

# JS实现简单的Vue

vue的使用相信大家都很熟练了，使用起来简单。但是大部分人不知道其内部的原理是怎么样的，今天我们就来一起实现一个简单的vue。

## Object.defineProperty()

实现之前我们得先看一下Object.defineProperty的实现，因为vue主要是通过数据劫持来实现的，通过get、set来完成数据的读取和更新。



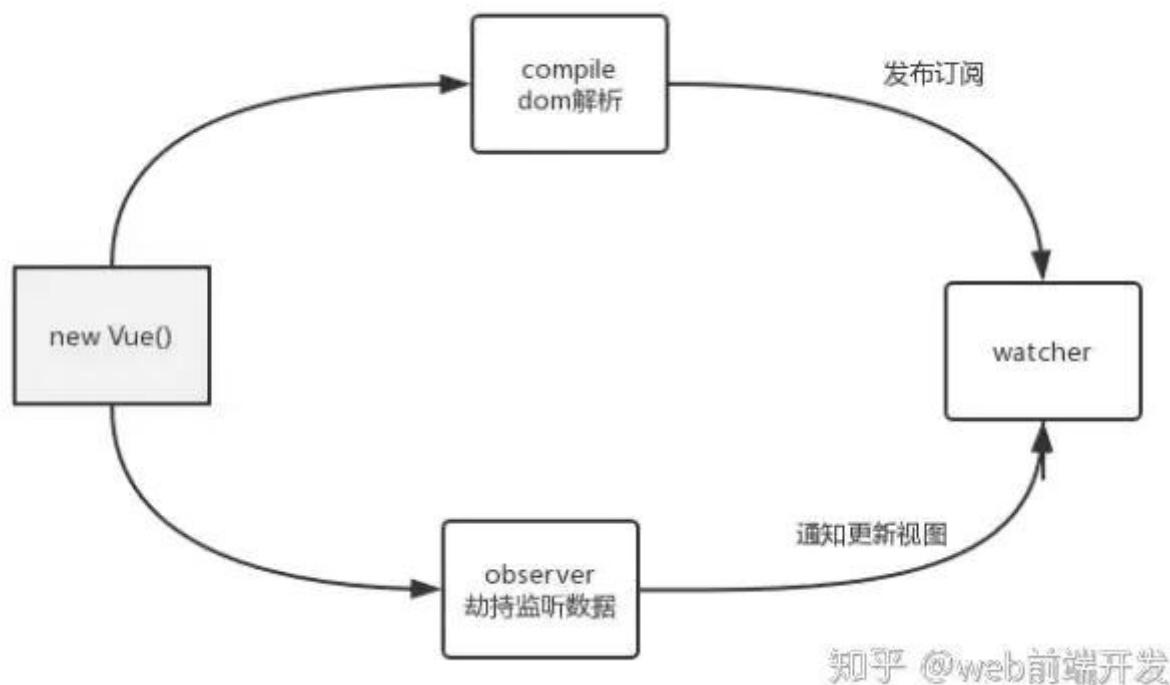
```
● ● ●

var obj = {name:'wclimb'}
var age = 24
Object.defineProperty(obj,'age',{
    enumerable: true, // 可枚举
    configurable: false, // 不能再define
    get () {
        return age
    },
    set (newVal) {
        console.log('我改变了',age +' -> '+newVal);
        age = newVal
    }
})
> obj.age
> 24
> obj.age = 25;
> 我改变了 24 -> 25
> 25

知乎的Vue和前端开发
```

从上面可以看到通过get获取数据，通过set监听到数据变化执行相应操作，还是不明白的话可以去看看Object.defineProperty文档。

流程图



```
<div id="wrap">
  <p v-html="test"></p>
  <input type="text" v-model="form">
  <input type="text" v-model="form">
  <button @click="changeValue">改变值</button>
  {{form}}
</div>
```

知乎 @web前端开发

## js调用

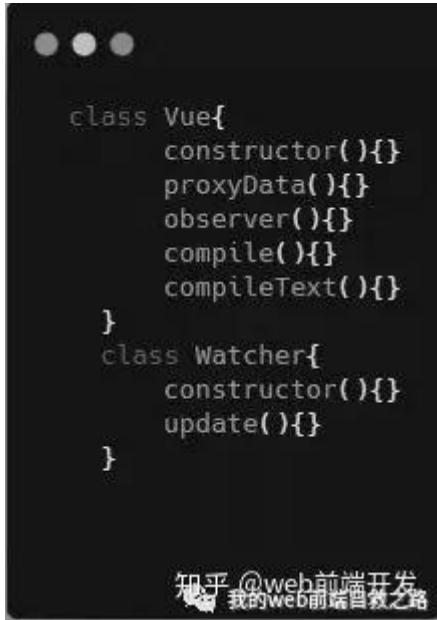


A screenshot of a mobile phone displaying a code editor. The code is written in JavaScript and defines a new Vue instance. The instance has an 'el' property set to '#wrap', a 'data' object containing 'form' and 'test' properties, and a 'methods' object with a 'changeValue' method. The 'changeValue' method logs the current value of 'form' to the console and then changes its value to '值被我改变了，气不气？'.

```
new Vue({
  el: '#wrap',
  data: {
    form: '这是form的值',
    test: '<strong>我是粗体</strong>',
  },
  methods: {
    changeValue(){
      console.log(this.form)
      this.form = '值被我改变了，气不气？'
    }
  }
})
```

©我的Vue前端开发

## Vue结构



```
class Vue{
    constructor(){}
    proxyData(){}
    observer(){}
    compile(){}
    compileText(){}
}
class Watcher{
    constructor(){}
    update(){}
}
```

知乎 @web前端开发  
我的web前端自我之路

1、Vue constructor 构造函数主要是数据的初始化

2、proxyData 数据代理

3、observer 劫持监听所有数据

4、compile 解析dom

5、compileText 解析dom里处理纯双花括号的操作

6、Watcher 更新视图操作

**Vue constructor 初始化**

```
class Vue{  
    constructor(options = {}){  
        this.$el = document.querySelector(options.el);  
        let data = this.data = options.data;  
        // 代理data，使其能直接this.xxx的方式访问data，正常的话需要this.data.xxx  
        Object.keys(data).forEach((key)=> {  
            this.proxyData(key);  
        });  
        this.methods = obj.methods // 事件方法  
        this.watcherTask = {}; // 需要监听的任务列表  
        this.observer(data); // 初始化劫持监听所有数据  
        this.compile(this.$el); // 解析dom  
    }  
}
```

↑和我的小伙伴一起跋涉

上面主要是初始化操作，针对传过来的数据进行处理

proxyData 代理data

```
class Vue{  
    constructor(options = {}){  
        //省略部分代码  
        proxyData(key){  
            let that = this;  
            Object.defineProperty(that, key, {  
                configurable: false,  
                enumerable: true,  
                get () {  
                    return that.data[key];  
                },  
                set (newVal) {  
                    that.data[key] = newVal;  
                }  
            });  
        }  
    }  
}
```

看我的Vue前端端开发

上面主要是代理data到最上层 · this.xxx的方式直接访问data

observer 劫持监听

```
class Vue{  
    constructor(options = {}){  
        this.$options = options  
    }  
    proxyData(key){  
        //...  
    }  
    observer(data){  
        let that = this  
        Object.keys(data).forEach(key=>{  
            let value = data[key]  
            this.watcherTask[key] = []  
            Object.defineProperty(data,key,{  
                configurable: false,  
                enumerable: true,  
                get(){  
                    return value  
                },  
                set(newValue){  
                    if(newValue !== value){  
                        value = newValue  
                        that.watcherTask[key].forEach(task => {  
                            task.update()  
                        })  
                    }  
                }  
            })  
        })  
    }  
}
```

## ④ 实现的Vue前端端追踪

同样是使用Object.defineProperty来监听数据，初始化需要订阅的数据。

把需要订阅的数据到push到watcherTask里，等到时候需要更新的时候就可以批量更新数据了。下面是；

遍历订阅池，批量更新视图。



compile 解析dom

```
class Vue{
  constructor(options = {}){
    .....
  }
  proxyData(key){
    .....
  }
  observer(data){
    .....
  }
  compile(el){
    var nodes = el.childNodes;
    for (let i = 0; i < nodes.length; i++) {
      const node = nodes[i];
      if(node.nodeType === 3){
        var text = node.textContent.trim();
        if (!text) continue;
        this.compileText(node, 'textContent')
      }else if(node.nodeType === 1){
        if(node.childNodes.length > 0){
          this.compile(node)
        }
        if(node.hasAttribute('v-model') && (node.tagName === 'INPUT' || node.tagName === 'TEXTAREA')){
          this.compileInput(node)
        }
      }
    }
  }
}
```

```
node.addEventListener('input',(()=>{
    let attrVal = node.getAttribute('v-model')
    this.watcherTask[attrVal].push(new Watcher(node,th
    node.removeAttribute('v-model')
    return () => {
        this.data[attrVal] = node.value
    }
})()))
}

if(node.hasAttribute('v-html')){
    let attrVal = node.getAttribute('v-html');
    this.watcherTask[attrVal].push(new Watcher(node,this,a
    node.removeAttribute('v-html')
}
this.compileText(node, 'innerHTML')
if(node.hasAttribute('@click')){
    let attrVal = node.getAttribute('@click')
    node.removeAttribute('@click')
    node.addEventListener('click',e => {
        this.methods[attrVal] && this.methods[attrVal].bin
    })
}
}
},
compileText(node,type){
    let reg = /\{\{(.*\)}\}/g, txt = node.textContent;
    if(reg.test(txt)){
        node.textContent = txt.replace(reg,(matched,value)=>{
            let tpl = this.watcherTask[value] || []
            tpl.push(new Watcher(node,this,value,type))
            return value.split('.').reduce((val, key) => {
                return this.data[key];
            }, this.$el);
        })
    }
}
```

```

    }
}

```

这里代码比较多，我们拆分看你就会觉得很简单了

首先我们先遍历el元素下面的所有子节点，`node.nodeType === 3` 的意思是当前元素是文本节点，`node.nodeType === 1` 的意思是当前元素是元素节点。因为可能有的是纯文本的形式，如纯双花括号就是纯文本的文本节点，然后通过判断元素节点是否还存在子节点，如果说有的话就递归调用`compile`方法。下面重头戏来了，我们拆开看：

```

...
if(node.hasAttribute('v-html')){
  let attrVal = node.getAttribute('v-html');
  this.watcherTask[attrVal].push(new Watcher(node,this,attrVal,'innerHTML'))
}
node.removeAttribute('v-html')

```

知乎我的Web前端面试之路

上面这个首先判断`node`节点上是否有`v-html`这种指令，如果存在的话，我们就发布订阅，怎么发布订阅呢？只需要把当前需要订阅的数据`push`到`watcherTask`里面，然后到时候在设置值的时候就可以批量更新了，实现双向数据绑定，也就是下面的操作

```

...
that.watcherTask[key].forEach(task => {
  task.update()
})

```

知乎我的Web前端面试之路

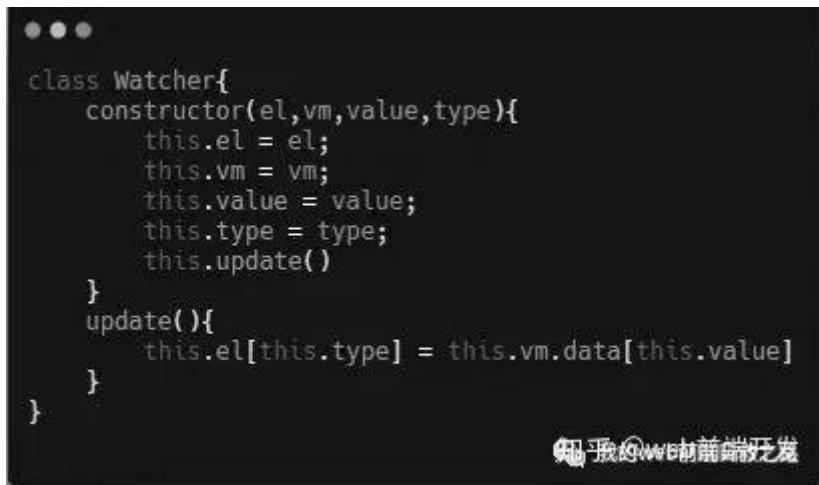
然后`push`的值是一个`Watcher`的实例，首先他`new`的时候会先执行一次，执行的操作就是去把纯双花括号`-> 1`，也就是说把我们写好的模板数据更新到模板视图上。

最后把当前元素属性剔除出去，我们用Vue的时候也是看不到这种指令的，不剔除也

不影响

至于Watcher是什么，看下面就知道了

## Watcher



```
class Watcher{
    constructor(el,vm,value,type){
        this.el = el;
        this.vm = vm;
        this.value = value;
        this.type = type;
        this.update()
    }
    update(){
        this.el[this.type] = this.vm.data[this.value]
    }
}
```

The screenshot shows a code editor with a dark theme. It displays a portion of a JavaScript file containing the definition of a 'Watcher' class. The class has a constructor that takes four parameters: 'el', 'vm', 'value', and 'type'. Inside the constructor, it initializes 'this.el', 'this.vm', 'this.value', and 'this.type' with the respective parameters. It then calls the 'update' method. The 'update' method sets the value at the specified 'type' index of 'this.el' to the value found in 'this.vm.data' at the index 'this.value'. The code is part of a larger project, as indicated by the watermark '知乎@我的前端开发'.

之前发布订阅之后走了这里面的操作，意思就是把当前元素如：node.innerHTML = '这是data里面的值'、  
node.value = '这个是表单的数据'

那么我们为什么不直接去更新呢，还需要update做什么，不是多此一举吗？

其实update记得吗？我们在订阅池里面需要批量更新，就是通过调用Watcher原型上的update方法。

## 效果

大家可以浏览器看一下效果，由于本人太懒了，gif效果图就先不放了，哈哈😊😊

## 完整代码

地址：[github.com/wclimb/MyVu...](https://github.com/wclimb/MyVu...)

## 参考

1、剖析Vue原理&实现双向绑定MVVM

2、仿Vue实现极简双向绑定

微信公众号：我的web前端自救之路

回复 加群，跟大佬们一起交流技术吧

# 一步一步实现自己的vue

## 1.准备工作

\*\* 我们先利用webpack构建项目:\*\*

- 初始化项目

```
npm init -y
```

- 安装webpack

```
npm i webpack webpack-cli webpack-dev-server html-webpack-plugin --save
```

- 配置webpack

```
// webpack.config.js
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: './src/index.js', // 以src下的index.js 作为入口文件进行打包
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  devtool: 'source-map', // 调试的时候可以快速找到错误代码
  resolve: {
    // 更改模块查找方方式（默认的是去node_modules里去找）去source文件里去找
    modules: [path.resolve(__dirname, 'source'), path.resolve('node_modules')]
  },
  plugins: [
    new HtmlWebpackPlugin({
      title: 'My Vue App'
    })
  ]
};
```

```

        template:path.resolve(__dirname,'public/index.html')
    })
]
}

```

- 配置package.json

```

"scripts": {
  "start": "webpack-dev-server",
  "build": "webpack"
},

```

## 2 实现数据监听

### 2.1 创建构造函数MyVue

并初始化用户传入的参数 `options`，我们先假设用户传入的 `options` 是只有 `data` 属性和 `el` 属性的。

```

export function initState(vm) {
  let opt = vm.$options
  if (opt.data){
    initData(vm);
  }
}

function initData(vm) {
  // 获取用户传入的data
  let data = vm.$options.data
  // 判断是不是函数，我们知道vue，使用data的时候可以data:{}这种形式，也可以data(){return{}}这种形式
  // 然后把把用户传入的打他数据赋值给vm._data
  data = vm._data = typeof data === 'function' ? data.call(vm) : data || {}
  observe(data)
}

```

到这里我们实现的是new MyVue的时候，通过\_init方法来初始化options，然后通过initData方法将data挂到vm实例的\_data上去了，接下来，我们要对data实现数据监听，上面的代码中observe代码就是用来实现数据监听的。

## 2.2 实现数据监听

```
export function observe(data) {
  if (typeof data !== 'object' || data == null){
    return
  }
  return new Observe(data)
}
```

在这段代码observe方法的代码中，observe()将传入的data先进行判断，如果data是对象，则new一个Observe对象来使这个data实现数据监听，我们再看下Observe是怎么实现的

```
class Observe {
  constructor(data){ // data就是我们定义的data vm._data实例
    // 将用户的数据使用defineProperty定义
    this.walk(data)
  }
  walk(data){
    let keys = Object.keys(data)
    for (let i = 0;i<keys.length;i++){
      let key = keys[i]; // 所有的key
      let value = data[keys[i]] //所有的value
      defineReactive(data,key,value)
    }
  }
}
```

可见，Observe 将data传入walk方法里，而在walk方法里对data进行遍历，然后将data的每一个属性和对应的值传入 `defineReactive`，我们不难猜测，这个 `defineReactive` 就是将data的每一个属性实现监听。我们再看下 `defineReactive`。

```
export function defineReactive(data, key, value) {  
  
    Object.defineProperty(data, key, {  
        get() {  
            return value  
        },  
        set(newValue) {  
            if (newValue === value) return  
            value = newValue  
            observe(value)  
        }  
    })  
}
```

可见，这是通过`defineProperty`，将每个key进行数据监听了。但是这里有一个问题，就是，这里只能监听一个层级，比如

```
data = {  
    wife: "迪丽热巴"  
}
```

这时没问题的，但是

```
data = {  
    wife: {  
        name: "迪丽热巴",  
        friend: {  
            name: "古力娜和"  
        }  
    }  
}
```

我们只能监听到wife.friend和wife.name是否改变与获取，无法监听到wife.friend.name这个属性的变化，因此，我们需要判断wife.friend是不是对象，然后将这个friend对象进行遍历对它的属性实现监听。

## 2.3 解决多层级监听的问题

因此我们在上面代码的基础上，添加上 `observe (value)` 就实现了递归监听。

```
export function defineReactive(data, key, value) {
    // 观察value是不是对象，是的话需要监听它的属性。
    observe(value)

    Object.defineProperty(data, key, {
        get(){
            return value
        },
        set(newValue){
            if (newValue === value) return
            value = newValue
        }
    })
}
```

基本完成。

但是到这里，还有一个问题，就是我们上面的data都是new MyVue的时候传进去的，因此要是我们再new 完改变data的某个值，如下面将message改成迪丽热巴对象，此时虽然我们依旧可以监听message，但是message.name是监听不到的。

```
let vm = new MyVue({
    el: '#app',
    data(){
        return{
            message:'大家好',
        }
    }
})
```

```
wife:{  
    name:"angelababy",  
    age:28  
}  
}  
}  
})  
vm._data.message = {  
    name:'迪丽热巴',  
    age:30  
}
```

## 2.4 解决data中某个属性变化后无法监听的问题

我们知道 message这个属性已经被我们监听了，所以改变message的时候，会触发set（）方法，因此我们只需要将wife再放进observe()中重新实现监听一遍即可，如代码所示

```
export function defineReactive(data, key, value) {  
    // 观察value是不是对象，是的话需要监听它的属性。  
    observe(value)  
  
    Object.defineProperty(data, key, {  
        get(){  
            return value  
        },  
        set(newValue){  
            if (newValue === value) return  
            value = newValue  
            observe(value)  
        }  
    })  
}
```

## 2.5 实现数据代理

我们用过vue的都知道，我们获取data中的属性的时候，都是直接通过this.xxx,获取值的，而我们上面只实现了想要获取值需要通过this.\_data.xxx,所以这一节来实现是数据代理，即将data中的属性挂载到vm上，我们可以实现一个proxy方法，该方法将传入的数据挂载到vm上，而当

我们访问`this.xxx`的时候，其实是访问了`this._data.xxx`，这就是代理模式。增加proxy后代码如下

```
function proxy(vm,source,key) {
    Object.defineProperty(vm,key,{ 
        get(){
            return vm[source][key]
        },
        set(newValue){
            return vm[source][key] = newValue
        }
    })
}

function initData(vm) {
    // 获取用户传入的data
    let data = vm.$options.data
    // 判断是不是函数，我们知道vue，使用data的时候可以data:{}这种形式，也可以data(){return{}}这种形式
    // 然后把把用户传入的数据赋值给vm._data
    data = vm._data = typeof data === 'function' ? data.call(vm) : data || {}

    for (let key in data) {
        proxy(vm,"_data",key)
    }

    observe(data)
}
```

实现原理非常简单，实际上就是当我们想要获取`this.wife`时，其实是去获取`this._data.wife`

至此，我们已经实现了数据监听，但是还有个问题，即`Object.defineProperty`的问题，也是面试常见的问题，即`Object.defineProperty`是无法监听数组的变化的

### 3 重写数组方法

如图所示，我们企图往数组arr中添加值，结果发现新添加进去的值是没办法被监听到的，因此，我们需要改写push等方法

```
let vm = new MyVue({
  el: '#app',
  data(){
    return{
      message:'大家好',
      wife:{
        name:"angelababy",
        age:28
      },
      arr:[1,2,{name:"赵丽颖"}]
    }
  }
})
vm.arr.push({hah:'dasd'})
```

基本思路就是之前我们调用push方法时，是从Aarray.prototype寻找这个方法，我们改成用一个空对象{} 继承 Aarray.prototype，然后再给空对象添加push方法

```
{
  push : function(){}
}
```

这样，我们调用push的时候，实际上就是调用上面{}中的push

现在，我们先区分出用户传入的Observe中接受监听的data是数组还是对象，如果是数组，则改变数组的原型链，这样才能改变调用push时，是调用我们自己设置的push，只需要在Observe添加判断是数组还是对象即可。

```
class Observe {
  constructor(data){ // data就是我们定义的data vm._data实例
    // 将用户的数据使用defineProperty定义
```

```

if (Array.isArray(data)){
    data.__proto__ = arrayMethods
} else {
    this.walk(data)
}
}

walk(data){
    let keys = Object.keys(data)
    for (let i = 0; i < keys.length; i++) {
        let key = keys[i]; // 所有的key
        let value = data[keys[i]] //所有的value
        defineReactive(data, key, value)
    }
}
}

```

其中的 `arrayMethods` 则是我们一直说的那个对象{},它里面添加push等方法属性

```

let oldArrayPrototypeMethods = Array.prototype
// 复制一份 然后改成新的
export let arrayMethods = Object.create(oldArrayPrototypeMethods)

// 修改的方法
let methods = ['push', 'shift', 'unshift', 'pop', 'reverse', 'sort', 'splice']

methods.forEach(method=>{
    arrayMethods[method] = function (...arg) {
        // 不光要返回新的数组方法，还要执行监听
        let res = oldArrayPrototypeMethods[method].apply(this, arg)
        // 实现新增属性的监听
        console.log("我是{}对象中的push,我在这里实现监听");
        return res
    }
})

```

实际上这是一种拦截的方法。接下来，我们就要着手实现新增属性的监听。基本思路，1.获得新增属性，2.实现监听

```

methods.forEach(method=>{
    arrayMethods[method] = function (...arg) {
        // 不光要返回新的数组方法，还要执行监听
        let res = oldArrayPrototypeMethods[method].apply(this, arg)
        // 实现新增属性的监听
        console.log("我是{}对象中的push,我在这里实现监听");
        // 实现新增属性的监听
        let inserted // 1.获得新增属性
        switch (method) {
            case 'push':
            case 'unshift':
                inserted = arg
                break
            case 'splice':
                inserted = arg.slice(2)
                break
            default:
                break
        }
        // 实现新增属性的监听
        if (inserted){
            observerArray(inserted)
        }
        return res
    }
})

```

这里用到了 `observerArray`，我们看一下

```

export function observerArray(inserted){
    // 循环监听每一个新增的属性
    for(let i =0;i<inserted.length;i++){
        observe(inserted[i])
    }
}

```

可见，它是将`inserted`进行遍历，对每一项实现监听。可能这里你会有疑问，为什么要进行遍历，原因是`inserted`不一定是一个值，也可能是多个，例如`[] .push(1,2,3)`。

现在已经实现了数组方法的拦截，还有个问题没有解决，就是当我们初始化的时候，`data`里面可能有数组，因此也要把这个数组进行监听

```
constructor(data){ // data就是我们定义的data vm._data实例
  // 将用户的数据使用defineProperty定义
  if (Array.isArray(data)){
    data.__proto__ = arrayMethods
    observerArray(data)
  }else {
    this.walk(data)
  }
}
```

现在已经实现了对数据的监听，不过这里还有问题没解决，也是vue2.0中没有解决的问题，就是这里并没有实现对数组的每一项实现了监听 例如，这样是不会监听到的。

```
let vm = new MyVue({
  el: '#app',
  data(){
    return{
      message:'大家好',
      wife:{
        name:"angelababy",
        age:28
      },
      arr:[1,2,{name:"赵丽颖"}]
    }
  }
})
vm.arr[0] = "我改了"
```

不仅如此，`vm.arr.length = 0`,当你这样设置数组长度时，也是无法监听到的。

## 4. 初始化渲染页面

数据初始化之后，接下来，就要把初始化好的数据渲染到页面上去了也就是说当dom中有{{name}}这样的引用时，要把{{name}}替换成data里对应的数据

```
MyVue.prototype._init = function (options) {
  let vm = this;
  // this.$options表示是Vue中的参数，如若我们细心的话我们发现vue框架的可读属性都是$开头的
  vm.$options = options;

  // MVVM原理 重新初始化数据  data
  initState(vm)

  // 初始化渲染页面
  if (vm.$options.el){
    vm.$mount()
  }
}
```

\$mount的功能很显然就是

1. 先获得dom树，
2. 然后替换dom树中的数据，
3. 然后再把新dom挂载到页面上去

我们看下实现代码

```
MyVue.prototype.$mount = function () {
  let vm = this
  let el = vm.$options.el
  el = vm.$el = query(el) //获取当前节点

  let updateComponent = () =>{
    console.log("更新和渲染的实现");
    vm._update()
}
```

```

    }
    new Watcher(vm, updateComponent)
}

```

显然，我们并没有看到上面所说的

1. 先获得dom树，
2. 然后替换dom树中的数据，
3. 然后再把新dom挂载到页面上去，

那肯定是把这些步骤放在 `vm._update()` 的时候实现了。我们来看下update代码

```

// 拿到数据更新视图
MyVue.prototype._update = function () {
  let vm = this
  let el = vm.$el
  // 1. 获取dom树
  let node = document.createDocumentFragment()
  let firstChild
  while (firstChild = el.firstChild){
    node.appendChild(firstChild)
  }
  // 2. 然后替换dom树中的数据，
  compiler(node,vm)

  // 3. 然后再把新dom挂载到页面上去，
  el.appendChild(node)
}

