2017年6月22日 星期四

Vue.js源码解析

1.判断是否使用new初始化Vue，通过其中的this指针指向判断
在使用new初始化时，创建一个新对象，并寻找它的原型链上的元素，把其中的this指向自身

```
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)
}
```

2.Vue自身的类非常简单，全部的功能都是用过mixin挂载上去的，这样有一个好处就是提高了解耦程度，但是既然是mixin那他就是不符合函数式编程的概念的，对于共有属性的一些操作必须遵从顺序

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

instance/index.js

initMixin(Vue)

3.全部使用的都是flow进行类型检查，这样的对比ts有什么好处呢？根据尤雨溪本人所说，这样的做法是为了避免使用ts带来的巨大工作量，flow可以对每个文件单独进行改动，难度较低一些，而且相对于全部推翻重做风险低很多

4.在对设置项option进行设置的时候，首先对内部模块进行了优化，因为其并不需要设置这些配置项，然后对配置项执行了mergeOptions的方法，此放方法较为复杂，首先进行了数据的扁平化

```
normalizeProps(child)
normalizeDirectives(child)
```

可以看到也没有做函数式的操作，都是通过共享参数直接传进去处理（奇妙）

```
const extendsFrom = child.extends
if (extendsFrom) {
  parent = mergeOptions(parent, extendsFrom, vm)
}
if (child.mixins) {
  for (let i = 0, l = child.mixins.length; i < l; i++) {
    parent = mergeOptions(parent, child.mixins[i], vm)
  }
}
```

随即进行了递归操作，对子元素的配置项进行了遍历（应该是之后添加上去的）

5.做了一个判断看是否原生支持Proxy代理，方法很粗暴，如果有就包装一层代理还设置了keyCode，不知道是要干嘛（开发模式使用，多半是快捷键刷新之类的）

```
const hasProxy =
  typeof Proxy !== 'undefined' &&
  Proxy.toString().match(/native code/)
```

6.喜闻乐见的有进行了几次初始化，当然中间有两个钩子函数的调用，可以看到在创建vue实例的时候进行了哪些工作

```
// expose real self
vm._self = vm
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')
```

