defineReactive() 方法，设置 getter/setter

defineReactive()

- src\core\observer\index.js
- defineReactive(obj, key, val, customSetter, shallow)
 - 为一个对象定义一个响应式的属性，每一个属性对应一个 dep 对象
 - 如果该属性的值是对象，继续调用 observe
 - 如果给属性赋新值，继续调用 observe
 - 如果数据更新发送通知

对象响应式处理

```

1  // 为一个对象定义一个响应式的属性
2  /**
3   * Define a reactive property on an Object.
4   */
5  export function defineReactive (
6    obj: Object,
7    key: string,
8    val: any,
9    customSetter?: ?Function,
10   shallow?: boolean
11 ) {
12   // 1. 为每一个属性，创建依赖对象实例
13   const dep = new Dep()
14   // 获取 obj 的属性描述符对象
15   const property = Object.getOwnPropertyDescriptor(obj, key)
16   if (property && property.configurable === false) {
17     return
18   }
19   // 提供预定义的存取器函数
20   // cater for pre-defined getter/setters
21   const getter = property && property.get
22   const setter = property && property.set
23   if ((!getter || setter) && arguments.length === 2) {
24     val = obj[key]
25   }
26   // 2. 判断是否递归观察子对象，并将子对象属性都转换成 getter/setter，返回子观察对象
27   let childob = !shallow && observe(val)
28   Object.defineProperty(obj, key, {

```

```

29     enumerable: true,
30     configurable: true,
31     get: function reactiveGetter () {
32         // 如果预定义的 getter 存在则 value 等于getter 调用的返回值
33         // 否则直接赋予属性值
34         const value = getter ? getter.call(obj) : val
35         // 如果存在当前依赖目标, 即 watcher 对象, 则建立依赖
36         if (Dep.target) {
37             // dep() 添加相互的依赖
38             // 1个组件对应一个 watcher 对象
39             // 1个watcher会对应多个dep (要观察的属性很多)
40             // 我们可以手动创建多个 watcher 监听1个属性的变化, 1个dep可以对应多个watcher
41             dep.depend()
42             // 如果子观察目标存在, 建立子对象的依赖关系, 将来 vue.set() 会用到
43             if (childOb) {
44                 childOb.dep.depend()
45                 // 如果属性是数组, 则特殊处理收集数组对象依赖
46                 if (Array.isArray(value)) {
47                     dependArray(value)
48                 }
49             }
50         }
51         // 返回属性值
52         return value
53     },
54     set: function reactiveSetter (newVal) {
55         // 如果预定义的 getter 存在则 value 等于getter 调用的返回值
56         // 否则直接赋予属性值
57         const value = getter ? getter.call(obj) : val
58         // 如果新值等于旧值或者新值旧值为null则不执行
59         /* eslint-disable no-self-compare */
60         if (newVal === value || (newVal !== newVal && value !== value)) {
61             return
62         }
63         /* eslint-enable no-self-compare */
64         if (process.env.NODE_ENV !== 'production' && customSetter) {
65             customSetter()
66         }
67         // 如果没有 setter 直接返回
68         // #7981: for accessor properties without setter
69         if (getter && !setter) return
70         // 如果预定义setter存在则调用, 否则直接更新新值
71         if (setter) {
72             setter.call(obj, newVal)
73         } else {
74             val = newVal
75         }
76         // 3. 如果新值是对象, 观察子对象并返回 子的 observer 对象
77         childOb = !shallow && observe(newVal)
78         // 4. 发布更改通知
79         dep.notify()
80     }
81 })
82 }

```

数组的响应式处理

- Observer 的构造函数中

```
1 // 数组的响应式处理
2 if (Array.isArray(value)) {
3   if (hasProto) {
4     protoAugment(value, arrayMethods)
5   } else {
6     copyAugment(value, arrayMethods, arrayKeys)
7   }
8   // 为数组中的每一个对象创建一个 observer 实例
9   this.observeArray(value)
10 } else {
11   // 编译对象中的每一个属性, 转换成 setter/getter
12   this.walk(value)
13 }
14 function protoAugment (target, src: Object) {
15   /* eslint-disable no-proto */
16   target.__proto__ = src
17   /* eslint-enable no-proto */
18 }
19 /* istanbul ignore next */
20 function copyAugment (target: Object, src: Object, keys: Array<string>) {
21   for (let i = 0, l = keys.length; i < l; i++) {
22     const key = keys[i]
23     def(target, key, src[key])
24   }
25 }
```

