源码串讲

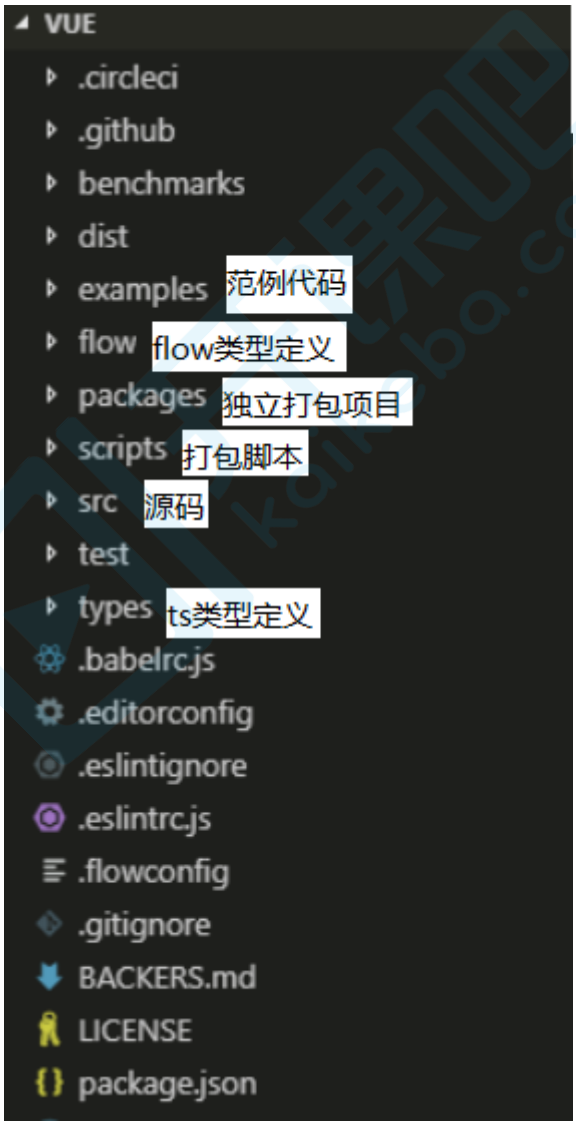
获取vue

项目地址：https://github.com/vuejs/vue

迁出项目：`git clone https://github.com/vuejs/vue.git`

当前版本号：2.6.9

文件结构



> vue源码使用flow编写，所以还要额外维护TypeScript类型定义，加上flow目前已停止维护，所以vue 3才会选择用TypeScript重写。

调试环境搭建

安装依赖：`npm i`

> win10需要管理员权限打开vscode

安装rollup：`npm i -g rollup`

> rollup是打包工具，用于打包纯代码项目

修改dev脚本，添加sourcemap，package.json

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

运行开发命令：`npm run dev`

引入前面创建的vue.js，samples/commits/index.html

```
<script src="../../dist/vue.js"></script>
```

接下来可以在浏览器愉快的调试代码了！

入口

src\platforms\web\entry-runtime-with-compiler.js

扩展$mount

```
// 扩展默认$mount方法：能够编译template或el指定的模板
const mount = Vue.prototype.$mount
Vue.prototype.$mount = function (
  el?: string | Element,
  hydrating?: boolean
): Component {
  // 获取选项
  const options = this.$options
  // 不存在render选项，则将template/el的设置转换为render函数
  if (!options.render) {
    let template = options.template
    if (template) {
      // 解析template选项
    } else if (el) {
      // 否则解析el选项
      template = getOuterHTML(el)
    }
    if (template) {
      // 编译得到render函数
      const { render, staticRenderFns } = compileToFunctions(template, {..}, this)
      options.render = render
    }
  }
```

```
  // 执行默认$mount函数
  return mount.call(this, el, hydrating)
}
```

