Vue2.1.7源码学习

2017-03-03

Vue 源码 WebFrontEnd

原本文章的名字叫做《源码解析》，不过后来想想，还是用“源码学习”来的合适一点，在没有彻底掌握源码中的每一个字母之前，“解析”就有点标题党了。建议在看这篇文章之前，最好打开2.1.7的源码对照着看，这样可能更容易理解。另外本人水平有限，文中有错误或不妥的地方望大家多多指正共同成长。

补充：Vue 2.2 刚刚发布，作为一个系列文章的第一篇，本篇文章主要从Vue代码的组织，Vue构造函数的还原，原型的设计，以及参数选项的处理和已经被写烂了的数据绑定与如何使用 Virtual DOM 更新视图入手。从整体的大方向观察框架，这么看来 `V2.1.7` 对于理解 `V2.2` 的代码不会有太大的影响。该系列文章的后续文章，都会从最新的源码入手，并对改动的地方做相应的提示。

很久之前写过一篇文章：JavaScript实现MVVM之我就是想监测一个普通对象的变化，文章开头提到了我写博客的风格，还是那句话，只写努力让小白，甚至是小学生都能看明白的文章。这不免会导致对于某些同学来说这篇文章有些墨迹，所以大家根据自己的喜好，可以详细的看，也可以跳跃着看。

一、从了解一个开源项目入手

要看一个项目的源码，不要一上来就看，先去了解一下项目本身的元数据和依赖，除此之外最好也了解一下 PR 规则，Issue Reporting 规则等等。特别是“前端”开源项目，我们在看源码之前第一个想到的应该是：`package.json` 文件。

在 `package.json` 文件中，我们最应该关注的就是 `scripts` 字段和 `devDependencies` 以及 `dependencies` 字段，通过 `scripts` 字段我们可以知道项目中定义的脚本命令，通过 `devDependencies` 和 `dependencies` 字段我们可以了解项目的依赖情况。

了解了这些之后，如果有依赖我们就 `npm install` 安装依赖就ok了。

除了 `package.json` 之外，我们还要阅读项目的贡献规则文档，了解如何开始，一个好的开源项目肯定会包含这部分内容的，Vue也不例外：https://github.com/vuejs/vue/blob/dev/.github/CONTRIBUTING.md，在这个文档里说明了一些行为准则，PR指南，Issue Reporting 指南，Development Setup 以及 项目结构。通过阅读这些内容，我们可以了解项目如何开始，如何开发以及目录的说明，下面是对重要目录和文件的简单介绍，这些内容你都可以去自己阅读获取：

```
├── build --------------------------------- 构建相关的文件，一般情况下
├── dist ---------------------------------- 构建后文件的输出目录
├── examples ------------------------------ 存放一些使用Vue开发的应用
├── flow ---------------------------------- 类型声明，使用开源项目 [F
├── package.json -------------------------- 不解释
├── test ---------------------------------- 包含所有测试文件
├── src ----------------------------------- 这个是我们最应该关注的目录
│   ├── entries --------------------------- 包含了不同的构建或包的入口
│   │   ├── web-runtime.js ---------------- 运行时构建的入口，输出 di
│   │   ├── web-runtime-with-compiler.js -- 独立构建版本的入口，输出
│   │   ├── web-compiler.js --------------- vue-template-compiler 包
│   │   ├── web-server-renderer.js -------- vue-server-renderer 包的
│   ├── compiler -------------------------- 编译器代码的存放目录，将
│   │   ├── parser ------------------------ 存放将模板字符串转换成元素
│   │   ├── codegen ----------------------- 存放从抽象语法树(AST)生成
│   │   ├── optimizer.js ------------------ 分析静态树，优化vdom渲染
│   ├── core ------------------------------ 存放通用的，平台无关的代码
│   │   ├── observer ---------------------- 反应系统，包含数据观测的核
│   │   ├── vdom -------------------------- 包含虚拟DOM创建(creation
│   │   ├── instance ---------------------- 包含Vue构造函数设计相关的
│   │   ├── global-api -------------------- 包含给Vue构造函数挂载全局
│   │   ├── components -------------------- 包含抽象出来的通用组件
│   ├── server ---------------------------- 包含服务端渲染(server-si
│   ├── platforms ------------------------- 包含平台特有的相关代码
│   ├── sfc ------------------------------- 包含单文件组件(.vue文件)
│   ├── shared ---------------------------- 包含整个代码库通用的代码
```

大概了解了重要目录和文件之后，我们就可以查看 Development Setup 中的常用命令部分，来了解如何开始这个项目了，我们可以看到这样的介绍：

```
# watch and auto re-build dist/vue.js
$ npm run dev

# watch and auto re-run unit tests in Chrome
$ npm run dev:test
```

现在，我们只需要运行 `npm run dev` 即可监测文件变化并自动重新构建输出 dist/vue.js，然后运行 `npm run dev:test` 来测试。不过为了方便，我会在 `examples` 目录新建一个例子，然后引用 dist/vue.js 这样，我们可以直接拿这个例子一边改Vue源码一边看自己写的代码想怎么玩怎么玩。

二、看源码的小提示

在真正步入源码世界之前，我想简单说一说看源码的技巧：

注重大体框架，从宏观到微观

当你看一个项目代码的时候，最好是能找到一条主线，先把大体流程结构摸清楚，再深入到细节，逐项击破，拿Vue举个栗子：假如你已经知道Vue中数据状态改变后会采用virtual DOM的方式更新DOM，这个时候，如果你不了解virtual DOM，那么听我一句“暂且不要去研究内部具体实现，因为这会是你丧失主线”，而你仅仅需要知道virtual DOM分为三个步骤：

> 一、createElement(): 用 JavaScript对象(虚拟树) 描述 真实DOM对象(真实树)
> 二、diff(oldNode, newNode) : 对比新旧两个虚拟树的区别，收集差异
> 三、patch() : 将差异应用到真实DOM树

有的时候 第二步 可能与 第三步 合并成一步(Vue 中的patch就是这样)，除此之外，还比如 `src/compiler/codegen` 内的代码，可能你不知道他写了什么，直接去看它会让你很痛苦，但是你只需要知道 codegen 是用来将抽象语法树(AST)生成render函数的就OK了，也就是生成类似下面这样的代码：

```
function anonymous() {
        with(this){return _c('p',{attrs:{"id":"app"}},[_v("\n      "
}
```

当我们知道了一个东西存在，且知道它存在的目的，那么我们就很容易抓住这条主线，这个系列的第一篇文章就是围绕大体主线展开的。了解大体之后，我们就知道了每部分内容都是做什么的，比如 codegen 是生成类似上面贴出的代码所示的函数的，那么再去看codegen下的代码时，目的性就会更强，就更容易理解。

三、Vue 的构造函数是什么样的

balabala一大堆，开始来干货吧。我们要做的第一件事就是搞清楚 Vue 构造函数到底是什么样子的。

我们知道，我们要使用 `new` 操作符来调用 `Vue`，那么也就是说 `Vue` 应该是一个构造函数，所以我们第一件要做的事儿就是把构造函数先扒的一清二楚，如何寻找 `Vue` 构造函数呢？当然是从 entry 开始啦，还记的我们运行 `npm run dev` 命令后，会输出 `dist/vue.js` 吗，那么我们就去看看 `npm run dev` 干了什么：

```
"dev": "TARGET=web-full-dev rollup -w -c build/config.js",
```

首先将 TARGET 得值设置为 ‘web-full-dev’，然后，然后，然后如果你不了解 rollup 就应该简单去看一下啦……，简单的说就是一个JavaScript模块打包器，你可以把它简

单的理解为和 webpack 一样，只不过它有他的优势，比如 Tree-shaking (webpack2也有)，但同样，在某些场景它也有他的劣势。。。废话不多说，其中 `-w` 就是 watch，`-c` 就是指定配置文件为 `build/config.js` ，我们打开这个配置文件看一看：

```
// 引入依赖，定义 banner
...

// builds 对象
const builds = {
        ...
        // Runtime+compiler development build (Browser)
        'web-full-dev': {
            entry: path.resolve(__dirname, '../src/entries/web-runt
            dest: path.resolve(__dirname, '../dist/vue.js'),
            format: 'umd',
            env: 'development',
            alias: { he: './entity-decoder' },
            banner
        },
        ...
}

// 生成配置的方法
function genConfig(opts){
        ...
}

if (process.env.TARGET) {
  module.exports = genConfig(builds[process.env.TARGET])
} else {
  exports.getBuild = name => genConfig(builds[name])
  exports.getAllBuilds = () => Object.keys(builds).map(name => genC
}
```

