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

```
src
  ├─compiler        编译相关
  ├─core            Vue 核心库
  ├─platforms       平台相关代码
  ├─server          SSR，服务端渲染
  ├─sfc             .vue 文件编译为 js 对象
  └─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

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

调试设置

打包

- 打包工具 Rollup

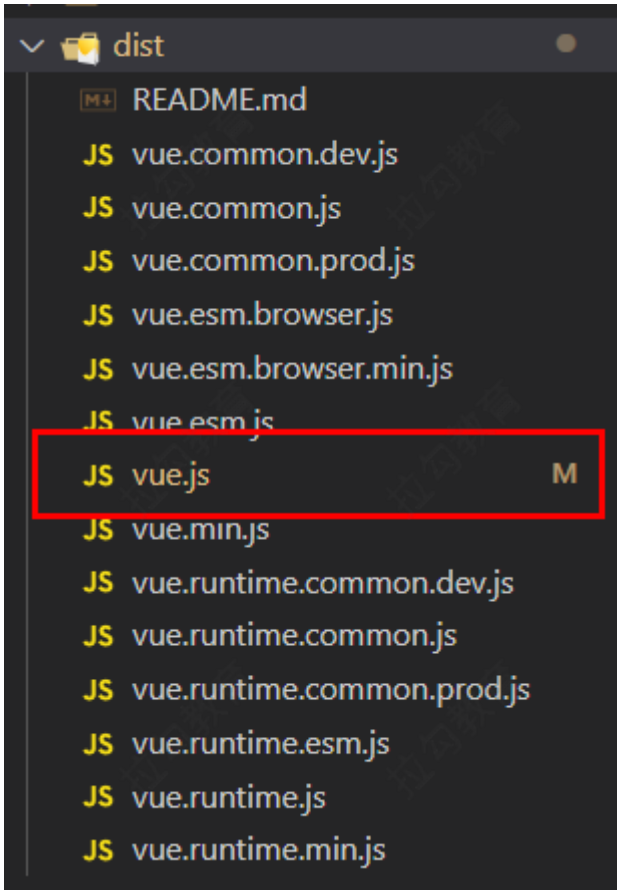
    - Vue.js 源码的打包工具使用的是 Rollup，比 Webpack 轻量
    - Webpack 把所有文件当做模块，Rollup 只处理 js 文件更适合在 Vue.js 这样的库中使用
    - Rollup 打包不会生成冗余的代码
- 安装依赖

```
npm i
```

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

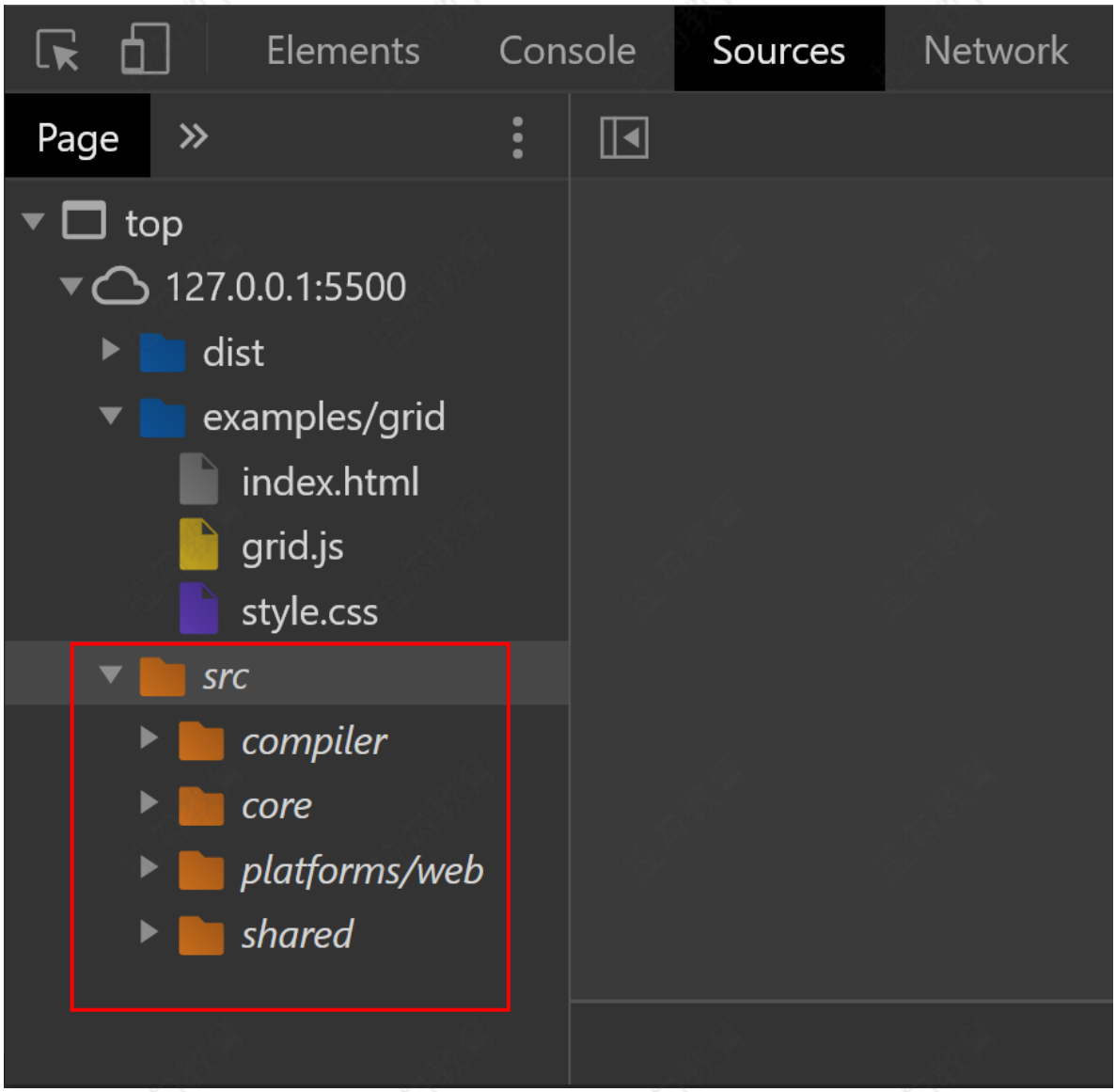
```
"dev": "rollup -w -c scripts/config.js --sourcemap --environment TARGET:web-full-dev"
```

- 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

