# Vue

<https://segmentfault.com/a/1190000038848131>

## 项目中何时开始使用vue

2018年10月份

## 生命周期

|  |  |  |
| --- | --- | --- |
| 生命周期钩子 | 组件状态 | 最佳实践 |
| beforeCreate | 实例创建之后，this指向创建的实例，不能访问到data，computed、watch、methods上的方法和数据 | 常用于初始化非响应式变量 |
| created  [kriˈeɪt] | 实例创建完成，可访问data、computed、watch、methods上的方法和数据，未挂载到DOM，不能访问$el属性，$ref属性内容为空数组。 | 常用于简单的ajax请求，页面的初始化 |
| beforeMount | 在挂载开始之前被调用，beforeMount之前，会找到对应的template，并编译成render函数 |  |
| Mounted | 实例挂载到DOM上，此时可以通过DOM API访问到DOM节点，$ref属性可以访问 | 常用于获取VNode信息和操作，ajax请求 |
| BeforeUpdate | 响应式数据更新时调用，发生在虚拟DOM打补丁之前 | 适合更新之前访问现有的DOM，比如手动移除已添加的事件监听器 |
| Updated | 虚拟DOM重新渲染和打补丁之后调用，组件DOM已经更新，可执行依赖于DOM的操作 | 避免在这个钩子函数中操作数据，可能陷入死循环 |
| BeforeDestroy  [dɪˈstrɔɪ] | 实例销毁之前调用。这一步，实例依然可用，this仍能获取实例 | 常用于销毁定时器，解绑全局变量、销毁插件对象等操作 |
| Destroyed | 实例销毁后调用，调用后，vue实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。 |  |

注意:  
created阶段的ajax请求与mounted请求的区别：前者页面视图未出现，如果请求信息过多，页面会长时间处于白屏状态。

## Vue的初始化过程(new Vue(options))都做了什么？

* 处理组件配置项
* 初始化根组件时进行了选项合并操作，将全局配置合并到根组件的局部配置上
* 初始化每个子组件时做了一些性能优化，将组件配置对象上的一些深层次属性扁平化放到vm.$options选项中，已提高代码的执行效率
* 初始化组件实例的关系属性，比如$parent、$children、$root、$refs等
* 处理自定义事件
* 调用beforeCreate钩子函数
* 初始化组件的inject配置项，得到result[key] = val显示的配置对象，然后对配置对象进行响应式处理，并代理每个key到vm实例上
* 数据响应式，处理props、methods、data、computed、watch等选项
* 解析组件配置项上的provide对象，将其挂载到vm.\_provided属性上
* 调用created钩子函数
* 如果发现配置项上有el选项，则自动调用$mount方法，也就是说有了el选项，就不需要手动调用$mount方法，反之，没提供el选项则必须调用$mount
* 接下来进入挂载阶段