```

可见，这三个步骤在update的时候实现了。

而这个update方法的执行需要

```

let updateComponent = () =>{
  console.log("更新和渲染的实现");
  vm._update()
}

```

这个方法的执行，显然，这个方法是new Watcher的时候执行的。

```
let id = 0

class Watcher {
    constructor(vm, exprOrFn, cb = ()=>{}, opts){
        this.vm = vm
        this.exprOrFn = exprOrFn
        this.cb = cb
        this.id = id++
        if (typeof exprOrFn === 'function'){
            this.getter = exprOrFn
        }
        this.get() // 创建一个watcher，默认调用此方法
    }
    get(){
        this.getter()
    }
}
export default Watcher
```

可见，this.getter就是我们传进去的 updateComponent，然后在new Watcher的时候，就自动执行了。

总结下思路，

1. new Watcher的时候执行了 updateComponent
2. 执行 updateComponent 的时候执行了 update
3. 执行update的时候执行了 1. 先获得dom树，2. 然后替换dom树中的数据，3. 然后再把新dom挂载到页面上去

现在我们就已经实现了初始化渲染。即把dom中{{}}表达式换成了data里的数据。

上面用到的compile方法我们还没解释，其实，很简单

```

const defaultRGE = /\{\{\{((?:.|\\r?\\n)+?)\}\}\}/g

export const util = {
  getValue(vm, exp) {
    let keys = exp.split('.')
    return keys.reduce((memo, current) => {
      memo = memo[current]
    }, vm)
  },
  compilerText(node, vm) {
    node.textContent = node.textContent.replace(defaultRGE, function (...arg) {
      return util.getValue(vm, arg[1])
    })
  }
}

export function compiler(node, vm) {
  // 1 取出子节点、
  let childNodes = node.childNodes
  childNodes = Array.from(childNodes)
  childNodes.forEach(child => {
    if (child.nodeType === 1) {
      compiler(child, vm)
    } else if (child.nodeType === 3) {
      util.compilerText(child, vm)
    }
  })
}

```

## 5. 更新数据渲染页面

我们上一节只实现了 初始化渲染，这一节来实现 数据一旦修改就重新渲染页面。上一节中，我们是通过new Watcher()来初始化页面的，也就是说这个watcher具有重新渲染页面的功能，因此，我们一旦改数据的时候，就再一次让这个watcher执行刷新页面的功能。这里有必要解释下一个watcher对应一个组件，也就是说你new Vue 机会生成一个watcher，因此有多个组件的时候就会生成多个watcher。

现在，我们给每一个data里的属性生成一个对应的dep。例如：

```
data: {
  age: 18,
  friend: {
    name: "赵丽颖",
    age: 12
  }
}
```

上面中，age,friend,friend.name,friend.age分别对应一个dep。一共四个dep。dep的功能是用来通知上面谈到的watcher执行刷新页面的功能的。

```
export function defineReactive(data, key, value) {
  // 观察value是不是对象，是的话需要监听它的属性。
  observe(value)
  let dep = new Dep() // 新增代码：一个key对应一个dep
  Object.defineProperty(data, key, {
    get(){
      return value
    },
    set(newValue){
      if (newValue === value) return
      value = newValue
      observe(value)
    }
  })
}
```

现在有一个问题，就是dep要怎么跟watcher关联起来，我们可以把watcher存储到dep里

```
let id = 0
class Dep {
  constructor(){
    this.id = id++
  }
}
```

```

    this.subs = []
}
addSub(watcher){ //订阅
    this.subs.push(watcher)
}
}

```

如代码所示，我们希望执行addSub方法就可以将watcher放到subs里。那什么时候可以执行addSub呢？

我们在执行compile的时候，也就是将dom里的{{}}表达式换成data里的值的时候，因为要获得data里的值，因此会触发get。这样，我们就可以在get里执行addSub。而watcher是放在全局作用域的，我们可以直接从全局作用域中拿这个watcher放到传入addSub。

好了，现在的问题就是，怎么把watcher放到全局作用域

```

let id = 0

class Watcher {
    constructor(vm,exprOrFn,cb = ()=>{},opts){
        this.vm = vm
        this.exprOrFn = exprOrFn
        this.cb = cb
        this.id = id++
        this.deps = []
        this.depsId = new Set()
        if (typeof exprOrFn === 'function'){
            this.getter = exprOrFn
        }
        this.get() // 创建一个watcher，默认调用此方法
    }
    get(){
        pushTarget(this)
        this.getter()
        popTarget()
    }
}
export default Watcher

```

可见，是通过pushTarget(this)放到全局作用域，再通过popTarget()将它移除。

要知道，wachter和dep是多对多的关系，dep里要保存对应的watcher，watcher也要保存对应的dep。因此，但我们触发get的时候，希望可以同时让当前的watcher保存当前的dep，也让当前的dep保存当前的wachter。

```
export function defineReactive(data, key, value) {
    // 观察value是不是对象，是的话需要监听它的属性。
    observe(value)
    let dep = new Dep()
    Object.defineProperty(data, key, {
        get(){
            if (Dep.target){
                dep.depend() //让dep保存watcher，也让watcher保存这个dep
            }
            return value
        },
        set(newValue){
            if (newValue === value) return
            value = newValue
            observe(value)

        }
    })
}
```

让我们看下depend方法怎么实现

```
let id = 0
class Dep {
    constructor(){
        this.id = id++
        this.subs = []
    }
    addSub(watcher){ //订阅
        this.subs.push(watcher)
    }
}
```

```

depend(){
  if (Dep.target){
    Dep.target.addDep(this)
  }
}

// 保存当前watcher
let stack = []

export function pushTarget(watcher) {
  Dep.target = watcher
  stack.push(watcher)
}

export function popTarget() {
  stack.pop()
  Dep.target = stack[stack.length - 1]
}

export default Dep

```

可见depend方法又执行了watcher里的addDep，看一下watcher里的addDep。

```

import {pushTarget , popTarget} from "./dep"

let id = 0

class Watcher {
  constructor(vm,exprOrFn,cb = ()=>{},opts){
    this.vm = vm
    this.exprOrFn = exprOrFn
    this.cb = cb
    this.id = id++
    this.deps = []
    this.depsId = new Set()
    if (typeof exprOrFn === 'function'){
      this.getter = exprOrFn
    }
    this.get() // 创建一个watcher，默认调用此方法
  }
  get(){
    pushTarget(this)
    this.getter()
  }
}

```

```

    popTarget()
}
update(){
    this.get()
}
addDep(dep){
    let id = dep.id
    if(this.depsId.has(id)){
        this.depsId.add(id)
        this.deps.push(dep)
    }
    dep.addSub(this)
}
}

export default Watcher

```

如此一来，就让dep和watcher实现了双向绑定。这里代码，你可能会有个疑问，就是为什么是用一个stack数组来保存watcher，这里必须解释下，因为每一个watcher是对应一个组件的，也就是说，当页面中有多个组件的时候，就会有多个watcher，而多个组件的执行是依次执行的，也就是说Dep.target中只会有当前被执行的组件所对应的watcher。

例如，有一道面试题：父子组件的执行顺序是什么？

答案：在组件开始生成到结束生成的过程中，如果该组件还包含子组件，则自己开始生成后，要让所有的子组件也开始生成，然后自己就等着，直到所有的子组件生成完毕，自己再结束。“父亲”先开始自己的created，然后“儿子”开始自己的created和mounted，最后“父亲”再执行自己的mounted。

为什么会这样，到这里我们就应该发现了，new Vue的时候是先执行initData，也就是初始化数据，然后执行\$mounted,也就是new Watcher。而初始化数据的时候，也要处理components里的数据。处理component里的数据的时候，每处理一个子组件就会new Vue，生成一个子组件。因此是顺序是这样的。也就对应了上面的答案。

1. 初始化父组件数据-->
2. 初始化 子组件数据 -->
3. new 子组件Wacther -->
4. new 父组件Watcher

好，言归正传，回到我们的项目来，接下来要实现的就是当有数据更改的时候，我们要重新渲染页面。而我们可以通过set来监听数据是否被更改，因此基本步骤为：

1. set监听到数据被更改
2. 让dep执行dep.notify()通知与它相关的watcher
3. watcher执行update，重新渲染页面

```
Object.defineProperty(data, key, {
  get(){
    if (Dep.target){
      dep.depend() //让dep保存watcher，也让watcher保存这个dep
    }
    return value
  },
  set(newValue){
    if (newValue === value) return
    value = newValue
    observe(value)

    // 当设置属性的时候，通知watcher更新
    dep.notify()
  }
})
```

dep添加notify方法

```
class Dep {
  constructor(){
    this.id = id++
    this.subs = []
  }
  addSub(watcher){ //订阅
    this.subs.push(watcher)
  }
  notify(){ //发布
    this.subs.forEach(watcher =>{
      watcher.update()
    })
  }
}
```

```
        })
    }
    depend(){
        if (Dep.target){
            Dep.target.addDep(this)
        }
    }
}
```

watcher添加update方法

```
class Watcher {
    constructor(vm,exprOrFn,cb = ()=>{},opts){
        this.vm = vm
        this.exprOrFn = exprOrFn
        this.cb = cb
        this.id = id++
        this.deps = []
        this.depsId = new Set()
        if (typeof exprOrFn === 'function'){
            this.getter = exprOrFn
        }
        this.get() // 创建一个watcher，默认调用此方法
    }
    get(){
        pushTarget(this)
        this.getter()
        popTarget()
    }
    update(){
        this.get()
    }
    addDep(dep){
        let id = dep.id
        if(this.depsId.has(id)){
            this.depsId.add(id)
            this.deps.push(dep)
        }
        dep.addSub(this)
    }
}
```

```

    }
export default Watcher

```

## 5. 批量更新防止重复渲染

上面我们是每更改一个数据，就会通知watcher重新渲染页面，显然，要是我们在一个组件里更改多个数据，那么就会多次通知watcher渲染页面，因此这节我们来实现 批量更新，防止重复渲染。怎么解决呢？

我们知道，每更新一个数据，就会触发`dep.notify`。而如果组件里的多个数据都更新的话，就会多次触发`dep.notify`。因为是同一个组件里的数据，因此，这些`dep.notify`通知的是同一个watcher 执行`update`。显然，这是没必要的，我们只希望先让所有的数据都修改完，再统一让watcher执行一次`update`。

该怎么实现呢？我们可以创建一个数组`queue`，来放置即将渲染页面的watcher。所以，我们先要判断这些`dep`通知的是不是同一个watcher。不相同的话就放入`queue`里，相同的就不放。（`queue`就是个去重数组）。基于此，我们可以更改Watcher里的`update`方法。

```

class Watcher {
  constructor(vm, exprOrFn, cb = ()=>{}, opts){
    ...
    this.get() // 创建一个watcher，默认调用此方法
  }
  get(){
    pushTarget(this)
    this.getter()
    popTarget()
  }
  update(){
    // this.get()
    queueWacther(this) //修改代码
  }
  // 新增代码
  run(){
    this.get()
  }
}

```

```

        }

    }

// 新增代码

let has = {}

let queue = []

function queueWacther(watcher) {

    let id = watcher.id

    if(has[id] == null){

        has[id] = true

        queue.push(watcher)

    }

}

```

这样一来，queue里放置的就是不同的wathcer。

接下，再执行queue里的watcher.run。

```

let has = {}

let queue = []

// 新增代码

function flushQueue() {

    console.log("执行了flushQueue");

    queue.forEach(watcher=>{

        watcher.run()

    })

    has = []

    queue = []

}

function queueWacther(watcher) {

    let id = watcher.id

    if(has[id] == null){

        has[id] = true

        queue.push(watcher)

    }

}

```

记得执行完要清空queue队列。

但是有个重要的事情，就是queue里的watcher.run必须要异步执行。

现在就是要异步执行queue里的watcher.run()。我们可以把重新渲染的动作放到异步队列里（可以通过promise.then放到微任务队列里）。而修改数据是在主线程上的，因此，会先执行完主线程才会执行异步队列里的方法。

```
let has = {}
let queue = []
function flushQueue() {
  console.log("执行了flushQueue");
  queue.forEach(watcher=>{
    watcher.run()
  })
  has = []
  queue = []
}
function queueWatcher(watcher) {
  let id = watcher.id
  if(has[id] == null){
    has[id] = true
    queue.push(watcher)
  }
  nextTick(flushQueue) //新增代码：异步执行flushQueue
}
//新增代码：异步执行flushQueue
function nextTick(flushQueue) {
  Promise.resolve().then(flushQueue)
}
```

实际上这就已经完成了批量更新和防止重复渲染。

但是为了贴近vue源码，我们更改下nextTick。使用vue的相信都用过nextTick，因此也就是说我们会在其他地方调用nextTick，而且我们是经常这样使用的

```
this.$nextTick(() => {
  this.msg2 = this.$refs.msgDiv.innerHTML
})
```

也就是说我们会传进个回调函数，而我们上面写的nextTick参数也是一个回调函数。那么我们可以把其他地方调用nextTick的回调函数一起整合进一个callback，然后统一执行callback。

因此，我们做如下更改

```
// 新增代码
let callbacks = []

function flushCallbacks() {
  callbacks.forEach(cb=>cb())
  callbacks = []
}

function nextTick(flushQueue) {
  callbacks.push(flushQueue) //新增代码

  Promise.resolve().then(flushCallbacks)
}
```

## 6. 实现数组依赖收集

上面我们只是对数组实现了方法的拦截，还没实现数据的更新渲染。现在要解决连个问题

1. 在哪里收集依赖
2. 依赖保存在哪里

实际上依赖收集依旧是在getter里实现的。

因为当我们 获取 `list:[1,2,3]` 的时候会触发get，所以可以在getter里收集依赖。

那保存在哪里呢？保存在Observe里，因为在拦截方法中可以获得observe，而在set里也可以获得observe。

```

class Observe {
  constructor(data){ // data就是我们定义的data vm._data实例
    // 将用户的数据使用defineProperty定义
    // 创建数组专用 的dep
    this.dep = new Dep()
    // 给我们的对象包括我们的数组添加一个属性__ob__ (这个属性即当前的observe)
    Object.defineProperty(data, '__ob__', {
      get:() => this
    })
    if (Array.isArray(data)){
      data.__proto__ = arrayMethods
      observerArray(data)
    }else {
      this.walk(data)
    }
  }
}

```

我们在这里返回了一个Observe对象。

然后我们需要在拦截方法里notify

```

methods.forEach(method=>{
  arrayMethods[method] = function (...arg) {
    // 不光要返回新的数组方法，还要执行监听
    let res = oldArrayPrototypeMethods[method].apply(this,arg)
    // 实现新增属性的监听
    console.log("我是{}对象中的push,我在这里实现监听");
    // 实现新增属性的监听
    let inserted
    switch (method) {
      case 'push':
      case 'unshift':
        inserted = arg
        break
      case 'splice':
        inserted = arg.slice(2)
    }
    notify();
    return res;
  }
})

```

```

        break
    default:
        break
    }
    // 实现新增属性的监听
    if (inserted){
        observerArray(inserted)
    }
    this.__ob__.dep.notify()
    return res
}
})

```

现在保存依赖和通知更新的问题都解决了，下一步就是在setter里依赖收集

## 7.watch的实现

现在在initState中添加初始化watch

```

export function initState(vm) {
    let opt = vm.$options
    if (opt.data){
        initData(vm);
    }
    if (opt.watch){
        initWatch(vm);
    }
}

```

我们现在原型对象上实现一个方法

```

MyVue.prototype.$watch = function (key, handler) {
    let vm = this
    new Watcher(vm, key, handler, {user:true})
}

```

}

这个方法为我们的watch中的key 单独创建了一个Watch实例，其中handler是回调方法。

我们希望初始化（即 new Watch ）的时候，先获得key的oldValue。方便后面和newValue比较是否发生变化。

```
class Watcher {
  constructor(vm, exprOrFn, cb = ()=>{}, opts) {
    // 省略其他代码
    if (typeof exprOrFn === 'function') {
      this.getter = exprOrFn
    } else {
      // 现在exprOrFn是我们传进来的key
      this.getter = function () {
        return util.getValue(vm, key)
      }
    }
    this.value = this.get() //获得老值oldValue
    // 创建一个watcher，默认调用此方法
  }
  get(){
    pushTarget(this)
    let value = this.getter()
    popTarget()
    return value
  }
}
```

当key的值改变的时候，会触发dep.notify。也就会触发wathcer.update ,然后触发watcher.run

我们在 run中获得新值，然后 将新值与老值进行比较，如果两者不等的话，就触发回调函数

```
run(){
  let value = this.get()
  if (this.value !== value){
```

```

        this.cb(value, this.value)
    }
}

```

ok，现在来继续初始化initWatch

```

function initWatch(vm) {
    let watch = vm.$options.watch
    for (let key in watch){
        let handler = watch[key]
        createWatch(vm, key, handler)
    }
}

function createWatch(vm, key, handler) {
    return this.$watch(vm, key, handler)
}

```

可见，其实核心思想就是给每个key 生成一个Watcher实例，来监听key的值的变化。

## 8. computed 实现

想要写computed，必须先知道computed 是有缓存的。

先来初始化computed

```

function initComputed(vm, computed) {
    let watchers = vm._watcherComputed = Object.create(null)
    for(let key in computed){
        let userDef = computed[key]
        watchers[key] = new Watcher(vm, userDef, ()=>{}, {lazy:true})
    }
}

```

可见，是先生成一个`_watcherComputed`的空对象挂载到`vm`里，然后遍历`computed`，给每个`computed`生成一个`Watcher`实例，一个key对应一个`Watcher`实例。然后保存到`_watcherComputed`里。

现在修改一下`watcher`，我们每new `Watcher`的时候就计算好key对应的值。然后保存在`Watcher`实例里。

现在我们不希望自动调用`Watcher`里的`get`方法。当`computed`的值改变时，再执行`get`，也就是`computed`的所有数据依赖有改变的时候再执行`get()`。

```
class Watcher {
  constructor(vm, exprOrFn, cb = ()=>{}, opts) {
    this.lazy = opts.lazy
    this.dirty = this.lazy
    if (typeof exprOrFn === 'function') {
      this.getter = exprOrFn
    } else {
      // 现在exprOrFn是我们传进来的key
      this.getter = function () {
        return util.getValue(vm, exprOrFn)
      }
    }
    this.value = this.lazy ? undefined : this.get() // 获得老值oldValue
    // 创建一个watcher，默认调用此方法
  }
}
```

当用户取值的时候，我们将key定义到`vm`上，并且返回`value`

```
function createComputedGetter(vm, key) {
  let watcher = vm._watcherComputed[key]
  return function () {
    if (watcher) {
      if (watcher.dirty) {
        // 页面取值的时候，dirty如果为true，就会调用get方法计算
        watcher.evalValue()
      }
    }
  }
}
```

```

        return watcher.value
    }
}

function initComputed(vm, computed) {
    let watchers = vm._watcherComputed = Object.create(null)
    for(let key in computed){
        let userDef = computed[key]
        watchers[key] = new Watcher(vm, userDef, ()=>{}, {lazy:true})

        // 当用户取值的时候，我们将key定义到vm上
        Object.defineProperty(vm, key, {
            get:createComputedGetter(vm, key)
        })
    }
}

```

`evalValue` 方法的实现非常简单

```

evalValue(){
    this.value = this.get()
    this.dirty = false
}

```

现在已经成功获得computed返回的值了，

接下来，要实现的是，当computed的依赖列表中，有变化的话，就要把dirty设置为true，重新赋予value新值

当computed里的依赖列表有变化时，就通知watcher.update。需要把dirty改为true。

```

update(){
    // this.get()

    // 批量更新， 防止重复渲染
    if (this.lazy){ // 如果是计算属性
        this.dirty = true
    }else{
        queueWatcher(this)
    }
}

```

```
    }  
}
```

现在解决依赖收集的问题

```
function createComputedGetter(vm, key) {  
  let watcher = vm._watcherComputed[key]  
  return function () {  
    if (watcher) {  
      if (watcher.dirty){  
        // 页面取值的时候，dirty如果为true，就会调用get方法计算  
        watcher.evalValue()  
      }  
      if (Dep.target){  
        watcher.depend()  
      }  
      return watcher.value  
    }  
  }  
}
```

```
depend(){  
  let i = this.deps.length  
  while(i--){  
    this.deps[i].depend()  
  }  
}
```

源码地址：<https://github.com/peigexing/myvue>

本文使用 mdnice 排版

# 手写Vue-router核心原理，再也不怕面试官问我Vue-router原理

## 手写vue-router核心原理

@[toc]

### 一、核心原理

---

#### 1.什么是前端路由？

在 Web 前端单页应用 SPA(Single Page Application)中，路由描述的是 URL 与 UI 之间的映射关系，这种映射是单向的，即 URL 变化引起 UI 更新（无需刷新页面）。

#### 2.如何实现前端路由？

要实现前端路由，需要解决两个核心：

1. 如何改变 URL 却不引起页面刷新？
2. 如何检测 URL 变化了？

下面分别使用 hash 和 history 两种实现方式回答上面的两个核心问题。

#### hash 实现

hash 是 URL 中 hash (#) 及后面的那部分，常用作锚点在页面内进行导航，改变 **URL** 中的 **hash** 部分不会引起页面刷新

通过 hashchange 事件监听 URL 的变化，改变 URL 的方式只有这几种：

1. 通过浏览器前进后退改变 URL
2. 通过 `<a>` 标签改变 URL

### 3. 通过window.location改变URL

#### history 实现

history 提供了 pushState 和 replaceState 两个方法，这两个方法改变 URL 的 **path** 部分不会引起页面刷新

history 提供类似 hashchange 事件的 popstate 事件，但 popstate 事件有些不同：

1. 通过浏览器前进后退改变 URL 时会触发 popstate 事件
2. 通过pushState/replaceState或 标签改变 URL 不会触发 popstate 事件。
3. 好在我们可以拦截 pushState/replaceState的调用和 标签的点击事件来检测 URL 变化
4. 通过js 调用history的back，go，forward方法课触发该事件

所以监听 URL 变化可以实现，只是没有 hashchange 那么方便。

## 二、原生js实现前端路由

### 1. 基于 hash 实现

html

```
<!DOCTYPE html>
<html lang="en">
<body>
<ul>
  <!-- 定义路由 -->
  <li><a href="#/home">home</a></li>
  <li><a href="#/about">about</a></li>

  <!-- 渲染路由对应的 UI -->
  <div id="routeView"></div>
</ul>
</ul>
</body>
<script>
  let routerView = routeView
```

```

window.addEventListener('hashchange', ()=>{
    let hash = location.hash;
    routerView.innerHTML = hash
})

window.addEventListener('DOMContentLoaded', ()=>{
    if(!location.hash){//如果不存在hash值，那么重定向到#/}
        location.hash="/"
    }else{//如果存在hash值，那就渲染对应UI
        let hash = location.hash;
        routerView.innerHTML = hash
    }
})

</script>
</html>

```

解释下上面代码，其实很简单：

1. 我们通过a标签的href属性来改变URL的hash值（当然，你触发浏览器的前进后退按钮也可以，或者在控制台输入window.location赋值来改变hash）
2. 我们监听**hashchange**事件。一旦事件触发，就改变**routerView**的内容，若是在vue中，这改变的应当是**router-view**这个组件的内容
3. 为何又监听了load事件？这时应为页面第一次加载完不会触发 hashchange，因而用load事件来监听hash值，再将视图渲染成对应的内容。

## 2. 基于 history 实现

```

<!DOCTYPE html>
<html lang="en">
<body>
<ul>
    <ul>
        <li><a href='/home'>home</a></li>
        <li><a href='/about'>about</a></li>
    </ul>
</ul>
<div id="routeView"></div>

```

```

        </ul>
    </ul>
</body>
<script>
    let routerView = routeView
    window.addEventListener('DOMContentLoaded', onLoad)
    window.addEventListener('popstate', ()=>{
        routerView.innerHTML = location.pathname
    })
    function onLoad () {
        routerView.innerHTML = location.pathname
        var linkList = document.querySelectorAll('a[href]')
        linkList.forEach(el => el.addEventListener('click', function (e) {
            e.preventDefault()
            history.pushState(null, '', el.getAttribute('href'))
            routerView.innerHTML = location.pathname
        }))
    }
</script>
</html>

```

解释下上面代码，其实也差不多：

1. 我们通过a标签的href属性来改变URL的path值（当然，你触发浏览器的前进后退按钮也可以，或者在控制台输入history.go,back,forward赋值来触发popState事件）。这里需要注意的就是，当改变path值时，默认会触发页面的跳转，所以需要拦截 `<a>` 标签点击事件默认行为，点击时使用 pushState 修改 URL 并更新手动 UI，从而实现点击链接更新 URL 和 UI 的效果。
2. 我们监听**popState**事件。一旦事件触发，就改变**routerView**的内容。
3. load事件则是一样的

有个问题：hash模式，也可以用 `history.go,back,forward` 来触发hashchange事件吗？

A：也是可以的。因为不管什么模式，浏览器为保存记录都会有一个栈。

### 三、基于Vue实现VueRouter

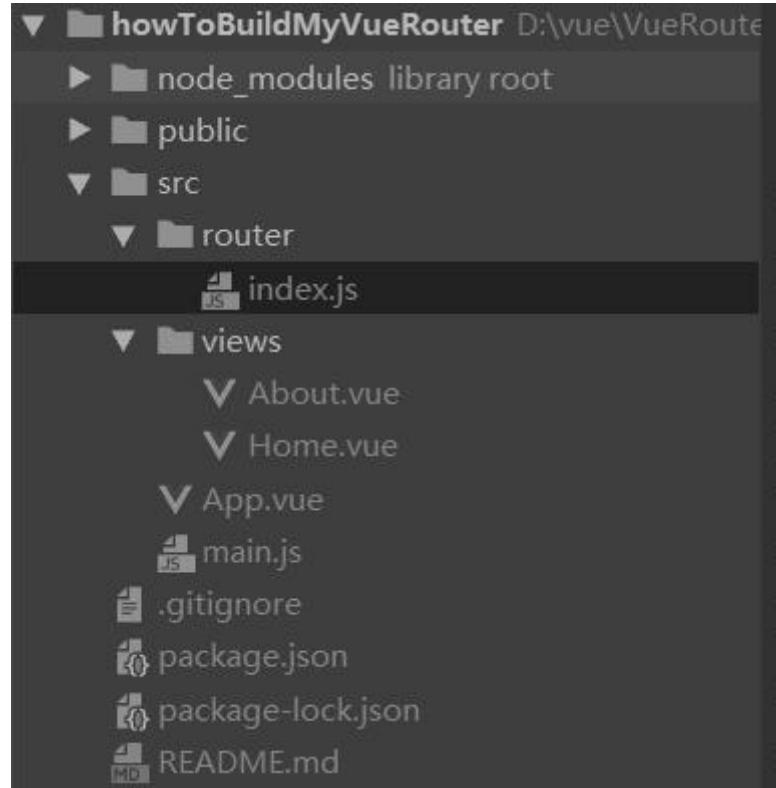
我们先利用vue-cli建一个项目

```
Vue CLI v3.10.0

[ Update available: 4.4.6 ]

? Please pick a preset: Manually select features
? Check the features needed for your project:
  ( ) Babel
  ( ) TypeScript
  ( ) Progressive Web App (PWA) Support
>(*) Router
  ( ) Vuex
  ( ) CSS Pre-processors
  ( ) Linter / Formatter
  ( ) Unit Testing
  ( ) E2E Testing
```