上面的代码是简化过的，当我们运行 `npm run dev` 的时候 `process.env.TARGET` 的值等于 'web-full-dev'，所以

```
module.exports = genConfig(builds[process.env.TARGET])
```

这句代码相当于：

```
module.exports = genConfig({
    entry: path.resolve(__dirname, '../src/entries/web-runtime-with-
```

```
    dest: path.resolve(__dirname, '../dist/vue.js'),
    format: 'umd',
    env: 'development',
    alias: { he: './entity-decoder' },
    banner
})
```

最终，genConfig 函数返回一个 config 对象，这个config对象就是Rollup的配置对象。那么我们就不难看到，入口文件是：

```
src/entries/web-runtime-with-compiler.js
```

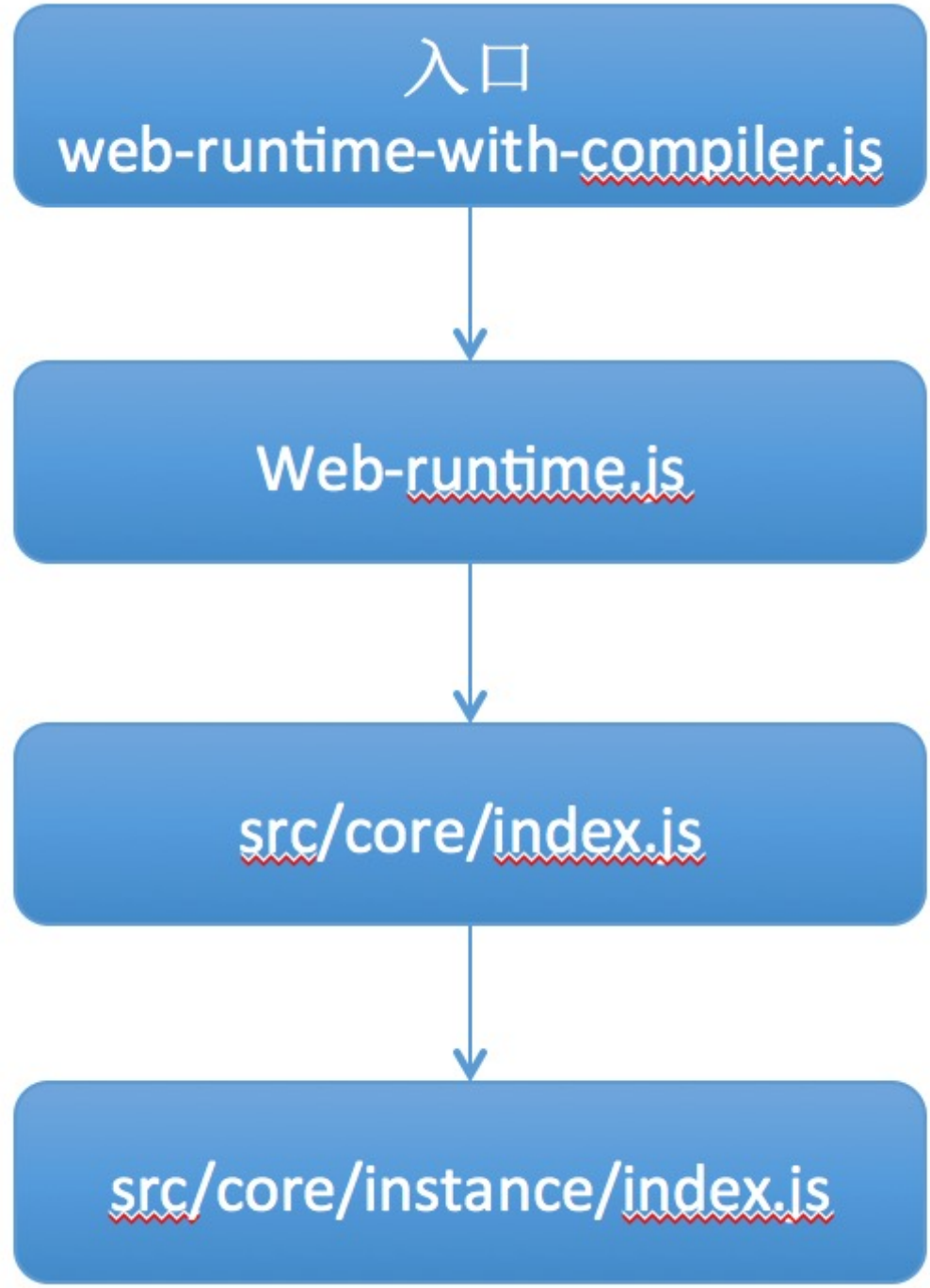
我们打开这个文件，不要忘了我们的主题，我们在寻找Vue构造函数，所以当我们看到这个文件的第一行代码是：

```
import Vue from './web-runtime'
```

这个时候，你就应该知道，这个文件暂时与你无缘，你应该打开 `web-runtime.js` 文件，不过当你打开这个文件时，你发现第一行是这样的：

```
import Vue from 'core/index'
```

依照此思路，最终我们寻找到Vue构造函数的位置应该是在 `src/core/instance/index.js` 文件中，其实我们猜也猜得到，上面介绍目录的时候说过：instance 是存放Vue构造函数设计相关代码的目录。总结一下，我们寻找的过程是这样的：



我们回头看一看 `src/core/instance/index.js` 文件，很简单：

```
import { initMixin } from './init'
import { stateMixin } from './state'
import { renderMixin } from './render'
import { eventsMixin } from './events'
import { lifecycleMixin } from './lifecycle'
import { warn } from '../util/index'

function Vue (options) {
  if (process.env.NODE_ENV !== 'production' &&
    !(this instanceof Vue)) {
    warn('Vue is a constructor and should be called with the `new`
  }
```

```
  this._init(options)
}

initMixin(Vue)
stateMixin(Vue)
eventsMixin(Vue)
lifecycleMixin(Vue)
renderMixin(Vue)

export default Vue
```

引入依赖，定义 Vue 构造函数，然后以Vue构造函数为参数，调用了五个方法，最后导出 Vue。这五个方法分别来自五个文件：`init.js` `state.js` `render.js` `events.js` 以及 `lifecycle.js`。

打开这五个文件，找到相应的方法，你会发现，这些方法的作用，就是在 Vue 的原型 prototype 上挂载方法或属性，经历了这五个方法后的Vue会变成这样：

```
// initMixin(Vue)       src/core/instance/init.js ****************
Vue.prototype._init = function (options?: Object) {}

// stateMixin(Vue)      src/core/instance/state.js ***************
Vue.prototype.$data
Vue.prototype.$set = set
Vue.prototype.$delete = del
Vue.prototype.$watch = function(){}

// renderMixin(Vue)     src/core/instance/render.js **************
Vue.prototype.$nextTick = function (fn: Function) {}
Vue.prototype._render = function (): VNode {}
Vue.prototype._s = _toString
Vue.prototype._v = createTextVNode
Vue.prototype._n = toNumber
Vue.prototype._e = createEmptyVNode
Vue.prototype._q = looseEqual
Vue.prototype._i = looseIndexOf
Vue.prototype._m = function(){}
Vue.prototype._o = function(){}
Vue.prototype._f = function resolveFilter (id) {}
Vue.prototype._l = function(){}
Vue.prototype._t = function(){}
Vue.prototype._b = function(){}
Vue.prototype._k = function(){}

// eventsMixin(Vue)     src/core/instance/events.js **************
Vue.prototype.$on = function (event: string, fn: Function): Compone
```

```
Vue.prototype.$once = function (event: string, fn: Function): Compo
Vue.prototype.$off = function (event?: string, fn?: Function): Comp
Vue.prototype.$emit = function (event: string): Component {}

// lifecycleMixin(Vue)  src/core/instance/lifecycle.js ************
Vue.prototype._mount = function(){}
Vue.prototype._update = function (vnode: VNode, hydrating?: boolean
Vue.prototype._updateFromParent = function(){}
Vue.prototype.$forceUpdate = function () {}
Vue.prototype.$destroy = function () {}
```

这样就结束了吗？并没有，根据我们之前寻找 Vue 的路线，这只是刚刚开始，我们追溯路线往回走，那么下一个处理 Vue 构造函数的应该是 `src/core/index.js` 文件，我们打开它：

```
import Vue from './instance/index'
import { initGlobalAPI } from './global-api/index'
import { isServerRendering } from 'core/util/env'

initGlobalAPI(Vue)

Object.defineProperty(Vue.prototype, '$isServer', {
  get: isServerRendering
})

Vue.version = '__VERSION__'

export default Vue
```

