

Vue源码初探

本文希望可以帮助那些想吃蛋糕，但又觉得蛋糕太大而又不知道从哪下口的人们。

一、如何开始第一步

- 将源码项目 `clone` 下来后，按照[CONTRIBUTING](#)中的 `Development Setup` 中的顺序，逐个执行下来

```
1 | $ npm install
2 |
3 | # watch and auto re-build dist/vue.js
4 | $ npm run dev
```

- 学会看package.json文件，就像你在使用MVVM去关注它的render一样。

既然 `$ npm run dev` 命令可以重新编译出 `vue.js` 文件，那么我们就从 `scripts` 中的 `dev` 开始看吧。

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

如果这里你还不清楚 `rollup` 是做什么的，可以[戳这里](#)，简单来说就是一个模块化打包工具。具体的介绍这里就跳过了，因为我们是来看vue的，如果太跳跃的话，基本就把这次主要想做的事忽略掉了，跳跳跳不一定跳哪里了，所以在阅读源码的时候，一定要牢记这次我们的目的是什么。

注意上面指令中的两个关键词 `scripts/config.js` 和 `web-full-dev`，接下来让我们看看 `script/config.js` 这个文件。

```
1 | if (process.env.TARGET) {
2 |   module.exports = genConfig(process.env.TARGET)
3 | } else {
4 |   exports.getBuild = genConfig
5 |   exports.getAllBuilds = () => Object.keys(builds).map(genConfig)
6 | }
```

回忆上面的命令，我们传入的 `TARGET` 是 `web-full-dev`，那么带入到方法中，最终会看到这样一个 `object`

```
1 | 'web-full-dev': {
2 |   // 入口文件
3 |   entry: resolve('web/entry-runtime-with-compiler.js'),
4 |   // 输出文件
5 |   dest: resolve('dist/vue.js'),
6 |   // 格式
7 |   format: 'umd',
8 |   // 环境
9 |   env: 'development',
10 |   // 别名
11 |   alias: { he: './entity-decoder' },
12 |   banner
13 | },
```

虽然这里我们还不知道它具体是做什么的，暂且通过语义来给它补上注释吧。既然有了入口文件，那么我们继续打开文件 `web/entry-runtime-with-compiler.js`。OK，打开这个文件后，终于看到了我们的一个目标关键词

```
1 | import Vue from './runtime/index'
```

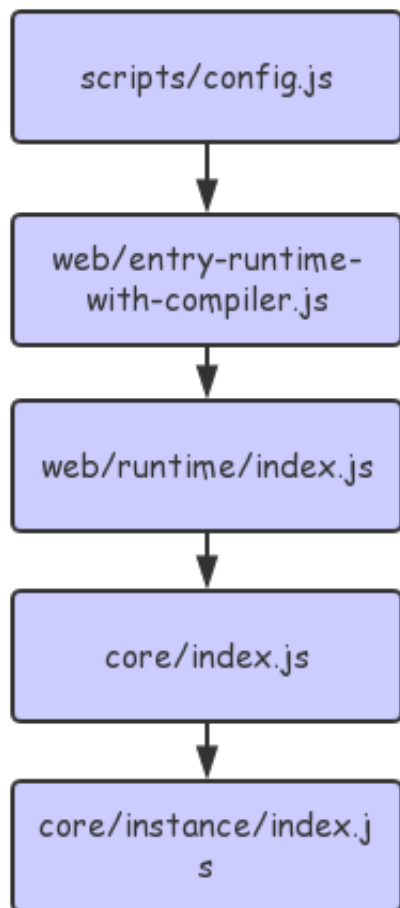
江湖规矩，继续往这个文件里跳，然后你就会看到：

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

是不是又看到了代码第一行中熟悉的关键词 `Vue`

```
1 | import Vue from './instance/index'
```

打开 `instance/index` 后，结束了我们的第一步，已经从package.json中到框架中的文件，找到了 `Vue` 的定义地方。让我们再回顾下流程：



二、学会利用demo

切记，在看源码时为了防止看着看着跑偏了，我们一定要按照代码执行的顺序看。

- 项目结构中有 `examples` 目录，让我们也创建一个属于自己的demo在这里面吧，随便copy一个目录，命名为demo，后面我们的代码都通过这个demo来进行测试、观察。

index.html内容如下：

```
1 | <!DOCTYPE html>
2 | <html>
3 |   <head>
4 |     <title>Demo</title>
5 |     <script src="../../dist/vue.js"></script>
6 |   </head>
7 |   <body>
8 |     <div id="demo">
9 |       <template>
10 |         <span>{{text}}</span>
11 |       </template>
12 |     </div>
13 |     <script src="app.js"></script>
14 |   </body>
15 | </html>
```

app.js文件内容如下：

```
1 | var demo = new Vue({
2 |   el: '#demo',
3 |   data() {
4 |     return {
5 |       text: 'hello world!'
6 |     }
7 |   }
8 | })
```

引入vue.js

上面demo的html中我们引入了dist/vue.js，那么window下，就会有 `Vue` 对象，暂且先将app.js的代码修改如下：

```
1 | console.dir(Vue);
```