```
// Compiler
// 需要编译器，把 template 转换成 render 函数
// const vm = new Vue({
//   el: '#app',
//   template: '<h1>{{ msg }}</h1>',
//   data: {
//     msg: 'Hello Vue'
//   }
// })
// Runtime
// 不需要编译器
const vm = new Vue({
  el: '#app',
  render (h) {
    return h('h1', this.msg)
  },
  data: {
    msg: 'Hello Vue'
  }
})
```

- 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

```
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

```
npm run dev
# "dev": "rollup -w -c scripts/config.js --sourcemap --environment TARGET:web-full-dev"
# --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

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

- 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

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

- resolve()
  - 获取入口和出口文件的绝对路径

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

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，还有服务器端渲染的库

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

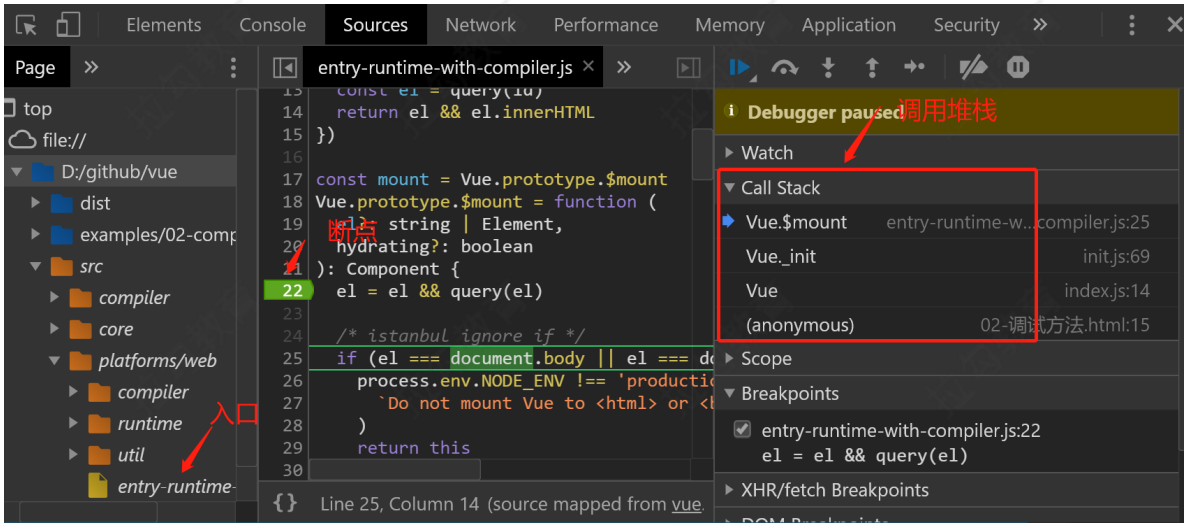
```
const vm = new Vue({
  el: '#app',
  template: '<h3>Hello template</h3>',
  render (h) {
    return h('h4', 'Hello render')
  }
})
```

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

- 调试代码
  - 调试的方法

```
const vm = new Vue({
  el: '#app',
  template: '<h3>Hello template</h3>',
  render (h) {
    return h('h4', 'Hello render')
  }
})
```



> 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**

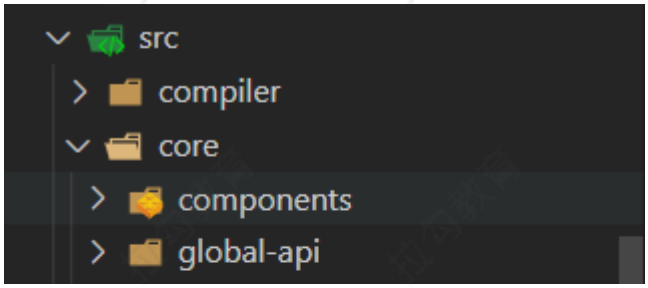
```
// install platform runtime directives & components
extend(Vue.options.directives, platformDirectives)
extend(Vue.options.components, platformComponents)

// install platform patch function
Vue.prototype.__patch__ = inBrowser ? patch : noop

// public mount method
Vue.prototype.$mount = function (
  el?: string | Element,
  hydrating?: boolean
): Component {
  el = el && inBrowser ? query(el) : undefined
  return mountComponent(this, el, hydrating)
}
```

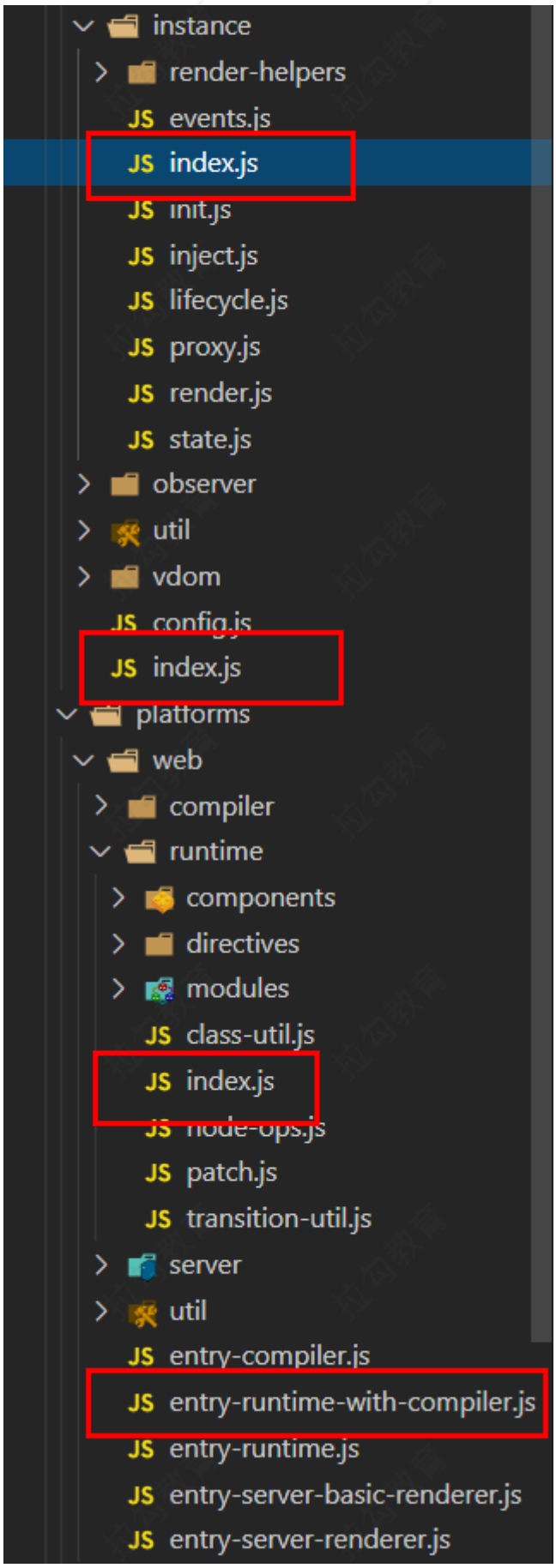
- src/platform/web/runtime/index.js 中引用了 'core/index'
- src/core/index.js
  - 定义了 Vue 的静态方法
  - initGlobalAPI(Vue)
- src/core/index.js 中引用了 './instance/index'
- src/core/instance/index.js
  - 定义了 Vue 的构造函数