## 什么是父组件，什么是子组件？

我们将某段代码封装成一个组件，而这个组件又在另一个组件中引入，而引入该封装组件的文件叫做父组件，被引用的组件叫做子组件。

## 父子组件，兄弟组件通信

1. 父组件向子组件传值

父组件引用子组件时，通过指令v-bind动态绑定一个属性，再在子组件中通过选项props接收来自父组件的数据。

* 父组件在子组件标签中绑定自定义属性；
* 子组件通过props[prɒps] 属性进行接收。

1. 子组件向父组件传值

子到父，在使用子组件的时候绑定自定义事件，在子组件内部拿到事件名称，再通过this.$emit(‘事件名’)触发并传递参数

* 父组件在子组件标签中绑定自定义事件；
* 子组件通过this.$emit()方法触发自定义事件，传值给父组件。

1. 兄弟组件之间传值

* 共同传给父组件，再有父组件分发（状态提升）；
* 使用Vuex
* 利用bus事件总线

其他的方式还有全局bus、provide/inject、refs/parent/children、attrs/$listeners

## 父组件有数据变更，如果通知到子组件

父组件data中的属性修改后，通过props传递给子组件的属性也会改变，且父组件和子组件的beforeUpdate和updated都会被触发，触发顺序为：

父组件beforeUpdate

子组件beforeUpdate

子组件updated

父组件updated

## 父组件子组件的生命周期顺序

父组件beforeCreate

父组件Created

父组件beforeMount

子组件beforeCreate

子组件Created

子组件beforeMount

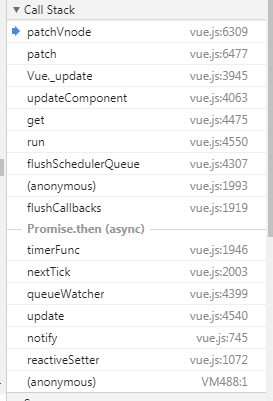
子组件Mounted

父组件Mounted

## 虚拟dom，diff算法

虚拟dom是轻量级的JavaScript对象，有渲染函数创建。

* 1. vue的diff算法
     1. 调用栈



## 什么是单向数据流？

数据通常按照一个方向来传递，通常从父传递子组件，子组件要修改父组件数据需要通知父组件自己修改，目的是便于状态的管理和解耦。

如果不加以限制，任何子组件内部直接修改父组件数据就会影响到其他引用相同数据的组件的显示，从而造成数据混乱。

## 单向绑定

## Vue2实现数据的双向绑定/响应式原理

Vue2.0使用Object.defineProperty()实现数据响应

Vue遍历data中的数据，使用Object.defineProperty()劫持getter和setter

在getter中做数据依赖收集

在setter中监听数据的变化，并通知订阅了当前数据的地方

Vue通过数据劫持（Object.defineProperty）+ 订阅发布模式

监听器Observer：用来劫持并通过Object.defineProperty监听所有属性（转变成setter和getter形式），如果属性发生变化，就通知订阅者

订阅器Dep：用来收集订阅者，对监听器Observer和订阅者Watcher进行统一管理

订阅者Watcher：监听器Observer和解释器Compile之间通信的桥梁；如果收到属性变化通知，就会执行相应的方法，从而更新视图。

解析器Compile：将template模版解析成能够使用响应式属性的渲染函数。

可以解析每个节点的相关指令，对模板数据和订阅器进行初始化。

主要做的事情是：

1. 在自身实例化时往属性订阅器（Dep）里面添加自己
2. 自身有一个update方法
3. 待属性变动dep.notice()通知时，就触发自身的update方法，并触发解析器（compile）中绑定的回调。

## Vue3.0采用proxy实现双向绑定

Object.defineProperty()的问题有：

1. 检测不到对象属性的添加和删除

Vue的set是能让vue知道你添加了属性

1. vue2.0不监控数组下标的变化，导致直接通过数组的下标给数组设置值，不能实时响应。

Js可以通过Object.defineProperty()监控数据下标的变化，但vue的设计者出于性能考虑取消了这个功能，并不是原生不支持。

1. 只能劫持对象的属性，因此我们需要对每个对象的每个属性进行遍历，如果属性值也是对象，就需要深度遍历，当data中的数据比较多且层级很深的时候，会有性能问题，因为要遍历data中所有的数据并给其设置成响应式。

使用Proxy的优势：

1. 可直接监听数组类型的数据变化。
2. 监听的目标为对象本身，不需要像Object.definePropery一样遍历每一个属性，有一定的性能提升。
3. 可拦截apply、ownkeys、has等13中方法。
4. 直接实现对象属性的新增和删除。

## 异步更新队列

当触发数据更新通知时，dep通知watcher进行数据更新，这时watcher会将自己加入到一个异步的更新队列中；

然后更新队列会将传入的更新操作进行批量处理；

这样就达到了多次更新同时完成，提高了用户体验，减少了浏览器的开销，增强了性能。

采用微任务执行刷新队列函数。

## nextTick

[JS的运行机制](#_Js执行机制)