这个文件也很简单，从 instance/index 中导入已经在原型上挂载了方法和属性后的 Vue，然后导入 `initGlobalAPI` 和 `isServerRendering`，之后将Vue作为参数传给 `initGlobalAPI`，最后又在 `Vue.prototype` 上挂载了 `$isServer`，在 `Vue` 上挂载了 `version` 属性。

`initGlobalAPI` 的作用是在 `Vue` 构造函数上挂载静态属性和方法，`Vue` 在经过 `initGlobalAPI` 之后，会变成这样：

```
// src/core/index.js / src/core/global-api/index.js
Vue.config
Vue.util = util
Vue.set = set
Vue.delete = del
Vue.nextTick = util.nextTick
Vue.options = {
    components: {
```

```
        KeepAlive
    },
    directives: {},
    filters: {},
    _base: Vue
}
Vue.use
Vue.mixin
Vue.cid = 0
Vue.extend
Vue.component = function(){}
Vue.directive = function(){}
Vue.filter = function(){}

Vue.prototype.$isServer
Vue.version = '__VERSION__'
```

其中，稍微复杂一点的就是 `Vue.options`，大家稍微分析分析就会知道他的确长成那个样子。下一个就是 `web-runtime.js` 文件了，`web-runtime.js` 文件主要做了三件事儿：

> 1、覆盖 `Vue.config` 的属性，将其设置为平台特有的一些方法
> 2、`Vue.options.directives` 和 `Vue.options.components` 安装平台特有的指令和组件
> 3、在 `Vue.prototype` 上定义 `__patch__` 和 `$mount`

经过 `web-runtime.js` 文件之后，`Vue` 变成下面这个样子：

```
// 安装平台特定的utils
Vue.config.isUnknownElement = isUnknownElement
Vue.config.isReservedTag = isReservedTag
Vue.config.getTagNamespace = getTagNamespace
Vue.config.mustUseProp = mustUseProp
// 安装平台特定的 指令 和 组件
Vue.options = {
    components: {
        KeepAlive,
        Transition,
        TransitionGroup
    },
    directives: {
        model,
        show
    },
    filters: {},
```

```
    _base: Vue
}
Vue.prototype.__patch__
Vue.prototype.$mount
```

这里大家要注意的是 `Vue.options` 的变化。另外这里的 `$mount` 方法很简单：

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

首先根据是否是浏览器环境决定要不要 `query(el)` 获取元素，然后将 `el` 作为参数传递给 `this._mount()` 。

最后一个处理 Vue 的文件就是入口文件 `web-runtime-with-compiler.js` 了，该文件做了两件事：

1、缓存来自 `web-runtime.js` 文件的 `$mount` 函数

```
const mount = Vue.prototype.$mount
```

然后覆盖覆盖了 `Vue.prototype.$mount`

2、在 Vue 上挂载 `compile`

```
Vue.compile = compileToFunctions
```

compileToFunctions 函数的作用，就是将模板 `template` 编译为render函数。

至此，我们算是还原了 Vue 构造函数，总结一下：

> 1、`Vue.prototype` 下的属性和方法的挂载主要是在 `src/core/instance` 目录中的代码处理的
> 2、`Vue` 下的静态属性和方法的挂载主要是在 `src/core/global-api` 目录下的代码处理的
> 3、`web-runtime.js` 主要是添加web平台特有的配置、组件和指令，`web-runtime-with-compiler.js` 给Vue的 `$mount` 方法添加 `compiler` 编译器，支持 `template` 。

四、一个贯穿始终的例子

在了解了 `Vue` 构造函数的设计之后，接下来，我们一个贯穿始终的例子就要登场了，掌声有请：

```
let v = new Vue({
        el: '#app',
        data: {
                a: 1,
                b: [1, 2, 3]
        }
})
```

好吧，我承认这段代码你家没满月的孩子都会写了。这段代码就是我们贯穿始终的例子，它就是这篇文章的主线，在后续的讲解中，都会以这段代码为例，当讲到必要的地方，会为其添加选项，比如讲计算属性的时候当然要加上一个 `computed` 属性了。不过在最开始，我只传递了两个选项 `el` 以及 `data`，“我们看看接下来会发生什么，让我们拭目以待“ —- NBA球星在接受采访时最喜欢说这句话。

当我们按照例子那样编码使用Vue的时候，Vue都做了什么？

想要知道Vue都干了什么，我们就要找到 Vue 初始化程序，查看 Vue 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` k
  }
  this._init(options)
}
```

我们发现，`_init()` 方法就是Vue调用的第一个方法，然后将我们的参数 `options` 透传了过去。在调用 `_init()` 之前，还做了一个安全模式的处理，告诉开发者必须使用 `new` 操作符调用 Vue。根据之前我们的整理，`_init()` 方法应该是在 `src/core/instance/init.js` 文件中定义的，我们打开这个文件查看 `_init()` 方法：

```
Vue.prototype._init = function (options?: Object) {
  const vm: Component = this
  // a uid
  vm._uid = uid++
  // a flag to avoid this being observed
  vm._isVue = true
  // merge options
```

```
    if (options && options._isComponent) {
      // optimize internal component instantiation
      // since dynamic options merging is pretty slow, and none of th
      // 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
    initLifecycle(vm)
    initEvents(vm)
    callHook(vm, 'beforeCreate')
    initState(vm)
    callHook(vm, 'created')
    initRender(vm)
  }
```

`_init()` 方法在一开始的时候，在 `this` 对象上定义了两个属性：`_uid` 和 `_isVue`，然后判断有没有定义 `options._isComponent`，在使用 Vue 开发项目的时候，我们是不会使用 `_isComponent` 选项的，这个选项是 Vue 内部使用的，按照本节开头的例子，这里会走 `else` 分支，也就是这段代码：

```
vm.$options = mergeOptions(
  resolveConstructorOptions(vm.constructor),
  options || {},
  vm
)
```

这样 `Vue` 第一步所做的事情就来了：*使用策略对象合并参数选项*

可以发现，Vue使用 `mergeOptions` 来处理我们调用Vue时传入的参数选项(options)，然后将返回值赋值给 `this.$options` (vm === this)，传给

`mergeOptions` 方法三个参数，我们分别来看一看，首先是：`resolveConstructorOptions(vm.constructor)`，我们查看一下这个方法：

```
export function resolveConstructorOptions (Ctor: Class<Component>)
  let options = Ctor.options
  if (Ctor.super) {
    const superOptions = Ctor.super.options
    const cachedSuperOptions = Ctor.superOptions
    const extendOptions = Ctor.extendOptions
    if (superOptions !== cachedSuperOptions) {
      // super option changed
      Ctor.superOptions = superOptions
      extendOptions.render = options.render
      extendOptions.staticRenderFns = options.staticRenderFns
      extendOptions._scopeId = options._scopeId
      options = Ctor.options = mergeOptions(superOptions, extendOpt
      if (options.name) {
        options.components[options.name] = Ctor
      }
    }
  }
  return options
}
```

这个方法接收一个参数 `Ctor`，通过传入的 `vm.constructor` 我们可以知道，其实就是 `Vue` 构造函数本身。所以下面这句代码：

```
let options = Ctor.options
```

相当于：

```
let options = Vue.options
```

大家还记得 `Vue.options` 吗？在寻找Vue构造函数一节里，我们整理了 `Vue.options` 应该长成下面这个样子：

```
Vue.options = {
    components: {
        KeepAlive,
        Transition,
        TransitionGroup
    },
    directives: {
        model,
```

```
        show
    },
    filters: {},
    _base: Vue
}
```

之后判断是否定义了 `Vue.super` ，这个是用来处理继承的，我们后续再讲，在本例中， `resolveConstructorOptions` 方法直接返回了 `Vue.options` 。也就是说，传递给 `mergeOptions` 方法的第一个参数就是 `Vue.options` 。

传给 `mergeOptions` 方法的第二个参数是我们调用Vue构造函数时的参数选项，第三个参数是 `vm` 也就是 `this` 对象，按照本节开头的例子那样使用 Vue，最终运行的代码应该如下：

```
vm.$options = mergeOptions(
        // Vue.options
    {
     components: {
        KeepAlive,
        Transition,
        TransitionGroup
     },
     directives: {
        model,
        show
     },
     filters: {},
     _base: Vue
},
// 调用Vue构造函数时传入的参数选项 options
    {
        el: '#app',
        data: {
                a: 1,
                b: [1, 2, 3]
        }
    },
    // this
    vm
 )
```

了解了这些，我们就可以看看 `mergeOptions` 到底做了些什么了，根据引用寻找到 `mergeOptions` 应该是在 `src/core/util/options.js` 文件中定义的。这个文件第一次看可能会头大，下面是我处理后的简略展示，大家看上去应该更容易理解了：

```
// 1、引用依赖
import Vue from '../instance/index'
其他引用...