如果这里你还不知道 `console.dir`，而只知道 `console.log`，那你就亲自试试然后记住他们之间的差异吧。

从控制台我们可以看出，`Vue` 对象以及原型上有一系列属性，那么这些属性是从哪儿来的，做什么的，就是我们后续去深入的内容。

三、从哪儿来的

是否还记得我们在第一章中找到最终 `Vue` 构造函数的文件？如果不记得了，就再回去看一眼吧，我们在本章会按照那个顺序倒着来看一遍 `Vue` 的属性挂载。

instance(src/core/instance/index.js)

```
1 import { initMixin } from './init'
2 import { stateMixin } from './state'
3 import { renderMixin } from './render'
4 import { eventsMixin } from './events'
5 import { lifecycleMixin } from './lifecycle'
6 import { warn } from '../util/index'
7
8 function Vue (options) {
9   if (process.env.NODE_ENV !== 'production' &&
10     !(this instanceof Vue)
11   ) {
12     warn('Vue is a constructor and should be
13       called with the `new` keyword')
14   }
15   this._init(options)
16 }
17 initMixin(Vue)
18 stateMixin(Vue)
19 eventsMixin(Vue)
20 lifecycleMixin(Vue)
21 renderMixin(Vue)
22
23 export default Vue
```

接下来我们就开始按照代码执行的顺序，先来分别看看这几个函数到底是弄啥嘞？

```
1 initMixin(Vue)
2 stateMixin(Vue)
3 eventsMixin(Vue)
4 lifecycleMixin(Vue)
5 renderMixin(Vue)
```

1. initMixin(src/core/instance/init.js)

```
1 | Vue.prototype._init = function (options?: Object) {}
```

在传入的 `vue` 对象的原型上挂载了 `_init` 方法。

2. stateMixin(src/core/instance/state.js)

```

1 // Object.defineProperty(Vue.prototype, '$data', dataDef)
2 // 这里$data只提供了get方法, set方法再非生产环境时会给予警告
3 Vue.prototype.$data = undefined;
4 // Object.defineProperty(Vue.prototype, '$props', propsDef)
5 // 这里$props只提供了get方法, set方法再非生产环境时会给予警告
6 Vue.prototype.$props = undefined;
7
8 Vue.prototype.$set = set
9 Vue.prototype.$delete = del
10
11 Vue.prototype.$watch = function() {}

```

如果这里你还不知道 `Object.defineProperty` 是做什么的, 我对你的建议是可以把对象的原型这部分好好看一眼, 对于后面的代码浏览会有很大的效率提升, 不然云里雾里的, 你浪费的只有自己的时间而已。

3. eventsMixin(src/core/instance/events.js)

```

1 Vue.prototype.$on = function() {}
2 Vue.prototype.$once = function() {}
3 Vue.prototype.$off = function() {}
4 Vue.prototype.$emit = function() {}

```

4. lifecycleMixin(src/core/instance/lifecycle.js)

```

1 Vue.prototype._update = function() {}
2 Vue.prototype.$forceUpdate = function () {}
3 Vue.prototype.$destroy = function () {}

```

5. renderMixin(src/core/instance/render.js)

```
1 // installRenderHelpers
2 Vue.prototype._o = markOnce
3 Vue.prototype._n = toNumber
4 Vue.prototype._s = toString
5 Vue.prototype._l = renderList
6 Vue.prototype._t = renderSlot
7 Vue.prototype._q = looseEqual
8 Vue.prototype._i = looseIndexOf
9 Vue.prototype._m = renderStatic
10 Vue.prototype._f = resolveFilter
11 Vue.prototype._k = checkKeyCodes
12 Vue.prototype._b = bindObjectProps
13 Vue.prototype._v = createTextVNode
14 Vue.prototype._e = createEmptyVNode
15 Vue.prototype._u = resolveScopedSlots
16 Vue.prototype._g = bindObjectListeners
17
18 //
19 Vue.prototype.$nextTick = function() {}
20 Vue.prototype._render = function() {}
```

将上面5个方法执行完成后，`instance` 中对 `vue` 的原型一波疯狂输出后，`vue` 的原型已经变成了：

```
> Vue.prototype
< ▼ {_init: f, $set: f, $delete: f, $watch: f, $on: f, ...} ⓘ
  ▶ $delete: f del(target, key)
  ▶ $destroy: f ()
  ▶ $emit: f (event)
  ▶ $forceUpdate: f ()
  ▶ $nextTick: f (fn)
  ▶ $off: f (event, fn)
  ▶ $on: f (event, fn)
  ▶ $once: f (event, fn)
  ▶ $set: f (target, key, val)
  ▶ $watch: f ( expOrFn, cb, options )
  ▶ _b: f bindObjectProps( data, tag, value, asProp, isSync )
  ▶ _e: f (text)
  ▶ _f: f resolveFilter(id)
  ▶ _g: f bindObjectListeners(data, value)
  ▶ _i: f looseIndexOf(arr, val)
  ▶ _init: f (options)
  ▶ _k: f checkKeyCodes( eventKeyCode, key, builtInKeyCode, eventKeyName, builtInKeyName )
  ▶ _l: f renderList( val, render )
  ▶ _m: f renderStatic( index, isInFor )
  ▶ _n: f toNumber(val)
  ▶ _o: f markOnce( tree, index, key )
  ▶ _q: f looseEqual(a, b)
  ▶ _render: f ()
  ▶ _s: f toString(val)
  ▶ _t: f renderSlot( name, fallback, props, bindObject )
  ▶ _u: f resolveScopedSlots( fns, // see flow/vnode res )
  ▶ _update: f (vnode, hydrating)
  ▶ _v: f createTextVNode(val)
    $data: undefined
    $props: undefined
  ▶ constructor: f Vue(options)
  ▶ get $data: f ()
  ▶ set $data: f (newData)
  ▶ get $props: f ()
  ▶ set $props: f ()
  ▶ __proto__: Object
```