src\platforms\web\runtime\index.js

实现$mount，核心就一个mountComponent；定义一个__patch__方法

```
Vue.prototype.__patch__ = inBrowser ? patch : noop
Vue.prototype.$mount = function (
  el?: string | Element,
  hydrating?: boolean
): Component {
  el = el && inBrowser ? query(el) : undefined
  // 挂载组件
  return mountComponent(this, el, hydrating)
}
```

mountComponent

定义更新函数，创建一个watcher，它会执行一次更新函数，完成挂载

```
updateComponent = () => {
    vm._update(vm._render(), hydrating)
}
new Watcher(vm, updateComponent, noop, {
    before () {
        if (vm._isMounted && !vm._isDestroyed) {
            callHook(vm, 'beforeUpdate')
        }
    }
}, true /* isRenderWatcher */)
```

core/instance/index.js

实现Vue构造函数，实现若干实例方法和属性

```
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)
}
```

```
initMixin(Vue) // _init
stateMixin(Vue) //
eventsMixin(Vue)
lifecycleMixin(Vue)
renderMixin(Vue)
```

initMixin

实现vue初始化函数_init

```
initLifecycle(vm)
initEvents(vm)
initRender(vm)
callHook(vm, 'beforeCreate')
initInjections(vm) // resolve injections before data/props
initState(vm)
initProvide(vm) // resolve provide after data/props
callHook(vm, 'created')
```

stateMixin

组件状态相关api如$set,$delete,$watch实现

eventsMixin

事件相关api如$on,$emit,$off,$once实现

lifecycleMixin

组件声明周期api如_update,$forceUpdate,$destroy实现

renderMixin(Vue)

实现组件渲染函数_render, $nextTick

下面是vue初始化过程

initLifecycle src\core\instance\lifecycle.js

把组件实例里面用到的常用属性初始化，比如$parent/$root/$children

```
  const options = vm.$options
  vm.$parent = parent
  vm.$root = parent ? parent.$root : vm

  vm.$children = []
  vm.$refs = {}

  vm._watcher = null
```

initEvents src\core\instance\events.js

父组件中定义的需要子组件处理的事件

```
vm._events = Object.create(null)
  vm._hasHookEvent = false
  // init parent attached events
  const listeners = vm.$options._parentListeners
  if (listeners) {
    updateComponentListeners(vm, listeners)
  }
```

initRenders

$slots $scopedSlots初始化

$createElement函数声明

$attrs和$listeners响应化

```
vm._vnode = null // the root of the child tree
  vm._staticTrees = null // v-once cached trees
  const options = vm.$options
  const parentVnode = vm.$vnode = options._parentVnode // the placeholder node in
parent tree
  const renderContext = parentVnode && parentVnode.context
  vm.$slots = resolveSlots(options._renderChildren, renderContext)
  vm.$scopedSlots = emptyObject
  // 把createElement函数挂载到当前组件上，编译器需要用到
  vm._c = (a, b, c, d) => createElement(vm, a, b, c, d, false)
  // 用户编写渲染函数使用这个render functions.
  vm.$createElement = (a, b, c, d) => createElement(vm, a, b, c, d, true)

  // $attrs & $listeners are exposed for easier HOC creation.
  // they need to be reactive so that HOCs using them are always updated
  const parentData = parentVnode && parentVnode.data

  defineReactive(vm, '$attrs', parentData && parentData.attrs || emptyObject, null,
true)
  defineReactive(vm, '$listeners', options._parentListeners || emptyObject, null, true)
```

initInjections

注入内容的响应化

```
  // 获取注入内容
  const result = resolveInject(vm.$options.inject, vm)
  if (result) {
    toggleObserving(false)
    // 注入内容响应化
    Object.keys(result).forEach(key => {
        defineReactive(vm, key, result[key])
    })
    toggleObserving(true)
  }
```

