# **我们知道Vue的三大核心知识是：数据驱动视图，虚拟Dom,diff算法。由于时间有限，我这里先介绍一个数据驱动。**

## **数据驱动**

# **数据驱动**

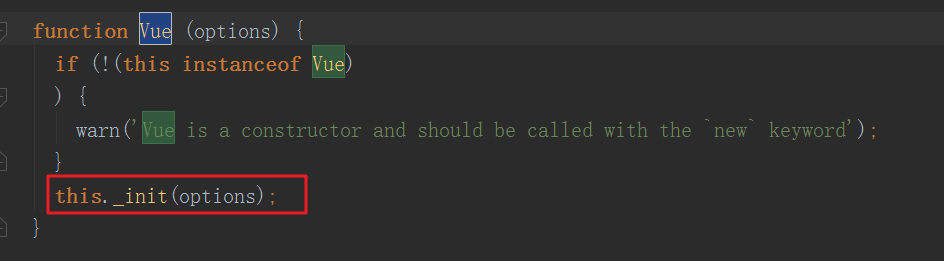
Vue.js 一个核心思想是数据驱动。所谓数据驱动，是指视图是由数据驱动生成的，我们对视图的修改，不是直接操作 DOM，而是通过修改数据。它相比我们传统的前端开发，如使用 jQuery 等前端库直接修改 DOM，大大简化了代码量，提高工作效率。特别是当交互复杂的时候，只关心数据的修改会让代码的逻辑变的非常清晰，这样的代码非常利于维护。

以下面这个例子来说(chapter02 test.html)：



我们会从数据驱动的角度来分析上述实现过程。

## 一．new Vue 发生了什么



此时的this就是vue实例，它本身是没有\_init方法的，那么这个方法定义在哪里呢？

该方法在 src/core/instance/init.js 中定义，我们可以通过debugger验证。

Vue.prototype.\_init = function (options?: Object) {

const vm: Component = this **//this就是生成的Vue实例**

vm.\_uid = uid++

vm.\_isVue = true

// merge options

vm.$options = mergeOptions(**//合并配置**

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)**//初始化生命周期**

initEvents(vm)**//初始化事件中心，**

initRender(vm)**//初始化渲染**

callHook(vm, 'beforeCreate')

initInjections(vm) // resolve injections before data/props

initState(vm)//**初始化 data、props、computed、watcher 等所有跟options相关的数据**

initProvide(vm) // resolve provide after data/props

callHook(vm, 'created')

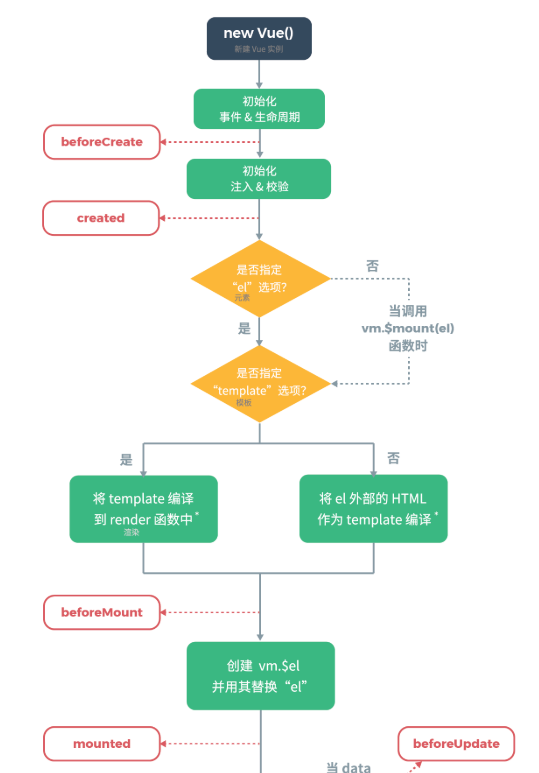
if (vm.$options.el) {

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

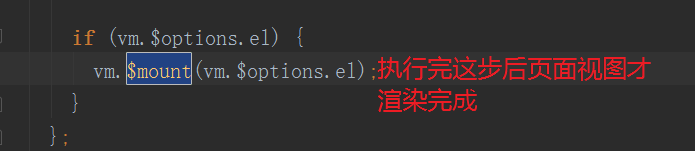
}

}

Vue 初始化主要就干了几件事情，合并配置，初始化生命周期，初始化事件中心，初始化渲染，初始化 data、props、computed、watcher 等等。对应下图：