实现原理：基于语言执行机制实现，直接创建一个异步任务，那么nextTick自然就达到再同步任务后执行的目的。

nextTick是vue的更新策略，也是性能优化手段，基于JS执行机制实现。

Vue中我们改变数据时不会立即触发视图，如果需要实时获取最新的DOM，这个时候可以手动调用nextTick。

$nextTick()会将我们传入的函数加入到异步更新队列中，

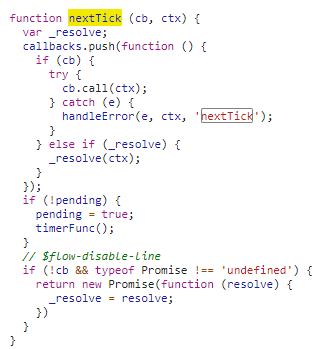
nextTick将函数放入一个callBacks的数组中；

然后推入到异步任务队列（微任务或者宏任务）中。

根据浏览器的渲染机制，渲染线程是在微任务执行完成之后运行的。渲染线程没运行，怎么拿到Dom呢？

因为，渲染线程只是把Dom树渲染成UI而已，vue更新Dom后，在Dm树中，新的Dom节点已经存在了，Js线程就可以拿到新的Dom了。除非开发者读取Dom的计算属性，触发了强制重流渲染线程才会打断js线程。

nextTick既是Vue的全局API，也是Vue的实例方法



## v-model的实现原理

|  |
| --- |
| <input v-model=”searchText” >  等价于:  <input v-bind:value=”searchText” v-on:input=”searchText = $event.target.value” > |

获取到页面表单上的v-model属性，拿到属性值val，为节点绑定input事件，同时把data中属性为val的属性值赋值给当前的表单value；

当前{{}}节点的时候，就创建一个观察者（Watcher），把这个观察者添加到当前val属性的发布者上。

## v-for的key为什么不建议使用数据下标

key是Vue识别节点的一个通用机制。

Vue更新使用v-for渲染的元素列表时，它默认使用“就地更新”的策略。如果数据项的顺序被改变，Vue将不会移动DOM元素来匹配数据项的顺序，而是就地更新每个元素，并且保证她们在每个索引位置正确渲染。

这个默认的模式是高效的，但是只适用于不依赖子组件状态或临时DOM状态（例如：表单输入值）的列表渲染输出。

为了给Vue一个提示，以便它能跟踪每个节点的身份，从而重用和重新排序现有元素，你需要为每项提供一个唯一key属性。

建议尽可能在使用v-for时提供key属性，除非遍历输出的DOM内容非常简单，或者是刻意依赖默认行为以获取性能上的提升。

简单的dom列表，不带key的时候只更新innerText。

Key是给每一个vnode的唯一id，可以依靠key更准确、更快的拿到oldVnode中对应的vnode节点；

1. 更准确

因为带key就不是就地复用了，在sameNode函数a.key ===b.key对比中可以避免就地复用的情况。所以也更加准确。

1. 更快

利用key的唯一性生成map对象来获取对应节点，比遍历方式更快。

patchVnode、updateChildren、sameVnode

## computed和watch的区别

computed是计算属性，它会根据所依赖的数据动态显示计算结果；

计算属性的结果会被缓存，除非依赖的响应式属性变化才会重新计算。

注意：如果某个依赖（比如非响应式属性）在该实例范畴之外，则计算属性是不会被更新的。

computed应用场景

1. 适用于一些重复使用数据或复杂及费时的运算
2. 如果我们需要的数据依赖其他的数据的话

computed的成员可以只定义一个函数作为只读属性，也可以定义为get/set变成可读写属性

watch 是对data的数据监听回调，当依赖的data的数据变化时，会执行回调。在回调中传入newVal和oldVal两个参数。

Watch应用场景：

当在data中的某个数据发生变化时，我们需要做一些操作

watch 普通监听和深度监听

数据变化时执行异步操作，这个时候使用watch最合适。

computed适合一个数据被多个数据影响，watch使用于一个数据影响多个数据。

computed实现原理：

var computedWatcherOptions = { lazy: true };

function initComputed (vm, computed) {

  // $flow-disable-line

  var watchers = vm.\_computedWatchers = Object.create(null);

  // computed properties are just getters during SSR

// 计算属性在服务器渲染的情况下只有getter。

  var isSSR = isServerRendering();

  for (var key in computed) {

    var userDef = computed[key];

    var getter = typeof userDef === 'function' ? userDef : userDef.get;

    if (getter == null) {

      warn(

        ("Getter is missing for computed property \"" + key + "\"."),

        vm

      );

    }

// 如果是服务器渲染，Vue不会为计算属性添加Wather

    if (!isSSR) {

      // create internal watcher for the computed property.