- 处理数组修改数据的方法

- src\core\observer\array.js

```
1 const arrayProto = Array.prototype
2 // 克隆数组的原型
3 export const arrayMethods = Object.create(arrayProto)
4 // 修改数组元素的方法
5 const methodsToPatch = [
6   'push',
7   'pop',
8   'shift',
9   'unshift',
10  'splice',
11  'sort',
12  'reverse'
13 ]
14
15 /**
16  * Intercept mutating methods and emit events
17  */
18 methodsToPatch.forEach(function (method) {
19   // cache original method
20   // 保存数组原方法
21   const original = arrayProto[method]
22   // 调用 Object.defineProperty() 重新定义修改数组的方法
23   def(arrayMethods, method, function mutator (...args) {
24     // 执行数组的原始方法
25     const result = original.apply(this, args)
```

```

26 // 获取数组对象的 ob 对象
27 const ob = this.__ob__
28 let inserted
29 switch (method) {
30   case 'push':
31   case 'unshift':
32     inserted = args
33     break
34   case 'splice':
35     inserted = args.slice(2)
36     break
37 }
38 // 对插入的新元素，重新遍历数组元素设置为响应式数据
39 if (inserted) ob.observeArray(inserted)
40 // notify change
41 // 调用了修改数组的方法，调用数组的ob对象发送通知
42 ob.dep.notify()
43 return result
44 })
45 })

```

Dep 类

- src\core\observer\dep.js
- 依赖对象
- 记录 watcher 对象
- depend() -- watcher 记录对应的 dep
- 发布通知

1. 在 `defineReactive()` 的 `getter` 中创建 `dep` 对象，并判断 `Dep.target` 是否有值（一会再来看有什么时候有值得），调用 `dep.depend()`
2. `dep.depend()` 内部调用 `Dep.target.addDep(this)`，也就是 `watcher` 的 `addDep()` 方法，它内部最调用 `dep.addSub(this)`，把 `watcher` 对象，添加到 `dep.subs.push(watcher)` 中，也就是把订阅者添加到 `dep` 的 `subs` 数组中，当数据变化的时候调用 `watcher` 对象的 `update()` 方法
3. 什么时候设置的 `Dep.target`? 通过简单的案例调试观察。调用 `mountComponent()` 方法的时候，创建了渲染 `watcher` 对象，执行 `watcher` 中的 `get()` 方法
4. `get()` 方法内部调用 `pushTarget(this)`，把当前 `Dep.target = watcher`，同时把当前 `watcher` 入栈，
因为有父子组件嵌套的时候先把父组件对应的 `watcher` 入栈，再去处理子组件的 `watcher`，子组件的处理完毕后，再把父组件对应的 `watcher` 出栈，继续操作
5. `Dep.target` 用来存放目前正在使用的 `watcher`。全局唯一，并且一次也只能有一个 `watcher` 被使用

```

1 // dep 是个可观察对象，可以有多个指令订阅它
2 /**
3  * A dep is an observable that can have multiple
4  * directives subscribing to it.
5  */
6 export default class Dep {
7   // 静态属性，watcher 对象
8   static target: ?Watcher;

```

```

9 // dep 实例 Id
10 id: number;
11 // dep 实例对应的 watcher 对象/订阅者数组
12 subs: Array<watcher>;
13
14 constructor () {
15     this.id = uid++;
16     this.subs = []
17 }
18
19 // 添加新的订阅者 watcher 对象
20 addSub (sub: watcher) {
21     this.subs.push(sub)
22 }
23
24 // 移除订阅者
25 removeSub (sub: watcher) {
26     remove(this.subs, sub)
27 }
28
29 // 将观察对象和 watcher 建立依赖
30 depend () {
31     if (Dep.target) {
32         // 如果 target 存在, 把 dep 对象添加到 watcher 的依赖中
33         Dep.target.addDep(this)
34     }
35 }
36
37 // 发布通知
38 notify () {
39     // stabilize the subscriber list first
40     const subs = this.subs.slice()
41     if (process.env.NODE_ENV !== 'production' && !config.async) {
42         // subs aren't sorted in scheduler if not running async
43         // we need to sort them now to make sure they fire in correct
44         // order
45         subs.sort((a, b) => a.id - b.id)
46     }
47     // 调用每个订阅者的update方法实现更新
48     for (let i = 0, l = subs.length; i < l; i++) {
49         subs[i].update()
50     }
51 }
52 }
53 // Dep.target 用来存放目前正在使用的watcher
54 // 全局唯一, 并且一次也只能有一个watcher被使用
55 // The current target watcher being evaluated.
56 // This is globally unique because only one watcher
57 // can be evaluated at a time.
58 Dep.target = null
59 const targetStack = []
60 // 入栈并将当前 watcher 赋值给Dep.target
61 export function pushTarget (target: ?watcher) {
62     targetStack.push(target)
63     Dep.target = target
64 }
65
66 export function popTarget () {