如果你认为到此就结束了？答案当然是，不。让我们顺着第一章整理的图，继续回到core/index.js中。

Core(src/core/index.js)


```
1 import Vue from './instance/index'
2 import { initGlobalAPI } from './global-api/index'
3 import { isServerRendering } from 'core/util/env'
4 import {
5   FunctionalRenderContext
6 } from 'core/vdom/create-functional-component'
7
8 // 初始化全局API
9 initGlobalAPI(Vue)
10
11 Object.defineProperty(Vue.prototype, '$isServer', {
12   get: isServerRendering
13 })
14
15 Object.defineProperty(Vue.prototype, '$ssrContext', {
16   get () {
17     /* istanbul ignore next */
18     return this.$vnode && this.$vnode.ssrContext
19   }
20 })
21
22 // expose FunctionalRenderContext for ssr runtime helper installation
23 Object.defineProperty(Vue, 'FunctionalRenderContext', {
24   value: FunctionalRenderContext
25 })
26
27 Vue.version = '__VERSION__'
28
29 export default Vue
```

按照代码执行顺序，我们看看 `initGlobalAPI(Vue)` 方法内容：

```

1 // Object.defineProperty(Vue, 'config', configDef)
2 Vue.config = { devtools: true, ...}
3 Vue.util = {
4   warn,
5   extend,
6   mergeOptions,
7   defineReactive,
8 }
9 Vue.set = set
10 Vue.delete = delete
11 Vue.nextTick = nextTick
12 Vue.options = {
13   components: {},
14   directives: {},
15   filters: {},
16   _base: Vue,
17 }
18 // extend(Vue.options.components, builtInComponents)
19 Vue.options.components.KeepAlive = { name: 'keep-alive' ...}
20 // initUse
21 Vue.use = function() {}
22 // initMixin
23 Vue.mixin = function() {}
24 // initExtend
25 Vue.cid = 0
26 Vue.extend = function() {}
27 // initAssetRegisters
28 Vue.component = function() {}
29 Vue.directive = function() {}
30 Vue.filter = function() {}

```

不难看出，整个Core在instance的基础上，又对 `vue` 的属性进行了一波输出。经历完Core后，整个 `vue` 变成了这样：

```

▼ f Vue(options)
  cid: 0
  ▶ component: f ( id, definition )
  ▶ delete: f del(target, key)
  ▶ directive: f ( id, definition )
  ▶ extend: f (extendOptions)
  ▶ filter: f ( id, definition )
  ▶ mixin: f (mixin)
  ▶ nextTick: f nextTick(cb, ctx)
  ▶ options: {components: {...}, directives: {...}, filters: {...}, _base: f}
  ▶ set: f (target, key, val)
  ▶ use: f (plugin)
  ▶ util: {warn: f, extend: f, mergeOptions: f, defineReactive: f}
  version: "2.5.17-beta.0"
  ▶ FunctionalRenderContext: f FunctionalRenderContext( data, props, children, parent, Ctor )
    arguments: (...)
    caller: (...)
    config: (...)
    length: 1
    name: "Vue"
  ▶ prototype: {_init: f, $set: f, $delete: f, $watch: f, $on: f, ...}
  ▶ get config: f ()
  ▶ set config: f ()
  ▶ __proto__: f ()
  [[FunctionLocation]]: vue.js:4700
  ▶ [[Scopes]]: Scopes[2]

```

继续顺着第一章整理的路线，来看看runtime又对 `Vue` 做了什么。

runtime(src/platforms/web/runtime/index.js)

这里还是记得先从宏观入手，不要去看每个方法的详细内容。可以通过 `debugger` 来暂停代码执行，然后通过控制台的 `console.dir(Vue)` 随时观察 `Vue` 的变化，

1. 这里首先针对web平台，对Vue.config来了一小波方法添加。

```

1 | Vue.config.mustUseProp = mustUseProp
2 | Vue.config.isReservedTag = isReservedTag
3 | Vue.config.isReservedAttr = isReservedAttr
4 | Vue.config.getTagNamespace = getTagNamespace
5 | Vue.config.isUnknownElement = isUnknownElement

```

2. 向options中directives增加了 `model` 以及 `show` 指令：

```

1 | // extend(Vue.options.directives, platformDirectives)
2 | Vue.options.directives = {
3 |   model: { componentUpdated: f ...}
4 |   show: { bind: f, update: f, unbind: f }
5 | }

```

3. 向options中components增加了 `Transition` 以及 `TransitionGroup` :

```
1 | // extend(Vue.options.components, platformComponents)
2 | Vue.options.components = {
3 |   KeepAlive: { name: "keep-alive" ...}
4 |   Transition: {name: "transition", props: {...} ...}
5 |   TransitionGroup: {props: {...}, beforeMount: f, ...}
6 | }
```