      watchers[key] = new Watcher(

        vm,

        getter || noop,

        noop,

        computedWatcherOptions

      );

    }

    // component-defined computed properties are already defined on the

    // component prototype. We only need to define computed properties defined

    // at instantiation here.

    if (!(key in vm)) {

      defineComputed(vm, key, userDef);

    } else {

      if (key in vm.$data) {

        warn(("The computed property \"" + key + "\" is already defined in data."), vm);

      } else if (vm.$options.props && key in vm.$options.props) {

        warn(("The computed property \"" + key + "\" is already defined as a prop."), vm);

      }

    }

  }

}

watchers 也就是vm.\_computedWatchers对象的引用

存放每个计算属性的观察者watcher实例

* 定义一个computed属性，watch可以监控吗？

答：可以

## computed缓存实现原理

computed的计算属性有缓存机制，只有当其依赖的响应式数据发送变化时才会清空缓存重新计算结果

1. 其缓存机制本质是通过一个dirty属性控制的，只有dirty为true时才会重新计算结果替换缓存。
2. dirty只有当其响应式数据发生变化时才会设置为true，重新计算后会再次被设置为false。

## v-if和v-show的区别

v-if在进行切换时，会直接对标签进行创建和销毁，不显示的标签不会加载在DOM树中。

v-show在进行切换时，会对标签的display属性进行切换，通过display：none来隐藏元素。

一般来说，v-if的性能开销会比v-show大，切换频繁的标签更适合使用v-show。

## 一个组件的data选项必须是一个函数

每个实例可以维护一份被返回对象的独立拷贝。

## vuex

vuex是vue的一个插件，用来做状态管理

其实我们可以把vuex看做是一个仓库，我们把需要共享的数据统一存放在仓库中，谁需要，直接取就可以。然后提供好一个统一修改机制（提交mutation）来修改仓库里的数据。

仓库解决了数据传递层级和同级组件数据传递问题。

1、State：存放状态（数据）的地方

2、Mutation：修改状态的地方，用于变更store里面的数据

* 只能使用mutation修改state
* 为什么要使用mutation,方便后期的维护。
* 一个对象，里面可以定义多个函数。
* Mutation对象里面的函数不能执行异步操作

**触发mutation的第一种方式：**

* 通过this.$store.commit(‘add’)调用mutation对象里面的函数add；
* Mutation对象里面的函数的第一个参数是state，第二个参数是this.$store.commit(‘add’,2)传递过来的数据

**触发mutation的第二种方式：**

|  |
| --- |
| Import { mapMutation } from ‘vuex’;  …mapMutations([‘add’])  btnHandler(){  this.add()  } |

3、Action：做异步操作，并提交mutation修改状态的地方

* 如果通过异步操作更新数据，必须调用Action，而不能实现mutation,但是Action中还是要通过触发mutation的方式间接更改数据。

const store = new Vuex.Store({

    state: {

        conut: 0

    },

    mutations:{

        add(state){

            state.count ++

        }

    },

    Actions: {

        addAsync(context){

            setTimeout(()=>{

                context.commit('add')

            })

        }

    }

})

* 触发Actions的方式一：

this.$store.dispatch('addAsync')

* 触发Actions的方式二：

import { mapActions } from 'Vuex';

new Vue({

    methods:{

        ...mapActions(['addAsync']),

        btnHandler(){

            this.addAsync()

        }

    }

})

4 、Getters：派生状态的地方

对Store中的数据进行加工处理形成新的数据，类似Vue的计算属性。

Store中数据发生变化，Getter的数据也会跟着变化。

const store = new Vuex.Store({

    state: {

        conut: 0

    },

    getters: {

        showNum(state){

            return "当前最新的数量是" + state.count

        }

    }

}

* 调用getters的方式一：

this.$store.getters.showNum

5、module

使用单一状态树，应用的所有状态会集中到一个比较大的对象。当应用变得非常复杂时，state对象就可能变得相当臃肿。

为了解决以上问题，Vuex允许我们将store分割成模块（module）。每个模块拥有自己的state、mutation、action、getter、甚至是嵌套子模块。