删除一些不必要的组建后项目目录暂时如下：



已经把项目放到 [github](#) : [github.com/Sunny-lucky/handwritten-vue-router](https://github.com/Sunny-lucky/handwritten-vue-router) 可以卑微地要个star吗。有什么不理解或者什么建议，欢迎下方评论

我们主要看下App.vue,About.vue,Home.vue,router/index.js

代码如下:

App.vue

```
<template>
<div id="app">
  <div id="nav">
    <router-link to="/home">Home</router-link> |
    <router-link to="/about">About</router-link>
  </div>
  <router-view/>
</div>
</template>
```

router/index.js

```
import Vue from 'vue'
import VueRouter from 'vue-router'
import Home from '../views/Home.vue'
import About from "../views/About.vue"
Vue.use(VueRouter)

const routes = [
{
  path: '/home',
  name: 'Home',
  component: Home
},
{
  path: '/about',
  name: 'About',
  component: About
}
]
const router = new VueRouter({
  mode:"history",
  routes
})
export default router
```

## Home.vue

```
<template>
<div class="home">
  <h1>这是Home组件</h1>
</div>
</template>
```

## About.vue

```
<template>
<div class="about">
  <h1>这是about组件</h1>
</div>
</template>
```

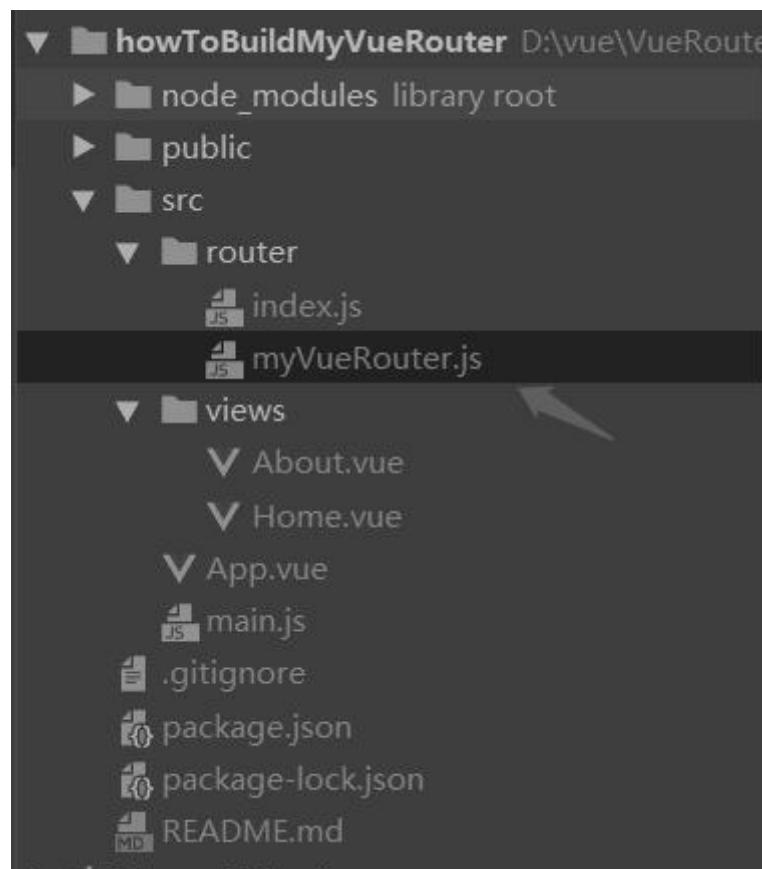
现在我们启动一下项目。看看项目初始化有没有成功。



ok，没毛病，初始化成功。

现在我们决定创建自己的VueRouter，于是创建myVueRouter.js文件

目前目录如下



再将VueRouter引入改成我们的myVueRouter.js

```
//router/index.js
import Vue from 'vue'
import VueRouter from './myVueRouter' //修改代码
import Home from '../views/Home.vue'
import About from "../views/About.vue"
Vue.use(VueRouter)
const routes = [
{
  path: '/home',
  name: 'Home',
  component: Home
},
{
  path: '/about',
  name: 'About',
  component: About
}
];
const router = new VueRouter({
  mode:"history",
  routes
})
export default router
```

## 四、剖析VueRouter本质

先抛出个问题，Vue项目中是怎么引入VueRouter。

1. 安装VueRouter，再通过 `import VueRouter from 'vue-router'` 引入
2. 先 `const router = new VueRouter({...})`，再把router作为参数的一个属性值，`new Vue({router})`
3. 通过`Vue.use(VueRouter)`使得每个组件都可以拥有store实例

从这个引入过程我们可以发现什么？

1. 我们是通过new VueRouter({...})获得一个router实例，也就是说，我们引入的VueRouter其实是一个类。

所以我们可以初步假设

```
class VueRouter{  
}  
}
```

2. 我们还使用了Vue.use(),而Vue.use的一个原则就是执行对象的install这个方法

所以，我们可以再一步 假设VueRouter有install这个方法。

```
class VueRouter{  
}  
VueRouter.install = function () {  
}  
}
```

到这里，你能大概地将VueRouter写出来吗？

很简单，就是将上面的VueRouter导出，如下就是myVueRouter.js

```
//myVueRouter.js  
class VueRouter{  
}  
VueRouter.install = function () {  
}  
  
export default VueRouter
```

## 五、分析Vue.use

```
Vue.use(plugin);
```

### (1) 参数

```
{ Object | Function } plugin
```

### (2) 用法

安装Vue.js插件。如果插件是一个对象，必须提供install方法。如果插件是一个函数，它会被作为install方法。调用install方法时，会将Vue作为参数传入。install方法被同一个插件多次调用时，插件也只会被安装一次。

关于如何上开发Vue插件，请看这篇文章，非常简单，不用两分钟就看完：[如何开发 Vue 插件？](#)

### (3) 作用

注册插件，此时只需要调用install方法并将Vue作为参数传入即可。但在细节上有两部分逻辑要处理：

1、插件的类型，可以是install方法，也可以是一个包含install方法的对象。

2、插件只能被安装一次，保证插件列表中不能有重复的插件。

### (4) 实现

```
Vue.use = function(plugin){
  const installedPlugins = (this._installedPlugins || (this._installedPlugins = []));
  if(installedPlugins.indexOf(plugin) > -1){
    return this;
  }
  // 其他参数
  const args = toArray(arguments, 1);
  args.unshift(this);
  if(typeof plugin.install === 'function'){
    plugin.install.apply(plugin, args);
  } else if(typeof plugin === 'function'){
    plugin.apply(null, plugin, args);
  }
  installedPlugins.push(plugin);
}
```

```

    return this;
}

```

1、在Vue.js上新增了use方法，并接收一个参数plugin。

2、首先判断插件是不是已经别注册过，如果被注册过，则直接终止方法执行，此时只需要使用indexOf方法即可。

3、toArray方法我们在就是将类数组转成真正的数组。使用toArray方法得到arguments。除了第一个参数之外，剩余的所有参数将得到的列表赋值给args，然后将Vue添加到args列表的最前面。这样做的目的是保证install方法被执行时第一个参数是Vue，其余参数是注册插件时传入的参数。

4、由于plugin参数支持对象和函数类型，所以通过判断plugin.install和plugin哪个是函数，即可知用户使用哪种方式注册的插件，然后执行用户编写的插件并将args作为参数传入。

5、最后，将插件添加到installedPlugins中，保证相同的插件不会反复被注册。(～让我想起了曾经面试官问我为什么插件不会被重新加载！！！哭唧唧，现在总算明白了)

第三点讲到，我们把Vue作为install的第一个参数，所以我们可以把Vue保存起来

```

//myVueRouter.js
let Vue = null;
class VueRouter{

}

VueRouter.install = function (v) {
    Vue = v;
};

export default VueRouter

```

然后再通过传进来的Vue创建两个组件router-link和router-view

```

//myVueRouter.js
let Vue = null;
class VueRouter{

```

```

}

VueRouter.install = function (v) {
  Vue = v;
  console.log(v);

  //新增代码
  Vue.component('router-link',{
    render(h){
      return h('a',{},'首页')
    }
  })
  Vue.component('router-view',{
    render(h){
      return h('h1',{},'首页视图')
    }
  })
};

export default VueRouter

```

我们执行下项目，如果没报错，说明我们的假设没毛病。



天啊，没报错。没毛病！

## 六、完善install方法

install一般是给每个vue实例添加东西的

在这里就是给每个组件添加\$route 和 \$router。

## \$route 和 \$router 有什么区别？

A：\$router 是VueRouter的实例对象，\$route 是当前路由对象，也就是说\$route 是\$router 的一个属性 注意每个组件添加的\$route 是同一个，\$router 也是同一个，所有组件共享的。

这是什么意思呢？？？

来看main.js

```
import Vue from 'vue'
import App from './App.vue'
import router from './router'

Vue.config.productionTip = false

new Vue({
  router,
  render: function (h) { return h(App) }
}).$mount('#app')
```

我们可以发现这里只是将router，也就是./router导出的store实例，作为Vue参数的一部分。

但是这里就是一个问题咯，这里的Vue是根组件啊。也就是说目前只有根组件有这个router值，而其他组件是还没有的，所以我们需要让其他组件也拥有这个router。

因此，install方法我们可以这样完善

```
//myVueRouter.js
let Vue = null;
class VueRouter{

}

VueRouter.install = function (v) {
  Vue = v;
  // 新增代码
  Vue.mixin({
    beforeCreate(){

```

```

if (this.$options && this.$options.router){ // 如果是根组件
    this._root = this; //把当前实例挂载到_root上
    this._router = this.$options.router;
} else { //如果是子组件
    this._root= this.$parent && this.$parent._root
}
Object.defineProperty(this, '$router',{
    get(){
        return this._root._router
    }
})
}

Vue.component('router-link',{
    render(h){
        return h('a',{},'首页')
    }
})
Vue.component('router-view',{
    render(h){
        return h('h1',{},'首页视图')
    }
})
};

export default VueRouter

```

解释下代码：

1. 参数Vue，我们在第四小节分析Vue.use的时候，再执行install的时候，将Vue作为参数传进去。
2. mixin的作用是将mixin的内容混合到Vue的初始参数options中。相信使用vue的同学应该使用过mixin了。
3. 为什么是beforeCreate而不是created呢？因为如果是在created操作的话，\$options已经初始化好了。
4. 如果判断当前组件是根组件的话，就将我们传入的router和\_root挂在到根组件实例上。

- 如果判断当前组件是子组件的话，就将我们\_root根组件挂载到子组件。注意是引用的复制，因此每个组件都拥有了同一个\_root根组件挂载在它身上。

这里有个问题，为什么判断当前组件是子组件，就可以直接从父组件拿到\_root根组件呢？这让我想起了曾经一个面试官问我的问题：父组件和子组件的执行顺序？

A：父beforeCreate->父created -> 父beforeMount -> 子beforeCreate ->子create ->子beforeMount ->子mounted -> 父mounted

可以得到，在执行子组件的beforeCreate的时候，父组件已经执行完beforeCreate了，那理所当然父组件已经有\_root了。

然后我们通过

```
Object.defineProperty(this, '$router', {
  get(){
    return this._root._router
  }
})
```

将 \$router 挂载到组件实例上。

其实这种思想也是一种代理的思想，我们获取组件的 \$router ，其实返回的是根组件的 \_root.\_router

到这里还install还没写完，可能你也发现了， \$route 还没实现，现在还实现不了，没有完善VueRouter的话，没办法获得当前路径

## 七、完善VueRouter类

我们先看看我们new VueRouter类时传进了什么东东

```
//router/index.js
import Vue from 'vue'
import VueRouter from './myVueRouter'
import Home from '../views/Home.vue'
import About from "../views/About.vue"
Vue.use(VueRouter)
```

```

const routes = [
  {
    path: '/home',
    name: 'Home',
    component: Home
  },
  {
    path: '/about',
    name: 'About',
    component: About
  }
];
const router = new VueRouter({
  mode:"history",
  routes
})
export default router

```

可见，传入了一个为数组的路由表routes，还有一个代表当前是什么模式的mode。因此我们可以先这样实现VueRouter

```

class VueRouter{
  constructor(options) {
    this.mode = options.mode || "hash"
    this.routes = options.routes || [] //你传递的这个路由是一个数组表
  }
}

```

先接收了这两个参数。

但是我们直接处理routes是十分不方便的，所以我们先要转换成 key : value 的格式

```

//myVueRouter.js
let Vue = null;
class VueRouter{

```

```

constructor(options) {
  this.mode = options.mode || "hash"
  this.routes = options.routes || [] //你传递的这个路由是一个数组表
  this.routesMap = this.createMap(this.routes)
  console.log(this.routesMap);
}

createMap(routes){
  return routes.reduce((pre,current)=>{
    pre[current.path] = current.component
    return pre;
  },{})
}

}

```

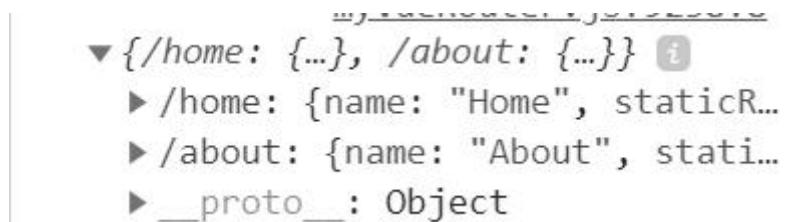
通过createMap我们将

```

const routes = [
  {
    path: '/home',
    name: 'Home',
    component: Home
  },
  {
    path: '/about',
    name: 'About',
    component: About
  }
]

```

转换成



```

▼ {/home: {...}, /about: {...}}
  ► /home: {name: "Home", staticR...
  ► /about: {name: "About", stati...
  ► __proto__: Object

```

路由中需要存放当前的路径，来表示当前的路径状态为了方便管理，可以用一个对象来表示

```
//myVueRouter.js
let Vue = null;
新增代码
class HistoryRoute {
  constructor(){
    this.current = null
  }
}
class VueRouter{
  constructor(options) {
    this.mode = options.mode || "hash"
    this.routes = options.routes || [] //你传递的这个路由是一个数组表
    this.routesMap = this.createMap(this.routes)
    新增代码
    this.history = new HistoryRoute();
  }
}

createMap(routes){
  return routes.reduce((pre,current)=>{
    pre[current.path] = current.component
    return pre;
  },{})
}

}
```

但是我们现在发现这个current也就是当前路径还是null，所以我们需要进行初始化。

初始化的时候判断是hash模式还是history模式。,然后将当前路径的值保存到current里

```
//myVueRouter.js
let Vue = null;
class HistoryRoute {
  constructor(){
    this.current = null
  }
}
```

```

    }

}

class VueRouter{
  constructor(options) {
    this.mode = options.mode || "hash"
    this.routes = options.routes || [] //你传递的这个路由是一个数组表
    this.routesMap = this.createMap(this.routes)
    this.history = new HistoryRoute();
    新增代码
    this.init()

  }
  新增代码
  init(){
    if (this.mode === "hash"){
      // 先判断用户打开时有没有hash值，没有的话跳转到#/ 
      location.hash? '' :location.hash = "/";
      window.addEventListener("load", ()=>{
        this.history.current = location.hash.slice(1)
      })
      window.addEventListener("hashchange", ()=>{
        this.history.current = location.hash.slice(1)
      })
    } else{
      location.pathname? '' :location.pathname = "/";
      window.addEventListener('load', ()=>{
        this.history.current = location.pathname
      })
      window.addEventListener("popstate", ()=>{
        this.history.current = location.pathname
      })
    }
  }

  createMap(routes){
    return routes.reduce((pre, current)=>{
      pre[current.path] = current.component
    })
  }
}

```

```

        return pre;
    },{})
}

}

```

监听事件跟上面原生js实现的时候一致。

## 八、完善\$route

前面那我们讲到，要先实现VueRouter的history.current的时候，才能获得当前的路径，而现在已经实现了，那么就可以着手实现\$route了。

很简单，跟实现\$router一样

```

VueRouter.install = function (v) {
    Vue = v;
    Vue.mixin({
        beforeCreate(){
            if (this.$options && this.$options.router){ // 如果是根组件
                this._root = this; //把当前实例挂载到_root上
                this._router = this.$options.router;
            }else { //如果是子组件
                this._root= this.$parent && this.$parent._root
            }
            Object.defineProperty(this, '$router', {
                get(){
                    return this._root._router
                }
            });
        新增代码
        Object.defineProperty(this, '$route', {
            get(){
                return this._root._router.history.current
            }
        })
    })
}

```

```

        }
    })
    Vue.component('router-link',{
        render(h){
            return h('a',{},'首页')
        }
    })
    Vue.component('router-view',{
        render(h){
            return h('h1',{},'首页视图')
        }
    })
});

```

## 九、完善router-view组件

现在我们已经保存了当前路径，也就是说现在我们可以获得当前路径，然后再根据当前路径从路由表中获取对应的组件进行渲染

```

Vue.component('router-view',{
    render(h){
        let current = this._self._root._router.history.current
        let routeMap = this._self._root._router.routesMap;
        return h(routeMap[current])
    }
})

```

解释一下：

render函数里的this指向的是一个Proxy代理对象，代理Vue组件，而我们前面讲到每个组件都有一个\_root属性指向根组件，根组件上有\_router这个路由实例。所以我们可以从\_router实例上获得路由表，也可以获得当前路径。然后再把获得的组件放到h()里进行渲染。

现在已经实现了router-view组件的渲染，但是有一个问题，就是你改变路径，视图是没有重新渲染的，所以需要将\_router.history进行响应式化。

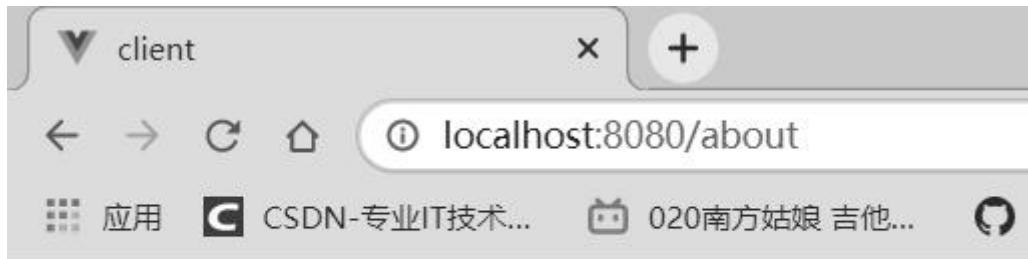
```

Vue.mixin({
  beforeCreate(){
    if (this.$options && this.$options.router){ // 如果是根组件
      this._root = this; //把当前实例挂载到_root上
      this._router = this.$options.router;
      新增代码
      Vue.util.defineReactive(this,"xxx",this._router.history)
    }else { //如果是子组件
      this._root= this.$parent && this.$parent._root
    }
    Object.defineProperty(this,'$router',{
      get(){
        return this._root._router
      }
    });
    Object.defineProperty(this,'$route',{
      get(){
        return this._root._router.history.current
      }
    })
  }
})
  
```

我们利用了Vue提供的API：defineReactive，使得this.\_router.history对象得到监听。

因此当我们第一次渲染**router-view**这个组件的时候，会获取到 this.\_router.history 这个对象，从而就会被监听到获取 this.\_router.history 。就会把**router-view**组件的依赖**watcher**收集到 this.\_router.history 对应的收集器**dep**中，因此 this.\_router.history 每次改变的时候。 this.\_router.history 对应的收集器**dep**就会通知**router-view**的组件依赖的**watcher**执行**update()**，从而使得 **router-view** 重新渲染（其实这就是**vue**响应式的内部原理）

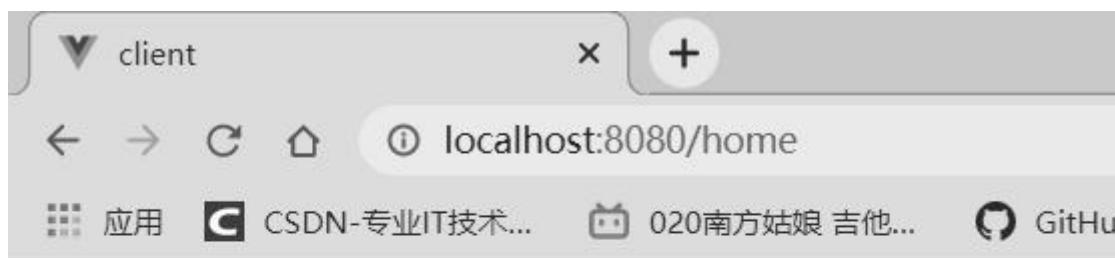
好了，现在我们来测试一下，通过改变url上的值，能不能触发**router-view**的重新渲染



首页 | 首页

# 这是about组件

path改成home



首页 | 首页

# 这是Home组件

可见成功实现了当前路径的监听。。

## 十、完善router-link组件

我们先看下router-link是怎么使用的。

```
<router-link to="/home">Home</router-link>
<router-link to="/about">About</router-link>
```

也就是说父组件间to这个路径传进去，子组件接收就好 因此我们可以这样实现

```
Vue.component('router-link',{
  props:{
    to:String
  },
  render(h){
    let mode = this._self._root._router.mode;
    let to = mode === "hash"? "#" + this.to : this.to
    return h('a',{attrs:{href:to}},this.$slots.default)
  }
})
```

我们把router-link渲染成a标签，当然这时最简单的做法。通过点击a标签就可以实现url上路径的切换。从而实现视图的重新渲染

ok，到这里完成此次的项目了。

看下VueRouter的完整代码吧

```
//myVueRouter.js
let Vue = null;
class HistoryRoute {
  constructor(){
    this.current = null
  }
}
class VueRouter{
  constructor(options) {
    this.mode = options.mode || "hash"
    this.routes = options.routes || [] //你传递的这个路由是一个数组表
    this.routesMap = this.createMap(this.routes)
    this.history = new HistoryRoute();
    this.init()
  }
  init(){
    this.history.start();
  }
}
```

```

if (this.mode === "hash"){
    // 先判断用户打开时有没有hash值，没有的话跳转到#/ 
    location.hash? '' : location.hash = "/";
    window.addEventListener("load", ()=>{
        this.history.current = location.hash.slice(1)
    })
    window.addEventListener("hashchange", ()=>{
        this.history.current = location.hash.slice(1)
    })
} else{
    location.pathname? '' : location.pathname = "/";
    window.addEventListener('load', ()=>{
        this.history.current = location.pathname
    })
    window.addEventListener("popstate", ()=>{
        this.history.current = location.pathname
    })
}

createMap(routes){
    return routes.reduce((pre, current)=>{
        pre[current.path] = current.component
        return pre;
    }, {})
}

VueRouter.install = function (v) {
    Vue = v;
    Vue.mixin({
        beforeCreate(){
            if ($options && $options.router){ // 如果是根组件
                _root = this; // 把当前实例挂载到_root上
                _router = $options.router;
                Vue.util.defineReactive(this, "xxx", _router.history)
            }else { // 如果是子组件
}
}

```

```

    this._root= this.$parent && this.$parent._root
}

Object.defineProperty(this, '$router',{
    get(){
        return this._root._router
    }
});

Object.defineProperty(this, '$route',{
    get(){
        return this._root._router.history.current
    }
})
}

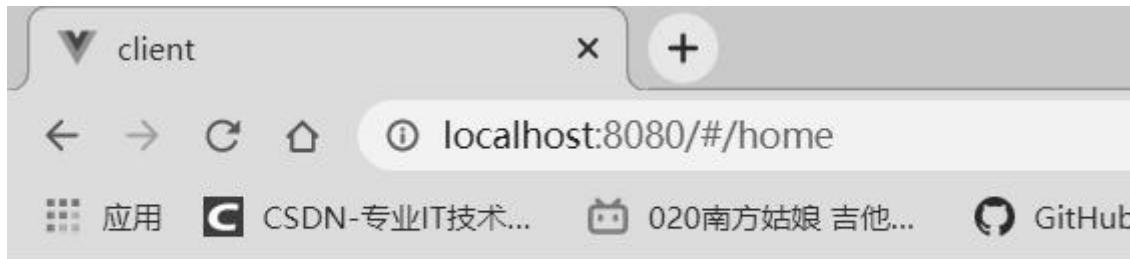
Vue.component('router-link',{
    props:{
        to:String
    },
    render(h){
        let mode = this._self._root._router.mode;
        let to = mode === "hash"? "#" + this.to : this.to
        return h('a',{attrs:{href:to}},this.$slots.default)
    }
})

Vue.component('router-view',{
    render(h){
        let current = this._self._root._router.history.current
        let routeMap = this._self._root._router.routesMap;
        return h(routeMap[current])
    }
})
};

export default VueRouter

```

现在测试下成功没



Home | About

# 这是Home组件



Home | About

# 这是about组件

点击确实视图切换了，成功。

完美收官！！！

有什么不理解或者什么建议，欢迎下方评论

感谢您也恭喜您看到这里，我可以卑微的求个star吗！！！

github : [github.com/Sunny-lucki...](https://github.com/Sunny-lucki...)

参考文献：文章前面一、二节原理部分 摘自：[blog.csdn.net/qq867263657...](http://blog.csdn.net/qq867263657...)