```
function Vue (options) {
  if (process.env.NODE_ENV !== 'production' &&
    !(this instanceof Vue)
  ) {
    warn('Vue is a constructor and should be called with the `new` keyword')
  }
  // 调用 _init() 方法
  this._init(options)
}
// 注册 vm 的 _init() 方法，初始化 vm
initMixin(Vue)
// 注册 vm 的 $data/$props/$set/$delete/$watch
stateMixin(Vue)
// 初始化事件相关方法
// $on/$once/$off/$emit
eventsMixin(Vue)
// 初始化生命周期相关的混入方法
// _update/$forceUpdate/$destroy
lifecycleMixin(Vue)
// 混入 render
// $nextTick/_render
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

```
// 注册 Vue 的静态属性/方法
initGlobalAPI(Vue)

// src/core/global-api/index.js
// 初始化 Vue.config 对象
Object.defineProperty(Vue, 'config', configDef)

// exposed util methods.
// NOTE: these are not considered part of the public API - avoid relying on
// them unless you are aware of the risk.
// 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
Vue.util = {
  warn,
  extend,
  mergeOptions,
  defineReactive
}
// 静态方法 set/delete/nextTick
Vue.set = set
Vue.delete = del
Vue.nextTick = nextTick

// 2.6 explicit observable API
// 让一个对象可响应
Vue.observable = <T>(obj: T): T => {
  observe(obj)
  return obj
}
// 初始化 Vue.options 对象，并给其扩展
// components/directives/filters/_base
Vue.options = Object.create(null)
ASSET_TYPES.forEach(type => {
  Vue.options[type + 's'] = Object.create(null)
})

// this is used to identify the "base" constructor to extend all plain-
object
// components with in Weex's multi-instance scenarios.
Vue.options._base = Vue

// 设置 keep-alive 组件
extend(Vue.options.components, builtInComponents)

// 注册 Vue.use() 用来注册插件
initUse(Vue)
// 注册 Vue.mixin() 实现混入
initMixin(Vue)
// 注册 Vue.extend() 基于传入的 options 返回一个组件的构造函数
initExtend(Vue)
// 注册 Vue.directive()、 Vue.component()、Vue.filter()
initAssetRegisters(Vue)
```

src/core/instance/index.js

- 定义 Vue 的构造函数
- 初始化 Vue 的实例成员

```
// 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
function Vue (options) {
  if (process.env.NODE_ENV !== 'production' &&
    !(this instanceof Vue)
  ) {
    warn('Vue is a constructor and should be called with the `new`
keyword')
  }
  this._init(options)
}
// 注册 vm 的 _init() 方法，初始化 vm
initMixin(Vue)
// 注册 vm 的 $data/$props/$set/$delete/$watch
stateMixin(Vue)
// 初始化事件相关方法
// $on/$once/$off/$emit
eventsMixin(Vue)
// 初始化生命周期相关的混入方法
// _update/$forceUpdate/$destroy
lifecycleMixin(Vue)
// 混入 render
// $nextTick/_render
renderMixin(Vue)
```