## vue v-cloak 指令

可以使用 v-cloak 指令设置样式，这些样式会在 Vue 实例编译结束时，从绑定的 HTML 元素上被移除。

当网络较慢，网页还在加载 Vue.js ，而导致 Vue 来不及渲染，这时页面就会显示出 Vue 源代码。我们可以使用 v-cloak 指令来解决这一问题。

在简单项目中，使用 v-cloak 指令是解决屏幕闪动的好方法。但在大型、工程化的项目中（webpack、vue-router）只有一个空的 div 元素，元素中的内容是通过路由挂载来实现的，这时我们就不需要用到 v-cloak 指令咯。

## **vue-rouder原理**

**什么是单页应用？**

单页应用的全称是 single-page application，简称 SPA，它是一种网站应用的模型，它可以动态重写当前的页面来与用户交互，而不需要重新加载整个页面。

SPA就是一个WEB项目只有一个HTML页面，一旦页面加载完成，SPA不会因为用户的操作而进行页面的重新加载或跳转。取而代之的是利用JS动态的变换HTML的内容，从而模拟多个视图间的跳转。

**SPA的问题：**

1. SPA无法记住用户的操作记录，无论是刷新、前进还是后退，都无法展示用户真实的期望内容。

**工作原理：**

SPA 的一个重要实现就是改变路由时，页面不刷新。实现这个功能，通常有两种方式：使用 window.history 对象或 location.hash。

1. hash模式：url后面的hash值的变化，浏览器既不会向服务器发出请求，浏览器也不会刷新，每次hash值的变化会触发hashchange事件。
2. history模式，利用HTML5中新增的pushState()和replaceState()方法，这两个方法应用于浏览器的历史记录栈，在当前已有的back、forward、go的基础之上，它们提供了对历史记录进行修改的功能。只是当它们执行修改时，虽然改变了当前的URL，但浏览器不会立即向后端发送请求。

* replaceState()方法使用state objects、title和URL作为参数，修改当前历史记录实体

语法：history.replaceState(stateObj, title[, url])

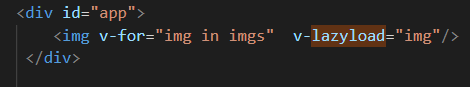
stateObject 状态对象是一个JavaScript对象；

url可选，新的URL和当前的URL必须是同源的。

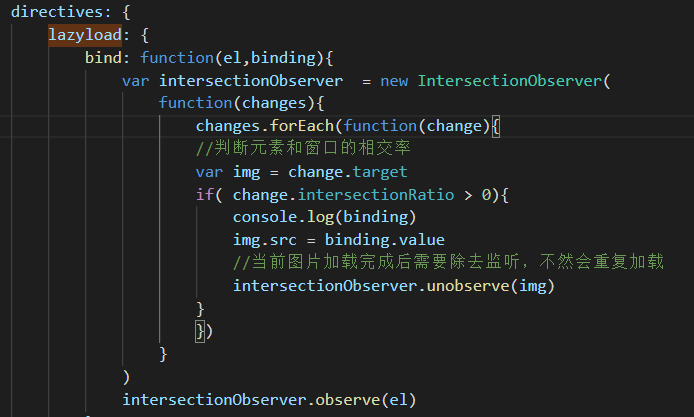
## Vue如何清除浏览器缓存

1. 项目打包的时候给每个打包文件加上hash值，一般是在文件后面加上时间戳；
2. 在html文件中加入meta标签，content属性设置为no-cache；
3. 在后端服务器中进行尽职缓存设置。

## Vue自定义指令懒加载







## CSS scoped穿透

[https://vue-loader.vuejs.org/zh/guide/scoped-css.html#](https://vue-loader.vuejs.org/zh/guide/scoped-css.html)

**什么是scoped？**

在vue文件中的style标签上有一个特殊的属性，scoped。当一个style标签有scoped属性的时候，它的css样式只能用于当前的vue组件，可以使组件的样式不相互污染。如果一个项目的所有style标签都加上了scoped属性，相当于实现了样式的模块化。

**Scoped的实现原理**

vue中国的scoped属性的效果主要是通过postCSS实现的，

|  |
| --- |
| <style scoped>  .example {  color: red;  }  </style>  <template>  <div class="example">hi</div>  </template> |