## 关注下面的标签，发现更多相似文章



# Vue.nextTick 的原理和用途

## 概览

官方文档说明：

- 用法：

在下次 DOM 更新循环结束之后执行延迟回调。在修改数据之后立即使用这个方法，获取更新后的 DOM。

疑问：

1. DOM 更新循环是指什么？
2. 下次更新循环是什么时候？
3. 修改数据之后使用，是加快了数据更新进度吗？
4. 在什么情况下要用到？

## 原理

### 异步说明

Vue 实现响应式并不是数据发生变化之后 DOM 立即变化，而是按一定的策略进行 DOM 的更新。

在 Vue 的文档中，说明 Vue 是异步执行 DOM 更新的。关于异步的解析，可以查看阮一峰老师的这篇文章。截取关键部分如下：

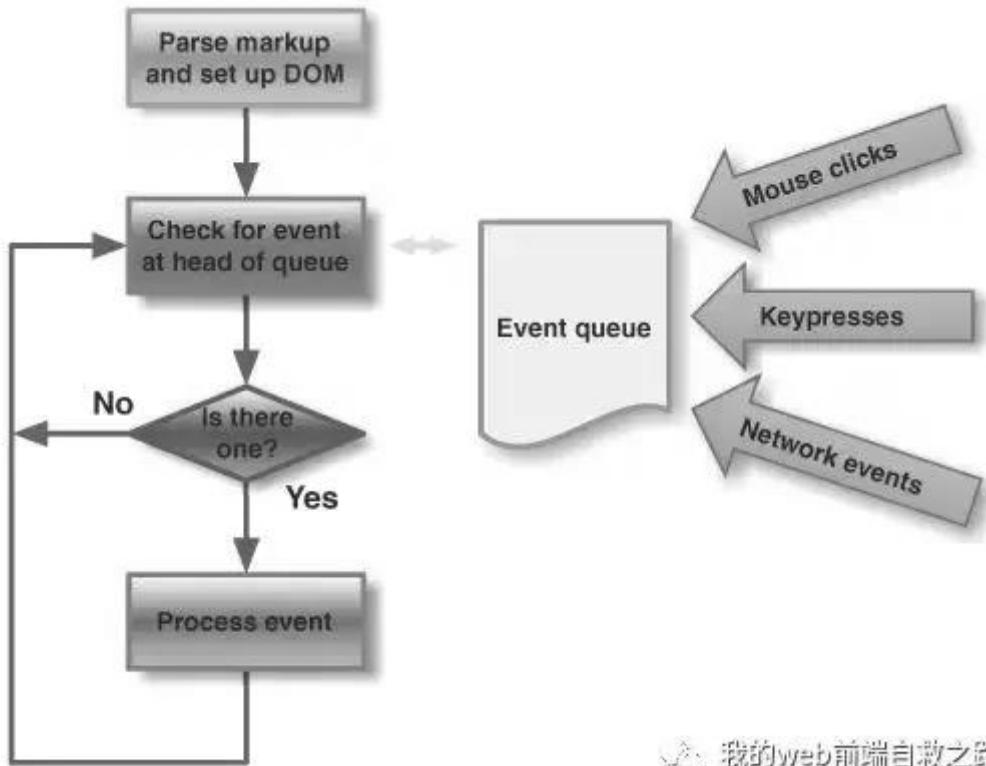
具体来说，异步执行的运行机制如下。

- (1) 所有同步任务都在主线程上执行，形成一个执行栈 (execution context stack)。
- (2) 主线程之外，还存在一个"任务队列" (task queue)。只要异步任务有了运行结果，就在"任务队列"之中放置一个事件。
- (3) 一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。

那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。

(4) 主线程不断重复上面的第三步。

下图就是主线程和任务队列的示意图。



我的web前端自救之路

## 事件循环说明

简单来说，Vue 在修改数据后，视图不会立刻更新，而是等同一事件循环中的所有数据变化完成之后，再统一进行视图更新。

知乎上的例子：

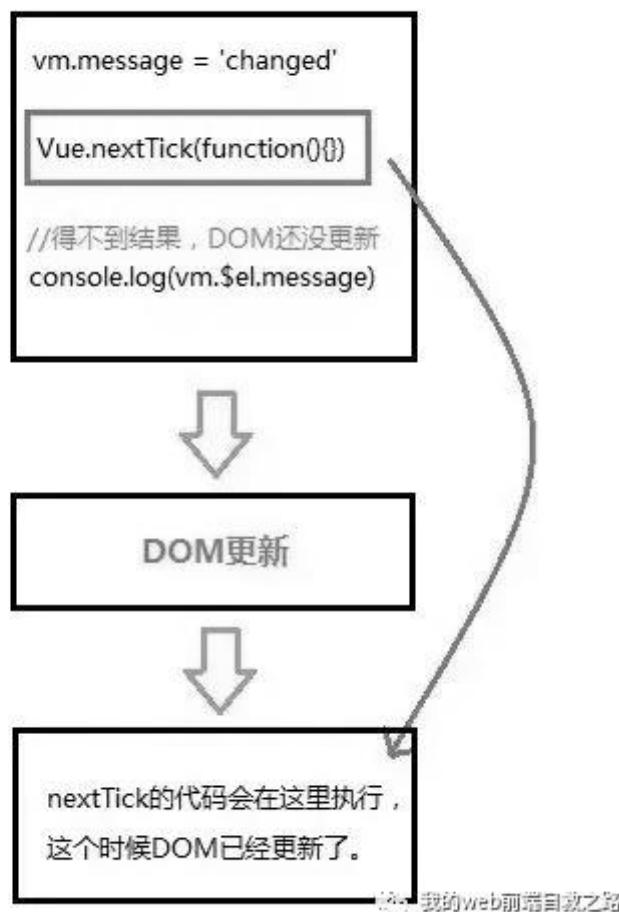
```

// 改变数据
vm.message = 'changed'

// 想要立即使用更新后的DOM。
这样不行，因为设置message后DOM还没有更新
console.log(vm.$el.textContent)
  
```

```
// 并不会得到'changed'  
//这样可以，nextTick里面的代码会在DOM更新后执行  
Vue.nextTick(function(){    console.log(vm.$el.textContent)  
//可以得到'changed'  
})
```

图解：



事件循环：

第一个 tick ( 图例中第一个步骤，即'本次更新循环' ) :

1. 首先修改数据，这是同步任务。同一事件循环的所有的同步任务都在主线程上执行，形成一个执行栈，此时还未涉及 DOM 。
2. Vue 开启一个异步队列，并缓冲在此事件循环中发生的所有数据改变。如果同一个 watcher 被多次触发，只会被推入到队列中一次。

第二个 tick ( 图例中第二个步骤，即'下次更新循环' ) :

同步任务执行完毕，开始执行异步 watcher 队列的任务，更新 DOM。Vue 在内部尝试对异步队列使用原生的 Promise.then 和 MessageChannel 方法，如果执行环境不支持，会采用 setTimeout(fn, 0) 代替。

第三个 tick ( 图例中第三个步骤 ) :

此时就是文档所说的

下次 DOM 更新循环结束之后

此时通过 Vue.nextTick 获取到改变后的 DOM。通过 setTimeout(fn, 0) 也可以同样获取到。

简单总结事件循环：

同步代码执行 -> 查找异步队列，推入执行栈，执行 Vue.nextTick[事件循环1] -> 查找异步队列，推入执行栈，执行 Vue.nextTick[事件循环2]...

总之，异步是单独的一个 tick，不会和同步在一个 tick 里发生，也是 DOM 不会马上改变的原因。

对于事件循环，可以在这里查看更详细的内容：<https://segmentfault.com/a/11...>

## 用途

应用场景：需要在视图更新之后，基于新的视图进行操作。

### created、mounted

需要注意的是，在 created 和 mounted 阶段，如果需要操作渲染后的试图，也要使用 nextTick 方法。

官方文档说明：

注意 mounted 不会承诺所有的子组件也都一起被挂载。如果你希望等到整个视图都渲染完毕，可以用 vm.\$nextTick 替换掉 mounted

```
mounted: function () {
  this.$nextTick(function () {
    // Code that will run only after the
    // entire view has been rendered
  })}
}
```

## 其他应用场景

其他应用场景如下三例：

例子1：

点击按钮显示原本以 v-show = false 隐藏起来的输入框，并获取焦点。

```
showsou(){
  this.showit = true
  //修改 v-show
  document.getElementById("keywords").focus()
  //在第一个 tick 里，获取不到输入框，自然也获取不到焦点
}
```

修改为：

```
showsou(){
  this.showit = true
  this.$nextTick(function () {
    // DOM 更新了
    document.getElementById("keywords").focus()
  })
}
```

## 实例理解 nextTick 应用

下面的例子来自 <https://www.cnblogs.com/hity-...>，稍有改动。各位可以复制运行一遍，加深理解。

```
<div>
  <ul>
    <li class="example" v-for="item in list1">{{item}}</li>
  </ul>
  <ul>
    <li class="example" v-for="item in list2">{{item}}</li>
```

```
</ul>
<ol>
  <li class="example" v-for="item in list3">{{item}}</li>
</ol>
<ol>
  <li class="example" v-for="item in list4">{{item}}</li>
</ol>
<ol>
  <li class="example" v-for="item in list5">{{item}}</li>
</ol>
</div>
</template>
<script type="text/javascript">
export default {
  data() {
    return {
      list1: [],
      list2: [],
      list3: [],
      list4: [],
      list5: []
    }
  },
  created() {
    this.composeList12()
    this.composeList34()
    this.composeList5()
    this.$nextTick(function() {
      // DOM 更新了
      console.log('finished test ' + new Date().toString(), document.querySelector('#app'))
    })
  },
  methods: {
    composeList12() {
      let me = this
      let count = 10000
```

```
for (let i = 0; i < count; i++) {
    this.$set(me.list1, i, 'I am a 测试信息～～啦啦啦' + i)
}
console.log('finished list1 ' + new Date().toString(),document.que

for (let i = 0; i < count; i++) {
    this.$set(me.list2, i, 'I am a 测试信息～～啦啦啦' + i)
}
console.log('finished list2 ' + new Date().toString(),document.que

this.$nextTick(function() {
    // DOM 更新了
    console.log('finished tick1&2 ' + new Date().toString(),document.que
})
},
composeList34() {
    let me = this
    let count = 10000

    for (let i = 0; i < count; i++) {
        this.$set(me.list3, i, 'I am a 测试信息～～啦啦啦' + i)
    }
    console.log('finished list3 ' + new Date().toString(),document.que

this.$nextTick(function() {
    // DOM 更新了
    console.log('finished tick3 ' + new Date().toString(),document.que
})

setTimeout(me.setTimeout1, 0)
},
setTimeout1() {
    let me = this
    let count = 10000

    for (let i = 0; i < count; i++) {
        this.$set(me.list4, i, 'I am a 测试信息～～啦啦啦' + i)
    }
}
```

```
        }

        console.log('finished list4 ' + new Date().toString(),document.que

        me.$nextTick(function() {
            // DOM 更新了
            console.log('finished tick4 ' + new Date().toString(),document
        })

    },
    composeList5() {
        let me = this
        let count = 10000

        this.$nextTick(function() {
            // DOM 更新了
            console.log('finished tick5-1 ' + new Date().toString(),docume
        })

        setTimeout(me.setTimeout2, 0)
    },
    setTimeout2() {
        let me = this
        let count = 10000

        for (let i = 0; i < count; i++) {
            this.$set(me.list5, i, 'I am a 测试信息～～啦啦啦' + i)
        }
        console.log('finished list5 ' + new Date().toString(),document.que

        me.$nextTick(function() {
            // DOM 更新了
            console.log('finished tick5 ' + new Date().toString(),document
        })

    }
}

</script>
```

结果：

```
finished list1 Tue Jan 16 2018 16:35:43 GMT+0800 (中国标准时间) 0
finished list2 Tue Jan 16 2018 16:35:44 GMT+0800 (中国标准时间) 0
finished list3 Tue Jan 16 2018 16:35:44 GMT+0800 (中国标准时间) 0
finished tick1&2 Tue Jan 16 2018 16:35:45 GMT+0800 (中国标准时间) 30000
finished tick3 Tue Jan 16 2018 16:35:45 GMT+0800 (中国标准时间) 30000
finished tick5-1 Tue Jan 16 2018 16:35:45 GMT+0800 (中国标准时间) 30000
finished test Tue Jan 16 2018 16:35:45 GMT+0800 (中国标准时间) 30000
finished list4 Tue Jan 16 2018 16:35:47 GMT+0800 (中国标准时间) 30000
finished tick4 Tue Jan 16 2018 16:35:47 GMT+0800 (中国标准时间) 40000
finished list5 Tue Jan 16 2018 16:35:47 GMT+0800 (中国标准时间) 50000
```



我的web前端自救之路

公众号回复 加群

和大佬们交流技术吧

微信公众号：我的web前端自救之路

回复 加群，跟大佬们一起交流技术吧

# 学习vue源码（1）手写与事件相关的实例方法

在Vue.js内部有这样一段代码

```
function Vue (options) {
  if (!(this instanceof Vue))
  ) {
    warn('Vue is a constructor and should be called with the `new` keyword');
  }
  this._init(options);
}

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

其中定义了Vue构造函数，然后分别调用initMixin，stateMixin，eventsMixin，lifecycleMixin，renderMixin，并将Vue构造函数当作参数传给这5个函数

这5个函数的作用就是向Vue的原型上挂载方法。

```
function initMixin (Vue) {
  Vue.prototype._init = function (options) {
```

当函数initMixin被调用时，会向Vue构造函数的prototype属性添加\_init方法，执行new Vue()时，会调用\_init方法。

好，接下来进入到本文的标题，与事件相关的实例方法有4个，分别是，**vm.on, vm.emit, vm.once, vm.off**;

这四个方法实在eventsMixin中挂在到Vue构造函数和prototype属性中的。

当eventsMixin被调用的时候，会向Vue构造函数的prototype属性添加4个实例方法。

```
function eventsMixin (Vue) {
  var hookRE = /^hook:/;
  Vue.prototype.$on = function (event, fn) {
    Vue.prototype.$once = function (event, fn) {
      var vm = this;
```

```
Vue.prototype.$off = function (event, fn) {
  var vm = this;
```

下面将介绍一下这四种方法

### 1、vm.\$on

**vm.\$on(event,callback)**

- 参数：
- `{string | Array<string>}` event (数组只在 2.2.0+ 中支持)
- `{Function}` callback

用法：监听当前实例上的自定义事件，事件可以由`vm.$emit`触发，回调函数会接收所有传入事件所触发的函数的额外参数

原理：只需要在注册事件时将回调函数收集起来，在触发事件时将收集起来的回调函数依次调用即可，代码如下

```
Vue.prototype.$on = function (event, fn) {
  var vm = this;
  if (Array.isArray(event)) {
    for (var i = 0, l = event.length; i < l; i++) {
      vm.$on(event[i], fn);
    }
  } else {
    (vm._events[event] || (vm._events[event] = [])).push(fn);
  }
  return vm
};
```

当`event`为数组的时候，需要遍历数组，使得数组中的每一项都调用`vm.$on`，使回调可以注册到数组中每项事件名所指定的事件列表中，当`event`参数不为数组时，就像事件列表中添加回调。

`vm._events`是一个对象，用来存储事件，使用事件名（`event`）从`vm._events`中取出事件列表，如果列表不存在，则使用空数组初始化，然后将回调函数添加到事件列表中。

`vm._events`在执行`new Vue()`时，会执行`this._init`方法进行一系列操作，其中就在`vue.js`的实例上创建一个`_events`

## 2、vm.\$emit

**vm.\$emit(eventName,[...args])**

参数：

- {string} eventName
- [... args]

用法：触发当前实例上的事件，附加参数都会传给监听器回调。

原理：vm.\$emit作用是触发事件，所有的事件监听器回调都会存储在vm.\_events中，所以触发事件的实现思路是使用事件名从vm.\_events中取出对应的事件监听器回调

列表，然后依次执行列表中的监听器回调并将参数传递给监听器回调，代码如下

```
Vue.prototype.$emit = function (event) {
  var vm = this;
  var cbs = vm._events[event];
  if (cbs) {
    // const args = toArray(arguments,1);
    cbs = cbs.length > 1 ? toArray(cbs) : cbs;
    var args = toArray(arguments, 1);
    var info = "event handler for '" + event + "'";
    for (var i = 0, l = cbs.length; i < l; i++) {
      try{
        cbs[i].apply(vm,args)
      }catch(e){
        handleError(e,vm,`event handler for "${event}"`)
      }
    }
  }
  return vm
};
```

使用event从vm.\_events中取出事件监听器回调函数列表，并将其赋值给变量cbs，如果cbs存在，依次调用每一个监听器回调并将其所有参数传给监听器回调。

toArray的作用是将类似于数组的数据换成真正的数组，它的第二个参数是起始位置，也就是说，args是一个数组，里面包含除第一个参数之外的所有参数。

**on**和**emit**可以结合起来使用，大部分用于父子组件传值，但是这里有一个问题

1、究竟是由子组件内部主动传数据给父组件，由父组件监听接收（由子组件中操作决定什么时候传值）

2、还是通过父组件决定子组件什么时候传值给父组件，然后再监听接收（由父组件中操作决定什么时候传值）

两种情况都有

**meit**事件触发，通过子组件内部的事件触发自定义事件**emit**

**meit**事件触发，可以通过父组件操作子组件(**ref**)的事件来触发自定义事件**emit**

第一种情况

父组件 所以msg打印出来就是蜡笔小柯南

```
<div id="aboutIndex">
  <About v-on:handleClick="handleClick" About>
    <!-- ref="handle"></ -->
    <!-- <input type="button" @click="handleButton" /> -->
    <div>{{msg}}</div>
    <!-- <router-view></router-view> -->
</div>
</template>
<script>
  import About from "./page/About.vue";
  export default {
    data() {
      return {
        msg: ""
      };
    },
    components: {
      About
    },
    methods: {
      handleClick(value) {
        // 直接监听，监听子组件触发事件传来的值
        console.log("-----");
        this.msg = value
      },
    }
  }
</script>
```

子组件

```
<div class="about">
  <button @click="handleEmit">按钮</button>
  <!-- <input type="button" @click="jumpTo" value="跳转" />
</div>
</template>
<script>
export default {
  data(){
    return{
      msg:"hello 酷狗",
      message:"蜡笔小柯南"
    }
  },
  methods:{
    handleEmit(){
      console.log("!!!!!!")
      //子组件主动触发自定义事件handleClick
      this.$emit('handleClick',this.message)
    },
  }
}
```

第二种情况

父组件 通过ref操作子组件触发事件

```

<div id="aboutIndex">
  <About v-on:handleClick="handleClick" ref="handle"></About>
  <input type="button" @click="handleButton" value="父组件主动要求触发" />
  <div>{{msg}}</div>
  <!-- <router-view></router-view> -->
</div>
</template>
<script>
import About from "./page/About.vue";
export default {
  data() {
    return {
      msg: ""
    };
  },
  components: {
    About
  },
  methods: {
    handleButton() {
      console.log("你走了吗");
      // 通过ref获取子组件的实例，在父组件主动操作子组件触发事件
      this.$refs.handle.target();
    },
    handleClick(value) {
      // 直接监听，监听子组件触发事件传来的值
    }
  }
}

```

子组件

```

  target(){
    // 父组件操作子组件触发自定义事件handleClick，将数据传给了父组件
    this.$emit('handleClick',this.msg)
  },
  jumpTo(){
    ...
  }
}

```

将两者情况对比，区别就在于\$emit 自定义事件的触发是有父组件还是子组件去触发

第一种，是在子组件中定义一个click点击事件来触发自定义事件\$emit,然后在父组件监听

第二种，是在父组件中第一个click点击事件，在组件中通过refs获取实例方法来直接触发事件，然后在父组件中监听

## 2、vm.\$off([event,callback])

参数：

- {string | Array<string>} event (只在 2.2.2+ 支持数组)
- {Function} [callback]

用法：

移除自定义事件监听器。

如果没有提供参数，则移除所有的事件监听器；

如果只提供了事件，则移除该事件所有的监听器；

如果同时提供了事件与回调，则只移除这个回调的监听器。

```
Vue.prototype.$off = function(event,fn){
  const vm = this;
  //1-1、没有提供参数的情况，此时需要移除所有事件的监听器
  if(!arguments.length){
    // 当arguments.length为0时，说明没有任何参数，这时需要移除所有事件监听器
    // 因此重置了vm._events属性，vm._events存储所有事件，所以将vm._events重置
    // 为初始状态就等同于将所有事件都移除了。
    vm._events = Object.create(null);
    return vm;
  }
  // 1-2、vm.$off的第一个参数支持数组，当event为数组的时候，只需要将数组遍历一遍，然后
  // 数组中的每一项依次调用vm.$off;
  if(Array.isArray(event)){
    for(let i = 0 ; i<event.length; i++){
      this.$off(event[i],fn)
    }
    return vm;
  }
  // 2、只提供了事件名，则移除该事件所有的监听器，只需要将vm._events中的event重置为空就行
  const cbs = vm._events[event];
  if(!cbs){
    return vm;
  }
  // 这里做了一个安全检测，如果这个事件没有被监听，vm._events内找不到任何监听器，直接退出
  // 程序，然后判断是否只有一个参数，如果是，将事件名在vm._events中所有事件都移除，只需要
  // 将vm._events上以该事件为属性的值设置为null即可
}
```

```

if(arguments.length === 1){
    vm._events[event] = null;
    return vm;
}

// 3、如果同时提供了事件与回调，那么只移除这个回调的监听器，将使用参数中提供的事件名从
// vm._events取出事件列表，然后从列表中找到与参数中提供的回调函数相同那个函数，并
// 将它从列表中移除

if(fn){
    // 先判断是否有fn参数，有则说明用户同时提供了event和fn两个参数，然后从vm._events
    // 中取出事件监听器列表并遍历它，如果列表中的某一项与fn相同，或者某一项的fn属性与fn相同，
    // 使用splice方法将它从列表中移除，当循环结束后，列表中所有与用户参数中提供的fn相同的监听器
    // 都会被移除

    const cbd = vm._events[event];
    let cb;
    let i = cbs.length;
    while(i--){
        // 这里有一个细节要注意，在代码中遍历列表是从后向前循环，这样在列表中移除当前
        // 位置的监听器，不会影响列表中未遍历到的监听器位置，如果是从前向后遍历，那么当从
        // 列表中移除一个监听器时，后面的监听器会自动向前移动一个位置，会导致下一轮循环
        // 时跳过一个元素。
        cb = cbs[i];
        if(cb === fn || cb.fn === fn){
            cbs.splice(i,1);
            break;
        }
    }
    return vm;
}

```

## 2、vm.\$once(event,callback)

参数：

- {string} event
- {Function} callback

用法：监听一个自定义事件，但是只触发一次。一旦触发之后，监听器就会被移除

```

Vue.prototype.$once = function (event, fn) {
    var vm = this;

```

```
function on () {
    vm.$off(event, on);
    fn.apply(vm, arguments);
}
on.fn = fn;
vm.$on(event, on);
return vm
};
```

在`vm.once`中调用`vm.on`来实现监听自定义事件的功能，当自定义事件触发后会执行拦截器，将监听器从事件列表中移除。

在`vm.once`使用`vm.on`来监听事件，首先，将函数`on`注册到事件中，当自定义事件被触发时，会执行函数`on`(在这个函数中，会用时`vm.$off`将自定义事件移除)，然后

手动执行函数，并将参数`arguments`传递给函数`fn`，这样就可以实现`vm.$once`的功能。

但是要注意`on.fn = fn`这行代码，前面介绍`vm.$off`时提到，在移除监听器时，需要将用户提供的监听器函数与列表中的监听器函数进行对比，相同部分会被移除，这导致当我们使用拦截器代替监听器注入到事件列表中时，

拦截器和用户提供的函数时不相同的，此时用户使用`vm.$off`来移除事件监听器，移除操作失败。

这个问题的解决方案是将用户提供的原始监听器保存到拦截器的`fn`属性中，当`vm.$off`方法遍历事件监听器列表时，同时会检查监听器和监听器的`fn`属性是否与用户提供的监听器函数相同，只要有一个相同，就说明需要被移除

的监听器找到了，将被找到的拦截器从监听器列表中移除

本文使用 [mdnice](#) 排版

关注下面的标签，发现更多相似文章

# 学习vue源码（2）手写Vue.extend方法



**Vue.extend( options );**

(1) 参数

{ object } options

(2) 用法

使用基础Vue构造器创建一个“子类”，其参数是一个包含“组件选项”的对象。`data`选项是特例，在`Vue.extend()`中，它必须是函数。

```
<div id="mount-point"></div>
<!-- 创建构造器 --&gt;
var Profile = Vue.extend({
  template: '&lt;p&gt;{{firstName}} {{lastName}} aka{{alias}}&lt;/p&gt;',
  data:function(){
    return{
      name:''
    }
  }
})</pre>
```

```

    firstName:'Walter',
    lastName:'White',
    alias:'Heisenberg'
  }
}

})
<!-- 创建Profile实例，并挂载到一个元素上 -->
new Profile().$mount('#mount-point');

```

(3) 全局API和示例方法不同，后者是在Vue的原型上挂载方法，也就是在Vue.prototype上挂载方法，而前者是直接在Vue上挂载方法。

```

Vue.extend = function(extendOptions){
  <!-- 做点什么 -->
}

```

#### (4) 作用

Vue.extend的作用是创建一个子类，所以可以创建一个子类，然后让它继承Vue身上的一些功能。

## 二、实现（1）创建一个子类

```

Vue.cid = 0
let cid = 1;

Vue.extend = function(extendOptions){
  extendOptions = extendOptions || {};
  const Super = this; // 指向 Vue这个构造函数
  const SuperId = Super.cid; 指向 Vue这个构造函数的id
  const cachedCtors = extendOptions._Ctor || (extendOptions._Ctor = {});
  if(cachedCtors[SuperId]){
    return cachedCtors[SuperId];
  }
  const name = extendOptions.name || Super.options.name;
  if(process.env.NODE_ENV !== 'production'){
    if(!/^[_a-zA-Z][\w-]*$/_.test(name)){
      warn(
        'invalid component name: "'+name+'".Component names'+
        'can only contain alphanumeric characters and the hyphen,'+
      )
    }
  }
}

```

```

    'and must start with a letter.'
)
}
}

const Sub = function VueComponent(options){
  this._init(options);
}

<!-- 缓存构造函数 -->
cachedCtors[SuperId] = Sub;
return sub;
}

```

1、为了性能考慮，在`Vue.extend`方法内增加了缓存策略。反复调用`Vue.extend`其实应该返回同一个结果。即这段代码所示

```

if(cachedCtors[SuperId]){
  return cachedCtors[SuperId];
}

```

那它是怎么判断是不是相同的呢？

我们举个例子。