// 2、合并父子选项值为最终值的策略对象，此时 strats 是一个空对象，因为 c
const strats = config.optionMergeStrategies
// 3、在 strats 对象上定义与参数选项名称相同的方法
strats.el =
strats.propsData = function (parent, child, vm, key){}
strats.data = function (parentVal, childVal, vm)

config._lifecycleHooks.forEach(hook => {
  strats[hook] = mergeHook
})

config._assetTypes.forEach(function (type) {
  strats[type + 's'] = mergeAssets
})

strats.watch = function (parentVal, childVal)

strats.props =
strats.methods =
strats.computed = function (parentVal: ?Object, childVal: ?Object)
// 默认的合并策略，如果有 `childVal` 则返回 `childVal` 没有则返回 `pare
const defaultStrat = function (parentVal: any, childVal: any): any
  return childVal === undefined
    ? parentVal
    : childVal
}

// 4、mergeOptions 中根据参数选项调用同名的策略方法进行合并处理
export function mergeOptions (
  parent: Object,
  child: Object,
  vm?: Component
): Object {

  // 其他代码
  ...

  const options = {}
  let key
  for (key in parent) {
    mergeField(key)
  }
  for (key in child) {
```

```
48      if (!hasOwn(parent, key)) {
49        mergeField(key)
50      }
51    }
52    function mergeField (key) {
53      const strat = strats[key] || defaultStrat
54      options[key] = strat(parent[key], child[key], vm, key)
55    }
56    return options
57
58  }
```

上面的代码中，我省略了一些工具函数，例如 `mergeHook` 和 `mergeAssets` 等等，唯一需要注意的是这段代码：

```
1  config._lifecycleHooks.forEach(hook => {
2    strats[hook] = mergeHook
3  })
4
5  config._assetTypes.forEach(function (type) {
6    strats[type + 's'] = mergeAssets
7  })
```

`config` 对象引用自 `src/core/config.js` 文件，最终的结果就是在 `strats` 下添加了相应的生命周期选项的合并策略函数为 `mergeHook`，添加指令(directives)、组件(components)、过滤器(filters)等选项的合并策略函数为 `mergeAssets`。

这样看来就清晰多了，拿我们贯穿本文的例子来说：

```
1  let v = new Vue({
2          el: '#app',
3          data: {
4                  a: 1,
5                  b: [1, 2, 3]
6          }
7  })
```

其中 `el` 选项会使用 `defaultStrat` 默认策略函数处理，`data` 选项则会使用 `strats.data` 策略函数处理，并且根据 `strats.data` 中的逻辑，`strats.data` 方法最终会返回一个函数：`mergedInstanceDataFn`。

这里就不详细的讲解每一个策略函数的内容了，后续都会讲到，这里我们还是抓住主线理清思路为主，只需要知道Vue在处理选项的时候，使用了一个策略对象对父子选

项进行合并。并将最终的值赋值给实例下的 `$options` 属性即：`this.$options`，那么我们继续查看 `_init()` 方法在合并完选项之后，又做了什么：

合并完选项之后，Vue 第二部做的事情就来了：*初始化工作与Vue实例对象的设计*

前面讲了 Vue 构造函数的设计，并且整理了 *Vue原型属性与方法* 和 *Vue静态属性与方法*，而 Vue 实例对象就是通过构造函数创造出来的，让我们来看一看 Vue 实例对象是如何设计的，下面的代码是 `_init()` 方法合并完选项之后的代码：

```
/* istanbul ignore else */
    if (process.env.NODE_ENV !== 'production') {
      initProxy(vm)
    } else {
      vm._renderProxy = vm
    }

    // expose real self
vm._self = vm
    initLifecycle(vm)
    initEvents(vm)
    callHook(vm, 'beforeCreate')
    initState(vm)
    callHook(vm, 'created')
    initRender(vm)
```

根据上面的代码，在生产环境下会为实例添加两个属性，并且属性值都为实例本身：

```
vm._renderProxy = vm
vm._self = vm
```

然后，调用了四个 `init*` 方法分别为：`initLifecycle`、`initEvents`、`initState`、`initRender`，且在 `initState` 前后分别回调了生命周期钩子 `beforeCreate` 和 `created`，而 `initRender` 是在 `created` 钩子执行之后执行的，看到这里，也就明白了为什么 created 的时候不能操作DOM了。因为这个时候还没有渲染真正的DOM元素到文档中。`created` 仅仅代表数据状态的初始化完成。

根据四个 `init*` 方法的引用关系打开对应的文件查看对应的方法，我们发现，这些方法是在处理Vue实例对象，以及做一些初始化的工作，类似整理Vue构造函数一样，我同样针对Vue实例做了属性和方法的整理，如下：

```
// 在 Vue.prototype._init 中添加的属性                    ********************
this._uid = uid++
this._isVue = true
this.$options = {
    components,
```

```
    directives,
    filters,
    _base,
    el,
    data: mergedInstanceDataFn()
}
this._renderProxy = this
this._self = this

// 在 initLifecycle 中添加的属性          ***************************
this.$parent = parent
this.$root = parent ? parent.$root : this

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

this._watcher = null
this._inactive = false
this._isMounted = false
this._isDestroyed = false
this._isBeingDestroyed = false

// 在 initEvents          中添加的属性             ******************
this._events = {}
this._updateListeners = function(){}

// 在 initState 中添加的属性              ***************************
this._watchers = []
    // initData
    this._data

// 在 initRender          中添加的属性     ***************************
this.$vnode = null // the placeholder node in parent tree
this._vnode = null // the root of the child tree
this._staticTrees = null
this.$slots
this.$scopedSlots
this._c
this.$createElement
```

以上就是一个Vue实例所包含的属性和方法，除此之外要注意的是，在 `initEvents` 中除了添加属性之外，如果有 `vm.$options._parentListeners` 还要调用 `vm._updateListeners()` 方法，在 `initState` 中又调用了一些其他init方法，如下：

```
export function initState (vm: Component) {
```

```
  vm._watchers = []
  initProps(vm)
  initMethods(vm)
  initData(vm)
  initComputed(vm)
  initWatch(vm)
}
```

最后在 `initRender` 中如果有 `vm.$options.el` 还要调用 `vm.$mount(vm.$options.el)`，如下：

```
if (vm.$options.el) {
  vm.$mount(vm.$options.el)
}
```

这就是为什么如果不传递 `el` 选项就需要手动 mount 的原因了。

那么我们依照我们本节开头的的例子，以及初始化的先后顺序来逐一看一看都发生了什么。我们将 `initState` 中的 `init*` 方法展开来看，执行顺序应该是这样的（从上到下的顺序执行）：

```
initLifecycle(vm)
initEvents(vm)
callHook(vm, 'beforeCreate')
initProps(vm)
initMethods(vm)
initData(vm)
initComputed(vm)
initWatch(vm)
callHook(vm, 'created')
initRender(vm)
```

首先是 `initLifecycle`，这个函数的作用就是在实例上添加一些属性，然后是 `initEvents`，由于 `vm.$options._parentListeners` 的值为 `undefined` 所以也仅仅是在实例上添加属性，`vm._updateListeners(listeners)` 并不会执行，由于我们只传递了 `el` 和 `data`，所以 `initProps`、`initMethods`、`initComputed`、`initWatch` 这四个方法什么都不会做，只有 `initData` 会执行。最后是 `initRender`，除了在实例上添加一些属性外，由于我们传递了 `el` 选项，所以会执行 `vm.$mount(vm.$options.el)`。

综上所述：按照我们的例子那样写，初始化工作只包含两个主要内容即：`initData` 和 `initRender`。

五、通过 initData 看Vue的数据响应系统

Vue的数据响应系统包含三个部分：`Observer`、`Dep`、`Watcher`。关于数据响应系统的内容真的已经被文章讲烂了，所以我就简单的说一下，力求大家能理解就ok，我们还是先看一下 `initData` 中的代码：