```

```

67 // 出栈操作
68 targetStack.pop()
69 Dep.target = targetStack[targetStack.length - 1]
70 }

```

Watcher 类

- Watcher 分为三种，Computed Watcher、用户 Watcher (侦听器)、**渲染 Watcher**
- 渲染 Watcher 的创建时机
 - /src/core/instance/lifecycle.js

```

1  export function mountComponent (
2    vm: Component,
3    el: ?Element,
4    hydrating?: boolean
5  ): Component {
6    vm.$el = el
7    .....
8    callHook(vm, 'beforeMount')
9
10   let updateComponent
11   /* istanbul ignore if */
12   if (process.env.NODE_ENV !== 'production' && config.performance && mark)
13   {
14     .....
15   } else {
16     updateComponent = () => {
17       vm._update(vm._render(), hydrating)
18     }
19   }
20   // 创建渲染 watcher, expOrFn 为 updateComponent
21   // we set this to vm._watcher inside the watcher's constructor
22   // since the watcher's initial patch may call $forceUpdate (e.g. inside
23   // child
24   // component's mounted hook), which relies on vm._watcher being already
25   // defined
26   new Watcher(vm, updateComponent, noop, {
27     before () {
28       if (vm._isMounted && !vm._isDestroyed) {
29         callHook(vm, 'beforeUpdate')
30       }
31     }
32   }, true /* isRenderWatcher */)
33   hydrating = false
34
35   // manually mounted instance, call mounted on self
36   // mounted is called for render-created child components in its inserted
37   // hook
38   if (vm.$vnode == null) {
39     vm._isMounted = true
40     callHook(vm, 'mounted')
41   }
42   return vm
43 }

```


- 渲染 watcher 创建的位置 lifecycle.js 的 mountComponent 函数中
- Watcher 的构造函数初始化，处理 expOrFn（渲染 watcher 和侦听器处理不同）
- 调用 this.get()，它里面调用 pushTarget() 然后 this.getter.call(vm, vm)（对于渲染 watcher 调用 updateComponent），如果是用户 watcher 会获取属性的值（触发 get 操作）
- 当数据更新的时候，dep 中调用 notify() 方法，notify() 中调用 watcher 的 update() 方法
- update() 中调用 queueWatcher()
- queueWatcher() 是一个核心方法，去除重复操作，调用 flushSchedulerQueue() 刷新队列并执行 watcher
- flushSchedulerQueue() 中对 watcher 排序，遍历所有 watcher，如果有 before，触发生命周期的钩子函数 beforeUpdate，执行 watcher.run()，它内部调用 this.get()，然后调用 this.cb()（渲染 watcher 的 cb 是 noop）
- 整个流程结束

调试响应式数据执行过程

- 数组响应式处理的核心过程和数组收集依赖的过程
- 当数组的数据改变的时候 watcher 的执行过程

```

1  <div id="app">
2    {{ arr }}
3  </div>
4
5  <script src="../../dist/vue.js"></script>
6  <script>
7    const vm = new Vue({
8      el: '#app',
9      data: {
10        arr: [2, 3, 5]
11      }
12    })
13  </script>

```

回答以下问题

- [检测变化的注意事项](#)

```

1  methods: {
2    handler () {
3      this.obj.count = 555
4      this.arr[0] = 1
5      this.arr.length = 0
6      this.arr.push(4)
7    }
8  }

```

- 转换成响应式数据


```
1 methods: {  
2   handler () {  
3     this.$set(this.obj, 'count', 555)  
4     this.$set(this.arr, 0, 1)  
5     this.arr.splice(0)  
6   }  
7 }
```

实例方法/数据

vm.\$set

- 功能

向响应式对象中添加一个属性，并确保这个新属性同样是响应式的，且触发视图更新。它必须用于向响应式对象上添加新属性，因为 Vue 无法探测普通的新增属性 (比如 `this.myObject.newProperty = 'hi'`)