```

const obj= {
  template: '<p></p>',
  data: function() {
    return {}
  }
}

var Profile = Vue.extend(obj)
var Profile2 = Vue.extend(obj)
console.log(Profile == Profile2) //true

```

如例子所示，只要`obj`相同，那么创建的子类就是同一个。

那怎么判断`obj`即源码中的`extendOptions`是否是同一个对象呢？

这就是这段代码的作用了

```
const Super = this; // 指向 Vue这个构造函数
const SuperId = Super.cid; 指向 Vue这个构造函数的id，值为0
const cachedCtors = extendOptions._Ctor || (extendOptions._Ctor = {});
```

我们给第一次作为参数的extendOptions添加一个属性\_Ctor={}。因此下次我们再用同一个extendOptions来创建子类的时候，这个extendOptions身上就会有\_Ctor这个属性了。然后再赋值

```
cachedCtors[SuperId] = Sub;
```

注意cachedCtors是引用extendOptions.\_Ctor的，也就是说他们是指向同一个对象。这时

```
extendOptions._Ctor = {
  0: Sub
}
```

因此它下次再用同一个extendOptions就会判断\_Ctor是否已经有缓存了没有。

2.对name校验，如果发现name选项不合格，会在开发环境下发出警告。

3.创建子类并将它返回，这一步并没有继承的逻辑，此时子类是不能用的，它还不具备Vue的能力。

## （2）子类继承Vue的能力

首先，将父类的原型继承到子类中

```
Sub.prototype = Object.create(Super.prototype);
Sub.prototype.constructor = Sub;
Sub.cid = cid++;
```

1、为子类添加了cid，它表示每个类的唯一标识。

```
将父类的options选项继承到子类中
Sub.options = mergeOptions(
  Super.options,
  extendOptions
)
Sub['super'] = Super;
```

1、合并了父类选项与子类选项的逻辑，并将父类保存到子类的super属性中。而mergeOptions方法会将两个选项合并为一个新对象。

如果选项中存在props属性，则初始化它

```
if(Sub.options.props){
  initProps(Sub);
}
```

1、初始化props的作用是将key代理到\_props中。例如，vm.name实际上可以访问到的是Sub.prototype.\_props.name。

```
function initProps(Comp){
  const props = Comp.options.props;
  for(const key in props){
    proxy(Comp.prototype, '_props', key)
  }
}
function proxy(target, sourceKey, key){
  sharedPropertyDefinition.get = function proxyGetter(){
    return this[sourceKey][key];
  }
  sharedPropertyDefinition.set = function proxySetter(val){
    this[sourceKey][key] = val;
  }
  Object.defineProperty(target, key, sharedPropertyDefinition);
}
```

如果选项中存在computed，则对它进行初始化

```

if(Sub.options.computed){
  initComputed(Sub);
}

function initComputed (Comp){
  const computed = Comp.options.computed;
  for(const key in computed){
    defineComputed(Comp.prototype,key,computed[key]);
  }
}

```

将父类中存在的属性依次复制到子类中

```

Sub.extend = Super.extend;
Sub.mixin = Super.mixin
Sub.use = Super.use;
// ASSET_TYPES = ['component','directive','filter']

ASSET_TYPES.forEach(function(type){
  Sub[type] = Super[type];
})

if(name){
  Sub.options.components[name] = Sub;
}

Sub.superOptions = Super.options;
Sub.extendOptions = extendOptions;
Sub.sealedOptions = extend({},Sub.options);

```

1、复制到子类中的方法包括extend、mixin、use、component、directive和filter

2、在子类上新增了superOptions、extendOptions和sealedOptions属性

总结：

创建了一个Sub函数并继承了父级。如果直接使用Vue.extend，则Sub继承于Vue构造函数。

### 三、完整代码

```

Vue.cid = 0
let cid = 1;

Vue.extend = function(extendOptions){
  extendOptions = extendOptions || {};

```

```
const Super = this;
const SuperId = Super.cid;
const cachedCtors = extendOptions._Ctor || (extendOptions._Ctor = {});
if(cachedCtors[SuperId]){
  return cachedCtors[SuperId];
}
const name = extendOptions.name || Super.options.name;
if(process.env.NODE_ENV !== 'production'){
  if(!/^[_a-zA-Z][\w-]*$/.test(name)){
    warn(
      'invalid component name: "'+name+'". Component names'+
      'can only contain alphanumeric characters and the hyphen,'+
      'and must start with a letter.'
    )
  }
}
const Sub = function VueComponent(options){
  this._init(options);
}
<!-- 将父类原型继承到子类中 -->
<!-- cid每个类的唯一标识 -->
Sub.prototype = Object.create(Super.prototype);
Sub.prototype.constructor = Sub;
Sub.cid = cid++;
<!-- 将父类的options选项继承到子类中 -->
Sub.options = mergeOptions(
  Super.options,
  extendOptions
)
Sub['super'] = Super;
<!-- 如果选项中存在props属性，则初始化它 -->
if(Sub.options.props){
  initProps(Sub);
}
<!-- 如果选项中存在computed属性，则对它进行初始化 -->
if(Sub.options.computed){
  initComputed(Sub);
}
<!-- 将父类中存在的属性依次复制到子类中 -->
Sub.extend = Super.extend;
Sub.mixin = Super.mixin'
```

```
Sub.use = Super.use;

<!-- ASSET_TYPES = ['component','directive','filter'] -->
ASSET_TYPES.forEach(function(type){
  Sub[type] = Super[type];
})

if(name){
  Sub.options.components[name] = Sub;
}

Sub.superOptions = Super.options;
Sub.extendOptions = extendOptions;
Sub.sealedOptions = extend({},Sub.options);

<!-- 缓存构造函数 -->
cachedCtors[SuperId] = Sub;
return sub;
}
```

本文使用 mdnice 排版

关注下面的标签，发现更多相似

# 学习vue源码（3）手写Vue.directive、 Vue.filter、Vue.component方法



## 一、Vue.directive

Vue.directive(id,[definition]);

1) 参数

```
{ string } id  
{ Function | Object } [ definition ]
```

## (2) 用法

注册或获取全局指令。

```
<!-- 注册 -->
Vue.directive('my-directive',{
  bind:function(){},
  inserted:function(){},
  update:function(){},
  componentUpdated:function(){},
  unbind:function(){},
})
<!-- 注册（指令函数） -->
Vue.directive('my-directive',function(){
  <!-- 这里将会被bind和update调用 -->
})
<!-- getter方法，返回已注册的指令 -->
var myDirective = Vue.directive('my-directive');
```

(3) 除了核心功能默认内置的指令外（v-model和v-show），Vue.js也允许注册自定义指令。虽然代码复用和抽象的主要形式是组件，但是有些情况下，仍然需要对普通DOM元素进行底层操作，这时就会用到自定义指令。

(4) Vue.directive方法的作用是注册或获取全局指令，而不是让指令生效。其区别是注册指令需要做的事是将指令保存在某个位置，而让指令生效是将指令从某个位置拿出来执行它。

## (5) 实现

```
<!-- 用于保存指令的位置 -->
Vue.options = Object.create(null);
Vue.options['directives'] = Object.create(null);

Vue.directive = function(id,definition){
  if(!definition){
    return this.options['directive'][id];
  }else{
    if(typeof definition === 'function'){
      definition = { bind: definition,update: definition};
    }
  }
}
```

```

this.optipns['directive'][id] = definition;

return definition;
}
}

```

1、在Vue构造函数上创建了options属性来存放选项，并在选项上新增了directive方法用于存在指令。

2、Vue.directive方法接受两个参数id和definition，它可以注册或获取指令，这取决于definition参数是否存在。

3、如果definition参数不存在，则使用id从this.options['directive']中读出指令并将它返回。

4、如果definition参数存在，则说明是注册操作，那么进而判断definition参数的类型是否是函数。

5、如果是函数，则默认监听bind和update两个事件，所以代码中将definition函数分别赋值给对象中的bind和update这两个方法，并使用这个对象覆盖definition。

6、如果definition不是函数，则说明它是用户自定义的指令对象，此时不需要做任何操作，直接将用户提供的指令对象保存在this.optipns['directive']上即可。

## 二、Vue.filter

Vue.filter(id,[definition]); (1) 参数

```

{ string } id
{ Function | Object } [definition]

```

(2) 用法

注册或获取全局过滤器

```

<!-- 注册 -->
Vue.filter('my-filter',function(value){
  <!-- 返回处理后的值 -->
})
<!-- getter方法，返回已注册的过滤器 -->
var myFilter = Vue.filter('my-filter');

```

(3) Vue.js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。过滤器应该被添加到Javascript表达式的尾部，由“管道”符号指示

```
<!-- 在双花括号中 -->
{{ message | capitalize }}
<!-- 在v-bind中 -->
<div v-bind:id="rawId | formatId "></div>
```

(4) Vue.filter的作用仅仅是注册或获取全局过滤器。

(5) 实现

```
Vue.options['filters'] = Object.create(null);

Vue.filter = function(id,definition){
  if(!definition){
    return this.options['filters'][id];
  }else{
    this.options['filters'][id] = definition;
    return definition;
}
```

1、在Vue.options中新增了filters属性用于存放过滤器，并在Vue.js上新增了filter方法，它接受两个参数id和definition。

2、Vue.filters方法可以注册或获取过滤器，这取决于definition参数是否存在。

3、如果definition不存在，则使用id从this.options['filters']中读出过滤器并将它返回。

4、如果definition存在，则说明是注册操作，直接将该参数保存到this.options['filters']中。

### 三、Vue.component

Vue.component(id,[definition]);

1) 参数

```
{ string } id
{ Function | Object } [definition]
```

## (2) 用法

注册或获取全局组件。注册组件时，还会自动使用给定的id设置组件的名称。

```
<!-- 注册组件，传入一个扩展过的构造器 -->
Vue.component('my-component',Vue.extend({/*...*/}));

<!-- 注册组件，传入一个选项对象（自动调用Vue.extend） -->
Vue.component('my-component',{/*...*/});

<!-- 获取注册的组件（始终返回构造器） -->
var MyComponent = Vue.component('my-component');
```

Vue.extend前面我们已经讲过，不了解的可以看下这篇文章：[学习vue源码（2）手写Vue.extend方法](#)

## (3) Vue.component只是注册或获取组件

## (4) 原理

```
Vue.options['components'] = Object.create(null);

Vue.component = function(id,definition){
  if(!definition){
    return this.options['components'][id];
  }else{
    if(isPlainObject(definition)){
      definition.name = definition.name || id;
      definition = Vue.extend(definition);
    }
    this.options['components'][id] = definition;
    return definition;
  }
}
```

1、在Vue.options中新增了components属性用于存放组件，并在Vue.js上新增了component方法，它接收两个参数id和definition。

2、Vue.component方法可以注册或获取过滤器，这取决于definition参数是否存在。

3、如果definition不存在，则使用id从this.options['components']中读出组件并将它返回。

4、如果definition存在，则说明是注册操作，那么需要将组件保存到this.options['components']中。

5、由于definition参数支持两种参数，分别是选项对象和构造器，而组件其实是一个构造函数，是使用Vue.extend生成的子类，所以需要将参数definition同一处理成构造器。

6、如果definition参数是Object类型，则调用Vue.extend方法将它变成Vue的子类，使用Vue.component方法注册组件。

7、如果选项对象中没有设置组件名，则自动使用给定的id设置组件的名称。

#### 四、合并Vue.directive、Vue.filter、Vue.component代码

```
Vue.options = Object.create(null);

<!-- ASSET_TYPES = ['component','directive','filter'] -->
ASSET_TYPES.forEach(type=>{
  Vue.options[type+s] = Object.create(null);
})

ASSET_TYPES.forEach(type=>{
  Vue[type] = function(id,definition){
    if(!definition){
      return this.options[type+s][id];
    }else{
      <!-- 组件 -->
      if(type==='component' && isPlainObject(definition)){
        definition.name = definition.name || id;
        definition = Vue.extend(definition);
      }
      <!-- 指令 -->
      if(type==='directive' && typeof definition === 'function'){
        definition = { bind: definition, update: definition};
      }
      this.options['components'][id] = definition;
      return definition;
    }
  }
})
```

文章参考：深入浅出Vue

本文使用 mdnice 排版

# 学习vue源码（4）手写vm.\$mount方法



一、概述 前面的文章 [https://mp.weixin.qq.com/s?\\_biz=MzU5NDM5MDg1Mw==&tempkey=MTA2M19WeHNUZGZ1VHVqRzFZcjZ0S2l0U014cnFwVDRtVkdfQ2dsaTVrOGdEVVU5WjZIUzdnYWR2aklESmoyc3BRMUREanYtQldtMENFSIZQX1lpb091RW1mS3lNYnBGUlhhTzRNUGpUbmh2b3JZVDJoUm5KMIbxbVk5NTlaNVptWnj2SUF1Z0MwT0dMVUc0a2x6alBZS1RlNWU2UjVpVy1pQlpON19ZYXBBfn4%3D&chksm=7e00beb1497737a70f8df5b6cce5222f8c6c35cdec15438766837cf0d8b4602e0b0cef410fb8#rd](https://mp.weixin.qq.com/s?_biz=MzU5NDM5MDg1Mw==&tempkey=MTA2M19WeHNUZGZ1VHVqRzFZcjZ0S2l0U014cnFwVDRtVkdfQ2dsaTVrOGdEVVU5WjZIUzdnYWR2aklESmoyc3BRMUREanYtQldtMENFSIZQX1lpb091RW1mS3lNYnBGUlhhTzRNUGpUbmh2b3JZVDJoUm5KMIbxbVk5NTlaNVptWnj2SUF1Z0MwT0dMVUc0a2x6alBZS1RlNWU2UjVpVy1pQlpON19ZYXBBfn4%3D&chksm=7e00beb1497737a70f8df5b6cce5222f8c6c35cdec15438766837cf0d8b4602e0b0cef410fb8#rd)

中我们谈到用Vue.extend创建出Vue的子类构造函数后，通过new 得到子类的实例，然后通过\$mount挂载到节点，如代码：

```
<div id="mount-point"></div>
<!-- 创建构造器 -->
var Profile = Vue.extend({
  template:'<p>{{firstName}} {{lastName}} aka{{alias}}</p>',
  data:function(){
    return{
      firstName:'Walter',
      lastName:'White',
      alias:'Heisenberg'
    }
  }
})
<!-- 创建Profile实例，并挂载到一个元素上 -->
new Profile().$mount('#mount-point');
```

我们没有讲\$mount方法是怎么实现的，这篇文章就来讲一下

## 二、使用方式

```
vm.$mount( [elementOrSelector] )
```

### (1) 参数

```
{ Element | string } [elementOrSelector]
```

### (2) 返回值

vm，即实例本身。

### (3) 用法

1、如果Vue.js实例在实例化时没有收到el选项，则它处于“未挂载”状态，没有关联的DOM元素。

2、可以使用vm.\$mount手动挂载一个未挂载的实例。

3、如果没有提供elementOrSelector参数，模板将被渲染为文档之外的元素，并且必须使用原生DOM的API把它插入文档中。

4、这个方法返回实例自身，因而可以链式调用其他实例方法。

## (4) 例子

```

var MyComponent = Vue.extend({
  template: '<div>Hello!</div>',
})
<!-- 创建并挂载到#app (会替换#app) -->
new MyComponent().$mount('#app');
<!-- 创建并挂载到#app (会替换#app) -->
new MyComponent().$mount({el:'#app'});
<!-- 创建并挂载到#app (会替换#app) -->
var component = new MyComponent().$mount();
document.getElementById('app').appendChild(component.$el);

```

1、在不同的构建版本中，`vm.$mount`的表现都不一样。其差异主要体现在完整版（`vue.js`）和只包含运行时版本（`vue.runtime.js`）之间。

2、完整版和只包含运行时版本之间的差异在于是否有编译器，而是否有编译器的差异主要在于`vm.$mount`方法的表现形式。

3、在只包含运行时的构建版本中，`vm.$mount`的作用如前面所述。而在完整的构建版本中，`vm.$mount`的作用会稍有不同，它首先会检查`template`或`el`选项所提供的模板是否已经转换成渲染函数（`render`函数）。如果没有，则立即进入编译过程，将模板编译成渲染函数，完成之后再进入挂载与渲染的流程中。

4、只包含运行时版本的`vm.$mount`没有编译步骤，它会默认实例上已经存在渲染函数，如果不存在，则会设置一个。并且，这个渲染函数在执行时会返回一个空节点的VNode，以保证执行时不会因为函数不存在而报错。同时如果是开发环境下运行，`Vue.js`会触发警告，提示我们当前使用的是只包含运行时的版本，会让我们提供渲染函数，或者去使用完整的构建版本。

5、从原理的角度来讲，完整版和只包含运行时版本之间是包含关系，完整版包含只包含运行时版本。

## 三、完整版`vm.$mount`的实现原理

### (1) 实现代码

```

const mount = Vue.prototype.$mount;
Vue.prototype.$mount = function(el){
  <!-- 做些什么 -->
}

```

```

    return mount.call(this,el);
}

```

1、将Vue原型上的\$mount方法保存在mount中，以便后续使用。

2、然后Vue原型上的\$mount方法被一个新的方法覆盖了。新方法中会调用原始的方法，这种做法通常被称为函数劫持。（看源码的同学可能发现了，vue多处用了函数劫持的做法，例如：对数组实现监听的时候...）

3、通过函数劫持，可以在原始功能上新增一些其他功能。上面代码中，vm.\$mount的原始方法就是mount的核心功能，而在完整版中需要将编译功能新增到核心功能上去。

(2) 由于el参数支持元素类型或者字符串类型的选择器，所以第一步是通过el获取DOM元素。

```

const mount = Vue.prototype.$mount;
Vue.prototype.$mount = function(el){
  el = el && query(el);
  return mount.call(this,el);
}

```

使用query获取DOM元素

```

function query(el){
  if(typeof el === 'string'){
    const selected = document.querySelector(el);
    if(!selected){
      return document.createElement('div');
    }
    return selected;
  }else{
    return el;
  }
}

```

1、如果el是字符串，则使用document.querySelector获取DOM元素，如果获取不到，则创建一个空的div元素。

2、如果el不是字符串，那么认为它是元素类型，直接返回el（如果执行vm.\$mount方法时没有传递el参数，则返回undefined）

### (3) 编译器

1、首先判断Vue.js实例中是否存在渲染函数，只有不存在时，才会将模板编译成渲染函数。

```
const mount = Vue.prototype.$mount;
Vue.prototype.$mount = function(el){
  el = el && query(el);
  const options = this.$options;
  if(!options.render){
    <!-- 将模板编译成渲染函数并赋值给options.render -->
  }
  return mount.call(this,el);
}
```

2、在实例化Vue.js时，会有一个初始化流程，其中会向Vue.js实例上新增一些方法，这里的this.\$options就是其中之一，它可以访问到实例化Vue.js时用户设置的一些参数，例如tempalte和render。

3、如果在实例化Vue.js时给出了render选项，那么template其实是无效的，因为不会进入模板编译的流程，而是直接使用render选项中提供的渲染函数。

4、Vue.js在官方文档的template选项中也给出了相应的提示。如果没有render选项，那么需要获取模板并将模板编译成渲染函数（render函数）赋值给render选项。

```
const mount = Vue.prototype.$mount;
Vue.prototype.$mount = function(el){
  el = el && query(el);
  const options = this.$options;
  if(!options.render){
    <!-- 新增获取模板相关逻辑 -->
    let template = options.template;
    if(template){
      }else if(el){
        template = getOuterHTML(el);
      }
    }
  return mount.call(this,el);
}
```

5、从选项中取出template选项，也就是取出用户实例化Vue.js时设置的模板。如果没有取到，说明用户没有设置tempalte选项。那么使用getOuterHTML方法从用户提供的el选项中获取模板。

```
function getOuterHTML(el){
  if(el.outerHTML){
    return el.outerHTML;
  }else{
    const container = document.createElement('div');
    container.appendChild(el.cloneNode(true));
    return container.innerHTML;
  }
}
```

6、getOuterHTML方法会返回参数中提供的DOM元素的HTML字符串。

## 7、整体逻辑

如果用户没有通过template选项设置模板，那么会从el选项中获取HTML字符串当作模板。如果用户提供了template选项，那么需要对它进一步解析，因为这个选项支持很多种使用方式。template选项可以直接设置成字符串模板，也可以设置为以#开头的选择符，还可以设置成DOM元素。

## 8、从不同的格式中将模板解析出来

```
const mount = Vue.prototype.$mount;
Vue.prototype.$mount = function(el){
  el = el && query(el);
  const options = this.$options;
  if(!options.render){
    <!-- 新增获取模板相关逻辑 -->
    let template = options.template;
    if(template){
      if(typeof tempalte === 'string'){
        if(tempalte.charAt(0) === "#"){
          template = idToTemplate(tempalte);
        }
      }else if(tempalte.nodeType){
        template = template.innerHTML;
      }else{
        if(process.env.NODE_ENV !== 'production'){

```

```

        warn('invalid template option:' + tempalte, this);
    }

    return this;
}

} else if(el){
    template = getOuterHTML(el);
}

}

return mount.call(this, el);
}

```

9、如果tempalte是字符串并且以#开头，则它将被用作选择符。通过选择符获取DOM元素后，会使用innerHTML作为模板。

10、使用idToTemplate方法从选择符中获取模板。idToTemplate使用选择符获取DOM元素之后，将它的innerHTML作为模板。

```

function idToTemplate(id){
    const el = query(id);
    return el && el.innerHTML;
}

```

11、如果template是字符串，但不是以#开头，就说明template是用户设置的模板，不需要进行任何处理，直接使用即可。

12、如果template选项的类型不是字符串，则判断它是否是一个DOM元素，如果是，则使用DOM元素的innerHTML作为模板。如果不是，只需要判断它是否具备nodeType属性即可。

13、如果tempalte选项既不是字符串，也不是DOM元素，那么Vue.js会触发警告，提示用户template选项是无效的。

14、获取模板之后，下一步是将模板编译成渲染函数，通过执行compileToFunctions函数可以将模板编译成渲染函数并设置到this.options上。

```

const mount = Vue.prototype.$mount;
Vue.prototype.$mount = function(el){
    el = el && query(el);
    const options = this.$options;
    if(!options.render){

```

```

<!-- 新增获取模板相关逻辑 -->
let template = options.template;
if(template){
  if(typeof tempalte === 'string'){
    if(tempalte.charAt(0) === "#"){
      template = idToTemplate(tempalte);
    }
  }else if(tempalte.nodeType){
    template = template.innerHTML;
  }else{
    if(process.env.NODE_ENV !== 'production'){
      warn('invalid template option:' +tempalte,this);
    }
    return this;
  }
}else if(el){
  template = getOuterHTML(el);
}
<!-- 新增编译相关逻辑 -->
if(tempalte){
  const { render } = compileToFunctions(
    template,
    {...},
    this
  )
  options.render = render;
}
return mount.call(this,el);
}

```

15、将模板编译成代码字符串并将代码字符串转换成渲染函数的过程是在compileToFunctions函数中完成的，其内部实现如下

```

function compileToFunctions(template,options,vm){
  options = extend({},options);
  <!-- 检查缓存 -->
  const key = options.delimiters
  ? String(options.delimiters)+tempalte
  :template;
  if(cache[key]){
    return cache[key];
  }
}

```

```

}

<!-- 编译 -->

const compiled = compile(template,options);

<!-- 将代码字符串转换为函数 -->

const res = {};

res.render = createFunction(compiled.render);

return (cache[key] = res)

}

function createFunction(code){

return new Function(code);

}

```

- 1) 首先，将options属性混合到空对象中，其目的是让options称为可选参数。
- 2) 检查缓存中是否已经存在编译后的模板。如果模板已经被编译，就会直接返回缓存中的结果，不会重复编译，保证不做无用功来提升性能。
- 3) 调用compile函数来编译模板，将模板编译成代码字符串并存储在compiled中的render属性中。
- 4) 调用createFunction函数将代码字符串转换成函数。其实现原理很简单，使用new Function(code)就可以完成。
- 5) 在代码字符串被new Function(code)转换成函数之后，当调用函数时，代码字符串会被执行。例如

```

const code = 'console.log("Hello Berwin")';

const render = new Function(code);

render(); //Hello Berwin

```

- 6) 最后，将渲染函数返回给调用方。
- 16、当通过compileToFunctions函数得到渲染函数之后，将渲染函数设置到this.\$options上。

#### 四、只包含运行时版本的vm.\$mount的实现原理

- (1) 只包含运行时版本的vm.**mount**方法包含了**vm.mount**方法的核心功能。实现如下

```
Vue.prototype.$mount = function(el){
  el = el && inBrowser ? query(el) : undefined;
  return mountComponent(this,el);
}
```

1、\$mount方法将ID转换为DOM元素后，使用mountComponent函数将Vue.js实例挂载到DOM元素上。

2、将实例挂载到DOM元素上指的是将模板渲染到指定的DOM元素中，而且是持续性的，以后当数据（状态）发生变化时，依然可以渲染到指定的DOM元素中。

3、实现这个功能需要开启watcher。

watcher将持续观察模板中用到的所有数据（状态），当这些数据（状态）被修改时它将得到通知，从而进行渲染操作。这个过程会持续到实例被销毁。

```
export function mountComponent(vm,el){
  if(!vm.$options.render){
    vm.$options.render = createEmptyVNode;
    if(process.env.NODE_ENV !== 'production'){
      <!-- 在开发环境发出警告 -->
    }
  }
}
```

4、mountComponent方法会判断实例上是否存在渲染函数。如果不存在，则设置一个默认的渲染函数createEmptyVNode，该渲染函数执行后，会返回一个注释类型的VNode节点。

5、事实上，如果在mountComponent方法中发现实例上没有渲染函数，则会将el参数指定页面中的元素节点替换成一个注释节点，并且在开发环境下在浏览器的控制台中给出警告。

（2）Vue.js实例在不同的阶段会触发不同的生命周期钩子，在挂载实例之前会触发beforeMount钩子函数。

```
export function mountComponent(vm,el){
  if(!vm.$options.render){
    vm.$options.render = createEmptyVNode;
    if(process.env.NODE_ENV !== 'production'){
      <!-- 在开发环境发出警告 -->
    }
  }
}
```

```

    callHook(vm, 'beforeMount')
}
}
}

```

1、钩子函数触发后，将执行真正的挂载操作。挂载操作与渲染类似，不同的是渲染指的是渲染一次，而挂载指的是持续性渲染。挂载之后，每当状态发生变化时，都会进行渲染操作。

### （3）mountComponent具体实现