通过postCSS转译后

|  |
| --- |
| <style>  .example[data-v-f3f3eg9] {  color: red;  }  </style>  <template>  <div class="example" data-v-f3f3eg9>hi</div>  </template> |

## Vue全局API—Mixin

全局注册一个混入，影响注册之后所有创建的每个vue实例。

插件作者可以使用混入，像组件注入自定义的行为。

## 如何写一个插件

插件通常用来为Vue添加全局功能。插件的功能范围没有严格的限制——一般有下面几种：

1. 添加全局方法或者property。如：vue-custom-element
2. 添加全局资源：指令/过滤器/过渡等。如：vue-touch
3. 通过全局混入来添加一些组件选项。如vue-router
4. 添加vue实例方法，通过把它们添加到Vue.prototype上实现。
5. 一个库，提供自己的API，同时提供上面提到的一个或多个功能。如vue-router

**使用插件**

通过全局方法Vue.use()使用插件。它需要在你调用new Vue()启动应用之前完成：

**开发插件**

Vue.js的插件应该暴露一个install [ɪnˈstɔːl] 方法。这个方法的第一个参数是Vue构造器，第二个参数是一个可选的选项对象。

## 编译之AST optimize优化

AST 优化（optimize）阶段标记静态节点和静态根节点

1. 递归遍历AST节点，判断每个节点是否是静态的；
2. 递归遍历父节点中所有子节点是否是静态的，如果所有子节点都为静态，则该父节点为静态根节点；
3. 在生成render函数阶段，判断一个节点是否是一个静态根节点，如果是静态根节点，生成静态render函数。
4. Render函数生成vnode阶段，如果是静态render函数，将其生成的vnode进行缓存，等到下次再执行render函数的时候直接从缓存中取出。

生成vnode的

标记静态根节点是为了缓存vnode，减少render函数生成vnode的时间。

## VUE单页面白屏问题

## VUE SSR 服务端渲染

## 实现Input输入框过滤字符c/d/e

方法1：input输入框使用v-on指令绑定input事件，使用v-bind指令绑定属性value

<input type="text" v-bind:value="val" v-on:input="inputVal" />

methods: {

inputVal(e){

        let \_this = this

        \_this.val = e.target.value ||  e.currentTarget.value

        var filterArr = ['c','d','e']

        if(filterArr.indexOf(e.data) >= 0){

           \_this.val = \_this.val.replace(e.data,'')

        }

},

}

方法二：使用watch监听属性监听

<input type="text" v-model="val"/>

watch: {

     val: function(newVal, oldVal){

         let \_this = this

         \_this.val = \_this.val.replace(/[cde]/g, '')

    }

},

方法三：使用computed计算属性

<input type="text" v-model="computedVal"/>

computed: {

      computedVal: {

           get: function(){

              let \_this = this

              return \_this.val.replace(/[cde]/g, '')

           },

           set: function(v){

               let \_this = this

               \_this.val = v

            }

      }

},

方法4：自定义指令的方式

<input v-input-filter-cde="val" />

directives:{

                'input-filter-cde': {

                    bind: function(el, binding, vnode){

                        var context = vnode.context

                        var property = binding.expression

                        el.value = context[property]

                        el.oninput = function(e){

                           var value = e.target.value || e.currentTarget.value

                           context[property] = value.replace(/[cde]/g, '')

                           el.value = value.replace(/[cde]/g, '')

                        }

                    },

                    update: function(el, binding, vnode){

                        console.log(vnode.context.val)

                    }

                }

            },

## Vue内置的组件—keep-alive

<keep-alive></keep-alive> 包裹动态组件时，会缓存不活动的组件实例,主要用于保留组件状态或避免重新渲染。

include定义缓存白名单，keep-alive会缓存命中的组件；

exclude定义缓存黑名单，被命中的组件将不会被缓存；

max定义缓存组件上限，超出上限使用LRU策略置换缓存数据。

大白话: 比如有一个列表和一个详情，那么用户就会经常执行打开详情=>返回列表=>打开详情…这样的话列表和详情都是一个频率很高的页面，那么就可以对列表组件使用<keep-alive></keep-alive>进行缓存，  
这样用户每次返回列表的时候，都能从缓存中快速渲染，而不是重新渲染