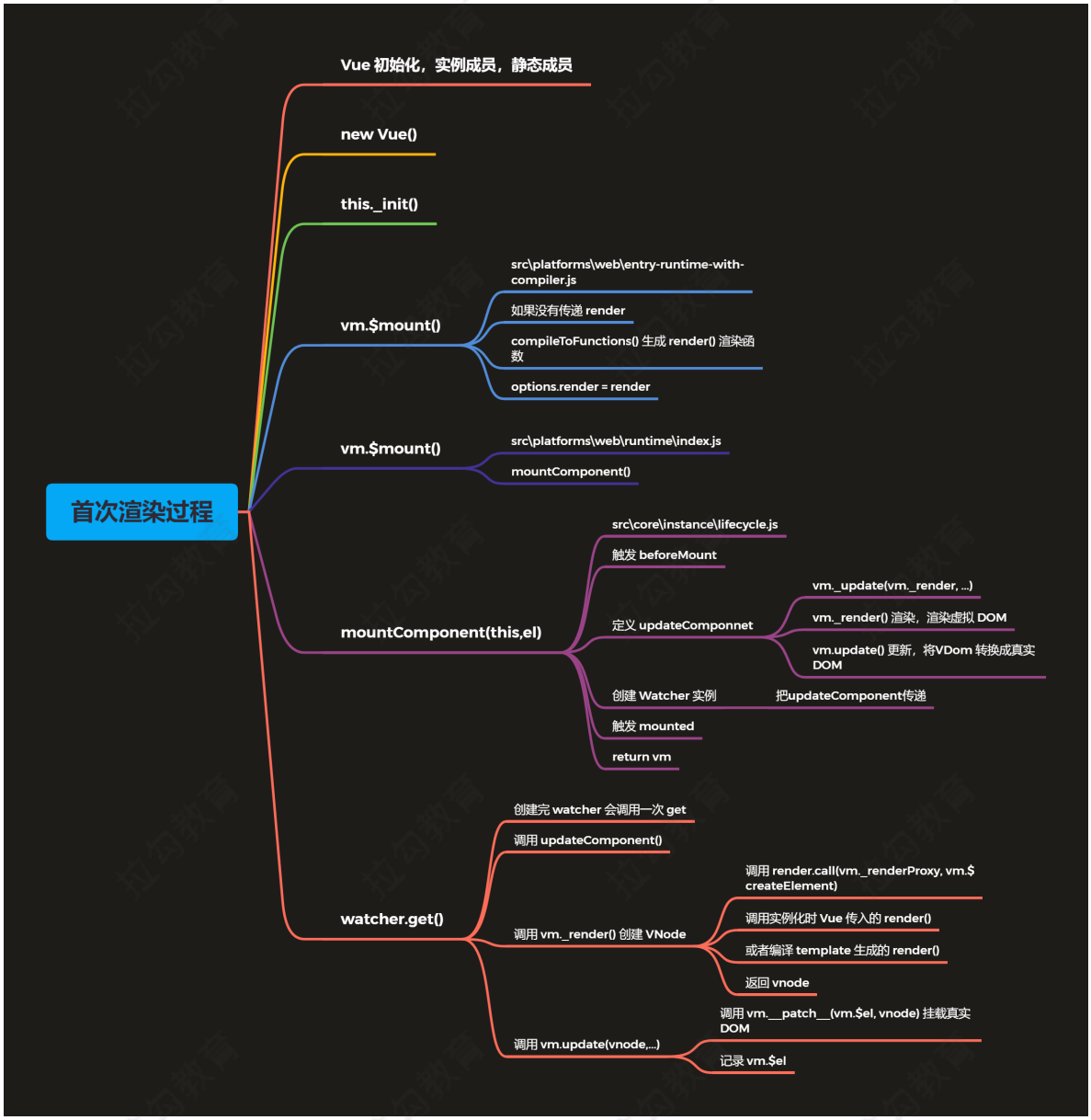
- initMixin(Vue)
  - 初始化 _init() 方法

```
// src\core\instance\init.js
export function initMixin (Vue: Class<Component>) {
  // 给 Vue 实例增加 _init() 方法
  // 合并 options / 初始化操作
  Vue.prototype._init = function (options?: Object) {
      // a flag to avoid this being observed
      // 如果是 Vue 实例不需要被 observe
      vm._isVue = true
      // merge options
      // 合并 options
      if (options && options._isComponent) {
        // optimize internal component instantiation
        // since dynamic options merging is pretty slow, and none of the
        // internal component options needs special treatment.
        initInternalComponent(vm, options)
      } else {
        vm.$options = mergeOptions(
          resolveConstructorOptions(vm.constructor),
          options || {},
          vm
        )
      }
      /* istanbul ignore else */
      if (process.env.NODE_ENV !== 'production') {
        initProxy(vm)
      } else {
        vm._renderProxy = vm
      }
      // expose real self
```

```
      vm._self = vm
      // vm 的生命周期相关变量初始化
      // $children/$parent/$root/$refs
      initLifecycle(vm)
      // vm 的事件监听初始化，父组件绑定在当前组件上的事件
      initEvents(vm)
      // vm 的编译render初始化
      // $slots/$scopedSlots/_c/$createElement/$attrs/$listeners
      initRender(vm)
      // beforeCreate 生命钩子的回调
      callHook(vm, 'beforeCreate')
      // 把 inject 的成员注入到 vm 上
      initInjections(vm) // resolve injections before data/props
      // 初始化状态 vm 的 _props/methods/_data/computed/watch
      initState(vm)
      // 初始化 provide
      initProvide(vm) // resolve provide after data/props
      // created 生命钩子的回调
      callHook(vm, 'created')

      /* istanbul ignore if */
      if (process.env.NODE_ENV !== 'production' && config.performance &&
mark) {
        vm._name = formatComponentName(vm, false)
        mark(endTag)
        measure(`vue ${vm._name} init`, startTag, endTag)
      }
      // 如果没有提供 el，调用 $mount() 挂载
      if (vm.$options.el) {
        vm.$mount(vm.$options.el)
      }
    }
}
```

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

```
// 数据的初始化
if (opts.data) {
  initData(vm)
} else {
  observe(vm._data = {}, true /* asRootData */)
}
```

- initData(vm) vm 数据的初始化

```
function initData (vm: Component) {
  let data = vm.$options.data
  // 初始化 _data，组件中 data 是函数，调用函数返回结果
  // 否则直接返回 data
  data = vm._data = typeof data === 'function'
    ? getData(data, vm)
    : data || {}
  ……

  // proxy data on instance
  // 获取 data 中的所有属性
  const keys = Object.keys(data)
  // 获取 props / methods
  const props = vm.$options.props
  const methods = vm.$options.methods
  let i = keys.length
  // 判断 data 上的成员是否和  props/methods 重名
  ……

  // observe data
  // 数据的响应式处理
  observe(data, true /* asRootData */)
}
```

- 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

```
export function observe (value: any, asRootData: ?boolean): Observer | void
{
  // 判断 value 是否是对象
  if (!isObject(value) || value instanceof VNode) {
    return
  }
  let ob: Observer | void
  // 如果 value 有 __ob__(observer对象) 属性 结束
  if (hasOwn(value, '__ob__') && value.__ob__ instanceof Observer) {
    ob = value.__ob__
  } else if (
    shouldObserve &&
    !isServerRendering() &&
    (Array.isArray(value) || isPlainObject(value)) &&
    Object.isExtensible(value) &&
    !value._isVue
  ) {
    // 创建一个 Observer 对象
```

```
    ob = new Observer(value)
  }
  if (asRootData && ob) {
    ob.vmCount++
  }
  return ob
}
```