```

export function mountComponent(vm,el){
  if(!vm.$options.render){
    vm.$options.render = createEmptyVNode;
    if(process.env.NODE_ENV !== 'production'){
      <!-- 在开发环境发出警告 -->
    }
    <!-- 触发生命周期钩子 -->
    callHook(vm, 'beforeMount');

    <!-- 挂载 -->
    vm._watcher = new Watcher(vm, ()=>{
      vm._update(vm._render())
    }, noop);

    <!-- 触发生命周期钩子 -->
    callHook(vm, 'mounted');
  }
  return vm;
}
}

```

1、vm.\_update作用：调用虚拟DOM中的patch方法来执行节点的比对与渲染操作。

2、vm.\_render作用：执行渲染函数，得到一份新的VNode节点树。

3、vm.\_update(vm.\_render())作用：先调用渲染函数得到一份最新的VNode节点树，然后通过vm.\_update方法对最新的VNode和上一次渲染用到的旧VNode进行对比并更新DOM节点。简单来说，就是执行了渲染操作。

### （4）挂载是持续性的，而持续性的关键就在于new Watcher这行代码。

1、Watcher的第二个参数支持函数，并且当它是函数时，会同时观察函数中所读取的所有Vue.js实例上的响应式数据。

2、当watcher执行函数时，函数中所读取的数据都将会触发getter去全局找到watcher并将其收集到函数的依赖列表中。即，函数中读取的所有数据都将被watcher观察。这些数据中的任何一个发生变化时，watcher都将得到通知。

3、当数据发生变化时，watcher会一次又一次地执行函数进入渲染流程，如此反复，这个过程会持续到实例被销毁。

4、挂载完毕后，会触发mounted钩子函数。

如果不懂watcher，其实可以去掉看，就简单很多

```
export function mountComponent(vm,el){
  if(!vm.$options.render){
    vm.$options.render = createEmptyVNode;
    if(process.env.NODE_ENV !== 'production'){
      <!-- 在开发环境发出警告 -->
    }
    <!-- 触发生命周期钩子 -->
    callHook(vm,'beforeMount');
    <!-- 挂载 -->

    vm._update(vm._render())
    <!-- 触发生命周期钩子 -->
    callHook(vm,'mounted');
  }
  return vm;
}
```

这样，是不是很容易理解了。整个 `mountComponent`，一句关键代码：`vm._update(vm._render())`，表示通过执行`vm._render()`得到VNode，再把VNode传入 `vm._update()`，`vm._update()`得功能是将传入的VNode 变成真实Dom渲染到页面。

简单地总结一下：

`$mount()`的思路就是，判断用户传入的option有没有`render`函数，

1.有的话就走运行时版本，

2.没有的话就自动生成`render`函数，然后在执行运行时版本（其实这就是编译时版本，比运行时版本多了异步生成`render`函数的步骤）。

执行运行时版本的时候，

1. 通过render()获得Vnode
2. 把Vnode传入\_update() 实现渲染

本文使用 mdn nice 排版

# 学习vue源码（5）手写Vue.use、Vue.mixin、Vue.compile



微博 @张娜拉

## 一、Vue.use

Vue.use(plugin);

### （1）参数

```
{ Object | Function } plugin
```

### （2）用法

安装Vue.js插件。如果插件是一个对象，必须提供install方法。如果插件是一个函数，它会被作为install方法。调用install方法时，会将Vue作为参数传入。install方法被同一个插件多次调用时，插件也只会被安装一次。

关于如何上开发Vue插件，请看这篇文章，非常简单，不用两分钟就看完：[如何开发 Vue 插件？](#)

### （3）作用

注册插件，此时只需要调用install方法并将Vue作为参数传入即可。但在细节上有两部分逻辑要处理：

- 1、插件的类型，可以是install方法，也可以是一个包含install方法的对象。

- 2、插件只能被安装一次，保证插件列表中不能有重复的插件。

#### （4）实现

```
Vue.use = function(plugin){
  const installedPlugins = (this._installedPlugins || (this._installedPlugins = []));
  if(installedPlugins.indexOf(plugin)>-1){
    return this;
  }
  <!-- 其他参数 -->
  const args = toArray(arguments,1);
  args.unshift(this);
  if(typeof plugin.install === 'function'){
    plugin.install.apply(plugin,args);
  }else if(typeof plugin === 'function'){
    plugin.apply(null,plugin,args);
  }
  installedPlugins.push(plugin);
  return this;
}
```

- 1、在Vue.js上新增了use方法，并接收一个参数plugin。

- 2、首先判断插件是不是已经别注册过，如果被注册过，则直接终止方法执行，此时只需要使用indexOf方法即可。

- 3、toArray方法我们在学习vue源码（1）手写与事件相关的实例方法已经提到过，就是将类数组转成真正的数组。使用toArray方法得到arguments。除了第一个参数之外，剩余的所有参数将得到的列表赋值给args，然后将Vue添加到args列表的最前面。这样做的目的是保证install方法被执行时第一个参数是Vue，其余参数是注册插件时传入的参数。

- 4、由于plugin参数支持对象和函数类型，所以通过判断plugin.install和plugin哪个是函数，即可知用户使用哪种方式注册的插件，然后执行用户编写的插件并将args作为参数传入。

- 5、最后，将插件添加到installedPlugins中，保证相同的插件不会反复被注册。

## 二、Vue.mixin

```
Vue.mixin(mixin);
```

### （1）参数

```
{ Object } mixin
```

### （2）用法

1、全局注册一个混入（mixin），影响之后创建的每个Vue.js实例。

2、插件作者可以使用混入向组件注入自定义行为（例如：监听生命周期钩子）。不推荐在应用代码中使用。

```
Vue.mixin({
  created:function(){
    var myOption = this.$options.myOption;
    if(myOption){
      console.log(myOption);
    }
  }
})

new Vue({
  myOption:'hello!'
})
// => "hello!"
```

（3）Vue.mixin方法注册后，会影响之后创建的每个Vue.js实例，因为该方法会更改Vue.options属性。

### （4）实现

```
import { mergeOptions } from '../util/index'

export function initMixin(Vue){
  Vue.mixin = function(mixinin){
```

```

this.options = mergeOptions(this.options,mixin);
return this;
}
}

```

1、mergeOptions会将用户传入的mixin与this.options合并成一个新对象，然后将这个生成的新对象覆盖this.options属性，这里的this.options其实就是Vue.options。mergeOptions的具体实现，我们后面再讲。

2、因为mixin方法修改了Vue.options属性，而之后创建的每个实例都会用到该属性，所以会影响创建的每个实例。

### 三、Vue.compile

```
Vue.compile(template);
```

#### （1）参数

```
{ string } template
```

#### （2）用法

编译模板字符串并返回包含渲染函数的对象。只在完整版中才有效。

```

var res = Vue.compile('<div><span>{{msg}}</span></div>');
new Vue({
  data:{
    msg:'hello'
  },
  render:res.render
})

```

（3）并不是所有Vue.js的构建版本都存在Vue.compile方法。与vm.\$mount类似，Vue.compile方法只存在于完整版中。（只有完整版包含编译器）

#### （4）实现

Vue.compile方法只需要调用编译器就可以实现功能。

```
Vue.compile = compileToFunctions;
```

compileToFunctions方法可以将模板编译成渲染函数。

compileToFunctions 函数，我们在学习 vue 源码（4）手写 vm.\$mount 方法已经讲到了，这里将不再赘述。

本文使用 mdn nice 排版

# 习vue源码（6）熟悉模板编译原理



前面我们在学习vue源码（4）手写vm.\$mount方法中谈到 compile 函数把模版转换成 渲染函数，

```
function compileToFunctions(template,options,vm){  
    options = extend({},options);  
    <!-- 检查缓存 -->  
    const key = options.delimiters  
    ? String(options.delimiters)+template  
    :template;  
    if(cache[key]){  
        return cache[key];  
    }  
    <!-- 编译 -->  
    const compiled = compile(template,options);  
    <!-- 将代码字符串转换为函数 -->  
    const res = {};  
    res.render = createFunction(compiled.render);  
    return (cache[key] = res)  
}  
function createFunction(code){  
    return new Function(code);  
}
```

但是没有谈到具体实现，这一次我们来具体实现。

先大概谈下模板编译原理。

## 一、模版编译原理概述

(1) 在Vue.js中创建HTML并不是只有模板这一种途径。既可以手动写渲染函数来创建HTML，也可以在Vue.js中使用JSX来创建HTML。

(2) 渲染函数是创建HTML最原始的方法。

(3) 模板最终会通过编译转换成渲染函数，渲染函数执行后，会得到一份vnode用于虚拟DOM渲染。所以模板编译其实是配合虚拟DOM进行渲染。

(4) 模板编译所介绍的内容是如何让虚拟DOM拿到vnode。（模板--> 模板编译--> 渲染函数（模板编译）--> vnode--> 用户界面（虚拟DOM））

(5) Vue.js提供了模板语法，允许声明式地描述状态和DOM之间地绑定关系，然后通过模板来生成真实DOM并将其呈现在用户界面上。

（6）在底层实现上，Vue.js会将模板编译成虚拟DOM渲染函数。当应用内部地状态发生变化时，Vue.js可以结合响应式系统，聪明地找出最小数量地组件进行重新渲染以及最少量地进行DOM操作。

## 二、模版编译原理概念

（1）平时使用模板时，可以在模板中使用一些变量来填充模板，还可以在模板中使用Javascript表达式，又或者是使用一些指令等。这些功能在HTML语法中是不存在的，这多亏了模板编译赋予了模板强大的功能。

（2）模板编译的主要目标就是生成渲染函数。而渲染函数的作用是每次执行它，它就会使用当前最新的状态生成一份新的vnode，然后使用这个vnode进行渲染。

## 三、将模板编译成渲染函数

（1）模板编译分三部分内容

1、将模板解析为AST。（Abstract Syntax Tree，抽象语法树）。

2、遍历AST标记静态节点。

3、使用AST生成渲染函数。

（2）由于静态节点不需要总是重新渲染，所以在生成AST之后、生成渲染函数之前这个节点，需要做一个操作，那就说遍历一遍AST，给所有静态节点做一个标记，这样在虚拟DOM中更新节点时，如果发现节点有这个标记，就不会重新渲染它。

（3）这三部分内容在模板编译中分别抽象出三个模块来实现各自的功能，分别是

1、解析器。

2、优化器。

3、代码生成器。

## 四、解析器

（1）作用：将模板解析成AST。

（2）在解析器内部，分成了很多小解析器，其中包括过滤器解析器、文本解析器和HTML解析器。然后通过一条主线将这些解析器组装在一起。

（3）在使用模板时，我们可以在其中使用过滤器，而过滤器解析器的作用就说用来解析过滤器的。

(4) 文本解析器就是用来解析文本的。其主要作用是用来解析带变量的文本。不带变量的文本是一段纯文本，不需要使用文本解析器来解析。

```
Hello {{name}}
```

(5) HTML解析器，它是解析器中最核心的模块，它的作用就是解析模板，每当解析到HTML标签的开始位置、结束位置、文本或则注释时，都会触发钩子函数，然后将相关信息通过参数传递进来。

(6) 主线上做的事就是监听HTML解析器。每当触发钩子函数时，就生成一个对应的AST节点。生成AST节点前，会根据类型使用不同的方式生成不同的AST。例如，如果是文本节点，就生成文本类型的AST。

(7) 这个AST其实和vnode有点类似，都是使用Javascript中的对象来表示节点。

(8) 当HTML解析器把所有模板都解析完毕后，AST也就生成好了。

五、优化器 (1) 目标：遍历AST，检测出所有静态子树（永远都不会发生变化的DOM节点）并给其打标记。

(2) 当AST中的静态子树被打上标记后，每次重新渲染时，就不需要为打上标记的静态节点创建新的虚拟节点，而是直接克隆已存在的虚拟节点。

(3) 在虚拟DOM的更新操作中，如果发现两个是同一个节点，正常情况下会对这两个节点进行更新，但是如果这两个节点是静态节点，则可以直接跳过更新节点的流程。

(4) 优化器的主要作用是避免一些无用功来提升性能。因为静态节点出了首次渲染，后续不需要任何重新渲染操作。

## 六、代码生成器

(1) 其是模板编译的最后一步，作用是将AST转换成渲染函数中的内容，这个内容可以称为“代码字符串”。

```
<p title="Berwin" @click="c">1</p>
```

生成后的代码字符串：

```

with(this){
  return _c(
    'p',
    {
      attrs:{'title':"Berwin"},
      on:{'click':c}
    },
    [_v("1")]
  )
}

```

(2) 这样一个代码字符串最终导出到外界使用时，会将代码字符串放到函数里，这个函数叫作渲染函数。

(3) 当渲染函数被导出到外界后，模板编译的任务就完成了。

```

const code = 'with(this){return \'Hello Berwin\'}';
const hello = new Function(code);
hello();
//Hello Berwin

```

(4) 渲染函数的作用是创建vnode。渲染函数之所以可以生成vnode，是因为代码字符串中会有很多函数调用（例如，上面生成的代码字符串中有两个函数调用\_c和\_v），这些函数是虚拟DOM提供的创建vnode的方法。

(5) vnode有很多种类型，不同类型对应不同的创建方法，所以代码字符串中的\_c和\_v其实都是创建vnode的方法，只是创建vnode的类型不同，例如\_c可以创建元素类型的vnode，而\_v可以创建文本类型的vnode。

本文使用 mdnice 排版

关注下面的标签，发现更多相似文章

Vue.js



前端工程师 @ 菜鸟  
获得点赞 1,076 · 获得阅读 49,112

关注

## 安装掘金浏览器插件

# 学习vue源码（7）手写解析器



微博 @张姐啦

通过 [学习vue源码（6）熟悉模板编译原理](#) 的学习，我们知道解析器在整个模板编译中的位置。我们只有将模板解析成AST后，才能基于AST做优化或者生成代码字符串，那么解析器是如何将模板解析成AST的呢？

这次，我们将详细介绍解析器内部的运行原理。

## 1 解析器的作用

解析器要实现的功能是将模板解析成AST。

例如：

```
<div>
  <p>{{name}}</p>
</div>
```

上面的代码是一个比较简单的模板，它转换成AST后的样子如下：

```
{  
  tag: "div"  
  type: 1,  
  staticRoot: false,  
  static: false,  
  plain: true,  
  parent: undefined,  
  attrsList: [],  
  attrsMap: {},  
  children: [  
    {  
      tag: "p"  
      type: 1,  
      staticRoot: false,  
      static: false,  
      plain: true,  
      parent: {tag: "div", ...},  
      attrsList: [],  
      attrsMap: {},  
      children: [{  
        type: 2,  
        text: "{{name}}",  
        static: false,  
        expression: "_s(name)"  
      }]  
    }  
  ]  
}
```

其实AST并不是什么很神奇的东西，不要被它的名字吓倒。它只是用JS中的对象来描述一个节点，一个对象代表一个节点，对象中的属性用来保存节点所需的各种数据。比如，`parent`属性保存了父节点的描述对象，`children`属性是一个数组，里面保存了一些子节点的描述对象。再比如，`type`属性代表一个节点的类型等。当很多个独立的节点通过`parent`属性和`children`属性连在一起时，就变成了一个树，而这样一个用对象描述的节点树其实就是AST。

## 2 解析器内部运行原理

事实上，解析器内部也分了好几个子解析器，比如HTML解析器、文本解析器以及过滤器解析器，其中最主要的是HTML解析器。顾名思义，HTML解析器的作用是解析HTML，它在解析HTML的过程中会不断触发各种钩子函数。这些钩子函数包括开始标签钩子函数、结束标签钩子函数、文本钩子函数以及注释钩子函数。

伪代码如下：

```
parseHTML(template, {
  start (tag, attrs, unary) {
    // 每当解析到标签的开始位置时，触发该函数
  },
  end () {
    // 每当解析到标签的结束位置时，触发该函数
  },
  chars (text) {
    // 每当解析到文本时，触发该函数
  },
  comment (text) {
    // 每当解析到注释时，触发该函数
  }
})
```

你可能不能很清晰地理解，下面我们举个简单的例子：

```
<div><p>我是Berwin</p></div>
```

当上面这个模板被HTML解析器解析时，所触发的钩子函数依次是：`start`、`start`、`chars`、`end`、`end`。

也就是说，解析器其实是从前向后解析的。解析到`<div>`时，会触发一个标签开始的钩子函数`start`；然后解析到`<p>`时，又触发一次钩子函数`start`；接着解析到我是Berwin这行文本，此时触发了文本钩子函数`chars`；然后解析到`</p>`，触发了标签结束的钩子函数`end`；接着继续解析到`</div>`，此时又触发一次标签结束的钩子函数`end`，解析结束。

因此，我们可以在钩子函数中构建AST节点。在`start`钩子函数中构建元素类型的节点，在`chars`钩子函数中构建文本类型的节点，在`comment`钩子函数中构建注释类型的节点。

当HTML解析器不再触发钩子函数时，就代表所有模板都解析完毕，所有类型的节点都在钩子函数中构建完成，即AST构建完成。

我们发现，钩子函数`start`有三个参数，分别是`tag`、`attrs`和`unary`，它们分别代表标签名、标签的属性以及是否是自闭合标签。

而文本节点的钩子函数`chars`和注释节点的钩子函数`comment`都只有一个参数，只有`text`。这是因为构建元素节点时需要知道标签名、属性和自闭合标识，而构建注释节点和文本节点时只需要知

道文本即可。

什么是自闭合标签？举个简单的例子，input标签就属于自闭合标签：

```
<input type="text" />
```

，而div标签就不属于自闭合标签：

```
<div></div>。
```

在start钩子函数中，我们可以使用这三个参数来构建一个元素类型的AST节点，例如：

```
function createASTElement (tag, attrs, parent) {
  return {
    type: 1,
    tag,
    attrsList: attrs,
    parent,
    children: []
  }
}

parseHTML(template, {
  start (tag, attrs, unary) {
    let element = createASTElement(tag, attrs, currentParent)
  }
})
```

在上面的代码中，我们在钩子函数start中构建了一个元素类型的AST节点。

如果是触发了文本的钩子函数，就使用参数中的文本构建一个文本类型的AST节点，例如：

```
parseHTML(template, {
  chars (text) {
    let element = {type: 3, text}
  }
})
```

如果是注释，就构建一个注释类型的AST节点，例如：

```
parseHTML(template, {
  comment (text) {
    let element = {type: 3, text, isComment: true}
  }
})
```

你会发现，看到的AST是有层级关系的，一个AST节点具有父节点和子节点，但是shang介绍的创建节点的方式，节点是被拉平的，没有层级关系。因此，我们需要一套逻辑来实现层级关系，让每一个AST节点都能找到它的父级。下面我们介绍一下如何构建AST层级关系。

构建AST层级关系其实非常简单，我们只需要维护一个栈（stack）即可，用栈来记录层级关系，这个层级关系也可以理解为DOM的深度。

HTML解析器在解析HTML时，是从前向后解析。每当遇到开始标签，就触发钩子函数start。每当遇到结束标签，就会触发钩子函数end。

基于HTML解析器的逻辑，我们可以在每次触发钩子函数start时，把当前构建的节点推入栈中；每当触发钩子函数end时，就从栈中弹出一个节点。

这样就可以保证每当触发钩子函数start时，栈的最后一个节点就是当前正在构建的节点的父节点，如图1所示。

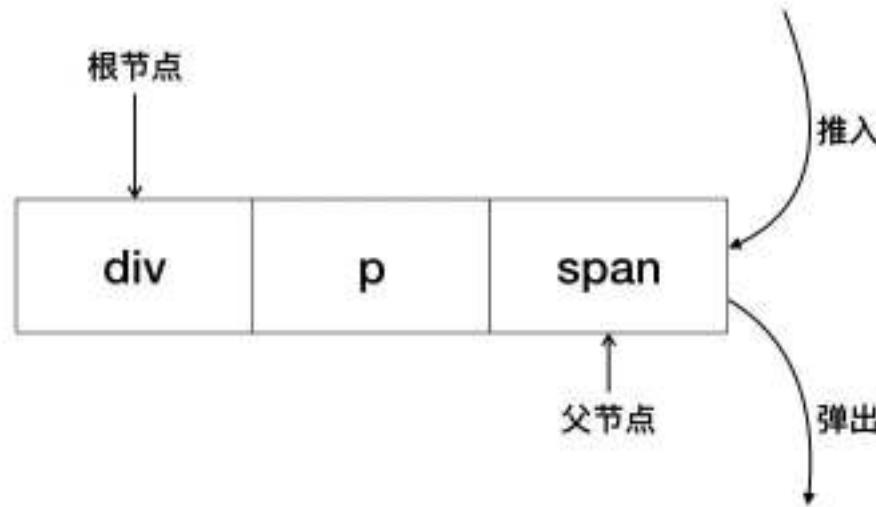


图1 使用栈记录DOM层级关系（英文为代码体）

下面我们用一个具体的例子来描述如何从0到1构建一个带层级关系的AST。

假设有这样一个模板：

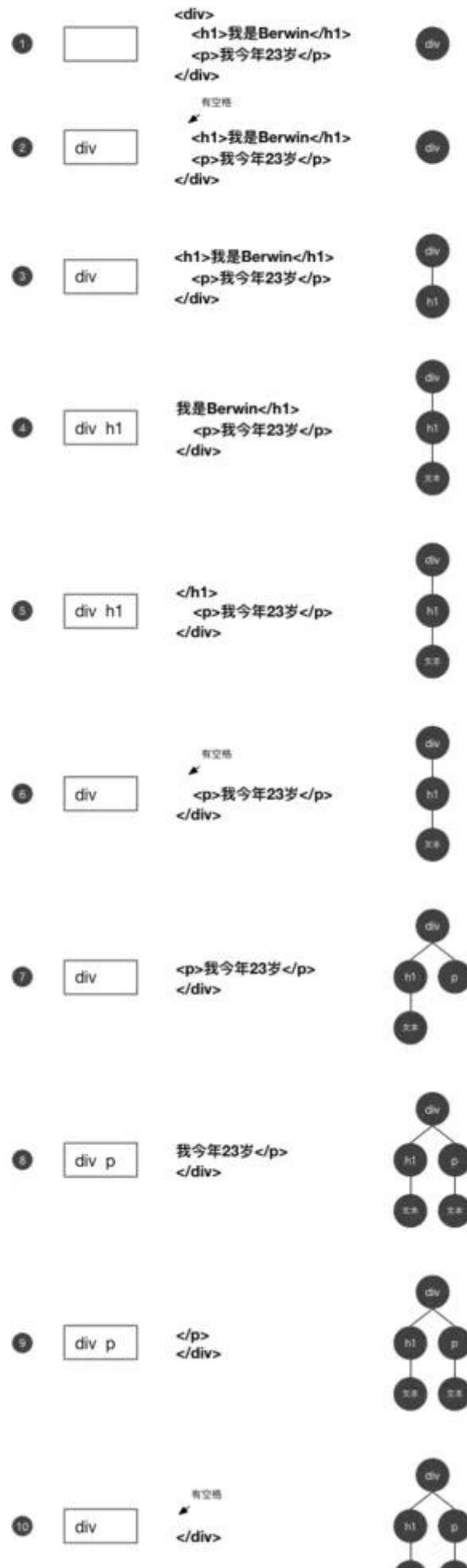
```
<div>
  <h1>我是Berwin</h1>
  <p>我今年23岁</p>
</div>
```

上面这个模板被解析成AST的过程如图9-2所示。

栈

模板

AST



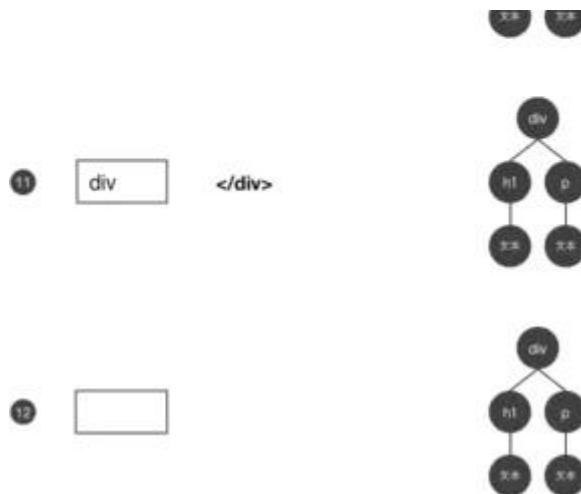


图9-2给出了构建AST的过程，图中的黑底白数字代表解析的步骤，具体如下。

- (1) 模板的开始位置是

的开始标签，于是会触发钩子函数start。start触发后，会先构建一个div节点。此时发现栈是空的，这说明div节点是根节点，因为它没有父节点。最后，将div节点推入栈中，并将模板字符串中的

开始标签从模板中截取掉。
- (2) 这时模板的开始位置是一些空格，这些空格会触发文本节点的钩子函数，在钩子函数里会忽略这些空格。同时会在模板中将这些空格截取掉。
- (3) 这时模板的开始位置是

# 的开始标签，于是会触发钩子函数start。与前面流程一样，start触发后，会先构建一个h1节点。此时发现栈的最后一个节点是div节点，这说明h1节点的父节点是div，于是将h1添加到div的子节点中，并且将h1节点推入栈中，同时从模板中将h1的开始标签截取掉。
- (4) 这时模板的开始位置是一段文本，于是会触发钩子函数chars。chars触发后，会先构建一个文本节点，此时发现栈中的最后一个节点是h1，这说明文本节点的父节点是h1，于是将文本节点添加到h1节点的子节点中。由于文本节点没有子节点，所以文本节点不会被推入栈中。最后，将文本从模板中截取掉。
- (5) 这时模板的开始位置是

# 结束标签，于是会触发钩子函数end。end触发后，会把栈中最后一个节点弹出来。
- (6) 与第(2)步一样，这时模板的开始位置是一些空格，这些空格会触发文本节点的钩子函数，在钩子函数里会忽略这些空格。同时会在模板中将这些空格截取掉。
- (7) 这时模板的开始位置是

开始标签，于是会触发钩子函数start。start触发后，会先构建一个p节点。由于第(5)步已经从栈中弹出了一个节点，所以此时栈中的最后一个节点是div，这说明p节点的父节点是div。于是将p推入div的子节点中，最后将p推入到栈中，并将p的开始标签从模板中截取掉。
- (8) 这时模板的开始位置又是一段文本，于是会触发钩子函数chars。当chars触发后，会先构建一个文本节点，此时发现栈中的最后一个节点是p节点，这说明文本节点的父节点是p节点。于是将文本节点推入p节点的子节点中，并将文本从模板中截取掉。

(9) 这时模板的开始位置是p的结束标签，于是会触发钩子函数end。当end触发后，会从栈中弹出一个节点出来，也就是把p标签从栈中弹出来，并将p的结束标签从模板中截取掉。

(10) 与第(2)步和第(6)步一样，这时模板的开始位置是一些空格，这些空格会触发文本节点的钩子函数并且在钩子函数里会忽略这些空格。同时会在模板中将这些空格截取掉。

(11) 这时模板的开始位置是div的结束标签，于是会触发钩子函数end。其逻辑与之前一样，把栈中的最后一个节点弹出来，也就是把div弹了出来，并将div的结束标签从模板中截取掉。

(12) 这时模板已经被截取空了，也就代表着HTML解析器已经运行完毕。这时我们会发现栈已经空了，但是我们得到了一个完整的带层级关系的AST语法树。这个AST中清晰写明了每个节点的父节点、子节点及其节点类型。

### 3 HTML解析器

通过前面的介绍，我们发现构建AST非常依赖HTML解析器所执行的钩子函数以及钩子函数中所提供的参数，你一定会非常好奇HTML解析器是如何解析模板的，接下来我们会详细介绍HTML解析器的运行原理。

#### 1 运行原理

事实上，解析HTML模板的过程就是循环的过程，简单来说就是用HTML模板字符串来循环，每轮循环都从HTML模板中截取一小段字符串，然后重复以上过程，直到HTML模板被截成一个空字符串时结束循环，解析完毕，如图9-2所示。

在截取一小段字符串时，有可能截取到开始标签，也有可能截取到结束标签，又或者是文本或者注释，我们可以根据截取的字符串的类型来触发不同的钩子函数。

循环HTML模板的伪代码如下：