注意：对象不能是 Vue 实例，或者 Vue 实例的根数据对象。

- 示例

```
1 vm.$set(obj, 'foo', 'test')
```

定义位置

- Vue.set()
 - global-api/index.js

```
1 // 静态方法 set/delete/nextTick  
2 Vue.set = set  
3 Vue.delete = del  
4 Vue.nextTick = nextTick
```

- vm.\$set()
 - instance/index.js

```
1 // 注册 vm 的 $data/$props/$set/$delete/$watch  
2 // instance/state.js  
3 stateMixin(Vue)  
4  
5 // instance/state.js  
6 Vue.prototype.$set = set  
7 Vue.prototype.$delete = del
```

源码

- set() 方法
 - observer/index.js

```

1  /**
2   * Set a property on an object. Adds the new property and
3   * triggers change notification if the property doesn't
4   * already exist.
5   */
6  export function set (target: Array<any> | Object, key: any, val: any): any
7  {
8    if (process.env.NODE_ENV !== 'production' &&
9      (isUndef(target) || isPrimitive(target))
10     ) {
11      warn(`Cannot set reactive property on undefined, null, or primitive
12      value: ${target: any}`)
13    }
14    // 判断 target 是否是对象, key 是否是合法的索引
15    if (Array.isArray(target) && isValidArrayIndex(key)) {
16      target.length = Math.max(target.length, key)
17      // 通过 splice 对key位置的元素进行替换
18      // splice 在 array.js进行了响应化的处理
19      target.splice(key, 1, val)
20      return val
21    }
22    // 如果 key 在对象中已经存在直接赋值
23    if (key in target && !(key in Object.prototype)) {
24      target[key] = val
25      return val
26    }
27    // 获取 target 中的 observer 对象
28    const ob = (target: any).__ob__
29    // 如果 target 是 vue 实例或者$data 直接返回
30    if (target._isVue || (ob && ob.vmCount)) {
31      process.env.NODE_ENV !== 'production' && warn(
32        'Avoid adding reactive properties to a Vue instance or its root $data
33        ' +
34        'at runtime - declare it upfront in the data option.'
35      )
36      return val
37    }
38    // 如果 ob 不存在, target 不是响应式对象直接赋值
39    if (!ob) {
40      target[key] = val
41      return val
42    }
43    // 把 key 设置为响应式属性
44    defineReactive(ob.value, key, val)
45    // 发送通知
46    ob.dep.notify()
47    return val
48  }

```

调试

```

1  <div id="app">
2    {{ obj.msg }}
3    <br>
4    {{ obj.foo }}
5  </div>
6

```

```

7   <script src="../../dist/vue.js"></script>
8   <script>
9     const vm = new Vue({
10       el: '#app',
11       data: {
12         obj: {
13           msg: 'hello set'
14         }
15       }
16     })
17     // 非响应式数据
18     // vm.obj.foo = 'test'
19     vm.$set(vm.obj, 'foo', 'test')
20   </script>

```

回顾 `defineReactive` 中的 `childOb`，给每一个响应式对象设置一个 **ob**。调用 `$set` 的时候，会获取 `ob` 对象，并通过 `ob.dep.notify()` 发送通知。

vm.\$delete

- 功能

删除对象的属性。如果对象是响应式的，确保删除能触发更新视图。这个方法主要用于避开 Vue 不能检测到属性被删除的限制，但是你应该很少会使用它。

注意：目标对象不能是一个 Vue 实例或 Vue 实例的根数据对象。

- 示例

```
1 | vm.$delete(vm.obj, 'msg')
```

定义位置

- `Vue.delete()`
 - `global-api/index.js`

```

1 | // 静态方法 set/delete/nextTick
2 | Vue.set = set
3 | Vue.delete = del
4 | Vue.nextTick = nextTick

```

- `vm.$delete()`
 - `instance/index.js`

```

1 // 注册 vm 的 $data/$props/$set/$delete/$watch
2 stateMixin(vue)
3
4 // instance/state.js
5 Vue.prototype.$set = set
6 Vue.prototype.$delete = del