Observer

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

```
export class Observer {
  // 观测对象
  value: any;
  // 依赖对象
  dep: Dep;
  // 实例计数器
  vmCount: number; // number of vms that have this object as root $data

  constructor (value: any) {
    this.value = value
    this.dep = new Dep()
    // 初始化实例的 vmCount 为0
    this.vmCount = 0
    // 将实例挂载到观测对象的 __ob__ 属性，设置为不可枚举
    def(value, '__ob__', this)
    if (Array.isArray(value)) {
      // 数组的响应式处理
      if (hasProto) {
        protoAugment(value, arrayMethods)
      } else {
        copyAugment(value, arrayMethods, arrayKeys)
      }
      // 为数组中的每一个对象创建一个 observer 实例
      this.observeArray(value)
    } else {
      // 对象的响应化处理
      // 遍历对象中的每一个属性，转换成 setter/getter
      this.walk(value)
    }
  }

  /**
   * Walk through all properties and convert them into
   * getter/setters. This method should only be called when
   * value type is Object.
   */
  walk (obj: Object) {
    // 获取观察对象的每一个属性
    const keys = Object.keys(obj)
    // 遍历每一个属性，设置为响应式数据
    for (let i = 0; i < keys.length; i++) {
      defineReactive(obj, keys[i])
```

```
    }
  }

  /**
   * Observe a list of Array items.
   */
  observeArray (items: Array<any>) {
    for (let i = 0, l = items.length; i < l; i++) {
      observe(items[i])
    }
  }
}
```

- 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

```
// 为一个对象定义一个响应式的属性
/**
 * Define a reactive property on an Object.
 */
export function defineReactive (
  obj: Object,
  key: string,
  val: any,
  customSetter?: ?Function,
  shallow?: boolean
) {
  // 1. 为每一个属性，创建依赖对象实例
  const dep = new Dep()
  // 获取 obj 的属性描述符对象
  const property = Object.getOwnPropertyDescriptor(obj, key)
  if (property && property.configurable === false) {
    return
  }
  // 提供预定义的存取器函数
  // cater for pre-defined getter/setters
  const getter = property && property.get
  const setter = property && property.set
  if ((!getter || setter) && arguments.length === 2) {
    val = obj[key]
  }
  // 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
  let childOb = !shallow && observe(val)
  Object.defineProperty(obj, key, {
```

```
    enumerable: true,
    configurable: true,
    get: function reactiveGetter () {
      // 如果预定义的 getter 存在则 value 等于getter 调用的返回值
      // 否则直接赋予属性值
      const value = getter ? getter.call(obj) : val
      // 如果存在当前依赖目标，即 watcher 对象，则建立依赖
      if (Dep.target) {
        // dep() 添加相互的依赖
        // 1个组件对应一个 watcher 对象
        // 1个watcher会对应多个dep（要观察的属性很多）
        // 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
        dep.depend()
        // 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
        if (childOb) {
          childOb.dep.depend()
          // 如果属性是数组，则特殊处理收集数组对象依赖
          if (Array.isArray(value)) {
            dependArray(value)
          }
        }
      }
      // 返回属性值
      return value
    },
    set: function reactiveSetter (newVal) {
      // 如果预定义的 getter 存在则 value 等于getter 调用的返回值
      // 否则直接赋予属性值
      const value = getter ? getter.call(obj) : val
      // 如果新值等于旧值或者新值旧值为null则不执行
      /* 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()
      }
      // 如果没有 setter 直接返回
      // #7981: for accessor properties without setter
      if (getter && !setter) return
      // 如果预定义setter存在则调用，否则直接更新新值
      if (setter) {
        setter.call(obj, newVal)
      } else {
        val = newVal
      }
      // 3. 如果新值是对象，观察子对象并返回 子的 observer 对象
      childOb = !shallow && observe(newVal)
      // 4. 发布更改通知
      dep.notify()
    }
  })
}
```

数组的响应式处理

- Observer 的构造函数中

```
// 数组的响应式处理
if (Array.isArray(value)) {
  if (hasProto) {
    protoAugment(value, arrayMethods)
  } else {
    copyAugment(value, arrayMethods, arrayKeys)
  }
  // 为数组中的每一个对象创建一个 observer 实例
  this.observeArray(value)
} else {
  // 编译对象中的每一个属性，转换成 setter/getter
  this.walk(value)
}
function protoAugment (target, src: Object) {
  /* eslint-disable no-proto */
  target.__proto__ = src
  /* eslint-enable no-proto */
}
/* istanbul ignore next */
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])
  }
}
```

- 处理数组修改数据的方法
  - src\core\observer\array.js

```
const arrayProto = Array.prototype
// 克隆数组的原型
export const arrayMethods = Object.create(arrayProto)
// 修改数组元素的方法
const methodsToPatch = [
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]

/**
 * Intercept mutating methods and emit events
 */
methodsToPatch.forEach(function (method) {
  // cache original method
  // 保存数组原方法
  const original = arrayProto[method]
  // 调用 Object.defineProperty() 重新定义修改数组的方法
  def(arrayMethods, method, function mutator (...args) {
    // 执行数组的原始方法
    const result = original.apply(this, args)
```

```
    // 获取数组对象的 ob 对象
    const ob = this.__ob__
    let inserted
    switch (method) {
      case 'push':
      case 'unshift':
        inserted = args
        break
      case 'splice':
        inserted = args.slice(2)
        break
    }
    // 对插入的新元素，重新遍历数组元素设置为响应式数据
    if (inserted) ob.observeArray(inserted)
    // notify change
    // 调用了修改数组的方法，调用数组的ob对象发送通知
    ob.dep.notify()
    return result
  })
})
```

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
```

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

```
  // dep 实例 Id
  id: number;
  // dep 实例对应的 watcher 对象/订阅者数组
  subs: Array<Watcher>;

  constructor () {
    this.id = uid++
    this.subs = []
  }

  // 添加新的订阅者 watcher 对象
  addSub (sub: Watcher) {
    this.subs.push(sub)
  }

  // 移除订阅者
  removeSub (sub: Watcher) {
    remove(this.subs, sub)
  }

  // 将观察对象和 watcher 建立依赖
  depend () {
    if (Dep.target) {
      // 如果 target 存在，把 dep 对象添加到 watcher 的依赖中
      Dep.target.addDep(this)
    }
  }