```
function parseHTML(html, options) {
  while (html) {
    // 截取模板字符串并触发钩子函数
  }
}
```

为了方便理解，我们手动模拟HTML解析器的解析过程。例如，下面这样一个简单的HTML模板：

```
<div>
  <p>{{name}}</p>
</div>
```

它在被HTML解析器解析的过程如下。

最初的HTML模板：

```
`<div>
  <p>{{name}}</p>
</div>`
```

第一轮循环时，截取出一段字符串

，并且触发钩子函数**start**，截取后的结果为：

```
`          `<p>{{name}}</p>
</div>`
```

第二轮循环时，截取出一段字符串：

并且触发钩子函数**chars**，截取后的结果为：

```
`<p>{{name}}</p>
</div>`
```

第三轮循环时，截取出一段字符串

，并且触发钩子函数**start**，截取后的结果为：

```
`{{name}}</p>
</div>`
```

第四轮循环时，截取出一段字符串{{name}}，并且触发钩子函数**chars**，截取后的结果为：

```
`</p>
</div>`
```

第五轮循环时，截取出一段字符串

，并且触发钩子函数end，截取后的结果为：

```
`</div>`
```

第六轮循环时，截取出一段字符串：

```
` `
```

并且触发钩子函数chars，截取后的结果为：

```
`</div>`
```

第七轮循环时，截取出一段字符串

，并且触发钩子函数end，截取后的结果为：

```
..
```

解析完毕。

HTML解析器的全部逻辑都是在循环中执行，循环结束就代表解析结束。接下来，我们要讨论的重点是HTML解析器在循环中都干了些什么事。

你会发现HTML解析器可以很聪明地知道它在每一轮循环中应该截取哪些字符串，那么它是如何做到这一点的呢？

通过前面的例子，我们发现一个很有趣的事，那就是每一轮截取字符串时，都是在整个模板的开始位置截取。我们根据模板开始位置的片段类型，进行不同的截取操作。

例如，上面例子中的第一轮循环：如果是以开始标签开头的模板，就把开始标签截取掉。再例如，上面例子中的第四轮循环：如果是以文本开始的模板，就把文本截取掉。

这些被截取的片段分很多种类型，示例如下。

- 开始标签，例如 `<div>`。
- 结束标签，例如 `</div>`。
- HTML注释，例如 `<!-- 我是注释 -->`。
- DOCTYPE，例如 `<!DOCTYPE html>`。
- 条件注释，例如 `<!--[if !IE]>-->我是注释<!--<![endif]-->`。
- 文本，例如 `我是Berwin`。
- 通常，最常见的是开始标签、结束标签、文本以及注释。

## 2 截取开始标签

上一节中我们说过，每一轮循环都是从模板的最前面截取，所以只有模板以开始标签开头，才需要进行开始标签的截取操作。

那么，如何确定模板是不是以开始标签开头？

在HTML解析器中，想分辨出模板是否以开始标签开头并不难，我们需要先判断HTML模板是不是以`<`开头。

如果HTML模板的第一个字符不是`<`，那么它一定不是以开始标签开头的模板，所以不需要进行开始标签的截取操作。

如果HTML模板以`<`开头，那么说明它至少是一个以标签开头的模板，但这个标签到底是什么类型的标签，还需要进一步确认。

如果模板以`<`开头，那么它有可能是以开始标签开头的模板，同时它也有可能是以结束标签开头的模板，还有可能是注释等其他标签，因为这些类型的片段都以`<`开头。那么，要进一步确定模板是不是以开始标签开头，还需要借助正则表达式来分辨模板的开始位置是否符合开始标签的特征。

那么，如何使用正则表达式来匹配模板以开始标签开头？我们看下面的代码：

```
const ncname = '[a-zA-Z_][\\w\\-\\.]*'
const qnameCapture = `((?:${ncname}\\:)?)?${ncname})`
const startTagOpen = new RegExp(`^<${qnameCapture}`)

// 以开始标签开始的模板
'<div></div>'.match(startTagOpen) // ["<div", "div", index: 0, input: "<div></div>"]

// 以结束标签开始的模板
```

```
'</div><div>我是Berwin</div>'.match(startTagOpen) // null
// 以文本开始的模板
'我是Berwin</p>'.match(startTagOpen) // null
```

通过上面的例子可以看到，只有 '`<div></div>`' 可以成功匹配，而以 `</div>` 开头的或者以文本开头的模板都无法成功匹配。

我们介绍了当HTML解析器解析到标签开始时，会触发钩子函数`start`，同时会给出三个参数，分别是标签名（`tagName`）、属性（`attrs`）以及自闭合标识（`unary`）。

因此，在分辨出模板以开始标签开始之后，需要将标签名、属性以及自闭合标识解析出来。

在分辨模板是否以开始标签开始时，就可以得到标签名，而属性和自闭合标识则需要进一步解析。

当完成上面的解析后，我们可以得到这样一个数据结构：

```
const start = '<div></div>'.match(startTagOpen)
if (start) {
  const match = {
    tagName: start[1],
    attrs: []
  }
}
```

这里有一个细节很重要：在前面的例子中，我们匹配到的开始标签并不全。例如：

```
const ncname = '[a-zA-Z_][\\w\\-\\.]*'
const qnameCapture = `((?:${ncname}\\:)?)?${ncname})`
const startTagOpen = new RegExp(`^<${qnameCapture}>`)

'<div></div>'.match(startTagOpen)
// [<div>, </div>, index: 0, input: "<div></div>"]

'<p></p>'.match(startTagOpen)
// [<p>, </p>, index: 0, input: "<p></p>"]
```

```
'<div class="box"></div>'.match(startTagOpen)  
// ["<div", "div", index: 0, input: "<div class='box'></div>"]
```

可以看出，上面这个正则表达式虽然可以分辨出模板是否以开始标签开头，但是它的匹配规则并不是匹配整个开始标签，而是开始标签的一小部分。

事实上，开始标签被拆分成三个小部分，分别是标签名、属性和结尾，如图3所示。



图3 开始标签被拆分成三个小部分（代码用代码体）

通过“标签名”这一段字符，就可以分辨出模板是否以开始标签开头，此后要想得到属性和自闭合标识，则需要进一步解析。**1. 解析标签属性** 在分辨模板是否以开始标签开头时，会将开始标签中的标签名这一小部分截取掉，因此在解析标签属性时，我们得到的模板是下面伪代码中的样子：

```
' class="box"></div>'
```

通常，标签属性是可选的，一个标签的属性有可能存在，也有可能不存在，所以需要判断标签是否存在属性，如果存在，对它进行截取。

下面的伪代码展示了如何解析开始标签中的属性，但是它只能解析一个属性：

```
const attribute = /^\\s*([\\s\"'<>\\/=]+)(?:\\s*(=)\\s*(?:\"([^\"]*)\"|'([^\']*)'|([\\s\"'=\\s\"'<>`]+)))?/
let html = ' class="box"></div>'
let attr = html.match(attribute)
html = html.substring(attr[0].length)
console.log(attr)
// [' class="box"', 'class', '=', 'box', undefined, undefined, index: 0, input: ' class="box"></div>'
```



如果标签上有很多属性，那么上面的处理方式就不足以支撑解析任务的正常运行。例如下面的代码：

```
const attribute = /^\\s*([\\s\"'<>\\/=]+)(?:\\s*(=)\\s*(?:\"([^\"]*)\"|'([^\']*)'|([\\s\"'=\\s\"'<>`]+)))?/
let html = ' class="box" id="el"></div>'
let attr = html.match(attribute)
html = html.substring(attr[0].length)
console.log(attr)
// [' class="box"', 'class', '=', 'box', undefined, undefined, index: 0, input: ' class="box" id="el"'
```

可以看到，这里只解析出了class属性，而id属性没有解析出来。

此时剩余的HTML模板是这样的：

```
' id="el"></div>'
```

所以属性也可以分成多个小部分，一小部分一小部分去解析与截取。

解决这个问题时，我们只需要每解析一个属性就截取一个属性。如果截取完后，剩下的HTML模板依然符合标签属性的正则表达式，那么说明还有剩余的属性需要处理，此时就重复执行前面的流程，直到剩余的模板不存在属性，也就是剩余的模板不存在符合正则表达式所预设的规则。

例如：

```
const startTagClose = /^\\s*(\\/?)>/
const attribute = /^\\s*([\\s\"'<>\\/=]+)(?:\\s*(=)\\s*(?:\"([^\"]*)\"|'([^\']*)'|([\\s\"'=\\s\"'<>`]+)))?/
```

```

let html = ' class="box" id="el"></div>'

let end, attr

const match = {tagName: 'div', attrs: []}

while (!(end = html.match(startTagClose)) && (attr = html.match(attribute))) {
    html = html.substring(attr[0].length)
    match.attrs.push(attr)
}

```

上面这段代码的意思是，如果剩余HTML模板不符合开始标签结尾部分的特征，并且符合标签属性的特征，那么进入到循环中进行解析与截取操作。

通过match方法解析出的结果为：

```
{
  tagName: 'div',
  attrs: [
    [' class="box"', 'class', '=', 'box', null, null],
    [' id="el"', 'id', '=', 'el', null, null]
  ]
}
```

可以看到，标签中的两个属性都已经解析好并且保存在了attrs中。

此时剩余模板是下面的样子：

```
"></div>"
```

我们将属性解析后的模板与解析之前的模板进行对比：

```

// 解析前的模板
' class="box" id="el"></div>'

// 解析后的模板
'></div>'

// 解析前的数据
{
  tagName: 'div',

```

```

    attrs: []
}

// 解析后的数据
{
  tagName: 'div',
  attrs: [
    [' class="box"', 'class', '=', 'box', null, null],
    [' id="el"', 'id', '=', 'el', null, null]
  ]
}

```

可以看到，标签上的所有属性都已经被成功解析出来，并保存在attrs属性中。

## 2. 解析自闭合标识

如果我们接着上面的例子继续解析的话，目前剩余的模板是下面这样的：

```
'></div>'
```

开始标签中结尾部分解析的主要目的是解析出当前这个标签是否是自闭合标签。

举个例子：

```
<div></div>
```

这样的div标签就不是自闭合标签，而下面这样的input标签就属于自闭合标签：

```
<input type="text" />
```

自闭合标签是没有子节点的，所以前文中我们提到构建AST层级时，需要维护一个栈，而一个节点是否需要推入到栈中，可以使用这个自闭合标识来判断。

那么，如何解析开始标签中的结尾部分呢？看下面这段代码：

```

function parseStartTagEnd (html) {
  const startTagClose = /^s*(\/?)>/

```

```

const end = html.match(startTagClose)
const match = {}

if (end) {
  match.unarySlash = end[1]
  html = html.substring(end[0].length)
  return match
}

console.log(parseStartTagEnd('></div>')) // {unarySlash: ""}
console.log(parseStartTagEnd('/><div></div>')) // {unarySlash: "/"}

```

这段代码可以正确解析出开始标签是否是自闭合标签。

从代码中打印出来的结果可以看到，自闭合标签解析后的`unarySlash`属性为`/`，而非自闭合标签为空字符串。

### 3. 实现源码

前面解析开始标签时，我们将其拆解成了三个部分，分别是标签名、属性和结尾。我相信你已经对开始标签的解析有了一个清晰的认识，接下来看一下Vue.js中真实的代码是什么样的：

```

const ncname = '[a-zA-Z_][\\w\\-\\.]*'
const qnameCapture = `((?:${ncname}\\:)?)?${ncname})`
const startTagOpen = new RegExp(`^<${qnameCapture}`)
const startTagClose = `/^\\s*(\\/?)>`
```

```

function advance (n) {
  html = html.substring(n)
}
```

```

function parseStartTag () {
  // 解析标签名，判断模板是否符合开始标签的特征
  const start = html.match(startTagOpen)
  if (start) {
    const match = {
      tagName: start[1],
      attrs: []
    }
    advance(start[0].length)
  }
}
```

```

// 解析标签属性
let end, attr
while (!(end = html.match(startTagClose)) && (attr = html.match(attribute))) {
    advance(attr[0].length)
    match.attrs.push(attr)
}

// 判断是否是自闭合标签
if (end) {
    match.unarySlash = end[1]
    advance(end[0].length)
    return match
}
}
}

```

上面的代码是Vue.js中解析开始标签的源码，这段代码中的html变量是HTML模板。

调用parseStartTag就可以将剩余模板开始部分的开始标签解析出来。如果剩余HTML模板的开始部分不符合开始标签的正则表达式规则，那么调用parseStartTag就会返回undefined。因此，判断剩余模板是否符合开始标签的规则，只需要调用parseStartTag即可。如果调用它后得到了解析结果，那么说明剩余模板的开始部分符合开始标签的规则，此时将解析出来的结果拿出来并调用钩子函数start即可：

```

// 开始标签
const startTagMatch = parseStartTag()
if (startTagMatch) {
    handleStartTag(startTagMatch)
    continue
}

```

前面我们说过，所有解析操作都运行在循环中，所以continue的意思是这一轮的解析工作已经完成，可以进行下一轮解析工作。

从代码中可以看出，如果调用parseStartTag之后有返回值，那么会进行开始标签的处理，其处理逻辑主要在handleStartTag中。这个函数的主要目的就是将tagName、attrs和unary等数据取出来，然后调用钩子函数将这些数据放到参数中。

### 3 截取结束标签

结束标签的截取要比开始标签简单得多，因为它不需要解析什么，只需要分辨出当前是否已经截取到结束标签，如果是，那么触发钩子函数就可以了。

那么，如何分辨模板已经截取到结束标签了呢？其道理其实和开始标签的截取相同。

如果HTML模板的第一个字符不是<，那么一定不是结束标签。只有HTML模板的第一个字符是<时，我们才需要进一步确认它到底是不是结束标签。

进一步确认时，我们只需要判断剩余HTML模板的开始位置是否符合正则表达式中定义的规则即可：

```
const ncname = '[a-zA-Z_][\\w\\-\\.]*'
const qnameCapture = `((?:${ncname}\\:)?)?${ncname})`
const endTag = new RegExp(`^<\\/${qnameCapture}[^>]*`)

const endTagMatch = '</div>'.match(endTag)
const endTagMatch2 = '<div>'.match(endTag)

console.log(endTagMatch) // [</div>, "div", index: 0, input: "</div>"]
console.log(endTagMatch2) // null
```

上面代码可以分辨出剩余模板是否是结束标签。当分辨出结束标签后，需要做两件事，一件事是截取模板，另一件事是触发钩子函数。而Vue.js中相关源码被精简后如下：

```
const endTagMatch = html.match(endTag)
if (endTagMatch) {
  html = html.substring(endTagMatch[0].length)
  options.end(endTagMatch[1])
  continue
}
```

可以看出，先对模板进行截取，然后触发钩子函数。

## 4 截取注释

分辨模板是否已经截取到注释的原理与开始标签和结束标签相同，先判断剩余HTML模板的第一个字符是不是<，如果是，再用正则表达式来进一步匹配：

```

const comment = /^<!--/

if (comment.test(html)) {
  const commentEnd = html.indexOf('-->')

  if (commentEnd >= 0) {
    if (options.shouldKeepComment) {
      options.comment(html.substring(4, commentEnd))
    }
    html = html.substring(commentEnd + 3)
    continue
  }
}

```

在上面的代码中，我们使用正则表达式来判断剩余的模板是否符合注释的规则，如果符合，就将这段注释文本截取出来。

这里有一个有意思的地方，那就是注释的钩子函数可以通过选项来配置，只有`options.shouldKeepComment`为真时，才会触发钩子函数，否则只截取模板，不触发钩子函数。

## 5 截取条件注释

条件注释不需要触发钩子函数，我们只需要把它截取掉就行了。

截取条件注释的原理与截取注释非常相似，如果模板的第一个字符是`<`，并且符合我们事先用正则表达式定义好的规则，就说明需要进行条件注释的截取操作。

在下面的代码中，我们通过`indexOf`找到条件注释结束位置的下标，然后将结束位置前的字符都截取掉：

```

const conditionalComment = /^<!\[/
if (conditionalComment.test(html)) {
  const conditionalEnd = html.indexOf(']>')

  if (conditionalEnd >= 0) {
    html = html.substring(conditionalEnd + 2)
    continue
  }
}

```

我们来举个例子：

```

const conditionalComment = /^<![/
let html = '<![if !IE]><link href="non-ie.css" rel="stylesheet"><![endif]>'
if (conditionalComment.test(html)) {
  const conditionalEnd = html.indexOf(']')
  if (conditionalEnd >= 0) {
    html = html.substring(conditionalEnd + 2)
  }
}

console.log(html) // '<link href="non-ie.css" rel="stylesheet"><![endif]>'

```

从打印结果中可以看到，HTML中的条件注释部分截取掉了。

通过这个逻辑可以发现，在Vue.js中条件注释其实没有用，写了也会被截取掉，通俗一点说就是写了也白写。

## 6 截取DOCTYPE

DOCTYPE与条件注释相同，都是不需要触发钩子函数的，只需要将匹配到的这一段字符截取掉即可。下面的代码将DOCTYPE这段字符匹配出来后，根据它的length属性来决定要截取多长的字符串：

```

const doctype = /^<!DOCTYPE [^>]+>/i
const doctypeMatch = html.match(doctype)
if (doctypeMatch) {
  html = html.substring(doctypeMatch[0].length)
  continue
}

```

示例如下：

```

const doctype = /^<!DOCTYPE [^>]+>/i
let html = '<!DOCTYPE html><html lang="en"><head></head><body></body></html>'
const doctypeMatch = html.match(doctype)
if (doctypeMatch) {
  html = html.substring(doctypeMatch[0].length)
}

```

```
console.log(html) // '<html lang="en"><head></head><body></body></html>'
```

从打印结果可以看到，HTML中的DOCTYPE被成功截取掉了。

## 7 截取文本

若想分辨在本轮循环中HTML模板是否已经截取到文本，其实很简单，我们甚至不需要使用正则表达式。

在前面的其他标签类型中，我们都会判断剩余HTML模板的第一个字符是否是<，如果是，再进一步确认到底是哪种类型。这是因为以<开头的标签类型太多了，如开始标签、结束标签和注释等。然而文本只有一种，如果HTML模板的第一个字符不是<，那么它一定是文本了。

例如：

```
我是文本</div>
```

上面这段HTML模板并不是以<开头的，所以可以断定它是以文本开头的。

那么，如何从模板中将文本解析出来呢？我们只需要找到下一个<在什么位置，这之前的所有字符都属于文本，如图4所示。



图4 尖括号前面的字符都属于文本

在代码中可以这样实现：

```
while (html) {
    let text
    let textEnd = html.indexOf('<')

    // 截取文本
    if (textEnd >= 0) {
        text = html.substring(0, textEnd)
        html = html.substring(textEnd)
    }

    // 如果模板中找不到<, 就说明整个模板都是文本
    if (textEnd < 0) {
        text = html
        html = ''
    }
}
```

```
// 触发钩子函数
if (options.chars && text) {
    options.chars(text)
}
}
```

上面的代码共有三部分逻辑。

第一部分是截取文本，这在前面介绍过了。<之前的所有字符都是文本，直接使用html.substring从模板的最开始位置截取到<之前的位置，就可以将文本截取出来。

第二部分是一个条件：如果在整个模板中都找不到<，那么说明整个模板全是文本。

第三部分是触发钩子函数并将截取出来的文本放到参数中。

关于文本，还有一个特殊情况需要处理：如果<是文本的一部分，该如何处理？

举个例子：

```
1<2</div>
```

在上面这样的模板中，如果只截取第一个<前面的字符，最后被截取出来的将只有1，而不能把所有文本都截取出来。

那么，该如何解决这个问题呢？

有一个思路是，如果将<前面的字符截取完之后，剩余的模板不符合任何需要被解析的片段的类型，就说明这个<是文本的一部分。

什么是需要被解析的片段的类型？我们说过HTML解析器是一段一段截取模板的，而被截取的每一段都符合某种类型，这些类型包括开始标签、结束标签和注释等。

说的再具体一点，那就是上面这段代码中的1被截取完之后，剩余模板是下面的样子：

<2 <2符合开始标签的特征么？不符合。

<2符合结束标签的特征么？不符合。

<2符合注释的特征么？不符合。

当剩余的模板什么都不符合时，就说明<属于文本的一部分。

当判断出<是属于文本的一部分后，我们需要做的事情是找到下一个<并将其前面的文本截取出来加到前面截取了一半的文本后面。

这里还用上面的例子，第二个<之前的字符是<2，那么把<2截取出来后，追加到上一次截取出来的1的后面，此时的结果是：

```
1<2
```

截取后剩余的模板是：

```
</div>
```

如果剩余的模板依然不符合任何被解析的类型，那么重复此过程。直到所有文本都解析完。

说完了思路，我们看一下具体的实现，伪代码如下：

```
while (html) {
    let text, rest, next
    let textEnd = html.indexOf('<')

    // 截取文本
    if (textEnd >= 0) {
        rest = html.slice(textEnd)
        while (
            !endTag.test(rest) &&
            !startTagOpen.test(rest) &&
            !comment.test(rest) &&
            !conditionalComment.test(rest)
        ) {
            // 如果'<'在纯文本中，将它视为纯文本对待
            next = rest.indexOf('<', 1)
            if (next < 0) break
            textEnd += next
            rest = html.slice(textEnd)
        }
        text = html.substring(0, textEnd)
        html = html.substring(textEnd)
    }

    // 如果模板中找不到<，那么说明整个模板都是文本
    if (textEnd < 0) {
        text = html
    }
}
```

```

    html = ''
}

// 触发钩子函数

if (options.chars && text) {
  options.chars(text)
}
}
}

```

在代码中，我们通过while来解决这个问题（注意是里面的while）。如果剩余的模板不符合任何被解析的类型，那么重复解析文本，直到剩余模板符合被解析的类型为止。

在上面的代码中，endTag、startTagOpen、comment和conditionalComment都是正则表达式，分别匹配结束标签、开始标签、注释和条件注释。

在Vue.js源码中，截取文本的逻辑和其他的实现思路一致。

## 8 纯文本内容元素的处理

什么是纯文本内容元素呢？script、style和textarea这三种元素叫作纯文本内容元素。解析它们的时候，会把这三种标签内包含的所有内容都当作文本处理。那么，具体该如何处理呢？

前面介绍开始标签、结束标签、文本、注释的截取时，其实都是默认当前需要截取的元素的父级元素不是纯文本内容元素。事实上，如果要截取元素的父级元素是纯文本内容元素的话，处理逻辑将完全不一样。

事实上，在while循环中，最外层的判断条件就是父级元素是不是纯文本内容元素。例如下面的伪代码：

```

while (html) {
  if (!lastTag || !isPlainTextElement(lastTag)) {
    // 父元素为正常元素的处理逻辑
  } else {
    // 父元素为script、style、textarea的处理逻辑
  }
}

```

在上面的代码中，lastTag代表父元素。可以看到，在while中，首先进行判断，如果父元素不存在或者不是纯文本内容元素，那么进行正常的处理逻辑，也就是前面介绍的逻辑。

而当父元素是script这种纯文本内容元素时，会进入到else这个语句里面。由于纯文本内容元素都被视作文本处理，所以我们的处理逻辑就变得很简单，只需要把这些文本截取出来并触发钩子函

数**chars**，然后再将结束标签截取出来并触发钩子函数**end**。

也就是说，如果父标签是纯文本内容元素，那么本轮循环会一次性将这个父标签给处理完毕。

伪代码如下：

```

while (html) {
  if (!lastTag || !isPlainTextElement(lastTag)) {
    // 父元素为正常元素的处理逻辑
  } else {
    // 父元素为script、style、textarea的处理逻辑
    const stackedTag = lastTag.toLowerCase()
    const reStackedTag = reCache[stackedTag] || (reCache[stackedTag] = new RegExp('([\s\S]*?)(<')
    const rest = html.replace(reStackedTag, function (all, text) {
      if (options.chars) {
        options.chars(text)
      }
      return ''
    })
    html = rest
    options.end(stackedTag)
  }
}

```

上面代码中的正则表达式可以匹配结束标签前包括结束标签自身在内的所有文本。

我们可以给**replace**方法的第二个参数传递一个函数。在这个函数中，我们得到了参数**text**（代表结束标签前的所有内容），触发了钩子函数**chars**并把**text**放到钩子函数的参数中传出去。最后，返回了一个空字符串，代表将匹配到的内容都截掉了。注意，这里的截掉会将内容和结束标签一起截取掉。

最后，调用钩子函数**end**并将标签名放到参数中传出去，代表本轮循环中的所有逻辑都已处理完毕。

假如我们现在有这样一个模板：

```

<div id="el">
  <script>console.log(1)</script>
</div>

```

当解析到script中的内容时，模板是下面的样子：