7.initLifecycle函数：对生命周期进行初始化，这个阶段挂载上了很多属性，应该是给后面做准备的用途

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

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

vm._watcher = null
vm._inactive = null
vm._directInactive = false
vm._isMounted = false
vm._isDestroyed = false
vm._isBeingDestroyed = false
```

8.其他函数也是大同小异，从注释中可以看出，这个init是对根节点做的特异化的init，很多是为了给vue这个tree中的子元素提供一个上下文环境

9.在生命周期中，可以看到在data和prop之前还有一个通过inject属性注入的东西，可以添加下这个属性试一试

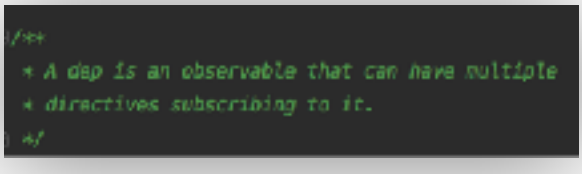
10.initState方法中对对象中的属性都进行了处理

prop，method，data，computed，watch都做了初始化

11.对于props，他是一个对象，vue对其中的属性都进行缓存处理，缓存的优势表现在转化成数组过后操作会更加便捷，并且性能也会更好

在这个过程中，会对每一个prop的可用性进行判断，同时使用比较核心的 **defineReactive** 方法对这个属性进行响应式处理（监听他的变化，并做出改变）

在defineReactive方法中，首先创建一个Dep对象的实例，这个对象拥有作为监视器，可以在上面挂载多个订阅方法，（directive 指令）



在创建过后对属性进行校验，获取对象的描述符，如果发现当前配置的属性configable被设置为false，那么直接跳过这个属性

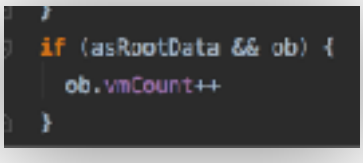
在对属性进行观察时，如果是数组则会一层层递归深入，但是如果是类数组元素呢？测试一下

Vue.extend方法会在vue对象的原型链上加上属性，所以在这个地方是自定义的属性的话会通过代理直接挂载到vm的_props属性上面，而并不是直接的vm对象

注：每个属性都会新创建一个dep对象，这个dep对象上可以挂载多个订阅者，事件的通知就是通过在set中调用其上的notify方法实现的，这个方法依次调用订阅的函数，并且递归的传递给它的子元素

12.props处理完成过后，method依次直接被挂载到vm上面

13.data属性初始化的时候会先判定是否是一个函数，如果是函数的话则使用起执行过后返回的值作为data挂载，如果这个时候不是返回的一个对象也会报错；在执行操作完成后属性都会被放到_data上面，如果与props产生冲突则发出一个警告



对于一个正常元素，会把data上的元素都用observe处理，绑定到一个观察者上面，同时相应的vmCount也会 +1

值得注意的是如果data属性为空时，这里会把他认为是一个根元素，并且把vmCount的数字加一，这里的ob指的是数据的观察者

14.computed的初始化，其实底层也是使用的watcher，首先会把其赋值为一个Object.create(null)这个对象，和平时的{}有什么区别呢？ 前一种方法创建的对象已经在原型链顶端了，他没有__proto__的属性

Watcher对象有一个lazy选项，可以惰性求值，即在constructor函数中先不求出结果，过后需要的时候再来

Watcher对象有两个用处：一个是用在这里，还有用途是在指令的时候用到

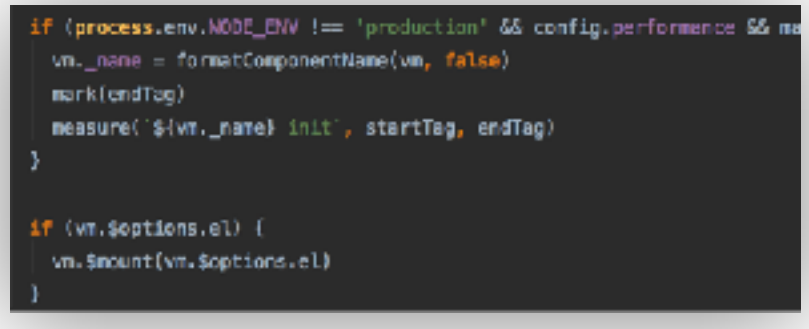
15.Watcher和Observer有什么区别呢？同是检测数据变化？

16.根据其优先级来看不可重复的顺序
Props > data > computed > watch

17.watch的初始化就是直接简是谁，没有发现创建watcher对象

18.在进行了以上的initState步骤过后，会执行provide的初始化，这个东西在官方文档中没有写到，不知道是拿来干嘛的，现在只知道是把provide属性安装到了_provided上面

完成这一步过后调用created的回调函数，执行相关操作

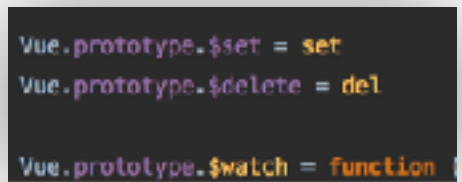


此时我们的组件也已经初始化完成，如果在性能测试下可以通过mark函数打印出这一段所花费的时间

再过后直接通过mount方法把组件挂载到对应元素上，如果没有就等待手动执行

StateMixin(Vue)

1.对$data和$props做好了setter的设置，用户试图更改直



接打出警告并且不会返回任何值

```
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()
  }
```

2.对$set方法进行了统一设置，函数中表明，在对其进行改动的时候会去拿到对应的target，并且把target上面的ob对象进行触发操作，如果没有的话，则使用createReactive方法新创建一个观察者绑定上去

3.在底下的watch部分实现代码中，发现调用watch函数会创建一个新的watch对象，并且返回一个函数，直接可以取消对表达式的监听

同时如果设置了immediate属性的话，回调函数会立即执行

在teardown函数中，只会对正在工作的（即active === true）的watcher进行解绑，由于这个操作非常耗时（一个一个的通过removeSub移除），所以在销毁元素时的直接跳过了这部分操作。

EventsMixin(Vue)

1.实现了一个基本的事件系统，意外的是没有组件的父元素子元素之间传递，不知道这一步是在哪里完成的