4. 在原型中追加 `__patch__` 以及 `$mount` :

```
1 | // 虚拟dom所用到的方法
2 | Vue.prototype.__patch__ = patch
3 | Vue.prototype.$mount = function() {}
```

5. 以及对devtools的支持。

entry(src/platforms/web/entry-runtime-with-compiler.js)

1. 在entry中, 覆盖了 `$mount` 方法。
2. 挂载compile, `compileToFunctions` 方法是将 `template` 编译为 `render` 函数

```
1 | Vue.compile = compileToFunctions
```

小结

至此, 我们完整的过了一遍在web中Vue的构造函数的变化过程:

- 通过instance对Vue.prototype进行属性和方法的挂载。
- 通过core对Vue进行静态属性和方法的挂载。
- 通过runtime添加了对platform === 'web'的情况下, 特有的配置、组件、指令。
- 通过entry来为\$mount方法增加编译 `template` 的能力。

四、做什么的

上一章我们从宏观角度观察了整个Vue构造函数的变化过程, 那么我们本章将从微观角度, 看看new Vue()后, 都做了什么。

将我们demo中的app.js修改为如下代码:

```

1  var demo = new Vue({
2    el: '#demo',
3    data() {
4      return {
5        text: 'hello world!'
6      }
7    }
8  })

```

还记得instance/init中的Vue构造函数吗？在代码执行了 `this._init(options)`，那我们就从 `_init` 入手，开始本章的旅途。

```

1  Vue.prototype._init = function (options?: Object) {
2    const vm: Component = this
3    // a uid
4    vm._uid = uid++
5
6    let startTag, endTag
7    /* istanbul ignore if */
8    // 浏览器环境&支持window.performance&非生产环境&配置了performance
9    if (process.env.NODE_ENV !== 'production'
10      && config.performance && mark) {
11      startTag = `vue-perf-start:${vm._uid}`
12      endTag = `vue-perf-end:${vm._uid}`
13      // 相当于 window.performance.mark(startTag)
14      mark(startTag)
15    }
16
17    // a flag to avoid this being observed
18    vm._isVue = true
19    // merge options
20    if (options && options._isComponent) {
21      // optimize internal component instantiation
22      // since dynamic options merging is pretty slow, and none of the
23      // internal component options needs special treatment.
24      initInternalComponent(vm, options)
25    } else {
26      // 将options进行合并
27      vm.$options = mergeOptions(
28        resolveConstructorOptions(vm.constructor),
29        options || {},
30        vm
31      )
32    }
33    /* istanbul ignore else */
34    if (process.env.NODE_ENV !== 'production') {

```

```

35     initProxy(vm)
36   } else {
37     vm._renderProxy = vm
38   }
39   // expose real self
40   vm._self = vm
41   initLifecycle(vm)
42   initEvents(vm)
43   initRender(vm)
44   callHook(vm, 'beforeCreate')
45   initInjections(vm) // resolve injections before data/props
46   initState(vm)
47   initProvide(vm) // resolve provide after data/props
48   callHook(vm, 'created')
49
50   /* istanbul ignore if */
51   if (process.env.NODE_ENV !== 'production'
52     && config.performance && mark) {
53     vm._name = formatComponentName(vm, false)
54     mark(endTag)
55     measure(`vue ${vm._name} init`, startTag, endTag)
56   }
57
58   if (vm.$options.el) {
59     vm.$mount(vm.$options.el)
60   }
61 }

```

这个方法都做了什么？

1. 在当前实例中，添加 `_uid` , `_isVue` 属性。
2. 当非生产环境时，用window.performance标记vue初始化的开始。
3. 由于我们的demo中，没有手动处理_isComponent，所以这里会进入到else分支，将Vue.options与传入options进行合并。
4. 为当前实例添加 `_renderProxy` , `_self` 属性。
5. 初始化生命周期， `initLifecycle`
6. 初始化事件， `initEvents`
7. 初始化render， `initRender`
8. 调用生命周期中的 `beforeCreate`
9. 初始化注入值 `initInjections`
10. 初始化状态 `initState`
11. 初始化XX `initProvide`
12. 调用生命周期中的 `created`
13. 非生产环境下，标识初始化结束，为当前实例增加 `_name` 属性
14. 根据 `options` 传入的 `el` , 调用当前实例的 `$mount`

OK, 我们又宏观的看了整个 `_init` 方法, 接下来我们结合我们的demo, 来细细的看下每一步产生的影响, 以及具体调用的方法。

mergeOptions(src/core/util/options.js)

```
1  vm.$options = mergeOptions(  
2    resolveConstructorOptions(vm.constructor),  
3    options || {},  
4    vm  
5  )  
6  
7  function resolveConstructorOptions (Ctor: Class<Component>) {  
8    let options = Ctor.options  
9    if (Ctor.super) {  
10     const superOptions = resolveConstructorOptions(Ctor.super)  
11     const cachedSuperOptions = Ctor.superOptions  
12     if (superOptions !== cachedSuperOptions) {  
13       // super option changed,  
14       // need to resolve new options.  
15       Ctor.superOptions = superOptions  
16       // check if there are any late-modified/attached options (#4976)  
17       const modifiedOptions = resolveModifiedOptions(Ctor)  
18       // update base extend options  
19       if (modifiedOptions) {  
20         extend(Ctor.extendOptions, modifiedOptions)  
21       }  
22       options = Ctor.options =  
23         mergeOptions(superOptions, Ctor.extendOptions)  
24       if (options.name) {  
25         options.components[options.name] = Ctor  
26       }  
27     }  
28   }  
29   return options  
30 }
```