<keep-alive>

<router-view v-if="$route.meta.alive"></router-view>

</keep-alive>

  <router-view v-if="!$route.meta.alive"></router-view>

结合vue-router使用

export default new Router({

    routes: [

        {

            path: '/banner',

            name: 'banner',

            component: Banner,

            meta:{

                alive: true,

            }

        },

]

})

**activated**::keep-alive组件激活时调用

## Vue全局API—Vue.extend

Vue.extend(options)

使用基础Vue构造器，创建一个“子类”。参数是一个包含组件选项的对象。

**大白话Vue.extend**

new Vue({}) options是一个空对象

new Vue({/\*something\*/}) 有【内容的实例】

Vue.extend({/\*something\*/}) 返回一个【有内容的类】，假设赋值给一个VueWithSoming，然后你就可以这么用：new VueWithSoming()。它返回的是【有内容的实例】，因为这个构造器本身带有【内容】。

我们日常使用Vue编写组件是这个样子的：

Vue.component('MyComponent',{

    template: '<div>这是一个组件</div>'

})

这是全局组件的注册方法，但其实这是一个语法糖，真正的运行过程是这样的：

let  myComponent = Vue.extend({

    template: '<div>这是一个组件</div>'

})

console.log(myComponent)

Vue.component('MyComponent',myComponent)

Vue.extend()返回的是一个函数VueComponent，可以看做是Vue的一个子类

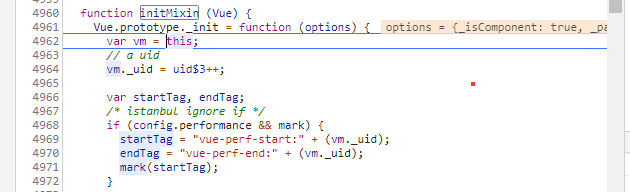


从源码上看VueComponent

1、有一个VueComponent函数



2、this.\_init调用的就是Vue原型上的\_init()



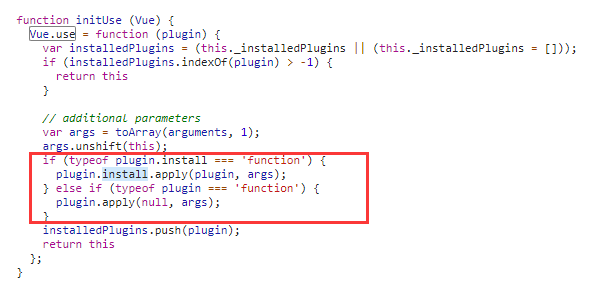
函数VueComponent可以new一个实例

let myComp = new myComponent()

console.log(myComp)

## Vue全局API—Vue.use

**源码**



1、\_installedPlugins 已安装插件列表

2、判断插件是否已经安装，已经安装就直接存缓存数组里面获取它

3、没有安装，则执行插件的install方法，没有install方法则直接执行插件

4、将新的插件放入数组，并返回

## Vue选项/资源—filters

过滤器

参数：第一个参数是过滤器别名；第二个参数是一个函数，称之为过滤器函数。

Vue允许自定义过滤器，可被用于一些常见的文本格式化。

过滤器可以用在两个地方：双花括号插值和v-bind表达式。

过滤器添加在JavaScript表达式的尾部，有“管道”符号指示。

过滤器函数总接收表达式的值作为第一个参数。

过滤器可以串联。

## Vue选项/资源—directives

除了核心功能默认内置的指令，Vue也允许注册自定义指令

* 钩子函数

一个指令定义对象可以提供如下几个钩子函数（均为可选）：

bind：只调用一次，指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置。

Inserted：被绑定元素插入父节点时调用（仅保证父节点存在，但不一定已被插入文档中）。

* 钩子函数参数

el：指令所绑定的元素，可以用来直接操作DOM。

binding：一个对象，包含以下属性

name：指令名，不包含v-前缀

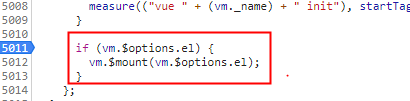
value：指令的绑定值，例如：v-my-directive=”1+1”中，绑定值为2.