通过debugger，我们先不关注具体逻辑，我们先关注数据驱动视图这个问题。



## 那么$mount做了什么？

Vue 中我们是通过 $mount 实例方法去挂载 vm 的。

下 src/platform/web/entry-runtime-with-compiler.js 文件中定义：

const mount = Vue.prototype.$mount

Vue.prototype.$mount = function (

el?: string | Element,

hydrating?: boolean

): Component {

el = el && query(el)**//el转换成真实DOM**

/\* istanbul ignore if \*/

**//对el进行限制**

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

// resolve template/el and convert to render function

if (!options.render) {**//先判断有没有render函数**

let template = options.template

if (template) {**//没有render就判断有没有template参数**

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.template}`,

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) {**//没有render，也没有template，就根据el生成template**

template = getOuterHTML(el)

}

if (template) {

/\* istanbul ignore if \*/

if (process.env.NODE\_ENV !== 'production' && config.performance && mark) {

mark('compile')

}

**//根据templ**ate**生成render函数，compileToFunctions涉及到编译的知识，暂时不讲**

const { render, staticRenderFns } = compileToFunctions(template, {

shouldDecodeNewlines,

shouldDecodeNewlinesForHref,

delimiters: options.delimiters,

comments: options.comments

}, this)

options.render = render

options.staticRenderFns = staticRenderFns

/\* istanbul ignore if \*/

if (process.env.NODE\_ENV !== 'production' && config.performance && mark) {

mark('compile end')

measure(`vue ${this.\_name} compile`, 'compile', 'compile end')

}

}

}

return mount.call(this, el, hydrating)**//最后再执行原型链上的mount方法进行挂载**

}

原型链上的 $mount 方法在 src/platform/web/runtime/index.js 中定义。

// public mount method

Vue.prototype.$mount = function (

el?: string | Element,

hydrating?: boolean

): Component {

**//el如果是字符串会转换成真实DOM**

el = el && inBrowser ? query(el) : undefined

return mountComponent(this, el, hydrating)**//执行mountComponent**

}

$mount 方法实际上会去调用 mountComponent 方法，这个方法定义在 src/core/instance/lifecycle.js 文件中：

export function mountComponent (

vm: Component,

el: ?Element,

hydrating?: boolean

): Component {

vm.$el = el

if (!vm.$options.render) {

vm.$options.render = createEmptyVNode

if (process.env.NODE\_ENV !== 'production') {

/\* istanbul ignore if \*/

if ((vm.$options.template && vm.$options.template.charAt(0) !== '#') ||

vm.$options.el || el) {

warn(

'You are using the runtime-only build of Vue where the template ' +

'compiler is not available. Either pre-compile the templates into ' +

'render functions, or use the compiler-included build.',

vm

)

} else {

warn(

'Failed to mount component: template or render function not defined.',

vm

)

}

}

}

callHook(vm, 'beforeMount')

let updateComponent

/\* istanbul ignore if \*/

if (process.env.NODE\_ENV !== 'production' && config.performance && mark) {

updateComponent = () => {

const name = vm.\_name

const id = vm.\_uid

const startTag = `vue-perf-start:${id}`

const endTag = `vue-perf-end:${id}`

mark(startTag)

const vnode = vm.\_render()

mark(endTag)

measure(`vue ${name} render`, startTag, endTag)

mark(startTag)

vm.\_update(vnode, hydrating)

mark(endTag)

measure(`vue ${name} patch`, startTag, endTag)

}

} else {**//生成updateComponent 方法**

updateComponent = () => {

**//**vm.\_update**调用 vm.\_render 方法生成虚拟 Node，更新真实Dom**

vm.\_update(vm.\_render(), hydrating)