```

源码

- src\core\observer\index.js

```

1 /**
2  * Delete a property and trigger change if necessary.
3  */
4 export function del (target: Array<any> | Object, key: any) {
5   if (process.env.NODE_ENV !== 'production' &&
6     (isUndef(target) || isPrimitive(target))
7   ) {
8     warn(`Cannot delete reactive property on undefined, null, or primitive
9     value: ${(target: any)}`)
10  }
11  // 判断是否是数组，以及 key 是否合法
12  if (Array.isArray(target) && isValidArrayIndex(key)) {
13    // 如果是数组通过 splice 删除
14    // splice 做过响应式处理
15    target.splice(key, 1)
16    return
17  }
18  // 获取 target 的 ob 对象
19  const ob = (target: any).__ob__
20  // target 如果是 Vue 实例或者 $data 对象，直接返回
21  if (target._isVue || (ob && ob.vmCount)) {
22    process.env.NODE_ENV !== 'production' && warn(
23      'Avoid deleting properties on a Vue instance or its root $data ' +
24      '- just set it to null.'
25    )
26    return
27  }
28  // 如果 target 对象没有 key 属性直接返回
29  if (!hasOwn(target, key)) {
30    return
31  }
32  // 删除属性
33  delete target[key]
34  if (!ob) {
35    return
36  }
37  // 通过 ob 发送通知
38  ob.dep.notify()

```

vm.\$watch

vm.\$watch(expOrFn, callback, [options])

- 功能

观察 Vue 实例变化的一个表达式或计算属性函数。回调函数得到的参数为新值和旧值。表达式只接受监督的键路径。对于更复杂的表达式，用一个函数取代。

- 参数

- expOrFn: 要监视的 \$data 中的属性，可以是表达式或函数
- callback: 数据变化后执行的函数
 - 函数: 回调函数
 - 对象: 具有 handler 属性(字符串或者函数)，如果该属性为字符串则 methods 中相应的定义
- options: 可选的选项
 - deep: 布尔类型，深度监听
 - immediate: 布尔类型，是否立即执行一次回调函数

- 示例

```
1  const vm = new Vue({
2    el: '#app',
3    data: {
4      a: '1',
5      b: '2',
6      msg: 'Hello Vue',
7      user: {
8        firstName: '诸葛',
9        lastName: '亮'
10   }
11 }
12 })
13 // expOrFn 是表达式
14 vm.$watch('msg', function (newVal, oldVal) {
15   console.log(newVal, oldVal)
16 })
17 vm.$watch('user.firstName', function (newVal, oldVal) {
18   console.log(newVal)
19 })
20 // expOrFn 是函数
21 vm.$watch(function () {
22   return this.a + this.b
23 }, function (newVal, oldVal) {
24   console.log(newVal)
25 })
26 // deep 是 true, 消耗性能
27 vm.$watch('user', function (newVal, oldVal) {
28   // 此时的 newVal 是 user 对象
29   console.log(newVal === vm.user)
30 }, {
31   deep: true
32 })
33 // immediate 是 true
34 vm.$watch('msg', function (newVal, oldVal) {
35   console.log(newVal)
36 }, {
37   immediate: true
38 })
```

三种类型的 Watcher 对象

- 没有静态方法，因为 \$watch 方法中要使用 Vue 的实例
- Watcher 分三种：计算属性 Watcher、用户 Watcher (侦听器)、渲染 Watcher
- 创建顺序：计算属性 Watcher、用户 Watcher (侦听器)、渲染 Watcher
- vm.\$watch()
 - src\core\instance\state.js

源码

```
1 Vue.prototype.$watch = function (
2   expOrFn: string | Function,
3   cb: any,
4   options?: Object
5 ): Function {
6   // 获取 vue 实例 this
7   const vm: Component = this
8   if (isPlainObject(cb)) {
9     // 判断如果 cb 是对象执行 createWatcher
10    return createWatcher(vm, expOrFn, cb, options)
11  }
12  options = options || {}
13  // 标记为用户 watcher
14  options.user = true
15  // 创建用户 watcher 对象
16  const watcher = new Watcher(vm, expOrFn, cb, options)
17  // 判断 immediate 如果为 true
18  if (options.immediate) {
19    // 立即执行一次 cb 回调，并且把当前值传入
20    try {
21      cb.call(vm, watcher.value)
22    } catch (error) {
23      handleError(error, vm, `callback for immediate watcher
24        "${watcher.expression}"`)
25    }
26    // 返回取消监听的方法
27    return function unwatchFn () {
28      watcher.teardown()
29    }
30  }
```