initState

执行各种数据状态初始化地方，包括数据响应化等等

```
vm._watchers = []
  const opts = vm.$options
  //初始化所有属性
  if (opts.props) initProps(vm, opts.props)
  // 初始化回调函数
  if (opts.methods) initMethods(vm, opts.methods)
  // 数据响应化
  if (opts.data) {
    initData(vm)
  } else {
    observe(vm._data = {}, true /* asRootData */)
  }
  if (opts.computed) initComputed(vm, opts.computed)
  if (opts.watch && opts.watch !== nativeWatch) {
    initWatch(vm, opts.watch)
  }
```

initProvide

为子组件提供数据

src/core/index.js

主要初始化全局api

```
initGlobalAPI(Vue)
```

initGlobalAPI

设置Vue全局API如set/delete/nextTick

```
// 核心代码
export function initGlobalAPI (Vue: GlobalAPI) {
  Vue.set = set
  Vue.delete = del
  Vue.nextTick = nextTick

  initUse(Vue) // 实现Vue.use函数
  initMixin(Vue) // 实现Vue.mixin函数
  initExtend(Vue) // 实现Vue.extend函数
  initAssetRegisters(Vue) // 注册实现Vue.component/directive/filter
}
```

####

vue数据响应式

vue数据响应化的代码都在src/core/observer里面

具体实现是在Vue初始化时，会调用initState，它会初始化data，props等，这里着重关注data初始化，src\core\instance\state.jsinitData核心代码是将data数据响应化

observe

observe方法返回一个Observer实例，core/observer/index.js

```
// value: 待响应化数据对象
export function observe (value: any, asRootData: ?boolean): Observer | void {
  // 观察者
  let ob: Observer | void
  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
```

```
  ) {// 创建观察者
    ob = new Observer(value)
  }
  if (asRootData && ob) {
    ob.vmCount++
  }
  return ob
}
```

Observer

Observer对象根据数据类型执行对应的响应化操作，core/observer/index.js

```
export class Observer {
  value: any;
  dep: Dep; // 保存数组类型数据的依赖

  constructor (value: any) {
    this.value = value
    this.dep = new Dep()
    def(value, '__ob__', this) // 在getter中可以通过__ob__可获取ob实例
    if (Array.isArray(value)) { // 数组响应化
      protoAugment(value, arrayMethods)
      this.observeArray(value)
    } else { // 对象响应化
      this.walk(value)
    }
  }

  /**
   * 遍历对象所有属性并转换为getter/setters
   */
  walk (obj: Object) {
    const keys = Object.keys(obj)
    for (let i = 0; i < keys.length; i++) {
      defineReactive(obj, keys[i])
    }
  }

  /**
   * 对数组每一项执行响应化
   */
  observeArray (items: Array<any>) {
    for (let i = 0, l = items.length; i < l; i++) {
      observe(items[i])
    }
  }
}
```

defineReactive

defineReactive定义对象属性的getter/setter，getter负责添加依赖，setter负责通知更新

```
export function defineReactive (
  obj: Object,
  key: string,
  val: any,
  customSetter?: ?Function,
  shallow?: boolean
) {
  const dep = new Dep() // 一个key一个Dep实例
  if (arguments.length === 2) {
    val = obj[key]
  }
  // 递归执行子对象响应化
  let childOb = !shallow && observe(val)
  // 定义当前对象getter/setter
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter () {
      // getter被调用时若存在依赖则追加
      if (Dep.target) {
        dep.depend()
        // 若存在子observer，则依赖也追加到子ob
        if (childOb) {
          childOb.dep.depend()
          if (Array.isArray(value)) {
            dependArray(value) // 数组需特殊处理
          }
        }
      }
      return value
    },
    set: function reactiveSetter (newVal) {
      if (newVal === value || (newVal !== newVal && value !== value)) {
        return
      }
      val = newVal // 更新值
      childOb = !shallow && observe(newVal) // 递归更新子对象
      dep.notify() // 通知更新
    }
  })
}
```