```
function initData (vm: Component) {
  let data = vm.$options.data
  data = vm._data = typeof data === 'function'
    ? data.call(vm)
    : data || {}
  if (!isPlainObject(data)) {
    data = {}
    process.env.NODE_ENV !== 'production' && warn(
      'data functions should return an object:\n' +
      'https://vuejs.org/v2/guide/components.html#data-Must-Be-a-Fu
      vm
    )
  }
  // proxy data on instance
  const keys = Object.keys(data)
  const props = vm.$options.props
  let i = keys.length
  while (i--) {
    if (props && hasOwn(props, keys[i])) {
      process.env.NODE_ENV !== 'production' && warn(
        `The data property "${keys[i]}" is already declared as a pr
        `Use prop default value instead.`,
        vm
      )
    } else {
      proxy(vm, keys[i])
    }
  }
  // observe data
  observe(data)
  data.__ob__ && data.__ob__.vmCount++
}
```

首先，先拿到 data 数据：`let data = vm.$options.data`，大家还记得此时 `vm.$options.data` 的值应该是通过 `mergeOptions` 合并处理后的 `mergedInstanceDataFn` 函数吗？所以在得到 data 后，它又判断了 data 的数据类型是不是 'function'，最终的结果是：data 还是我们传入的数据选项的 data，即：

```
data: {
        a: 1,
```

```
        b: [1, 2, 3]
}
```

然后在实例对象上定义 `_data` 属性，该属性与 `data` 是相同的引用。

然后是一个 `while` 循环，循环的目的是在实例对象上对数据进行代理，这样我们就能通过 `this.a` 来访问 `data.a` 了，代码的处理是在 `proxy` 函数中，该函数非常简单，仅仅是在实例对象上设置与 `data` 属性同名的访问器属性，然后使用 `_data` 做数据劫持，如下：

```
function proxy (vm: Component, key: string) {
  if (!isReserved(key)) {
    Object.defineProperty(vm, key, {
      configurable: true,
      enumerable: true,
      get: function proxyGetter () {
        return vm._data[key]
      },
      set: function proxySetter (val) {
        vm._data[key] = val
      }
    })
  }
}
```

做完数据的代理，就正式进入响应系统，

```
observe(data)
```

我们说过，数据响应系统主要包含三部分：`Observer`、`Dep`、`Watcher`，代码分别存放在：`observer/index.js`、`observer/dep.js` 以及 `observer/watcher.js` 文件中，这回我们换一种方式，我们先不看其源码，大家先跟着我的思路来思考，最后回头再去看代码，你会有一种：”奥，不过如此“的感觉。

假如，我们有如下代码：

```
var data = {
    a: 1,
    b: {
        c: 2
    }
}

observer(data)

```

```
new Watch('a', () => {
    alert(9)
})
new Watch('a', () => {
    alert(90)
})
new Watch('b.c', () => {
    alert(80)
})
```

这段代码目的是，首先定义一个数据对象 `data`，然后通过 observer 对其进行观测，之后定义了三个观察者，当数据有变化时，执行相应的方法，这个功能使用Vue的实现原来要如何去实现？其实就是在问 `observer` 怎么写？ `Watch` 构造函数又怎么写？接下来我们逐一实现。

首先，observer 的作用是：将数据对象data的属性转换为访问器属性：

```
class Observer {
    constructor (data) {
        this.walk(data)
    }
    walk (data) {
        // 遍历 data 对象属性，调用 defineReactive 方法
        let keys = Object.keys(data)
        for(let i = 0; i < keys.length; i++){
            defineReactive(data, keys[i], data[keys[i]])
        }
    }
}

// defineReactive方法仅仅将data的属性转换为访问器属性
function defineReactive (data, key, val) {
        // 递归观测子属性
    observer(val)

    Object.defineProperty(data, key, {
        enumerable: true,
        configurable: true,
        get: function () {
            return val
        },
        set: function (newVal) {
            if(val === newVal){
                return
            }
            // 对新值进行观测
```

```
            observer(newVal)
        }
    })
}

// observer 方法首先判断data是不是纯JavaScript对象，如果是，调用 Observ
function observer (data) {
    if(Object.prototype.toString.call(data) !== '[object Object]')
        return
    }
    new Observer(data)
}
```

上面的代码中，我们定义了 observer 方法，该方法检测了数据data是不是纯JavaScript对象，如果是就调用 `Observer` 类，并将 `data` 作为参数透传。在 `Observer` 类中，我们使用 `walk` 方法对数据data的属性循环调用 `defineReactive` 方法， `defineReactive` 方法很简单，仅仅是将数据data的属性转为访问器属性，并对数据进行递归观测，否则只能观测数据data的直属子属性。这样我们的第一步工作就完成了，当我们修改或者获取data属性值的时候，通过 `get` 和 `set` 即能获取到通知。

我们继续往下看，来看一下 `Watch`：

```
new Watch('a', () => {
    alert(9)
})
```

现在的问题是， `Watch` 要怎么和 `observer` 关联？？？？？？？我们看看 `Watch` 它知道些什么，通过上面调用 `Watch` 的方式，传递给 `Watch` 两个参数，一个是 'a' 我们可以称其为表达式，另外一个是回调函数。所以我们目前只能写出这样的代码：

```
class Watch {
    constructor (exp, fn) {
        this.exp = exp
        this.fn = fn
    }
}
```

那么要怎么关联呢，大家看下面的代码会发生什么：

```
class Watch {
    constructor (exp, fn) {
        this.exp = exp
```

```
        this.fn = fn
        data[exp]
    }
}
```

多了一句 `data[exp]`，这句话是在干什么？是不是在获取 `data` 下某个属性的值，比如 exp 为 'a' 的话，那么 `data[exp]` 就相当于在获取 `data.a` 的值，那这会放生什么？大家不要忘了，此时数据 `data` 下的属性已经是访问器属性了，所以这么做的结果会直接触发对应属性的 `get` 函数，这样我们就成功的和 `observer` 产生了关联，但这样还不够，我们还是没有达到目的，不过我们已经无限接近了，我们继续思考看一下可不可以这样：

> 既然在 `Watch` 中对表达式求值，能够触发 `observer` 的 `get`，那么可不可以在 `get` 中收集 `Watch` 中函数呢？

答案是可以的，不过这个时候我们就需要 `Dep` 出场了，它是一个依赖收集器。我们的思路是：`data` 下的每一个属性都有一个唯一的 `Dep` 对象，在 `get` 中收集仅针对该属性的依赖，然后在 `set` 方法中触发所有收集的依赖，这样就搞定了，看如下代码：

```
class Dep {
    constructor () {
        this.subs = []
    }
    addSub () {
        this.subs.push(Dep.target)
    }
    notify () {
        for(let i = 0; i < this.subs.length; i++){
            this.subs[i].fn()
        }
    }
}
Dep.target = null
function pushTarget(watch){
    Dep.target = watch
}

class Watch {
    constructor (exp, fn) {
        this.exp = exp
        this.fn = fn
        pushTarget(this)
```

```
        data[exp]
    }
}
```

上面的代码中，我们在 `Watch` 中增加了 `pushTarget(this)`，可以发现，这句代码的作用是将 `Dep.target` 的值设置为该Watch对象。在 `pushTarget` 之后我们才对表达式进行求值，接着，我们修改 `defineReactive` 代码如下