调试

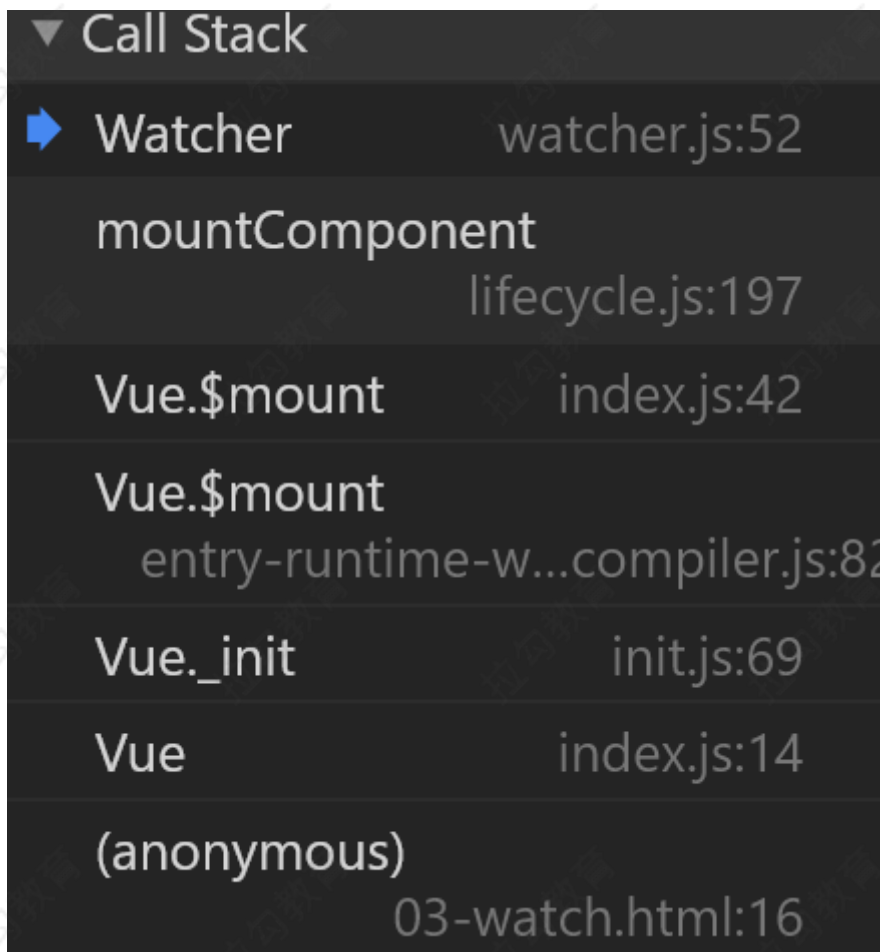
- 查看 watcher 的创建顺序
 - 计算属性 watcher

▼ Call Stack	
➡ Watcher	watcher.js:52
initComputed	state.js:187
initState	state.js:58
Vue._init	init.js:57
Vue	index.js:14
(anonymous)	03-watch.html:16

- 用户 watcher(侦听器)

➡ Watcher	watcher.js:52
Vue.\$watch	state.js:356
createWatcher	state.js:316
initWatch	state.js:298
initState	state.js:60
Vue._init	init.js:57
Vue	index.js:14
(anonymous)	03-watch.html:16

- 渲染 watcher



- 查看渲染 watcher 的执行过程
 - 当数据更新，defineReactive 的 set 方法中调用 dep.notify()
 - 调用 watcher 的 update()
 - 调用 queueWatcher(), 把 watcher 存入队列，如果已经存入，不重复添加
 - 循环调用 flushSchedulerQueue()
 - 通过 nextTick(), 在消息循环结束之前时候调用 flushSchedulerQueue()
 - 调用 watcher.run()
 - 调用 watcher.get() 获取最新值
 - 如果是渲染 watcher 结束
 - 如果是用户 watcher，调用 this.cb()

异步更新队列-nextTick()