  // 发布通知
  notify () {
    // stabilize the subscriber list first
    const subs = this.subs.slice()
    if (process.env.NODE_ENV !== 'production' && !config.async) {
      // subs aren't sorted in scheduler if not running async
      // we need to sort them now to make sure they fire in correct
      // order
      subs.sort((a, b) => a.id - b.id)
    }
    // 调用每个订阅者的update方法实现更新
    for (let i = 0, l = subs.length; i < l; i++) {
      subs[i].update()
    }
  }
}
// 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
// The current target watcher being evaluated.
// This is globally unique because only one watcher
// can be evaluated at a time.
Dep.target = null
const targetStack = []
// 入栈并将当前 watcher 赋值给Dep.target
export function pushTarget (target: ?Watcher) {
  targetStack.push(target)
  Dep.target = target
}

export function popTarget () {
```

```
  // 出栈操作
  targetStack.pop()
  Dep.target = targetStack[targetStack.length - 1]
}
```

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

```
export function mountComponent (
  vm: Component,
  el: ?Element,
  hydrating?: boolean
): Component {
  vm.$el = el
  ……
  callHook(vm, 'beforeMount')

  let updateComponent
  /* istanbul ignore if */
  if (process.env.NODE_ENV !== 'production' && config.performance && mark)
{
    ……
  } else {
    updateComponent = () => {
      vm._update(vm._render(), hydrating)
    }
  }
  // 创建渲染 Watcher，expOrFn 为 updateComponent
  // we set this to vm._watcher inside the watcher's constructor
  // since the watcher's initial patch may call $forceUpdate (e.g. inside
child
  // component's mounted hook), which relies on vm._watcher being already
defined
  new Watcher(vm, updateComponent, noop, {
    before () {
      if (vm._isMounted && !vm._isDestroyed) {
        callHook(vm, 'beforeUpdate')
      }
    }
  }, true /* isRenderWatcher */)
  hydrating = false

  // manually mounted instance, call mounted on self
  // mounted is called for render-created child components in its inserted
hook
  if (vm.$vnode == null) {
    vm._isMounted = true
    callHook(vm, 'mounted')
  }
  return vm
}
```

- 渲染 wacher 创建的位置 lifecycle.js 的 mountComponent 函数中
- Wacher 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 wacher 调用 updateComponent），如果是用户 wacher 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 wacher 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 wacher 排序，遍历所有 wacher ，如果有 before，触发生命周期的钩子函数 beforeUpdate，执行 wacher.run()，它内部调用 this.get()，然后调用 this.cb() (渲染 wacher 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

```
<div id="app">
  {{ arr }}
</div>

<script src="../../dist/vue.js"></script>
<script>
  const vm = new Vue({
    el: '#app',
    data: {
      arr: [2, 3, 5]
    }
  })
</script>
```

回答以下问题

- 检测变化的注意事项

```
methods: {
  handler () {
    this.obj.count = 555
    this.arr[0] = 1
    this.arr.length = 0
    this.arr.push(4)
  }
}
```

- 转换成响应式数据

```
methods: {
  handler () {
    this.$set(this.obj, 'count', 555)
    this.$set(this.arr, 0, 1)
    this.arr.splice(0)
  }
}
```

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

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

定义位置

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

```
  // 静态方法 set/delete/nextTick
  Vue.set = set
  Vue.delete = del
  Vue.nextTick = nextTick
```

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

```
// 注册 vm 的 $data/$props/$set/$delete/$watch
// instance/state.js
stateMixin(Vue)

// instance/state.js
Vue.prototype.$set = set
Vue.prototype.$delete = del
```

源码

- set() 方法
  - observer/index.js

```
/**
 * Set a property on an object. Adds the new property and
 * triggers change notification if the property doesn't
 * already exist.
 */
export function set (target: Array<any> | Object, key: any, val: any): any
{
  if (process.env.NODE_ENV !== 'production' &&
    (isUndef(target) || isPrimitive(target))
  ) {
    warn(`Cannot set reactive property on undefined, null, or primitive
value: ${(target: any)}`)
  }
  // 判断 target 是否是对象，key 是否是合法的索引
  if (Array.isArray(target) && isValidArrayIndex(key)) {
    target.length = Math.max(target.length, key)
    // 通过 splice 对key位置的元素进行替换
    // splice 在 array.js进行了响应化的处理
    target.splice(key, 1, val)
    return val
  }
  // 如果 key 在对象中已经存在直接赋值
  if (key in target && !(key in Object.prototype)) {
    target[key] = val
    return val
  }
  // 获取 target 中的 observer 对象
  const ob = (target: any).__ob__
  // 如果 target 是 vue 实例或者$data 直接返回
  if (target._isVue || (ob && ob.vmCount)) {
    process.env.NODE_ENV !== 'production' && warn(
      'Avoid adding reactive properties to a Vue instance or its root $data
' +
      'at runtime - declare it upfront in the data option.'
    )
    return val
  }
  // 如果 ob 不存在，target 不是响应式对象直接赋值
  if (!ob) {
    target[key] = val
    return val
  }
  // 把 key 设置为响应式属性
  defineReactive(ob.value, key, val)
  // 发送通知
  ob.dep.notify()
  return val
}
```

调试

```
<div id="app">
  {{ obj.msg }}
    <br>
      {{ obj.foo }}
</div>

```

```
<script src="../../dist/vue.js"></script>
<script>
  const vm = new Vue({
    el: '#app',
    data: {
      obj: {
        msg: 'hello set'
      }
    }
  })
  // 非响应式数据
  // vm.obj.foo = 'test'
  vm.$set(vm.obj, 'foo', 'test')
</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

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

定义位置

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

```
  // 静态方法 set/delete/nextTick
  Vue.set = set
  Vue.delete = del
  Vue.nextTick = nextTick
```

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

```
// 注册 vm 的 $data/$props/$set/$delete/$watch
stateMixin(Vue)

// instance/state.js
Vue.prototype.$set = set
Vue.prototype.$delete = del
```

源码

- src\core\observer\index.js

```
/**
 * Delete a property and trigger change if necessary.
 */
export function del (target: Array<any> | Object, key: any) {
  if (process.env.NODE_ENV !== 'production' &&
    (isUndef(target) || isPrimitive(target))
  ) {
    warn(`Cannot delete reactive property on undefined, null, or primitive value: ${(target: any)}`)
  }
  // 判断是否是数组，以及 key 是否合法
  if (Array.isArray(target) && isValidArrayIndex(key)) {
    // 如果是数组通过 splice 删除
    // splice 做过响应式处理
    target.splice(key, 1)
    return
  }
  // 获取 target 的 ob 对象
  const ob = (target: any).__ob__
  // target 如果是 Vue 实例或者 $data 对象，直接返回
  if (target._isVue || (ob && ob.vmCount)) {
    process.env.NODE_ENV !== 'production' && warn(
      'Avoid deleting properties on a Vue instance or its root $data ' +
      '- just set it to null.'
    )
    return
  }
  // 如果 target 对象没有 key 属性直接返回
  if (!hasOwn(target, key)) {
    return
  }
  // 删除属性
  delete target[key]
  if (!ob) {
    return
  }
  // 通过 ob 发送通知
  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