Dep

负责管理一组Watcher，包括watcher实例的增删及通知更新，core/observer/dep.js

```
export default class Dep {
  static target: ?Watcher; // 依赖收集时的wacher引用
  subs: Array<Watcher>; // watcher数组

  constructor () {
    this.subs = []
  }
```

```
  //添加watcher实例
  addSub (sub: Watcher) {
    this.subs.push(sub)
  }
  //删除watcher实例
  removeSub (sub: Watcher) {
    remove(this.subs, sub)
  }
  //watcher和dep相互保存引用
  depend () {
    if (Dep.target) {
      Dep.target.addDep(this)
    }
  }

  notify () {
    // stabilize the subscriber list first
    const subs = this.subs.slice()

    for (let i = 0, l = subs.length; i < l; i++) {
      subs[i].update()
    }
  }
}
```

Watcher

Watcher解析一个表达式并收集依赖，当数值变化时触发回调函数，常用于$watch API和指令中。

每个组件也会有对应的Watcher，数值变化会触发其update函数导致重新渲染

```
export default class Watcher {
  constructor (
    vm: Component,
    expOrFn: string | Function,
    cb: Function,
    options?: ?Object,
    isRenderWatcher?: boolean
  ) {
    this.vm = vm
    // 组件保存render watcher
    if (isRenderWatcher) {
      vm._watcher = this
    }
    // 组件保存非render watcher
    vm._watchers.push(this)

    // options...

    // 将表达式解析为getter函数
    // 那些和组件实例对应的Watcher创建时会传递组件更新函数进来
    if (typeof expOrFn === 'function') {
      this.getter = expOrFn
```

```
    } else {
      // 这种是$watch传递进来的表达式，它们需要解析为函数
      this.getter = parsePath(expOrFn)
      if (!this.getter) {
        this.getter = noop
      }
    }
    // 若非延迟watcher，立即调用getter
    this.value = this.lazy ? undefined : this.get()
  }

  /**
   * 模拟getter，重新收集依赖re-collect dependencies.
   */
  get () {
    // Dep.target = this
    pushTarget(this)
    let value
    const vm = this.vm
    try {
      // 从组件中获取到value同时触发依赖收集
      value = this.getter.call(vm, vm)
    }
    catch (e) {}
    finally {
      // deep watching，递归触发深层属性
      if (this.deep) {
        traverse(value)
      }
      popTarget()
      this.cleanupDeps()
    }
    return value
  }

  addDep (dep: Dep) {
    const id = dep.id
    if (!this.newDepIds.has(id)) {
      // watcher保存dep引用
      this.newDepIds.add(id)
      this.newDeps.push(dep)
      // dep添加watcher
      if (!this.depIds.has(id)) {
        dep.addSub(this)
      }
    }
  }

  update () {
    // 更新逻辑
    if (this.lazy) {
      this.dirty = true
    } else if (this.sync) {
```

```
      this.run()
    } else {
      //默认lazy和sync都是false，所以会走该逻辑
      queueWatcher(this)
    }
  }
}
```

数组响应化

数组数据变化采取的策略是拦截push、pop、splice等方法执行dep通知。

为数组原型中的7个可以改变内容的方法定义拦截器，src\core\observer\array.js

```
// 数组原型
const arrayProto = Array.prototype
// 修改后的原型
export const arrayMethods = Object.create(arrayProto)
// 七个待修改方法
const methodsToPatch = [
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]

/**
 * 拦截这些方法，额外发送变更通知
 */
methodsToPatch.forEach(function (method) {
  // 原始数组方法
  const original = arrayProto[method]
  // 修改这些方法的descriptor
  def(arrayMethods, method, function mutator (...args) {
    // 原始操作
    const result = original.apply(this, args)
    // 获取ob实例用于发送通知
    const ob = this.__ob__
    // 三个能新增元素的方法特殊处理
    let inserted
    switch (method) {
      case 'push':
      case 'unshift':
        inserted = args
        break
      case 'splice':
        inserted = args.slice(2)
```

```
            break
        }
        // 若有新增则做响应处理
        if (inserted) ob.observeArray(inserted)
        // 通知更新
        ob.dep.notify()
        return result
      })
    })
```