oldValue: 指令绑定的前一个值，仅在update和componentUpdated钩子中可用。无论值是否改变都可用。

expression：字符串形式的指令表达式

## Vue 选项/DOM—el

**如果没有el怎么处理**



如上，有el的时候才会挂载。

如果在实例化时存在这个选项，实例将立即进入编译过程，否则，需要显示调用vm.$mount()，手动启动编译。



手动挂载会判断options里面有没有template 和render，没有的话，会输出上面的警告。

## Vue选项/其它—name

类型String

限制：只有作为组件选项时起作用。

作用：1、允许组件模版递归调用自身。

2、便于调试。有名字的组件有更好的警告信息。另外，当在有vue-devtools，未命名组件将显示成<AnonymousComponent> [əˈnɒnɪməs] [kəmˈpəʊnənt]匿名组件，这很没有语义。通过提供 name选项，可以获取更有语义信息的组件树。

注意：组件在全局用Vue.component()注册时，全局ID自动作为组件的name。

## **Vue实例属性**—**refs**

# Vue3

## Composition （组合式）API

Vue2中，随着功能的增加，组件变得越来越复杂，越来越难维护，而难以维护的根本原因是Vue的API设计迫使开发者使用watch、computed、methods选项组织代码，而不是实际的业务逻辑。

另外vue2缺少一种较为简洁的低成本的机制来完成逻辑复用，虽然可以minxis完成逻辑复用，但是当minxi变多的时候，会使得难以找到对应的data、computed或者methods来源于哪个mixin，使得类型推断难以进行。

所以composition API的出现，主要是为了解决options API带来的问题，第一个是代码组织问题，composition API可以让开发者根据业务逻辑组织自己的代码，让代码具备更好的可读性和可拓展性。第二个是实现代码的逻辑提取和复用。

Composition API在setup函数中执行。

Mixins的最大的问题是命名冲突。



## 片段（多根节点的组件）

Vue3，组件可以包含多个根节点

## 一个元素上同时使用v-if和v-for

Vue2.x中v-for优先左右

Vue3.x中v-if总是优先于v-for生效

## 编译方面的优化根节点

标记和提升所有的静态节点，diff的时候只需要对比动态节点的内容。

* **静态提升**

所有的静态节点都被提升到render方法之外，只会在应用启动的时候被创建一次，只会使用只需要应用提取的静态节点，随着每次的渲染被不断的复用。

## Patch flag

在动态标签末尾加上响应的标记，只有带patchFlag的节点才被认为是动态的元素，会被追踪属性的修改，能快速的找到动态节点，而不用逐个逐层遍历，提高了虚拟dom diff的性能。

# Vue性能优化

* **代码层面的优化**

1、v-if和v-show区分使用场景

v-if适用于在运行时很少改变条件，不需要频繁切换条件的场景；

v-show则适用于需要非常频繁切换条件的场景。

2、computed和watch区分使用场景

3、v-for遍历必须为item添加key，且避免同时使用v-if

4、长列表性能优化

纯粹的数据展示，不会有任何改变，我们不需要vue来劫持我们的数据，在大量数据展示的情况下，这能够明显的减少组件初始化的时间。使用Object.freeze来冻结一个对象。

5、事件销毁

Vue组件销毁时，会自动清理它与其它实例的连接，解绑它的全部指令及事件监听器，但是仅限于组件本身的事件。如果在js内使用addEventListene等方式是不会自动销毁的，所以我们在组件销毁时需要手动移除这些事件的监听，以免造成内存泄露。

6、图片懒加载

7、路由懒加载

Vue是单页面应用，可能会有很多的路由引入，这样使用webpack打包后的文件很大，当进入首页时，加载的资源过多，页面会出现白屏的情况，不利于用户体验。

如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应的组件，这样就更加高效了。

1. 第三方插件按需引入
2. 优化无限列表性能

如果是非常长或者无限滚动的列表，那么需要采用窗口化的技术来优化性能，只需要渲染少部分区域dom节点的时间。

可参考以下开源项目：vue-virtual-scroll-list和vue-virtual-scroller

10、服务器渲染SSR或者预渲染

* **Webpack配置层面的优化**
* **基础的web技术层面的优化**