}

}

// 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

**//再实例化一个渲染Watcher,在它的回调函数中会调用 updateComponent 方法**

new Watcher(vm, updateComponent, noop, { before () {

if (vm.\_isMounted) {

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

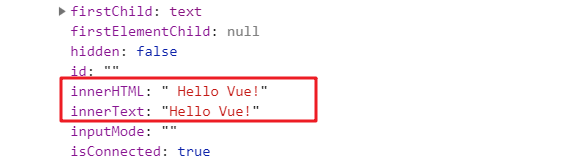
}

## **总结：**

## 整个$mounted的过程，思路很清晰：它会完成整个渲染工作。首先根据template生成一个render方法，然后再实例化一个渲染Watcher 。Watcher 的回调函数是updateComponent 方法，updateComponent又最终会调用vm.\_update 更新 DOM。

Watcher 在这里起到两个作用，一个是初始化的时候会执行回调函数，另一个是当 vm 实例中的监测的数据发生变化的时候执行回调函数。

我们在render函数里面打印出vnode可以看出：



点击按钮后：



这里简单介绍一下几个概念：

### 虚拟DOM

真正的 DOM 元素是非常庞大的，因为浏览器的标准就把 DOM 设计的非常复杂。当我们频繁的去做 DOM 更新，会产生一定的性能问题。（看virtualDom.js）

VNode 是对真实 DOM 的一种抽象描述，它的核心定义无非就几个关键属性，标签名、数据、子节点、键值等，其它属性都是都是用来扩展 VNode 的灵活性以及实现一些特殊 feature 的。由于 VNode 只是用来映射到真实 DOM 的渲染，不需要包含操作 DOM 的方法，因此它是非常轻量和简单的。

### **vm.\_render 用于创建VNode**

vm.\_render 的核心是 createElement，用于创建一个VNode Tree。

### **vm.\_update用于把**VNode**渲染真实Dom**

### Vue 的 \_update 是实例的一个私有方法，它被调用的时机有 2 个，一个是首次渲染，一个是数据更新的时候。

到目前为止，我们真正明白了，数据驱动是如何实现的：新建Vue实例，数据初始化--》根据template生成render方法---》初始化渲染watcher，调用render方法，生成虚拟Dom，再调用\_update生成真实Dom。--》数据变化，watcher的回调函数继续执行---》调用render方法，生成新虚拟Dom，再调用\_update对比虚拟Dom更新真实Dom。从而实现了数据驱动视图的实现。所以这里与有一个关键点：如何捕获数据变化？

## 如何实现捕获数据变化？

测试代码：chapter02

第一步：普通数据变成可监听对象

变化监听目前使用的是Object.defineProperty.。Vue3.0会使用ES6的proxy来实现。

demo01:用Object.defineProperty实现一个数据监听的方法。

存在的问题：

1.Object的属性增加或者删除的时候，都不能监听到变化。用vm.$set()，

第二步：收集依赖

1.在get里面收集依赖， 在set里面触发依赖。看demo02.js

每个key对应一个依赖数组，用于存储这个key的依赖，每个依赖假定是一个函数。当数据变化的时候，执行这些函数，实现数据变化，通知依赖数据变更的目的。

暂定依赖函数放在window.target里面。

2.直接把依赖写在数据监听函数里面，不够解耦，需要抽象出一个专门处理依赖的对象Dep.js

接下来在main.js里面运行一下defineReactiveData01.js

3.存在的问题：实际依赖是不可能是target这么简单,并且依赖的来源可以是各种情况，比如写一个watcher:

vm.watch('a.b.c',function(newVal,oldVal){

console.log(newVal,oldVal)

})，

还比如是computed的。

那么如何在上面的基础上，抽象一个可以处理各种类型的依赖的类？---watcher

看Watcher.js

简单来说，就是当遇到一个watch或者compute的时候，就新建一个watcher对象，并执行watcher对象的get方法,而因为一旦执行了watcher的get方法，就把整个watcher对象添加到对应的key的依赖数组中了，就可以实现监听数据变化。而当数据变化的时候，就执行watcher对象的update方法，这样就实现数据响应了。