```
const vm = new Vue({
  el: '#app',
  data: {
    a: '1',
    b: '2',
    msg: 'Hello Vue',
    user: {
      firstName: '诸葛',
      lastName: '亮'
    }
  }
})
// expOrFn 是表达式
vm.$watch('msg', function (newVal, oldVal) {
  console.log(newVal, oldVal)
})
vm.$watch('user.firstName', function (newVal, oldVal) {
  console.log(newVal)
})
// expOrFn 是函数
vm.$watch(function () {
  return this.a + this.b
}, function (newVal, oldVal) {
  console.log(newVal)
})
// deep 是 true，消耗性能
vm.$watch('user', function (newVal, oldVal) {
  // 此时的 newVal 是 user 对象
  console.log(newVal === vm.user)
}, {
  deep: true
})
// immediate 是 true
vm.$watch('msg', function (newVal, oldVal) {
  console.log(newVal)
}, {
  immediate: true
})
```

三种类型的 Watcher 对象

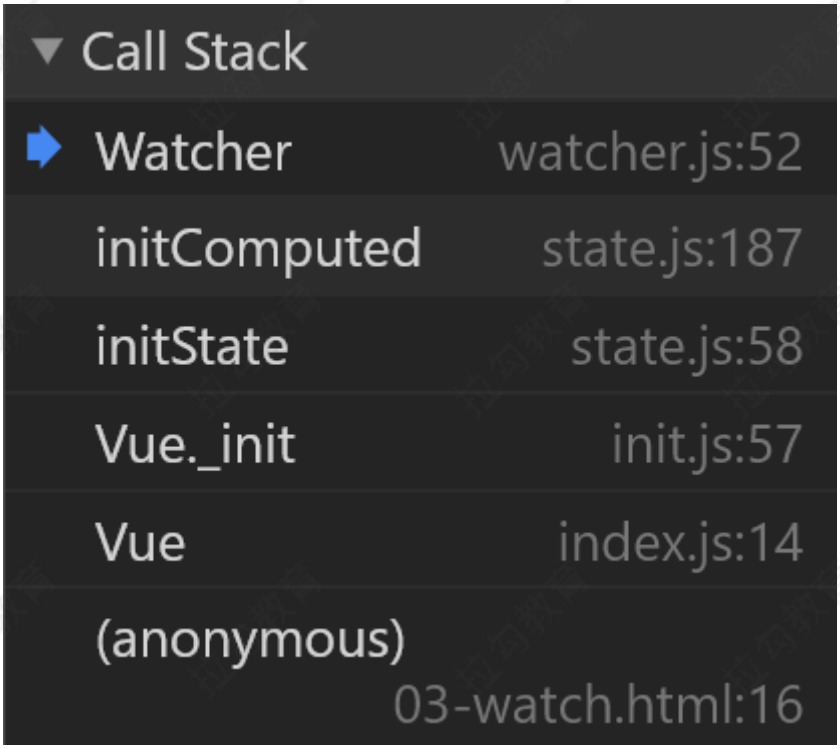
- 没有静态方法，因为 $watch 方法中要使用 Vue 的实例
- Watcher 分三种：计算属性 Watcher、用户 Watcher (侦听器)、渲染 Watcher
- 创建顺序：计算属性 Watcher、用户 Watcher (侦听器)、渲染 Watcher
- vm.$watch()
    - src\core\instance\state.js

源码

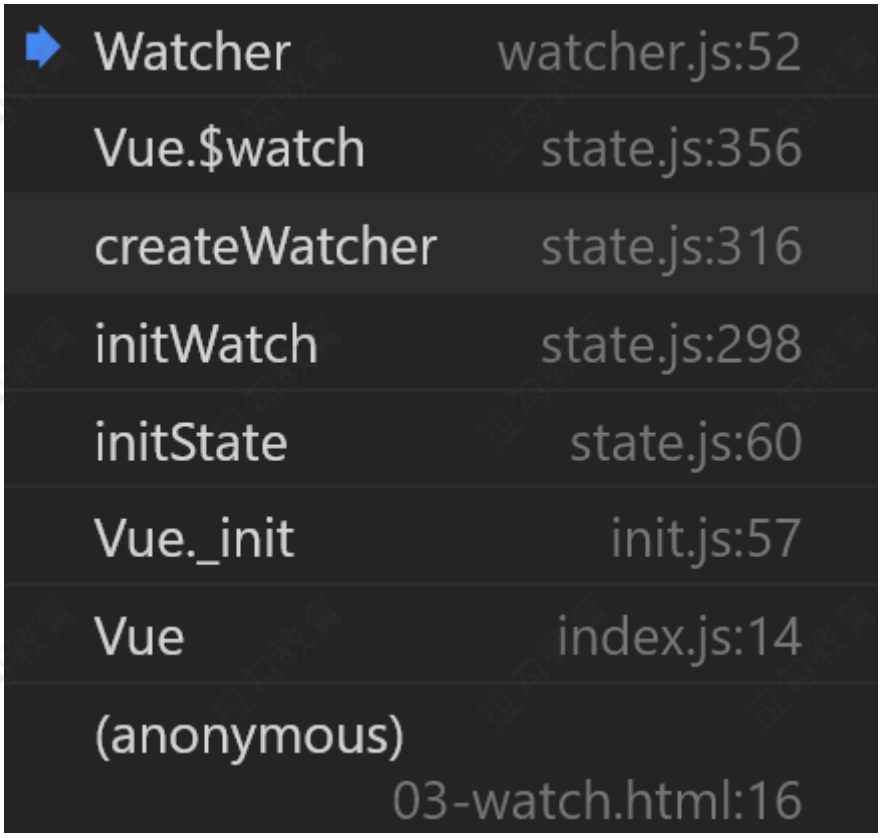
```
Vue.prototype.$watch = function (
  expOrFn: string | Function,
  cb: any,
  options?: Object
): Function {
  // 获取 Vue 实例 this
  const vm: Component = this
  if (isPlainObject(cb)) {
    // 判断如果 cb 是对象执行 createWatcher
    return createWatcher(vm, expOrFn, cb, options)
  }
  options = options || {}
  // 标记为用户 watcher
  options.user = true
  // 创建用户 watcher 对象
  const watcher = new Watcher(vm, expOrFn, cb, options)
  // 判断 immediate 如果为 true
  if (options.immediate) {
    // 立即执行一次 cb 回调，并且把当前值传入
    try {
      cb.call(vm, watcher.value)
    } catch (error) {
      handleError(error, vm, `callback for immediate watcher "${watcher.expression}"`)
    }
  }
  // 返回取消监听的方法
  return function unwatchFn () {
    watcher.teardown()
  }
}
```

调试

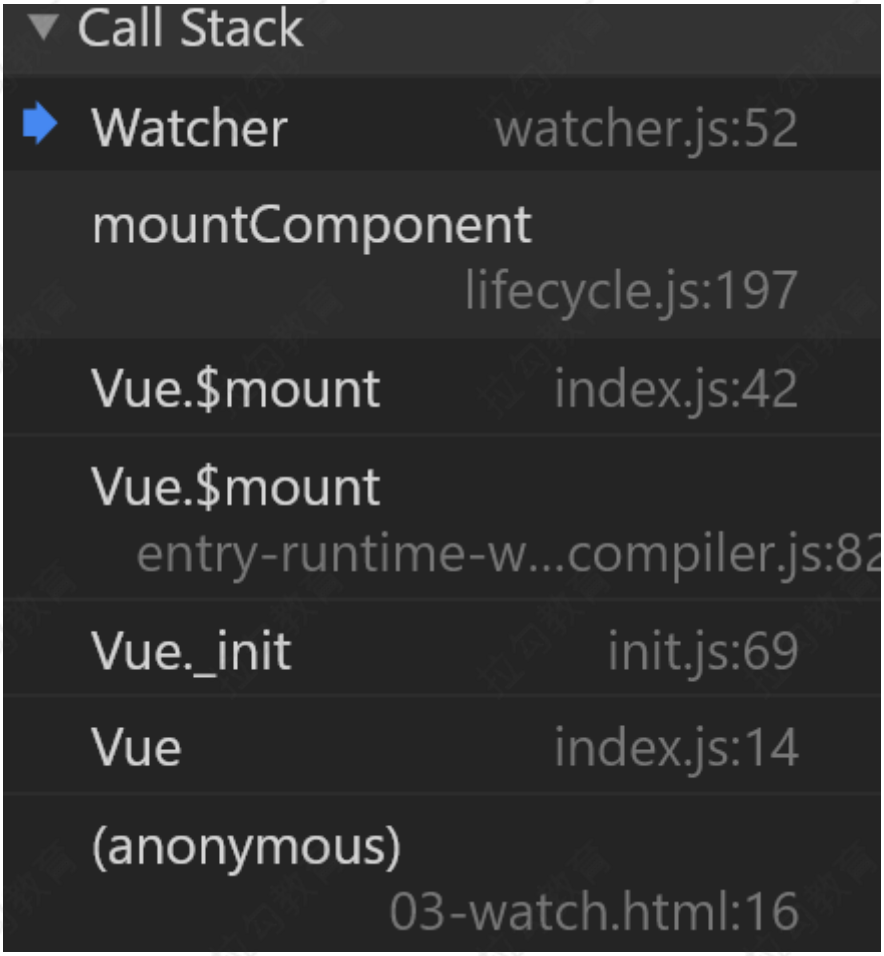
- 查看 watcher 的创建顺序
    - 计算属性 watcher



- 用户 wacher(侦听器)



- 渲染 wacher



- 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 wacher 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 wacher.run()
    - 调用 wacher.get() 获取最新值
    - 如果是渲染 wacher 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