```
function defineReactive (data, key, val) {
    observer(val)
    let dep = new Dep()          // 新增
    Object.defineProperty(data, key, {
        enumerable: true,
        configurable: true,
        get: function () {
            dep.addSub()         // 新增
            return val
        },
        set: function (newVal) {
            if(val === newVal){
                return
            }
            observer(newVal)
            dep.notify()         // 新增
        }
    })
}
```

如标注，新增了三句代码，我们知道，`Watch` 中对表达式求值会触发 `get` 方法，我们在 `get` 方法中调用了 `dep.addSub`，也就执行了这句代码：`this.subs.push(Dep.target)`，由于在这句代码执行之前，`Dep.target` 的值已经被设置为一个 `Watch` 对象了，所以最终结果就是收集了一个 `Watch` 对象，然后在 `set` 方法中我们调用了 `dep.notify`，所以当data属性值变化的时候，就会通过 `dep.notify` 循环调用所有收集的Watch对象中的回调函数：

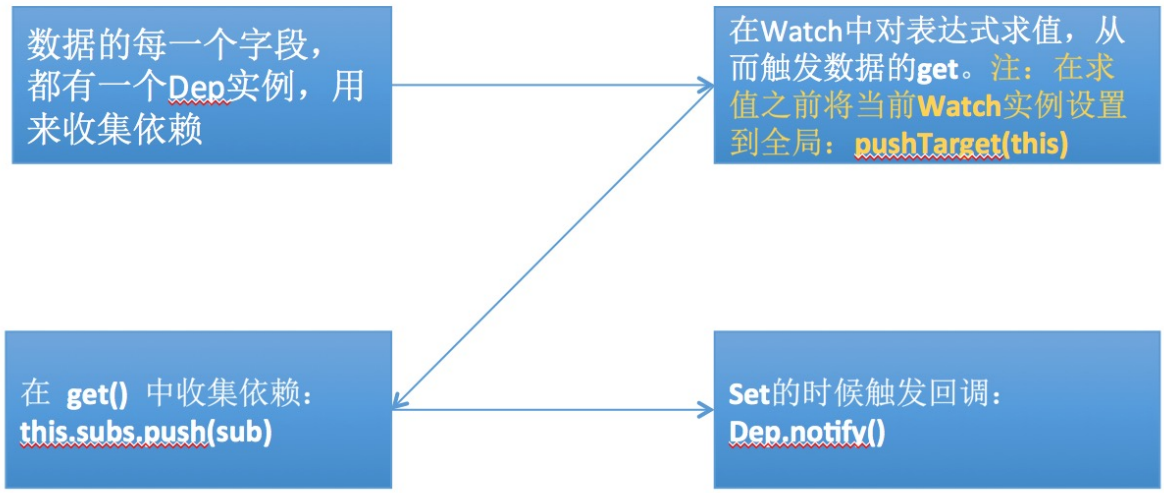
```
notify () {
    for(let i = 0; i < this.subs.length; i++){
        this.subs[i].fn()
    }
}
```

这样 `observer`、`Dep`、`Watch` 三者就联系成为一个有机的整体，实现了我们最初的目标，完整的代码可以戳这里：observer-dep-watch。这里还给大家挖了个坑，因为我们没有处理对数组的观测，由于比较复杂并且这又不是我们讨论的重点，如果

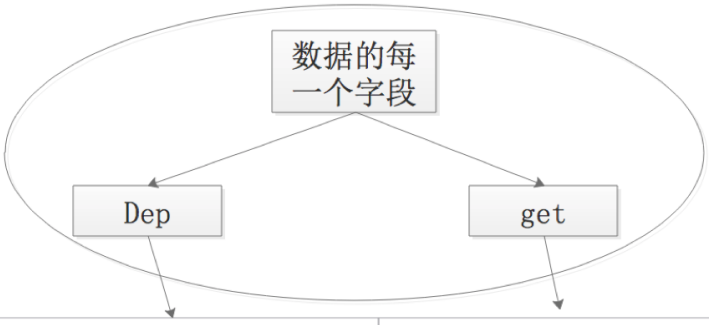
大家想了解可以戳我的这篇文章：JavaScript实现MVVM之我就是想监测一个普通对象的变化，另外，在 Watch 中对表达式求值的时候也只做了直接子属性的求值，所以如果 exp 的值为 'a.b' 的时候，就不可以用了，Vue的做法是使用 `.` 分割表达式字符串为数组，然后遍历一下对其进行求值，大家可以查看其源码。如下：

```
/**
 * Parse simple path.
 */
const bailRE = /[^\w.$]/
export function parsePath (path: string): any {
  if (bailRE.test(path)) {
    return
  } else {
    const segments = path.split('.')
    return function (obj) {
      for (let i = 0; i < segments.length; i++) {
        if (!obj) return
        obj = obj[segments[i]]
      }
      return obj
    }
  }
}
```

Vue 的求值代码是在 `src/core/util/lang.js` 文件中 `parsePath` 函数中实现的。总结一下Vue的依赖收集过程应该是这样的：



实际上，Vue并没有直接在 `get` 中调用 `addSub`，而是调用的 `dep.depend`，目的是将当前的 dep 对象收集到 watch 对象中，如果要完整的流程，应该是这样的：（大家注意数据的每一个字段都拥有自己的 `dep` 对象和 `get` 方法。）



		1、pushTarget()
2、Dep.target = Watcher实例		
		3、对表达式求值
	4、触发 get() 调用 dep.depend()	
5、将当前dep实例添加到watch实例中： Dep.target.addDep(Dep实例)		
		6、将watch实例添加到dep中： dep.addSub(Watcher实例)
7、收集依赖： this.subs.push(sub)		

这样 Vue 就建立了一套数据响应系统，之前我们说过，按照我们的例子那样写，初始化工作只包含两个主要内容即：`initData` 和 `initRender`。现在 `initData` 我们分析完了，接下来看一看 `initRender`

六、通过 initRender 看Vue的 render(渲染) 与 re-render(重新渲染)

在 `initRender` 方法中，因为我们的例子中传递了 `el` 选项，所以下面的代码会执行：

```
if (vm.$options.el) {
  vm.$mount(vm.$options.el)
}
```

这里，调用了 `$mount` 方法，在还原Vue构造函数的时候，我们整理过所有的方法，其中 `$mount` 方法在两个地方出现过：

1、在 `web-runtime.js` 文件中：

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

它的作用是通过 `el` 获取相应的DOM元素，然后调用 `lifecycle.js` 文件中的 `_mount` 方法。

2、在 `web-runtime-with-compiler.js` 文件中：

```
// 缓存了来自 web-runtime.js 的 $mount 方法
const mount = Vue.prototype.$mount
// 重写 $mount 方法
Vue.prototype.$mount = function (
  el?: string | Element,
  hydrating?: boolean
): Component {
  // 根据 el 获取相应的DOM元素
  el = el && query(el)
  // 不允许你将 el 挂载到 html 标签或者 body 标签
  if (el === document.body || el === document.documentElement) {
    process.env.NODE_ENV !== 'production' && warn(
      `Do not mount Vue to <html> or <body> - mount to normal eleme
    )
    return this
  }

  const options = this.$options
  // 如果我们没有写 render 选项，那么就尝试将 template 或者 el 转化为 r
  if (!options.render) {
    let template = options.template
    if (template) {
      if (typeof template === 'string') {
        if (template.charAt(0) === '#') {
          template = idToTemplate(template)
          /* istanbul ignore if */
          if (process.env.NODE_ENV !== 'production' && !template) {
            warn(
              `Template element not found or is empty: ${options.te
              this
            )
          }
        }
      } else if (template.nodeType) {
        template = template.innerHTML
      } else {
        if (process.env.NODE_ENV !== 'production') {
          warn('invalid template option:' + template, this)
        }
        return this
      }
    } else if (el) {
```