- Vue 更新 DOM 是异步执行的，批量的
 - 在下次 DOM 更新循环结束之后执行延迟回调。在修改数据之后立即使用这个方法，获取更新后的 DOM。
- `vm.$nextTick(function () { /* 操作 DOM */ })` / `Vue.nextTick(function () {})`

vm.\$nextTick() 代码演示

```

1 <div id="app">
2   <p ref="p1">{{ msg }}</p>
3 </div>
4 <script src="../../dist/vue.js"></script>
5 <script>
6   const vm = new Vue({
7     el: '#app',
8     data: {
9       msg: 'Hello nextTick',
10      name: 'Vue.js',
11      title: 'Title'
12    },
13    mounted() {
14      this.msg = 'Hello world'
15      this.name = 'Hello snabbdom'
16      this.title = 'vue.js'
17
18      this.$nextTick(() => {
19        console.log(this.$refs.p1.textContent)
20      })
21    }
22  })
23 </script>

```

客服



我
我想咨询价格



机器人融融

对不起，我目前只能回答常见的业务相关问题！此问题暂不在我知识范围内，我会继续努力的！您也可以换个简单的问法向我提问，或许我就可以回答您了...

我
Hello



机器人融融

对不起，我目前只能回答常见的业务相关问题！此问题暂不在我知识范围内，我会继续努力的！您也可以换个简单的问法向我提问，或许我就可以回答您了...

转人工服务

请输入文字...

发送

定义位置

- src\core\instance\render.js

```
1 Vue.prototype.$nextTick = function (fn: Function) {  
2   return nextTick(fn, this)  
3 }
```

源码

- 手动调用 vm.\$nextTick()
- 在 Watcher 的 queueWatcher 中执行 nextTick()
- src\core\util\next-tick.js

```
1 let timerFunc  
2  
3 // The nextTick behavior leverages the microtask queue, which can be  
4 // accessed  
5 // via either native Promise.then or MutationObserver.  
6 // MutationObserver has wider support, however it is seriously bugged in  
7 // UIWebView in iOS >= 9.3.3 when triggered in touch event handlers. It  
8 // completely stops working after triggering a few times... so, if native  
9 // Promise is available, we will use it:  
10 /* istanbul ignore next, $flow-disable-line */  
11 if (typeof Promise !== 'undefined' && isNative(Promise)) {  
12   const p = Promise.resolve()  
13   timerFunc = () => {  
14     p.then(flushCallbacks)  
15     // In problematic UIWebViews, Promise.then doesn't completely break,  
16     // but  
17     // it can get stuck in a weird state where callbacks are pushed into  
18     // the  
19     // microtask queue but the queue isn't being flushed, until the browser  
20     // needs to do some other work, e.g. handle a timer. Therefore we can  
21     // "force" the microtask queue to be flushed by adding an empty timer.  
22     if (isIOS) setTimeout(noop)  
23   }  
24   isUsingMicroTask = true  
25 } else if (!isIE && typeof MutationObserver !== 'undefined' && (  
26   isNative(MutationObserver) ||
```

```

24 // PhantomJS and iOS 7.x
25 MutationObserver.toString() === '[object MutationObserverConstructor]'
26 }) {
27 // Use MutationObserver where native Promise is not available,
28 // e.g. PhantomJS, iOS7, Android 4.4
29 // (#6466 MutationObserver is unreliable in IE11)
30 let counter = 1
31 const observer = new MutationObserver(flushCallbacks)
32 const textNode = document.createTextNode(String(counter))
33 observer.observe(textNode, {
34   characterData: true
35 })
36 timerFunc = () => {
37   counter = (counter + 1) % 2
38   textNode.data = String(counter)
39 }
40 isUsingMicroTask = true
41 } else if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
42 // Fallback to setImmediate.
43 // Technically it leverages the (macro) task queue,
44 // but it is still a better choice than setTimeout.
45 timerFunc = () => {
46   setImmediate(flushCallbacks)
47 }
48 } else {
49 // Fallback to setTimeout.
50 timerFunc = () => {
51   setTimeout(flushCallbacks, 0)
52 }
53 }
54
55 export function nextTick (cb?: Function, ctx?: Object) {
56   let _resolve
57   // 把 cb 加上异常处理存入 callbacks 数组中
58   callbacks.push(() => {
59     if (cb) {
60       try {
61         // 调用 cb()
62         cb.call(ctx)
63       } catch (e) {
64         handleError(e, ctx, 'nextTick')
65       }
66     } else if (_resolve) {
67       _resolve(ctx)
68     }
69   })
70   if (!pending) {
71     pending = true
72     timerFunc()
73   }
74   // $flow-disable-line
75   if (!cb && typeof Promise !== 'undefined') {
76     // 返回 promise 对象
77     return new Promise(resolve => {
78       _resolve = resolve
79     })
80   }
81 }

```