```html
<div id="app">
  <p ref="p1">{{ msg }}</p>
</div>
<script src="../../dist/vue.js"></script>
<script>
  const vm = new Vue({
    el: '#app',
    data: {
      msg: 'Hello nextTick',
      name: 'Vue.js',
      title: 'Title'
    },
    mounted() {
      this.msg = 'Hello World'
      this.name = 'Hello snabbdom'
      this.title = 'Vue.js'

      this.$nextTick(() => {
        console.log(this.$refs.p1.textContent)
      })
    }
  })
</script>
```

定义位置

- src\core\instance\render.js

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

源码

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

```
let timerFunc

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

```
  // PhantomJS and iOS 7.x
  MutationObserver.toString() === '[object MutationObserverConstructor]'
)) {
  // Use MutationObserver where native Promise is not available,
  // e.g. PhantomJS, iOS7, Android 4.4
  // (#6466 MutationObserver is unreliable in IE11)
  let counter = 1
  const observer = new MutationObserver(flushCallbacks)
  const textNode = document.createTextNode(String(counter))
  observer.observe(textNode, {
    characterData: true
  })
  timerFunc = () => {
    counter = (counter + 1) % 2
    textNode.data = String(counter)
  }
  isUsingMicroTask = true
} else if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
  // Fallback to setImmediate.
  // Technically it leverages the (macro) task queue,
  // but it is still a better choice than setTimeout.
  timerFunc = () => {
    setImmediate(flushCallbacks)
  }
} else {
  // Fallback to setTimeout.
  timerFunc = () => {
    setTimeout(flushCallbacks, 0)
  }
}

export function nextTick (cb?: Function, ctx?: Object) {
  let _resolve
  // 把 cb 加上异常处理存入 callbacks 数组中
  callbacks.push(() => {
    if (cb) {
      try {
        // 调用 cb()
        cb.call(ctx)
      } catch (e) {
        handleError(e, ctx, 'nextTick')
      }
    } else if (_resolve) {
      _resolve(ctx)
    }
  })
  if (!pending) {
    pending = true
    timerFunc()
  }
  // $flow-disable-line
  if (!cb && typeof Promise !== 'undefined') {
    // 返回 promise 对象
    return new Promise(resolve => {
      _resolve = resolve
    })
  }
}
```