```
        template = getOuterHTML(el)
      }
      if (template) {
        const { render, staticRenderFns } = compileToFunctions(templa
          warn,
          shouldDecodeNewlines,
          delimiters: options.delimiters
        }, this)
        options.render = render
        options.staticRenderFns = staticRenderFns
      }
    }
    // 调用已经缓存下来的 web-runtime.js 文件中的 $mount 方法
    return mount.call(this, el, hydrating)
  }
```

分析一下可知 `web-runtime-with-compiler.js` 的逻辑如下：

1、缓存来自 `web-runtime.js` 文件的 `$mount` 方法

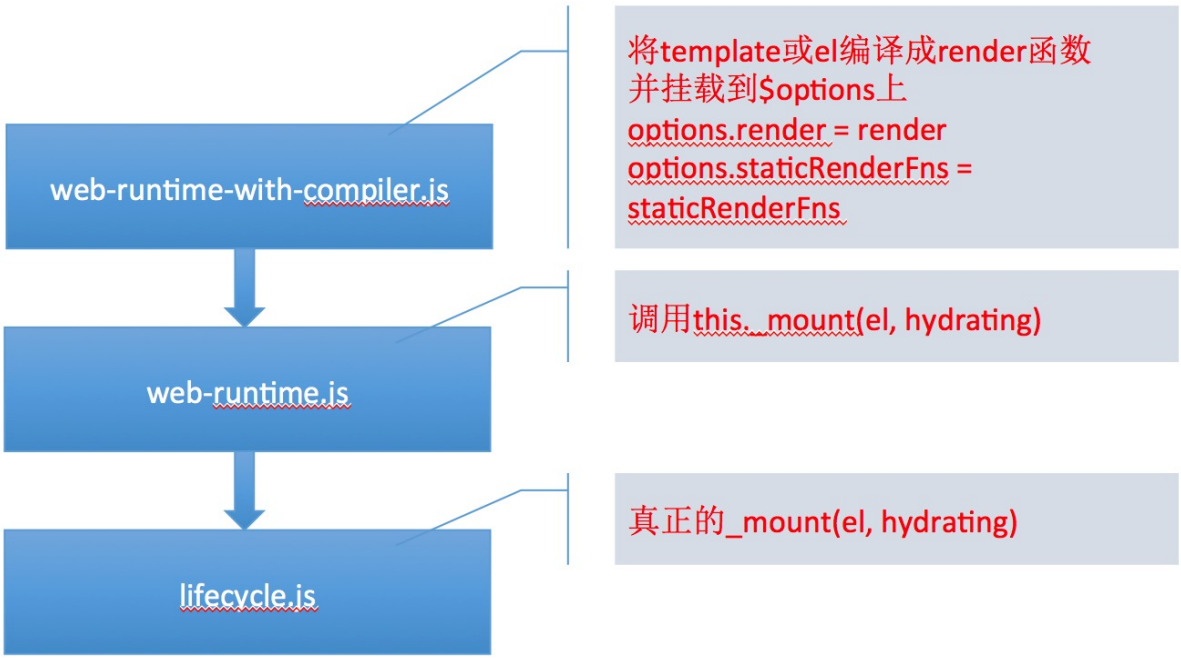
2、判断有没有传递 `render` 选项，如果有直接调用来自 `web-runtime.js` 文件的 $mount 方法

3、如果没有传递 `render` 选项，那么查看有没有 `template` 选项，如果有就使用 `compileToFunctions` 函数根据其内容编译成 `render` 函数

4、如果没有 `template` 选项，那么查看有没有 `el` 选项，如果有就使用 `compileToFunctions` 函数将其内容(template = getOuterHTML(el))编译成 `render` 函数

5、将编译成的 `render` 函数挂载到 `this.$options` 属性下，并调用缓存下来的 `web-runtime.js` 文件中的 $mount 方法

简单的用一张图表示 `mount` 方法的调用关系，从上至下调用：



不过不管怎样，我们发现这些步骤的最终目的是生成 `render` 函数，然后再调用 `lifecycle.js` 文件中的 `_mount` 方法，我们看看这个方法做了什么事情，查看 `_mount` 方法的代码，这是简化过得：

```
Vue.prototype._mount = function (
  el?: Element | void,
  hydrating?: boolean
): Component {
  const vm: Component = this

  // 在Vue实例对象上添加 $el 属性，指向挂载点元素
  vm.$el = el

  // 触发 beforeMount 生命周期钩子
  callHook(vm, 'beforeMount')

  vm._watcher = new Watcher(vm, () => {
    vm._update(vm._render(), hydrating)
  }, noop)

  // 如果是第一次mount则触发 mounted 生命周期钩子
  if (vm.$vnode == null) {
    vm._isMounted = true
    callHook(vm, 'mounted')
  }
  return vm
}
```

上面的代码很简单，该注释的都注释了，唯一需要看的就是这段代码：

```
vm._watcher = new Watcher(vm, () => {
  vm._update(vm._render(), hydrating)
}, noop)
```

看上去很眼熟有没有？我们平时使用Vue都是这样使用 watch的：

```
this.$watch('a', (newVal, oldVal) => {

})
// 或者
this.$watch(function(){
        return this.a + this.b
}, (newVal, oldVal) => {

})
```

第一个参数是 表达式或者函数，第二个参数是回调函数，第三个参数是可选的选项。原理是 `Watch` 内部对表达式求值或者对函数求值从而触发数据的 `get` 方法收集依赖。可是 `_mount` 方法中使用 `Watcher` 的时候第一个参数 `vm` 是什么鬼。我们不妨去看看源码中 `$watch` 函数是如何实现的，根据之前还原Vue构造函数中所整理的内容可知： `$warch` 方法是在 `src/core/instance/state.js` 文件中的 `stateMixin` 方法中定义的，源码如下：

```
Vue.prototype.$watch = function (
  expOrFn: string | Function,
  cb: Function,
  options?: Object
): Function {
  const vm: Component = this
  options = options || {}
  options.user = true
  const watcher = new Watcher(vm, expOrFn, cb, options)
  if (options.immediate) {
    cb.call(vm, watcher.value)
  }
  return function unwatchFn () {
    watcher.teardown()
  }
}
```

我们可以发现， `$warch` 其实是对 `Watcher` 的一个封装，内部的 `Watcher` 的第一个参数实际上也是 `vm` 即：Vue实例对象，这一点我们可以在 `Watcher` 的源码中得到验证，代开 `observer/watcher.js` 文件查看：

```
export default class Watcher {
```

```
2
3     constructor (
4       vm: Component,
5       expOrFn: string | Function,
6       cb: Function,
7       options?: Object = {}
8     ) {
9
10    }
11  }
```

可以发现真正的 `Watcher` 第一个参数实际上就是 `vm` 。第二个参数是表达式或者函数，然后以此类推，所以现在再来看 `_mount` 中的这段代码：

```
1  vm._watcher = new Watcher(vm, () => {
2    vm._update(vm._render(), hydrating)
3  }, noop)
```

忽略第一个参数 `vm` ，也就说， `Watcher` 内部应该对第二个参数求值，也就是运行这个函数：

```
1  () => {
2    vm._update(vm._render(), hydrating)
3  }
```

所以 `vm._render()` 函数被第一个执行，该函数在 `src/core/instance/render.js` 中，该方法中的代码很多，下面是简化过的：

```
1  Vue.prototype._render = function (): VNode {
2    const vm: Component = this
3    // 解构出 $options 中的 render 函数
4    const {
5      render,
6      staticRenderFns,
7      _parentVnode
8    } = vm.$options
9    ...
10
11   let vnode
12   try {
13     // 运行 render 函数
14     vnode = render.call(vm._renderProxy, vm.$createElement)
15   } catch (e) {
16     ...
17   }
18
```

```
    // set parent
    vnode.parent = _parentVnode
    return vnode
  }
```

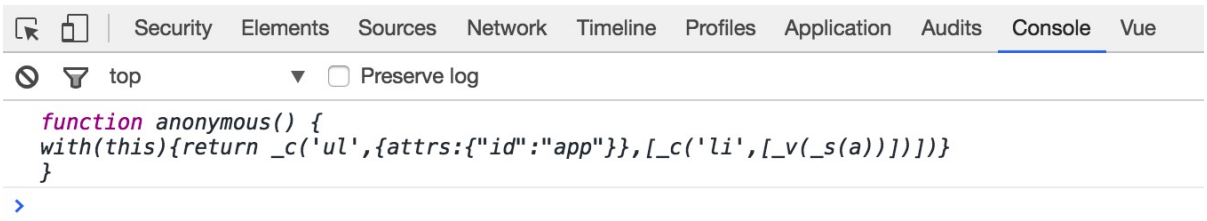
`_render` 方法首先从 `vm.$options` 中解构出 `render` 函数，大家应该记得：`vm.$options.render` 方法是在 `web-runtime-with-compiler.js` 文件中通过 `compileToFunctions` 方法将 `template` 或 `el` 编译而来的。解构出 `render` 函数后，接下来便执行了该方法：

```
vnode = render.call(vm._renderProxy, vm.$createElement)
```

其中使用 `call` 指定了 `render` 函数的作用域环境为 `vm._renderProxy`，这个属性在我们整理实例对象的时候知道，他是在 `Vue.prototype._init` 方法中被添加的，即：`vm._renderProxy = vm`，其实就是Vue实例对象本身，然后传递了一个参数：`vm.$createElement`。那么 `render` 函数到底是干什么的呢？让我们根据上面那句代码猜一猜，我们已经知道 `render` 函数是从 `template` 或 `el` 编译而来的，如果没错的话应该是返回一个虚拟DOM对象。我们不妨使用 `console.log` 打印一下 `render` 函数，当我们的模板这样编写时：

```
<ul id="app">
  <li>{{a}}</li>
</ul>
```

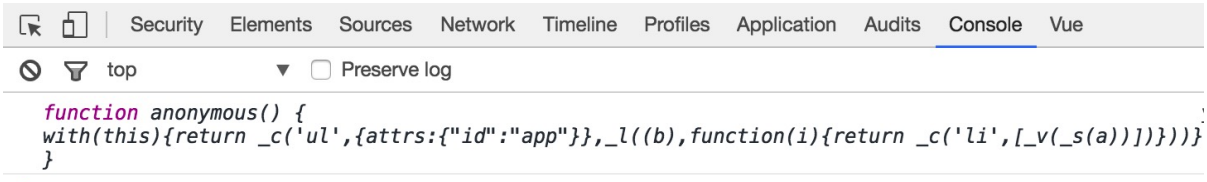
打印的 `render` 函数如下：



我们修改模板为：