还记得我们在第三章中, runtime对 `vue` 的变更之后, options变成了什么样吗? 如果你忘了, 这里我们再回忆一下:

```

1  Vue.options = {
2    components: {
3      KeepAlive: { name: "keep-alive" ...}
4      Transition: {name: "transition", props: {...} ...}
5      TransitionGroup: {props: {...}, beforeMount: f, ...}
6    },
7    directives: {
8      model: { componentUpdated: f ...}
9      show: { bind: f, update: f, unbind: f }
10   },
11   filters: {},
12   _base: f Vue
13 }

```

我们将上面的代码进行拆解，首先将 `this.constructor` 传入 `resolveConstructorOptions` 中，因为我们的demo中没有进行继承操作，所以在 `resolveConstructorOptions` 方法中，没有进入if，直接返回得到的结果，就是在 `runtime` 中进行处理后的 `options` 选项。而 `options` 就是我们在调用 `new Vue({})` 时，传入的 `options` 。此时，`mergeOptions`方法变为：

```

1  vm.$options = mergeOptions(
2    {
3      components: {
4        KeepAlive: { name: "keep-alive" ...}
5        Transition: {name: "transition", props: {...} ...}
6        TransitionGroup: {props: {...}, beforeMount: f, ...}
7      },
8      directives: {
9        model: { componentUpdated: f ...}
10       show: { bind: f, update: f, unbind: f }
11     },
12     filters: {},
13     _base: f Vue
14   },
15   {
16     el: '#demo',
17     data: f data()
18   },
19   vm
20 )

```


接下来开始调用 `mergeOptions` 方法。打开文件后，我们发现在引用该文件时，会立即执行一段代码：

```

1  // config.optionMergeStrategies = Object.create(null)
2  const strats = config.optionMergeStrategies

```


仔细往下看后面，还有一系列针对 `strats` 挂载方法和属性的操作，最终 `strats` 会变为：

```
▼ Object   
  ▶ activated: f mergeHook( parentVal, childVal )  
  ▶ beforeCreate: f mergeHook( parentVal, childVal )  
  ▶ beforeDestroy: f mergeHook( parentVal, childVal )  
  ▶ beforeMount: f mergeHook( parentVal, childVal )  
  ▶ beforeUpdate: f mergeHook( parentVal, childVal )  
  ▶ components: f mergeAssets( parentVal, childVal, vm, key )  
  ▶ computed: f ( parentVal, childVal, vm, key )  
  ▶ created: f mergeHook( parentVal, childVal )  
  ▶ data: f ( parentVal, childVal, vm )  
  ▶ deactivated: f mergeHook( parentVal, childVal )  
  ▶ destroyed: f mergeHook( parentVal, childVal )  
  ▶ directives: f mergeAssets( parentVal, childVal, vm, key )  
  ▶ el: f (parent, child, vm, key)  
  ▶ errorCaptured: f mergeHook( parentVal, childVal )  
  ▶ filters: f mergeAssets( parentVal, childVal, vm, key )  
  ▶ inject: f ( parentVal, childVal, vm, key )  
  ▶ methods: f ( parentVal, childVal, vm, key )  
  ▶ mounted: f mergeHook( parentVal, childVal )  
  ▶ props: f ( parentVal, childVal, vm, key )  
  ▶ propsData: f (parent, child, vm, key)  
  ▶ provide: f mergeDataOrFn( parentVal, childVal, vm )  
  ▶ updated: f mergeHook( parentVal, childVal )  
  ▶ watch: f ( parentVal, childVal, vm, key )
```

其实这些散落在代码中的挂载操作，有点没想明白尤大没有放到一个方法里去统一处理一波？

继续往下翻，看到了我们进入这个文件的目标，那就是 `mergeOptions` 方法：

```

1 function mergeOptions (
2   parent: Object,
3   child: Object,
4   vm?: Component
5 ): Object {
6   debugger;
7   if (process.env.NODE_ENV !== 'production') {
8     // 根据用户传入的options, 检查合法性
9     checkComponents(child)
10  }
11
12  if (typeof child === 'function') {
13    child = child.options
14  }
15  // 标准化传入options中的props
16  normalizeProps(child, vm)
17  // 标准化注入
18  normalizeInject(child, vm)
19  // 标准化指令
20  normalizeDirectives(child)
21  const extendsFrom = child.extends
22  if (extendsFrom) {
23    parent = mergeOptions(parent, extendsFrom, vm)
24  }
25  if (child.mixins) {
26    for (let i = 0, l = child.mixins.length; i < l; i++) {
27      parent = mergeOptions(parent, child.mixins[i], vm)
28    }
29  }
30  const options = {}
31  let key
32  for (key in parent) {
33    mergeField(key)
34  }
35  for (key in child) {
36    if (!hasOwn(parent, key)) {
37      mergeField(key)
38    }
39  }
40  function mergeField (key) {
41    const strat = strats[key] || defaultStrat
42    options[key] = strat(parent[key], child[key], vm, key)
43  }
44  return options
45 }