Observer中覆盖数组原型

```
if (Array.isArray(value)) {
    // 替换数组原型
    protoAugment(value, arrayMethods) // value.__proto__ = arrayMethods
    this.observeArray(value)
}
```

defineReactive中数组的特殊处理：

```
// getter处理中
if (Array.isArray(value)) {
    dependArray(value)
}

// 数组中所有项添加依赖，将来数组里面就可以通过__ob__.dep发送通知
function dependArray (value: Array<any>) {
  for (let e, i = 0, l = value.length; i < l; i++) {
    e = value[i]
    e && e.__ob__ && e.__ob__.dep.depend()
    if (Array.isArray(e)) {
      dependArray(e)
    }
  }
}
```

vue异步更新队列

Vue 在更新 DOM 时是**异步**执行的。只要侦听到数据变化，Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据变更。如果同一个 watcher 被多次触发，只会被推入到队列中一次。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的。然后，在下一个的事件循环“tick”中，Vue 刷新队列并执行实际 (已去重的) 工作。Vue 在内部对异步队列尝试使用原生的 `Promise.then`、`MutationObserver` 和 `setImmediate`，如果执行环境不支持，则会采用 `setTimeout(fn, 0)` 代替。

queueWatcher

执行watcher入队操作，若存在重复id则跳过

```
// watcher入队
export function queueWatcher (watcher: Watcher) {
  const id = watcher.id
  if (has[id] == null) { // id不存在才会入队
    has[id] = true
    if (!flushing) { // 没有在执行刷新则进入队尾
      queue.push(watcher)
    } else {
      // 若已刷新，按id顺序插入到队列
      // 若已经过了，则下次刷新立即执行
      let i = queue.length - 1
      while (i > index && queue[i].id > watcher.id) {
        i--
      }
      queue.splice(i + 1, 0, watcher)
    }
    // 刷新队列
    if (!waiting) {
      waiting = true
      nextTick(flushSchedulerQueue)
    }
  }
}
```

nextTick(flushSchedulerQueue)

nextTick按照特定异步策略执行队列刷新操作

```
// nextTick异步执行策略，src\core\util\next-tick.js
export function nextTick (cb?: Function, ctx?: Object) {
  let _resolve
  // 注意cb不是立刻执行，而是加入到回调数组，等待调用
  callbacks.push(() => {
    if (cb) {
      try {
        cb.call(ctx) // 真正执行cb
      } catch (e) {
        handleError(e, ctx, 'nextTick')
      }
    } else if (_resolve) {
      _resolve(ctx)
    }
  })
  // 没有处在挂起状态则开始异步执行过程
  if (!pending) {
    pending = true
    timerFunc()
  }
  // $flow-disable-line
  if (!cb && typeof Promise !== 'undefined') {
    return new Promise(resolve => {
```

```
      _resolve = resolve
    })
  }
}

let timerFunc

// nextTick异步行为利用微任务队列，可通过Promise或MutationObserver交互
// 首选Promise，次选MutationObserver
if (typeof Promise !== 'undefined' && isNative(Promise)) {
  const p = Promise.resolve()
  timerFunc = () => {
    p.then(flushCallbacks)
    // IOS hack
    if (isIOS) setTimeout(noop)
  }
  isUsingMicroTask = true
} else if (!isIE && typeof MutationObserver !== 'undefined' && (
  isNative(MutationObserver) ||
  // PhantomJS and iOS 7.x
  MutationObserver.toString() === '[object MutationObserverConstructor]'
)) {
  // 不能用Promise时：PhantomJS, iOS7, Android 4.4
  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)) {
  // 回退到setImmediate.它利用的是宏任务队列
  timerFunc = () => {
    setImmediate(flushCallbacks)
  }
} else {
  // 最后选择setTimeout.
  timerFunc = () => {
    setTimeout(flushCallbacks, 0)
  }
}
```

> 宏任务和微任务相关知识补充请看这里