```
<ul id="app">
  <li v-for="i in b">{{a}}</li>
</ul>
```

打印出来的 `render` 函数如下：



其实了解Vue2.x版本的同学都知道，Vue提供了 `render` 选项，作为 `template` 的代替方案，同时为JavaScript提供了完全编程的能力，下面两种编写模板的方式实际是等价的：

```
// 方案一：
new Vue({
        el: '#app',
        data: {
                a: 1
        },
        template: '<ul><li>{{a}}</li><li>{{a}}</li></ul>'
})

// 方案二：
new Vue({
        el: '#app',
        render: function (createElement) {
                createElement('ul', [
                        createElement('li', this.a),
                        createElement('li', this.a)
                ])
        }
})
```

现在我们再来看我们打印的 `render` 函数：

```
function anonymous() {
        with(this){
                return _c('ul', {
                        attrs: {"id": "app"}
                },[
                        _c('li', [_v(_s(a))])
                ])
        }
}
```

是不是与我们自己写 `render` 函数很像？因为 render 函数的作用域被绑定到了Vue实例，即：`render.call(vm._renderProxy, vm.$createElement)`，所以上面代码中 `_c`、`_v`、`_s` 以及变量 `a` 相当于Vue实例下的方法和变量。大家还记得诸如 `_c`、`_v`、`_s` 这样的方法在哪里定义的吗？我们在整理Vue构造函数的时候知

道，他们在 `src/core/instance/render.js` 文件中的 `renderMixin` 方法中定义，除了这些之外还有诸如：`_l`、`_m`、`_o` 等等。其中 `_l` 就在我们使用 `v-for` 指令的时候出现了。所以现在大家知道为什么这些方法都被定义在 `render.js` 文件中了吧，因为他们就是为了构造出 `render` 函数而存在的。

现在我们已经知道了 `render` 函数的长相，也知道了 `render` 函数的作用域是Vue实例本身即：`this`(或 `vm`)。那么当我们执行 `render` 函数时，其中的变量如：`a`，就相当于：`this.a`，我们知道这是在求值，所以 `_mount` 中的这段代码：

```
vm._watcher = new Watcher(vm, () => {
  vm._update(vm._render(), hydrating)
}, noop)
```

当 `vm._render` 执行的时候，所依赖的变量就会被求值，并被收集为依赖。按照Vue中 `watcher.js` 的逻辑，当依赖的变量有变化时不仅仅回调函数被执行，实际上还要重新求值，即还要执行一遍：

```
() => {
  vm._update(vm._render(), hydrating)
}
```

这实际上就做到了 `re-render`，因为 `vm._update` 就是文章开头所说的虚拟DOM中的最后一步：`patch`

`vm_render` 方法最终返回一个 `vnode` 对象，即虚拟DOM，然后作为 `vm_update` 的第一个参数传递了过去，我们看一下 `vm_update` 的逻辑，在 `src/core/instance/lifecycle.js` 文件中有这么一段代码：

```
if (!prevVnode) {
  // initial render
  vm.$el = vm.__patch__(
    vm.$el, vnode, hydrating, false /* removeOnly */,
    vm.$options._parentElm,
    vm.$options._refElm
  )
} else {
  // updates
  vm.$el = vm.__patch__(prevVnode, vnode)
}
```

如果还没有 `prevVnode` 说明是首次渲染，直接创建真实DOM。如果已经有了 `prevVnode` 说明不是首次渲染，那么就采用 `patch` 算法进行必要的DOM操作。这就是Vue更新DOM的逻辑。只不过我们没有将 virtual DOM 内部的实现。

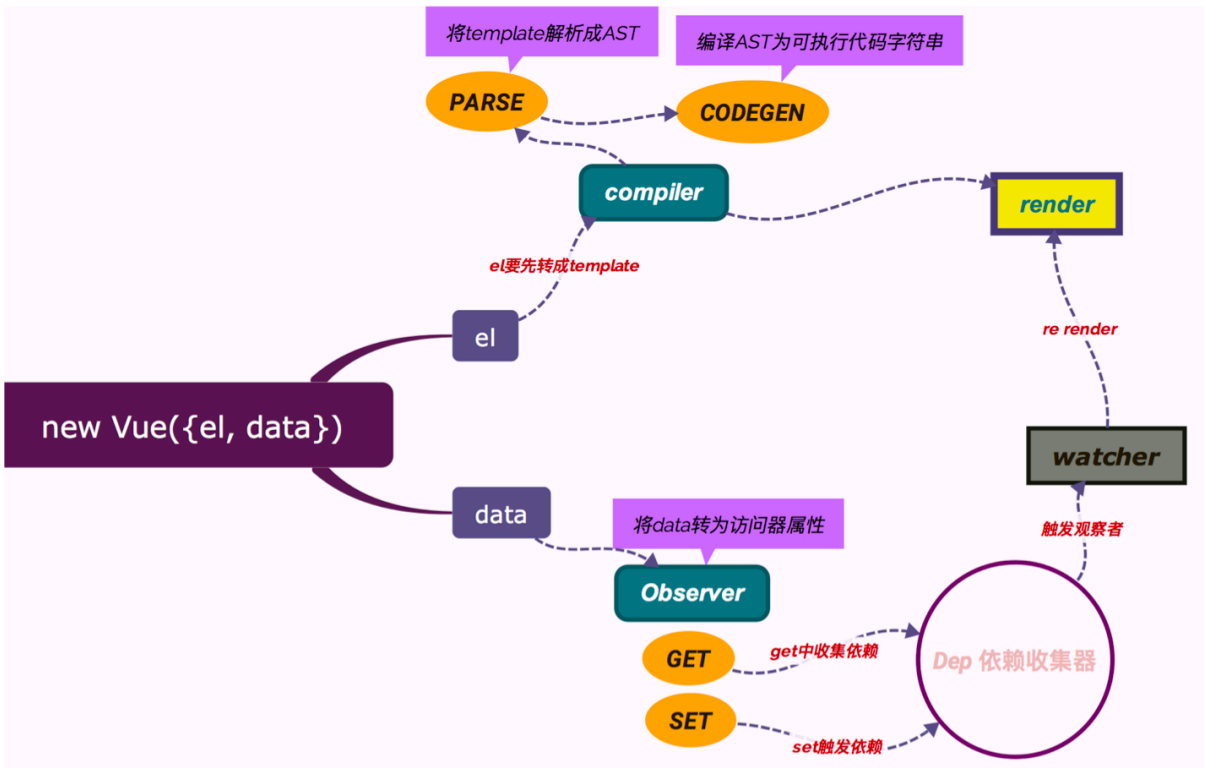
现在我们来好好理理思路，当我们写如下代码时：

```
new Vue({
        el: '#app',
        data: {
                a: 1,
                b: [1, 2, 3]
        }
})
```

Vue 所做的事：

> 1、构建数据响应系统，使用 `Observer` 将数据data转换为访问器属性；将 `el` 编译为 `render` 函数， `render` 函数返回值为虚拟DOM
> 2、在 `_mount` 中对 `_update` 求值，而 `_update` 又会对 `render` 求值，`render` 内部又会对依赖的变量求值，收集为被求值的变量的依赖，当变量改变时， `_update` 又会重新执行一遍，从而做到 `re-render` 。

用一张详细一点的图表示就是这样的：



到此，我们从大体流程，挑着重点的走了一遍Vue，但是还有很多细节我们没有提及，比如：

1、将模板转为 `render` 函数的时候，实际是先生成的抽象语法树（AST），再将抽象语法树转成的 `render` 函数，而且这一整套的代码我们也没有提及，因为他在复杂了，其实这部分内容就是在完正则。

2、我们也没有详细的讲 Virtual DOM 的实现原理，网上已经有文章讲了，大家可以搜一搜

3、我们的例子中仅仅传递了 `el` ， `data` 选项，大家知道 Vue 支持的选项很多，比如我们都没有讲到，但都是触类旁通的，比如你搞清楚了 `data` 选项再去看 `computed` 选项或者 `props` 选项就会很容易，比如你知道了 `Watcher` 的工作机制再去看 `watch` 选项就会很容易。

本篇文章作为Vue源码的启蒙文章，也许还有很多缺陷，全当抛砖引玉了。

< 从矩阵与空间操作的关系理解CSS3的transform

使用weinre调试移动端页面 >