```

因为我们这里使用了最简单的 `hello world`，所以在 `mergeOptions` 中，可以直接从30行开始看，这

里初始化了变量 `options`，32行、35行的 `for` 循环分别根据合并策略进行了合并。看到这里，恍然大悟，原来 `strats` 是定义一些标准合并策略，如果没有定义在其中，就使用默认合并策略 `defaultStrat`。

这里有个小细节，就是在循环子`options`时，仅合并父`options`中不存在的项，来提高合并效率。

让我们继续来用最直白的方式，回顾下上面的过程：

```

1 // 初始化合并策略
2 const strats = config.optionMergeStrategies
3 strats.el = strats.propsData = function (parent, child, vm, key) {}
4 strats.data = function (parentVal, childVal, vm) {}
5 constants.LIFECYCLE_HOOKS.forEach(hook => strats[hook] = mergeHook)
6 constants.ASSET_TYPES.forEach(type => strats[type + 's'] = mergeAssets)
7 strats.watch = function(parentVal, childVal, vm, key) {}
8 strats.props =
9 strats.methods =
10 strats.inject =
11 strats.computed = function(parentVal, childVal, vm, key) {}
12 strats.provide = mergeDataOrFn
13
14 // 默认合并策略
15 const defaultStrat = function (parentVal, childVal) {
16   return childVal === undefined
17     ? parentVal
18     : childVal
19 }
20
21 function mergeOptions (parent, child, vm) {
22   // 本次demo没有用到省略前面代码
23   ...
24
25   const options = {}
26   let key
27   for (key in parent) {
28     mergeField(key)
29   }
30   for (key in child) {
31     if (!hasOwn(parent, key)) {
32       mergeField(key)
33     }
34   }
35   function mergeField (key) {
36     const strat = strats[key] || defaultStrat
37     options[key] = strat(parent[key], child[key], vm, key)
38   }
39   return options
40 }

```

怎么样，是不是清晰多了？本次的demo经过 `mergeOptions` 之后，变为了如下：

```
▼ Object ⓘ
  ► components: {}
  ► data: f mergedInstanceDataFn()
  ► directives: {}
    el: "#demo"
  ► filters: {}
  ► _base: f Vue(options)
  ► __proto__: Object
```

OK，因为我们本次是来看 `_init` 的，所以到这里，你需要清除 `Vue` 通过合并策略，将parent与child进行了合并即可。接下来，我们继续回到 `_init` 对 `options` 合并处理完之后做了什么？

initProxy(src/core/instance/proxy.js)

在merge完options后，会判断如果是非生产环境时，会进入initProxy方法。

```
1 | if (process.env.NODE_ENV !== 'production') {
2 |   initProxy(vm)
3 | } else {
4 |   vm._renderProxy = vm
5 | }
6 | vm._self = vm
```

带着雾水，进入到方法定义的文件，看到了 `Proxy` 这个关键字，如果这里你还不清楚，可以看下阮老师的ES6，上面有讲。

- 这里在非生产环境时，对config.keyCodes的一些关键字做了禁止赋值操作。
- 返回了 `vm._renderProxy = new Proxy(vm, handlers)`，这里的 `handlers`，由于我们的options中没有render，所以这里取值是hasHandler。

这部分具体是做什么用的，暂且知道有这么个东西，主线还是不要放弃，继续回到主线吧。

initLifecycle(src/core/instance/lifecycle.js)

初始化了与生命周期相关的属性。

```

1 function initLifecycle (vm) {
2   const options = vm.$options
3   // 省去部分与本次demo无关代码
4   ...
5   vm.$parent = undefined
6   vm.$root = vm
7
8   vm.$children = []
9   vm.$refs = {}
10
11  vm._watcher = null
12  vm._inactive = null
13  vm._directInactive = false
14  vm._isMounted = false
15  vm._isDestroyed = false
16  vm._isBeingDestroyed = false
17 }

```

initEvents(src/core/instance/events.js)

```

1 function initEvents (vm) {
2   vm._events = Object.create(null)
3   vm._hasHookEvent = false
4   // 省去部分与本次demo无关代码
5   ...
6 }

```

initRender(src/core/instance/render.js)

```

1 function initRender (vm: Component) {
2   vm._vnode = null // the root of the child tree
3   vm._staticTrees = null // v-once cached trees
4   vm.$slots = {}
5   vm.$scopedSlots = {}
6   vm._c = (a, b, c, d) => createElement(vm, a, b, c, d, false)
7   vm.$createElement = (a, b, c, d) => createElement(vm, a, b, c, d, true)
8   vm.$attrs = {}
9   vm.$listeners = {}
10 }

```

callHook(vm, 'beforeCreate')