```
Vue.prototype.$once = function (event: string, fn: Function): Component {
  const vm: Component = this
  function on () {
    vm.$off(event, on)
    fn.apply(vm, arguments)
  }
  on.fn = fn
  vm.$on(event, on)
  return vm
}
```

2.基本的$on, $once, $off, $emit功能都加了上去，挂载到了_events属性上面的一个对象，通过事件名字分组，在调用时依次执行

3.值得一看的还有once的实现方式，通过对绑定的函数进行修饰，使用高阶函数，在执行操作时把事件解绑（注意一定要先解除绑定再执行操作，不然可能造成递归调用系统卡死）

LifecycleMixin(Vue)

1.在这个处理过程中没有完整的生命周期，背后应该是scheduler在操控，update实际上是_update方法，在执行更新前会调用beforeUpdate的回调函数，

这里出现了一个很重要的_vnode属性，应该存储的是这个节点挂载的信息，

当第一次挂载时（第一次render应该也是调用了这个方法），会使用__patch__方法，在$el元素上渲染我们的元素，这一点和react非常相似

还有一个相似的地方就是，vue中也存在高阶组件的概念，通过比较自身vnode和父节点的_vnode如果相同，则认为其是高阶组件

```
// if parent is an HOC, update its $el as well
if (vm.$vnode && vm.$parent && vm.$vnode === vm.$parent._vnode) {
  vm.$parent.$el = vm.$el
}
```

2.这个时候很容易出现一个疑问，为什么没有进行递归更新呢？其实这个步骤是有一个叫scheduler的对象来控制的

```
// updated hook is called by the scheduler to ensure that children are
// updated in a parent's updated hook.
```

3.$forceUpdate干了什么？其实只做了一个操作，就是把所有watcher重新计算一次，（watcher中保存的是所有watch的表达式，那computed是不会受影响咯？）

4.由于销毁操作会消耗一定时间，并且可能重复调用导致出现一些东西undefined的问题，所以有一个名叫_isBeingDestroyed的属性保存了是否正在被删除，就像前面说过的一样，这个变量还被用在removeSub中做优化

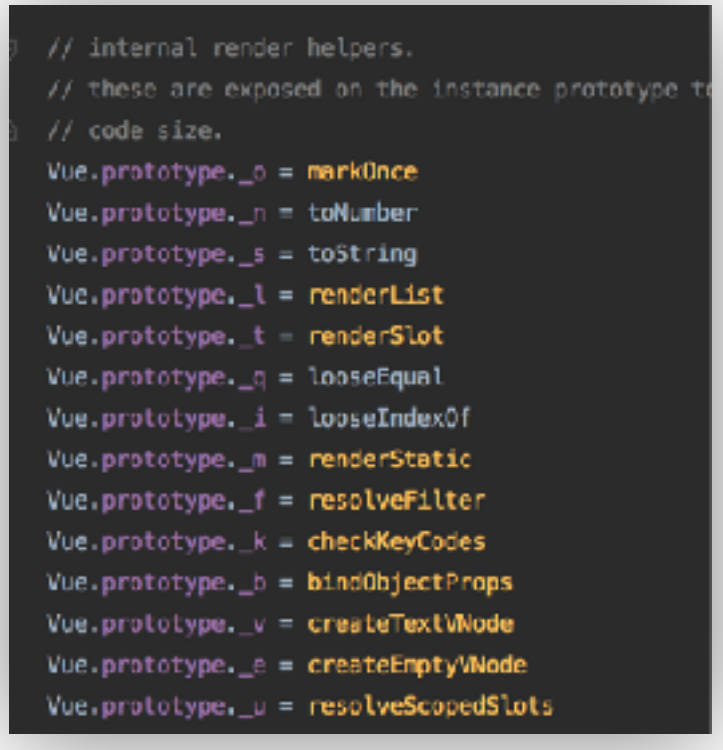
卸载时候总得来说做了几项工作：
· 调用beforeDestroy的钩子函数
· 将自身状态设置为正在销毁
· 从父组件中移除自身的连接
· 移除所有watcher（代码有一部分很迷，如右图）
· 将data的观察者减一 vm._data.__ob__.vmCount－
· 调用__patch__方法进行最后一次渲染，这次渲染传入null
· 调用destroyed的钩子函数
· 通过$off关闭事件监听器
· 清除vue的依赖和DOM的依赖