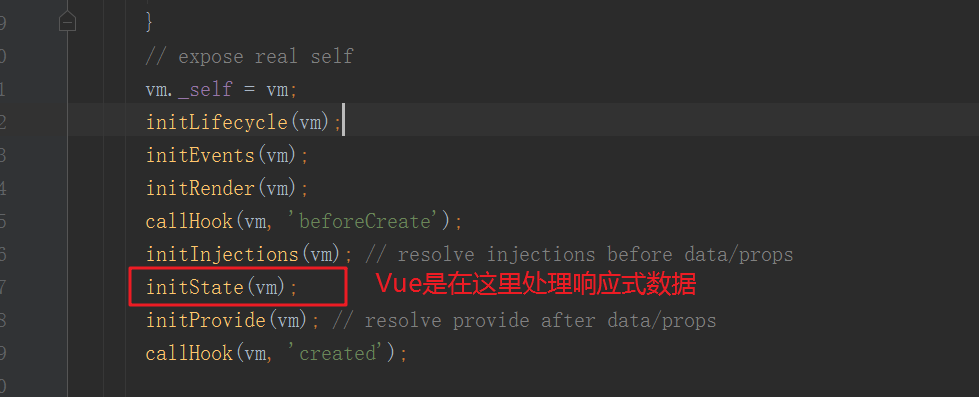
在main.js里面运行一下demo03.js

第三步：递归监听data所有的key.

到目前为止，我们知道了数据监听的原理，也抽象出了Dep类来专门处理依赖收集，同时也抽象了一个依赖类:watcher。那接下来就是简单的，递归所有的data的key设置数据监听---Observer类。Observer.js

重新写一个干净的defineReactiveData.js。然后在main.js里面运行一下demo04.js

## Vue在哪里处理数据？



function initState (vm) {  
 vm.\_watchers = [];  
 var opts = vm.$options;  
 if (opts.props) { initProps(vm, opts.props); }  
 if (opts.methods) { initMethods(vm, opts.methods); }  
 if (opts.data) {  
 initData(vm);**//这里处理响应式数据** } else {  
 observe(vm.\_data = {}, true /\* asRootData \*/);  
 }  
 if (opts.computed) { initComputed(vm, opts.computed); }  
 if (opts.watch && opts.watch !== nativeWatch) {  
 initWatch(vm, opts.watch);**//如果有watch，也是在这里进行初始化** }  
}

function initData (vm) {  
 var data = vm.$options.data;  
 data = vm.\_data = typeof data === 'function'  
 ? getData(data, vm)**//获取data** : data || {};  
 // observe data  
 observe(data, true /\* asRootData \*/);**//新建observe**}

function observe (value) {  
 if (!isObject(value) || value instanceof VNode) {  
 return  
 }  
 var ob;  
ob = new Observer(value);//**//新建Observer**  
 return ob  
}

**//initWatch是新建一个个自定义的watcher对象**

function initWatch (vm, watch) {  
 for (var key in watch) {  
 var handler = watch[key];**//对每个watch新建一个wather**  
 createWatcher(vm, key, handler);  
 }  
}

function createWatcher (vm,expOrFn,handler,options) {  
 return vm.$watch(expOrFn, handler, options)  
}

Vue.prototype.$watch = function (expOrFn,cb,options) {  
 var vm = this;  
 if (isPlainObject(cb)) {  
 return createWatcher(vm, expOrFn, cb, options)  
 }  
 options = options || {};  
 options.user = true;  
 var watcher = new Watcher(vm, expOrFn, cb, options);**//新建Watcher对象**  
}

总结：在Vue初始化的时候，对data/props/watch/computed等数据进行初始化，对data/props是进行Observer响应式数据处理，而对watch/computed等生成对应的watcher对象。这样当数据变化的时候，就可以触发对应的watcher，执行其update方法。

而在Vue实例生成的时候，会生成一个渲染watcher对象。这个对象传入的是updateComponent方法作为回调函数。这个方法在执行render函数的时候，会触发模板数据的getter，把这个渲染watcher加入到对应数据的Dep里面。当数据变化的时候，就会通知渲染watcher重新执行updateComponent方法,从而重新执行render和update方法，更新真实Dom。这个就是数据驱动的整个过程。注意一个vue组件对应生成一个渲染watcher。