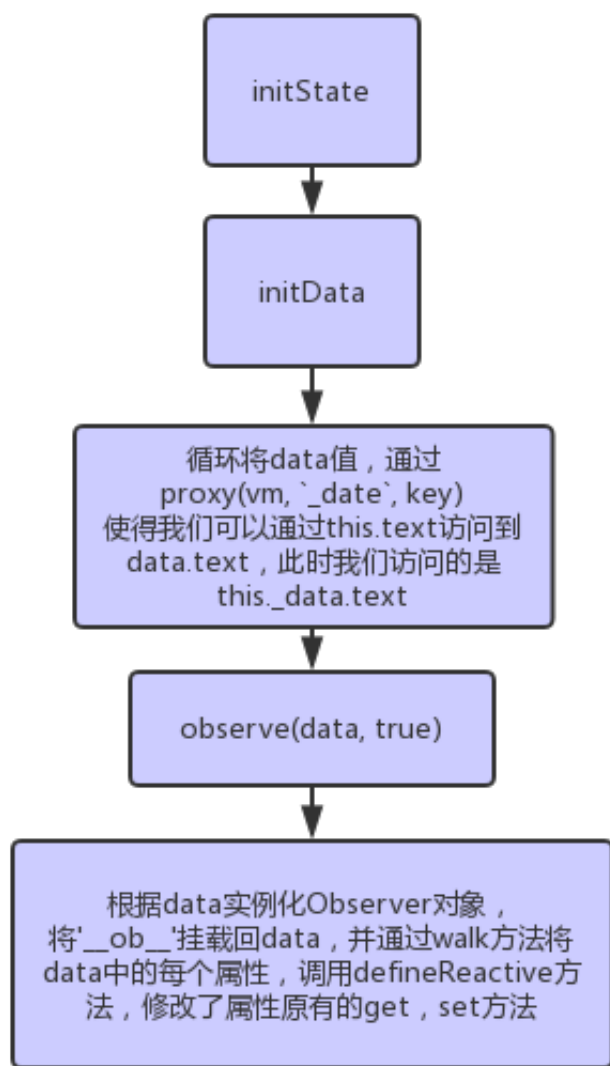
调用生命周期函数beforeCreate

initInjections(src/core/instance/inject.js)

由于本demo没有用到注入值，对本次vm并无实际影响，所以这一步暂且忽略，有兴趣可以自行翻阅。

initState(src/core/instance/state.js)

本次的只针对这最简单的demo，分析 `initState`，可能忽略了很多过程，后续我们会针对更复杂的demo来继续分析一波。



这里你可以先留意到几个关键词 `Observer`，`Dep`，`Watcher`。每个 `Observer` 都有一个独立的 `Dep`。关于 `Watcher`，暂时没用到，但是请相信，马上就可以看到了。

initProvide(src/core/instance/inject.js)

由于本demo没有用到，对本次vm并无实际影响，所以这一步暂且忽略，有兴趣可以自行翻阅。

callHook(vm, 'created')

这里知道为什么在 `created` 时候，没法操作DOM了吗？因为在这里，还没有涉及到实际的DOM渲染。

vm.\$mount(vm.\$options.el)

这里前面有个if判断，所以当你如果没有在 `new Vue` 中的 `options` 没有传入 `el` 的话，就不会触发实际的渲染，就需要自己手动调用了 `$mount`。

这里的 `$mount` 最终会调向哪里？还记得我们在第三章看到的 `compiler` 所做的事情吗？就是覆盖 `Vue.prototype.$mount`，接下来，我们一起进入 `$mount` 函数看看它都做了什么吧。

```
1 // 只保留与本次相关代码，其余看太多会影响视线
2 const mount = Vue.prototype.$mount
3 Vue.prototype.$mount = function (
4   el?: string | Element,
5   hydrating?: boolean
6 ): Component {
7   el = el && query(el)
8
9   const options = this.$options
10  if (!options.render) {
11    let template = getOuterHTML(el)
12    if (template) {
13      const { render, staticRenderFns } = compileToFunctions(template, {
14        shouldDecodeNewlines,
15        shouldDecodeNewlinesForHref,
16        delimiters: options.delimiters,
17        comments: options.comments
18      }, this)
19      options.render = render
20      options.staticRenderFns = staticRenderFns
21    }
22  }
23  return mount.call(this, el, hydrating)
24 }
```

这里在覆盖 `$mount` 之前，先将原有的 `$mount` 保留至变量 `mount` 中，整个覆盖后的方法是将 `template` 转为 `render` 函数挂载至 `vm` 的 `options`，然后调用调用原有的 `mount`。所以还记得 `mount` 来自于哪嘛？那就继续吧 `runtime/index`，方法很简单，调用了生命周期中 `mountComponent`。


```

1 // 依然只保留和本demo相关的内容
2 function mountComponent (
3   vm: Component,
4   el: ?Element,
5   hydrating?: boolean
6 ): Component {
7   vm.$el = el
8   callHook(vm, 'beforeMount')
9
10  let updateComponent = () => {
11    vm._update(vm._render(), hydrating)
12  }
13
14  new Watcher(vm, updateComponent, noop, {
15    before () {
16      if (vm._isMounted) {
17        callHook(vm, 'beforeUpdate')
18      }
19    }
20  }, true /* isRenderWatcher */)
21  hydrating = false
22
23  if (vm.$vnode == null) {
24    vm._isMounted = true
25    callHook(vm, 'mounted')
26  }
27  return vm
28 }