```
// teardown watchers
if (vm._watcher) {
  vm._watcher.teardown()
}
let i = vm._watchers.length
while (i--) {
  vm._watchers[i].teardown()
}
```

RenderMixin(Vue)

1.添加上了一个nextTrick的方法，这个方法的实现非常有趣，优先使用了Promise，并在Promise不可用的时候使用兼容性好一些的MutationObserver接口，作者还表示第二个东西会在iOS下面有一些BUG，其中也牵扯到一些关于MacroTask和MicroTask的问题，非常有意思

做了一些降级处理，本来是想用microTask的，如果前两个不能用最后还是会换成在marco中工作的setTimeout函数



2._render函数中做了一些优化处理，比如将slot进行缓存，在render时首先把他们克隆一遍

在这一步中，操作和返回的元素并不是真实的DOM节点，而是名为vnode的vue节点，如果出现了错误的情况，可能会返回一个空的vnode

同时，为了节省代码大小，也是蛮拼的

这些操作**都是针对指令做的**

· markOnce用于v-once的指令，使用一个带_once_前缀的固定key，将元素标记为不变的元素

· renderList用于v-for的指令，可以对数组，数字，字符串以及对象进行遍历操作（其实数字部分可以传入NaN或者Infinity都会出现BUG）

· renderSlot用于渲染slot标签，区别了一下有两种slot，一种是带有作用域的有外部标签传递了属性给他，还有一种是直接拿来渲染的，两者只是渲染的函数有点不同

· renderStatic渲染静态组件树？不知道有什么用，但是前缀是用的_static_

· resolveFilter底层调用的是resolveAsset函数，不过给他传递了一个filter参数进去，然后转化成了驼峰属性？很迷

· checkKeyCodes用于检查按键的监听，主要是用到v-on上面（忘了怎么用的了）

· bindObjectProps应用到v-bind上面，其中比较有意思的是，对class和style的处理是单独提出来的

style是一个对象还可以理解，那么class也单独提出来的原因是什么呢？

根据属性的设置，可以选择设定为props还是dom上的attribute

· createTextVNode只是纯粹的生成了字符节点

· createEmptyVNode也是类似的，生成的是一个空节点，其isComment属性被设置为true（就是在HTML文件中看到的空注释符）

·resolveScopedSlot 即为创建slot相关对象，没有什么特殊的

Vue.runtime.common.js

工具函数设计解析

1.在很多函数中都会发现没有直接通过{}创建对象，而是使用的Object.create(null)，个人觉得应该是有两个原因，第一是如果用{}可以在原型链上添加别的属性，影响数据处理，还有一点是可以省一定的内存空间

2.cache函数可以缓存纯函数操作的结果，注意这里的要求一定要是纯函数！其他函数的话会跟执行的环境有关，导致缓存的结果不可信

3.把驼峰表达式转化成连接线连接的函数也很有意思

```
var hyphenateRE = /([^-])([A-Z])/g;
var hyphenate = cached(function (str) {
  return str
    .replace(hyphenateRE, '$1-$2')
    .replace(hyphenateRE, '$1-$2')
    .toLowerCase()
});
```

这个函数应该也是做了很多考虑的，由于设置的匹配每次会匹配两个连续的，所有有必要执行两次

同时没有用网上流传的/([a-z\d])([A-Z])/g类似的方法也是极大的增强了兼容性，像ABCDE这种奇葩的也可以转化成为a-b-c-d-e

4.在2.3.4版本的158行代码可以看到作者提到：*Simple bind, faster than native*，可见作者表示自定义的简单封装的bind还要比原生的bind函数更快，这个需要测试一下

5.将类数组转化为数组的方法中没有使用slice等操作，说起来你可能不信，他是一个一个遍历的，应该是处于兼容性的考虑（但是这样会不会出现一个问题：本来是稀疏数组类似的元素结果相对应的值全部变为了undefined，用in操作符变得无法检测了）

6.looseEqual方法对比两个对象没有使用递归遍历的方法，直接使用的JSON.stringify方法变换后对比，没有考虑特殊数字等的问题

7.once函数的实现，非常经典，利用了闭包的特性

```
function once (fn) {
  var called = false;
  return function () {
    if (!called) {
      called = true;
      fn.apply(this, arguments);
    }
  }
}
```

8.在生命周期的钩子函数中有两个不为人熟知的钩子：activated和deactivated，分别是用于keep-alive组件激活和隐藏的时候调用的函数

9.在判断是否以横线开始的属性时用到了数字检测，也许是为了避免UTF-16等编码不同，或者双字节文字（如emoji表情）的影响吧