```

OK，精彩的部分来了，一个 `Watcher`，盘活了整个我们前面铺垫的一系列东西。打开 `src/core/observer/watcher.js`，让我们看看 `Watcher` 的构造函数吧。为了清楚的看到 `Watcher` 的流程。依旧只保留方法我们需要关注的东西：

```

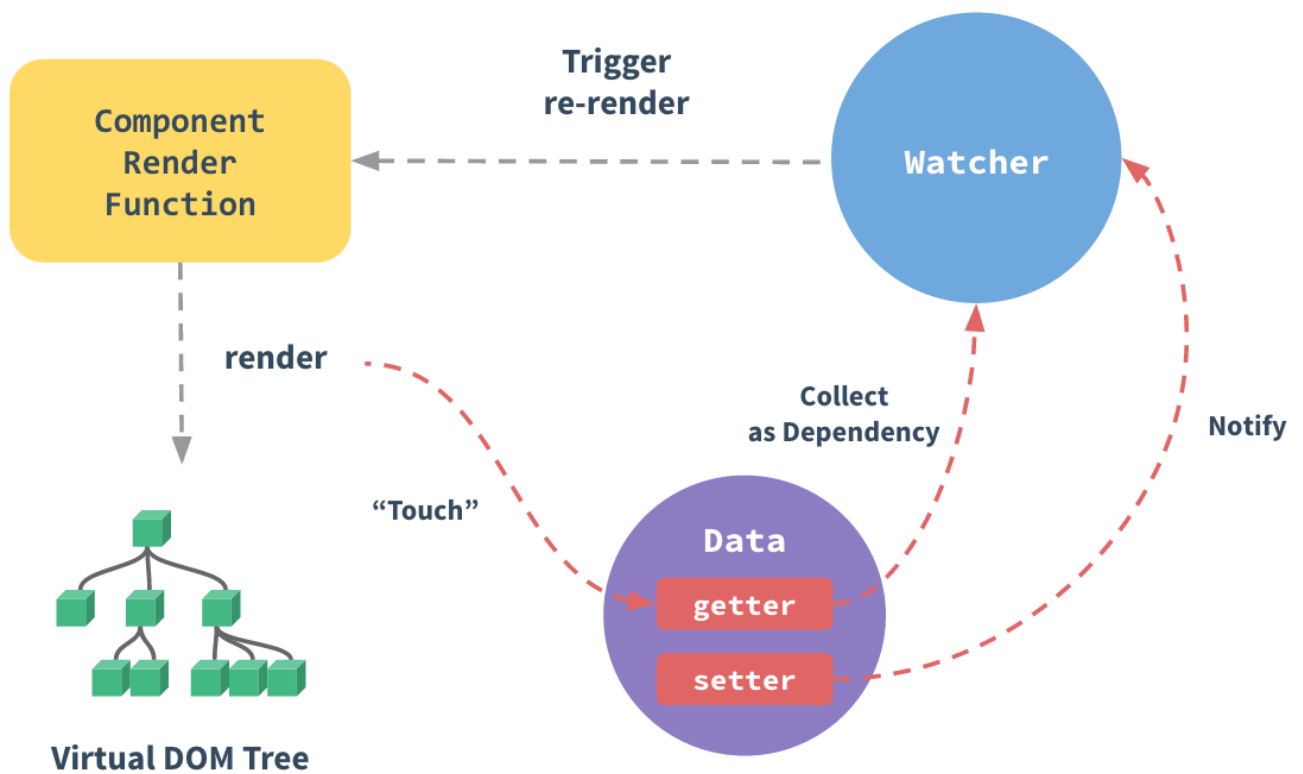
1   constructor (vm, expOrFn, cb, options, isRenderWatcher) {
2     this.vm = vm
3     vm._watcher = this
4     vm._watchers.push(this)
5     this.getter = expOrFn
6     this.value = this.get()
7   }
8
9   get () {
10    pushTarget(this)
11    let value
12    const vm = this.vm
13    value = this.getter.call(vm, vm)
14    popTarget()
15    this.cleanupDeps()
16    return value
17  }

```

1. 在 `watcher` 的构造函数中，本次传入的 `updateComponent` 作为 `watcher` 的 `getter`。
2. 在 `get` 方法调用时，又通过 `pushTarget` 方法，将当前 `watcher` 赋值给 `Dep.target`。
3. 调用 `getter`，相当于调用 `vm._update`，先调用 `vm._render`，而这时 `vm._render`，此时会将已经准备好的 `render` 函数进行调用。
4. `render` 函数中又用到了 `this.text`，所以又会调用 `text` 的 `get` 方法，从而触发了 `dep.depend()`。
5. `dep.depend()` 会调回 `watcher` 的 `addDep`，这时 `watcher` 记录了当前 `dep` 实例。
6. 继续调用 `dep.addSub(this)`，`dep` 又记录了当前 `watcher` 实例，将当前的 `watcher` 存入 `dep.subs` 中。
7. 这里顺带提一下本次 `demo` 还没有使用的，也就是当 `this.text` 发生改变时，会触发 `Observer` 中的 `set` 方法，从而触发 `dep.notify()` 方法来进行 `update` 操作。

最后这段文字太干了，可以自己通过断点，耐心的走一遍整个过程。

就这样，`vue` 的数据响应系统，通过 `Observer`、`Watcher`、`Dep` 完美的串在了一起。也希望经历这个过程后，你能对真正的这张图，有一定的理解。



当然，`$mount` 中还有一步被我轻描淡写了，那就是这部分，将template转换为render，render实际调用时，会经历 `_render`，`$createElement`，`__patch__`，方法，有兴趣可以自己浏览下'src/core/vdom/'目录下的文件，来了解 `vue` 针对虚拟dom的使用。