```
function isReserved (str) {
  var c = (str + '').charCodeAt(0);
  return c === 0x24 || c === 0x5F
}
```

基础类解析

1.Dep类：A dep is an observable that can have multiple directives subscribing to it.

有一个变量uid$1是用来唯一标识它的

他还有一个静态的属性target，他是一个数组，其中存储的是Watcher，调用depend方法会把二者连起来

2.Observer：最主要的功能就是将某一个对象的所有属性变为了setter和getter处理，即属性劫持

在创建时会把他和一个新创建的dep连接起来，她还会把属性自身上的__ob__设置为自己（但是自己并没有返回值）

当然上面的方法是针对于对象的，那数组的话就是单独拿出来对上面的方法进行修改，添加上了监视功能

3.Watcher

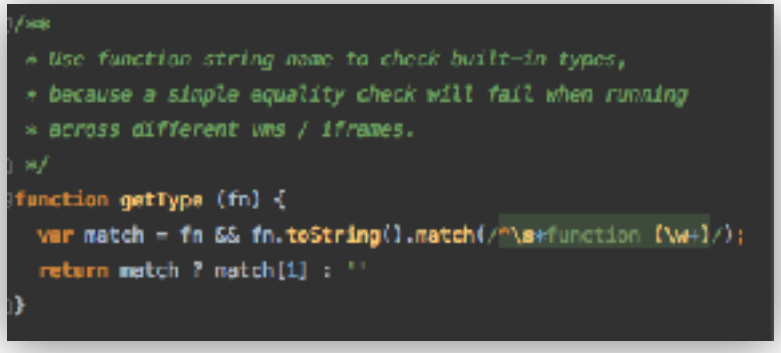
修改操作

1.对数组原型上的方法进行了改造，涉及到
'push',
'pop',
'shift',
'unshift',
'splice',
'sort',
‘reverse'
调用这些方法都会触发dep的notify事件

2.对于在属性传递时，可能出现向组件子元素传递一些嵌套元素被freeze的元素的情形，这个时候便不能对其进行监听，所以出现了shouldConvert这样的属性标识避免被监视

3.官方文档里面说的props是只能传递一个数组进去，但是实际上传对象进去都可以，他只会把键取出来拼到另一边

4.有时候会出现跨域（iframe）的问题，这个时候默认的window不是指向一个地方的，有可能导致平时使用的一些类型对比函数出现错误，使用字符串转化后比较就要稳妥一些



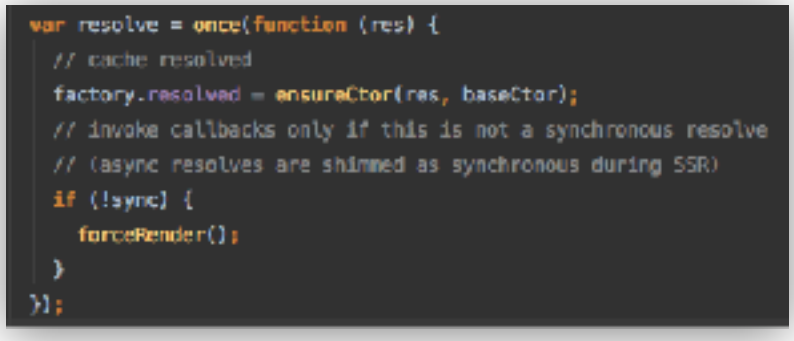
这个函数只是会把传入的Number对象等转化为字符串，这是针对跨iframe做的处理

5.在对属性进行处理时，面对几种符号开头的事件会有特殊处理方式
& - passive不会被阻止，~ - 只执行一次，！- 捕获阶段执行执行

6.vue自带了异步加载组件的功能，利用的是resolveAsyncComponent函数（vue.runtime.common.js:1963），第一个参数就是一个组件经过处理的factory工厂函数，我们在配置时设置的超时都在这里作用

里面有个forceRender函数会对factory所在的上下文遍历进行forceUpdate，这里的上下文当然也是过后传进去的，即需要加载组件地方的上下文

同时这里的设计也有需要注意的一点就是对SSR的处理，在进行异步操作之前先将同步操作标识sync设置为true，然后进行（异步或同步的）加载操作，在marcotask中继续向下将sync置为false，这时候变为要求异步加载；用处在哪里？



若之前里面的加载是异步的话，变量变为false会保证异步加载完成后forceRender执行；如果是同步的话实际上到改变sync值的那一步已经加载好了资源，这个时候直接返回加载好的组件就行了；

7.在挂载（mount）组件的函数中，如果没有配置render函数，则会默认分配一个空VNode的生成函数，之后再_watcher上添加Watcher对象，并向其中传入update的回调函数；处理完次步骤，在根节点（$vnode为null）触发mounted事件

8.flushSchedulerQueue是一个很神奇的方法，在这个过程中会触发组件的updated事件