```
console.log(1)</script>
</div>
```

此时父元素为script，所以会进入到else中的逻辑进行处理。在其处理过程中，会触发钩子函数chars和end。

钩子函数chars的参数为script中的所有内容，本例中大概是下面的样子：

```
chars('console.log(1)')
```

钩子函数end的参数为标签名，本例中是script。

处理后的剩余模板如下：

```
</div>
```

## 9 使用栈维护DOM层级

通过前面几节的介绍，你一定会感到很奇怪，如何知道父元素是谁？

在前面几节中，我们并没有介绍HTML解析器内部其实也有一个栈来维护DOM层级关系，其逻辑与：就是每解析到开始标签，就向栈中推进去一个；每解析到标签结束，就弹出来一个。因此，想取到父元素并不难，只需要拿到栈中的最后一项即可。

同时，HTML解析器中的栈还有另一个作用，它可以检测出HTML标签是否正确闭合。例如：

```
<div><p></div>
```

在上面的代码中，p标签忘记写结束标签，那么当HTML解析器解析到div的结束标签时，栈顶的元素却是p标签。这个时候从栈顶向栈底循环找到div标签，在找到div标签之前遇到的所有其他标签都是忘记了闭合的标签，而Vue.js会在非生产环境下在控制台打印警告提示。

关于使用栈来维护DOM层级关系的具体实现思路

## 10 整体逻辑

前面我们把开始标签、结束标签、注释、文本、纯文本内容元素等的截取方式拆分开，单独进行了详细介绍。本节中，我们就来介绍如何将这些解析方式组装起来完成HTML解析器的功能。

首先，HTML解析器是一个函数。HTML解析器最终的目的是实现这样的功能：

```
parseHTML(template, {
  start (tag, attrs, unary) {
    // 每当解析到标签的开始位置时，触发该函数
  },
  end () {
    // 每当解析到标签的结束位置时，触发该函数
  },
  chars (text) {
    // 每当解析到文本时，触发该函数
  },
  comment (text) {
    // 每当解析到注释时，触发该函数
  }
})
```

所以HTML解析器在实现上肯定是一个函数，它有两个参数——模板和选项：

```
export function parseHTML (html, options) {
  // 做点什么
}
```

我们的模板是一小段一小段去截取与解析的，所以需要一个循环来不断截取，直到全部截取完毕：

```
export function parseHTML (html, options) {
  while (html) {
    // 做点什么
  }
}
```

在循环中，首先要判断父元素是不是纯文本内容元素，因为不同类型父节点的解析方式将完全不同：

```

export function parseHTML (html, options) {
  while (html) {
    if (!lastTag || !isPlainTextElement(lastTag)) {
      // 父元素为正常元素的处理逻辑
    } else {
      // 父元素为script、style、textarea的处理逻辑
    }
  }
}

```

在上面的代码中，我们发现这里已经把整体逻辑分成了两部分，一部分是父标签是正常标签的逻辑，另一部分是父标签是script、style、textarea这种纯文本内容元素的逻辑。

如果父标签为正常的元素，那么有几种情况需要分别处理，比如需要分辨出当前要解析的一小段模板到底是什么类型。是开始标签？还是结束标签？又或者是文本？

我们把所有需要处理的情况都列出来，有下面几种情况：

- 文本
- 注释
- 条件注释
- DOCTYPE
- 结束标签
- 开始标签

我们会发现，在这些需要处理的类型中，除了文本之外，其他都是以标签形式存在的，而标签是以<开头的。

所以逻辑就很清晰了，我们先根据<来判断需要解析的字符是文本还是其他的：

```

export function parseHTML (html, options) {
  while (html) {
    if (!lastTag || !isPlainTextElement(lastTag)) {
      let textEnd = html.indexOf('<')
      if (textEnd === 0) {
        // 做点什么
      }
    }
  }
}

```

```
let text, rest, next

if (textEnd >= 0) {
    // 解析文本
}

if (textEnd < 0) {
    text = html
    html = ''
}

if (options.chars && text) {
    options.chars(text)
}

} else {
    // 父元素为script、style、textarea的处理逻辑
}
}
```

在上面的代码中，我们可以通过<来分辨是否需要进行文本解析。

如果通过<分辨出即将解析的这一小部分字符不是文本而是标签类，那么标签类有那么多类型，我们需要进一步分辨具体是哪种类型：

```
export function parseHTML (html, options) {
    while (html) {
        if (!lastTag || !isPlainTextElement(lastTag)) {
            let textEnd = html.indexOf('<')

            if (textEnd === 0) {
                // 注释
                if (comment.test(html)) {
                    // 注释的处理逻辑
                    continue
                }
            }

            // 条件注释
            if (conditionalComment.test(html)) {
                // 条件注释的处理逻辑
                continue
            }
        }
    }
}
```

```
// DOCTYPE
const doctypeMatch = html.match(doctype)
if (doctypeMatch) {
    // DOCTYPE的处理逻辑
    continue
}

// 结束标签
const endTagMatch = html.match(endTag)
if (endTagMatch) {
    // 结束标签的处理逻辑
    continue
}

// 开始标签
const startTagMatch = parseStartTag()
if (startTagMatch) {
    // 开始标签的处理逻辑
    continue
}

let text, rest, next
if (textEnd >= 0) {
    // 解析文本
}

if (textEnd < 0) {
    text = html
    html = ''
}

if (options.chars && text) {
    options.chars(text)
}
} else {
    // 父元素为script、style、textarea的处理逻辑
}
}
```

关于不同类型的具体处理方式，前面已经详细介绍过，这里不再重复。

## 4 文本解析器

文本解析器的作用是解析文本。你可能会觉得很奇怪，文本不是在HTML解析器中被解析出来了么？准确地说，文本解析器是对HTML解析器解析出来的文本进行二次加工。为什么要进行二次加工？

文本其实分两种类型，一种是纯文本，另一种是带变量的文本。例如下面这样的文本是纯文本：

```
Hello Berwin
```

而下面这样的是带变量的文本：

```
Hello {{name}}
```

在Vue.js模板中，我们可以使用变量来填充模板。而HTML解析器在解析文本时，并不会区分文本是否是带变量的文本。如果是纯文本，不需要进行任何处理；但如果是带变量的文本，那么需要使用文本解析器进一步解析。因为带变量的文本在使用虚拟DOM进行渲染时，需要将变量替换成变量中的值。

我们介绍过，每当HTML解析器解析到文本时，都会触发`chars`函数，并且从参数中得到解析出的文本。在`chars`函数中，我们需要构建文本类型的AST，并将它添加到父节点的`children`属性中。

而在构建文本类型的AST时，纯文本和带变量的文本是不同的处理方式。如果是带变量的文本，我们需要借助文本解析器对它进行二次加工，其代码如下：

```
parseHTML(template, {
  start (tag, attrs, unary) {
    // 每当解析到标签的开始位置时，触发该函数
  },
  end () {
    // 每当解析到标签的结束位置时，触发该函数
  },
  chars (text) {
    text = text.trim()
    if (text) {
      const children = currentParent.children
      let expression
```

```
        if (expression = parseText(text)) {
            children.push({
                type: 2,
                expression,
                text
            })
        } else {
            children.push({
                type: 3,
                text
            })
        }
    }
},
comment (text) {
    // 每当解析到注释时，触发该函数
}
})
```

在chars函数中，如果执行parseText后有返回结果，则说明文本是带变量的文本，并且已经通过文本解析器（parseText）二次加工，此时构建一个带变量的文本类型的AST并将其添加到父节点的children属性中。否则，就直接构建一个普通的文本节点并将其添加到父节点的children属性中。而代码中的currentParent是当前节点的父节点，也就是前面介绍的栈中的最后一个节点。

假设`chars`函数被触发后，我们得到的`text`是一个带变量的文本：

"Hello {{name}}"

这个带变量的文本被文本解析器解析之后，得到的expression变量是这样的：

```
"Hello "+_s(name)
```

上面代码中的`_s`其实是下面这个`toString`函数的别名：

```
function toString (val) {  
    return val == null  
        ? ''  
        : typeof val === 'object'  
            ? JSON.stringify(val, null, 2)
```

```
: String(val)
}
```

假设当前上下文中有一个变量name，其值为Berwin，那么expression中的内容被执行时，它的内容是不是就是Hello Berwin了？

我们举个例子：

```
var obj = {name: 'Berwin'}
with(obj) {
    function toString (val) {
        return val == null
            ? ''
            : typeof val === 'object'
                ? JSON.stringify(val, null, 2)
                : String(val)
    }
    console.log("Hello "+toString(name)) // "Hello Berwin"
}
```

在上面的代码中，我们打印出来的结果是"Hello Berwin"。

事实上，最终AST会转换成代码字符串放在with中执行，接着，我们详细介绍如何加工文本，也就是文本解析器的内部实现原理。

在文本解析器中，第一步要做的事情就是使用正则表达式来判断文本是否是带变量的文本，也就是检查文本中是否包含{{xxx}}这样的语法。如果是纯文本，则直接返回undefined；如果是带变量的文本，再进行二次加工。所以我们的代码是这样的：

```
function parseText (text) {
    const tagRE = /\{\{((?:.|\\n)+?)\}\}/g
    if (!tagRE(text)) {
        return
    }
}
```

在上面的代码中，如果是纯文本，则直接返回。如果是带变量的文本，该如何处理呢？

一个解决思路是使用正则表达式匹配出文本中的变量，先把变量左边的文本添加到数组中，然后把变量改成\_s(x)这样的形式也添加到数组中。如果变量后面还有变量，则重复以上动作，直到所有变量都添加到数组中。如果最后一个变量的后面有文本，就将它添加到数组中。

这时我们其实已经有一个数组，数组元素的顺序和文本的顺序是一致的，此时将这些数组元素用`+`连起来变成字符串，就可以得到最终想要的效果，如图9-5所示。

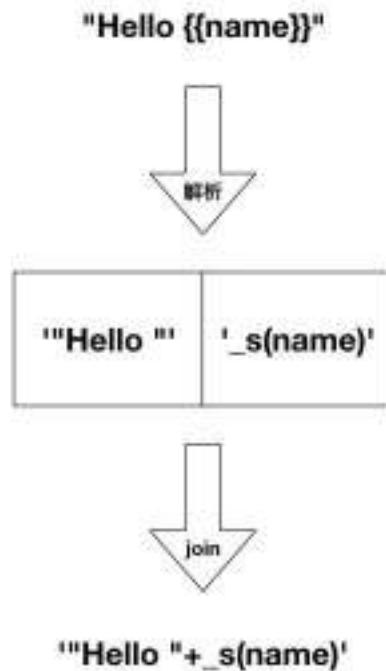


图5 文本解析过程

在图5中，最上面的字符串代表即将解析的文本，中间两个方块代表数组中的两个元素。最后，使用数组方法`join`将这两个元素合并成一个字符串。

具体实现代码如下：

```
function parseText (text) {  
  const tagRE = /\{\{((?:.|\\n)+?)\}\}/g  
  if (!tagRE.test(text)) {  
    return  
  }  
  
  const tokens = []  
  let lastIndex = tagRE.lastIndex = 0  
  let match, index  
  while ((match = tagRE.exec(text))) {  
    index = match.index  
  }  
}
```

```

// 先把 {{ 前边的文本添加到tokens中
if (index > lastIndex) {
  tokens.push(JSON.stringify(text.slice(lastIndex, index)))
}

// 把变量改成 `_s(x)` 这样的形式也添加到数组中
tokens.push(`_s(${match[1].trim()})`)

// 设置lastIndex来保证下一轮循环时，正则表达式不再重复匹配已经解析过的文本
lastIndex = index + match[0].length
}

// 当所有变量都处理完毕后，如果最后一个变量右边还有文本，就将文本添加到数组中
if (lastIndex < text.length) {
  tokens.push(JSON.stringify(text.slice(lastIndex)))
}

return tokens.join('+')
}

```

这是文本解析器的全部代码，代码并不多，逻辑也不是很复杂。

这段代码有一个很关键的地方在lastIndex：每处理完一个变量后，会重新设置lastIndex的位置，这样可以保证如果后面还有其他变量，那么在下一轮循环时可以从lastIndex的位置开始向后匹配，而lastIndex之前的文本将不再被匹配。

下面用文本解析器解析不同的文本看看：

```

parseText('你好{{name}}')
// '"你好 "+_s(name)'

parseText('你好Berwin')
// undefined

parseText('你好{{name}}， 你今年已经{{age}}岁啦')
// '"你好"+_s(name)+"， 你今年已经"+_s(age)+"岁啦"'

```

从上面代码的打印结果可以看到，文本已经被正确解析了。

## 5 总结

解析器的作用是通过模板得到AST（抽象语法树）。

生成AST的过程需要借助HTML解析器，当HTML解析器触发不同的钩子函数时，我们可以构建出不同的节点。

随后，我们可以通过栈来得到当前正在构建的节点的父节点，然后将构建出的节点添加到父节点的下面。

最终，当HTML解析器运行完毕后，我们就可以得到一个完整的带DOM层级关系的AST。

HTML解析器的内部原理是一小段一小段地截取模板字符串，每截取一小段字符串，就会根据截取出来的字符串类型触发不同的钩子函数，直到模板字符串截空停止运行。

文本分两种类型，不带变量的纯文本和带变量的文本，后者需要使用文本解析器进行二次加工。

本文使用 mdnice 排版

## 关注下面的标签，发现更多相似文章

Vue.js



阳光是sunny 

前端工程师 @ 菜鸟

获得点赞 1,076 · 获得阅读 49,103

关注

## 安装掘金浏览器插件

打开新标签页发现好内容，掘金、GitHub、Dribbble、ProductHunt 等站点内容轻松获取。快来安装掘金浏览器插件获取高质量内容吧！

阳光是sunny Lv3

2020年06月06日 阅读 142

关于作者

# 学习vue源码（8）手写优化器



前面的 学习 vue 源码（6）熟悉模板编译原理 我们谈到 模板编译分为解析器和代码生成器。

在 学习 vue 源码（7）手写解析器 里我们已经学会了解析器怎么实现，现在就来看看 优化器怎么实现吧。

优化器的目标是找出那些静态节点并打上标记

优化器的实现原理主要分两步：

第一步：用递归的方式将所有节点添加 static 属性，标识是不是静态节点

第二步：标记所有静态根节点

而静态节点指的是 DOM 不需要发生变化的节点，例如：

复制代码

```
<p>我是静态节点，我不需要发生变化</p>
```

阳光是sunny Lv3

前端工程师 @ 菜鸟

获得点赞 1,076

文章被阅读 49,106

相关文章

手写 webpack 核心原理，再也不怕面试官问我 webpack 原理

1,506 点赞 53 评论

手写 Promise 核心原理，再也不怕面试官问我 Promise 原理

1,201 点赞 20 评论

手写 Vue-router 核心原理，再也不怕面试官问我 Vue-router 原理

1,201 点赞 19 评论

手写 axios 核心原理，再也不怕面试官问我 axios 原理

1,111 点赞 7 评论

手写 Vuex 核心原理，再也不怕面试官问我 Vuex 原理

80 点赞 17 评论

落实到AST中，静态节点指的是static属性为true的节点，例如

```
{  
  type:1,  
  tag:'p',  
  staticRoot:false,  
  static:true,  
  ....  
}
```

标记静态节点有两个好处：

1. 每次重新渲染的时候不需要为静态节点创建新节点
2. 在Virtual DOM中patching的过程可以被跳过

什么是静态根节点？

答：子节点全是静态节点的节点就是静态根节点，例如：

```
<ul>  
  <li>我是静态节点，我不需要发生变化</li>  
  <li>我是静态节点2，我不需要发生变化</li>  
  <li>我是静态节点3，我不需要发生变化</li>  
</ul>
```

**ul**就是静态根节点。

落实到AST中，静态根节点指的是staticRoot属性为true的节点，例如

```
{  
  type:1,  
  tag:'ul',  
  staticRoot:true,  
  static:true,  
  ....  
}
```

上面说到：

优化器的实现原理主要分两步：

第一步：用递归的方式将所有节点添加 static 属性，标识是不是静态节点

第二步：标记所有静态根节点

源码中是这样实现的：

```
function optimize(root, options) {  
  
    if (!root) return  
  
    // first pass: mark all non-static nodes.  
    markStatic(root);  
  
    // second pass: mark static roots.  
    markStaticRoots(root);  
}  

```

现在先看看第一步：

如何将所有节点标记 static 属性？

vue 判断一个节点是不是静态节点的做法其实并不难：

1. 先根据自身是不是静态节点做一个标记 `node.static = isStatic(node)`
2. 然后在循环 `children`，如果 `children` 中出现了哪怕一个节点不是静态节点，在将当前节点的标记修改成 `false`：`node.static = false`。

如代码所示：

```
function markStatic (node) {  
    node.static = isStatic(node);  
    if (node.type === 1) {  
        for (var i = 0, l = node.children.length; i < l; i++) {  
            var child = node.children[i];  
            markStatic(child);  
  
        }  
    }  
}
```

如何判断一个节点是不是静态节点？

也就是说 `isStatic` 这个函数是如何判断静态节点的？

原创作者

```
function isStatic (node: ASTNode): boolean {
  if (node.type === 2) { // expression
    return false
  }
  if (node.type === 3) { // text
    return true
  }
  return !(node.pre || (
    !node.hasBindings && // no dynamic bindings
    !node.if && !node.for && // not v-if or v-for or v-else
    !isBuiltInTag(node.tag) && // not a built-in
    isPlatformReservedTag(node.tag) && // not a component
    !isDirectChildOfTemplateFor(node) &&
    Object.keys(node).every(isStaticKey)
  ))
}
```

先解释一下，在上文讲的解析器中将 模板字符串 解析成 AST 的时候，会根据不同的文本类型设置一个 `type`：

所以上面 `isStatic` 中的逻辑很明显，如果 `type === 2` 那肯定不是 静态节点 返回 `false`，如果 `type === 3` 那就是静态节点，返回 `true`。

那如果 `type === 1`，就有点复杂了，元素节点判断是不是静态节点的条件很多，咱们先一个个看。

首先如果 `node.pre` 为 `true` 直接认为当前节点是静态节点，（`v-pre`是vue的一个指令）

其次 `node.hasBindings` 不能为 `true`。

`node.hasBindings` 属性是在解析器转换 AST 时设置的，如果当前节点的 `attrs` 中，有 `v-`、`@`、`:`开头的 attr，就会把 `node.hasBindings` 设置为 `true`。

```
const dirRE = /^v-|@|^:/  
if (dirRE.test(attr)) {  
  // mark element as dynamic  
  el.hasBindings = true  
}
```

并且元素节点不能有 **if** 和 **for**属性。

`node.if` 和 `node.for` 也是在解析器转换 AST 时设置的。

在解析的时候发现节点使用了 `v-if`，就会在解析的时候给当前节点设置一个 `if` 属性。

就是说元素节点不能使用 `v-if` `v-for` `v-else` 等指令。

并且元素节点不能是 **slot** 和 **component**。

并且元素节点不能是组件。

例如：

```
<List></List>
```

不能是上面这样的自定义组件

并且元素节点的父级节点不能是带 **v-for** 的 **template**

并且元素节点上不能出现额外的属性。

额外的属性指的是不能出现 `type` `tag` `attrsList` `attrsMap` `plain` `parent` `children` `attrs` `staticClass` `staticStyle` 这几个属性之外的其他属性，如果出现其他属性则认为当前节点不是静态节点。

只有符合上面所有条件的节点才会被认为是静态节点。

不过有个问题：递归是从上到下一次标记的，如果父节点被标记为静态节点，而递归到后面的过程中子节点被标记为动态节点，那么就会有矛盾，因此需要在子节点打上标记后，重新给父节点打标记，如代码所示

```
function markStatic (node) {
  node.static = isStatic(node);
  if (node.type === 1) {
    for (var i = 0, l = node.children.length; i < l; i++) {
      var child = node.children[i];
      markStatic(child);

      if (!child.static) {
        node.static = false;
      }
    }
  }
}
```

好了，现在来谈谈优化器的第二步：

如何标记静态根节点？

标记静态根节点其实也是递归的过程。

思路跟第一步类似，有一点不同的就是 标记静态根节点 时当判断此节点是静态根节点就不会往下走了，直接return；

vue 中的实现大概是这样的：

```
function markStaticRoots (node: ASTNode, isInFor: boolean) {
  if (node.type === 1) {
    // For a node to qualify as a static root, it should have children that
    // are not just static text. Otherwise the cost of hoisting out will
    // outweigh the benefits and it's better off to just always render it fresh.
    if (node.static && node.children.length && !(

      node.children.length === 1 &&
      node.children[0].type === 3
    )) {
      node.staticRoot = true
      return
    } else {
      node.staticRoot = false
    }
    if (node.children) {
      for (let i = 0, l = node.children.length; i < l; i++) {
        markStaticRoots(node.children[i], isInFor || !node.for)
      }
    }
  }
}
```

```
    }  
}  
}  
}
```

这段代码其实就一个意思：

有两种特殊的情况，不会被标记为静态根节点

- 根节点只有一个文本节点
- 一个没有子节点的静态节点

这两种情况，优化成本大于收益。

当前节点是静态节点，并且有子节点，并且子节点不是单个静态文本节点这种情况会将当前节点标记为根静态节点。

额，，可能有点绕口，重新解释下。

上面我们标记 静态节点 的时候有一段逻辑是只有所有 子节点 都是 静态节点，当前节点才是真正的 静态节点。

所以这里我们如果发现一个节点是 静态节点，那就能证明它的所有 子节点 也都是静态节点，而我们要标记的是 静态根节点，所以如果一个静态节点只包含了一个文本节点那就不会被标记为 静态根节点。

其实这么做也是为了性能考虑，vue 在注释中也说了，如果把一个只包含静态文本的节点标记为根节点，那么它的成本会超过收益~

总结一下

整体逻辑其实就是递归 AST 这棵树，然后将 静态节点 和 静态根节点 找到并打上标记。

本文使用 [mdnice](#) 排版

关注下面的标签，发现更多相似文章

Vue.js

阳光是sunny Lv1 前端工程师 @ 菜鸟  
发布了 73 篇专栏 · 获得点赞 1,076 · 获得阅读 49,106

关注

安装掘金浏览器插件



打开新标签页发现好内容，掘金、GitHub、Dribbble、ProductHunt 等站点内容轻松获取。快来安装掘金浏览器插件获取高质量内容吧！

输入评论...

## 相关推荐

技术胖 · 1天前 · TypeScript / Vue.js

**技术胖的 TypeScript 免费视频图文教程 ( 2W 字 )**

 360  72

全栈然叔 · 4天前 · Vue.js

**Vue3.0全球发布会干货总结**

 169  26

童欧巴 · 6小时前 · Vue.js / 前端

**海贼王 One Piece，一起康康Vue版本号中的彩蛋**

 8  1

小小洋 · 1天前 · Vue.js

**你要知道的vue面试题汇总**

 96  5

我是十三 · 5天前 · Vue.js / 面试

**Vue 3.0 来了，我们该做些什么？**

 342  58

大前端晨曦 · 1天前 · Vue.js

**Vue中实现输入框Input输入限制**

 29  7

前端Up主 · 3天前 · Vue.js

**源码分析 | 透过表象看本质，Vue3来了、看看里面到底有什么**

 81  17



有赞前端 · 4天前 · Vue.js

## Vant 3.0 Beta 版本发布 🎉

154 42

axuebin · 6天前 · Vue.js / 前端

## 尤大 3 天前发在 GitHub 上的 vue-lit 是啥？

251 45

前端森林 · 7天前 · Vue.js / 前端框架

## 那个男人 他带着Vue3来了~

124 53

李白不吃茶v · 5天前 · Vue.js

## Vue3.0 不畏惧祖传代码的 Composition API

155 32

Gopal · 12天前 · Vue.js

## 一个合格的中级前端工程师应该掌握的 20 个 Vue 技巧

748 55

doodlewind · 16天前 · Vue.js / 面试

## 他写出了 Vue，却做不对这十道 Vue 笔试题

553 191

TinssonTai · 2天前 · Vue.js

## Vue3拥抱TypeScript的正确姿势

27 14

Younglina · 4天前 · Vue.js

## Vue 源码中的一些辅助函数

59 8

zeka · 8天前 · Vue.js / 前端

## 或许这就是下一代组件库

110 33



Shenfq · 18天前 · Vue.js / Webpack

## 面向未来的前端构建工具-vite

608 69

qjuliang · 21小时前 · Vue.js

(记录) vue、element表格首行跑到最后一行去了

2 2

大海我来了 · 7天前 · Vue.js

## 34条我能告诉你的Vue之实操篇

169 24

南方小菜 · 2天前 · Vue.js

## Vue|思路篇|编译ast

7



# 学习vue源码（9）手写代码生成器



前面的学习vue源码（6）熟悉模板编译原理我们谈到，模板编译分为解析器，优化器，代码生成器。

前面两部分已经实现，现在就来看看代码生成器怎么实现吧。

代码生成器的作用是使用 AST 生成 render 函数代码字符串。

解析器主要干的事是将 模板字符串 转换成 element ASTs，例如：

```
<div>
  <p>{{name}}</p>
</div>
```

上面这样一个简单的 模板 转换成 AST 后是这样的：

```
{
  tag: "div",
  type: 1,
  staticRoot: false,
  static: false,
  plain: true,
  parent: undefined,
  attrsList: [],
```

```

  attrsMap: {},
  children: [
    {
      tag: "p"
      type: 1,
      staticRoot: false,
      static: false,
      plain: true,
      parent: {tag: "div", ...},
      attrsList: [],
      attrsMap: {},
      children: [{{
        type: 2,
        text: "{{name}}",
        static: false,
        expression: "_s(name)"
      }}]
    }
  ]
}

```

使用例子中的模板生成后的 AST 来生成 render 后是这样的：

```
{
  render: `with(this){return _c('div',[_c('p',[_v(_s(name))])])}`
}
```

格式化后是这样的：

```

with(this){
  return _c(
    'div',
    [
      _c(
        'p',
        [
          _v(_s(name))
        ]
      )
    ]
  )
}

```

```
    ]  
  )  
}
```

生成后的代码字符串中看到了有几个函数调用 `_c`，`_v`，`_s`。

`_c` 对应的是 `createElement`，它的作用是创建一个元素。

- 第一个参数是一个HTML标签名
- 第二个参数是元素上使用的属性所对应的数据对象，可选项
- 第三个参数是 `children`

例如：

一个简单的模板：

```
<p title="Berwin" @click="c">1</p>
```

生成后的代码字符串是：

```
`with(this){return _c('p',{attrs:{'title':'Berwin'},on:{'click':c}},[_v("1")])}`
```

格式化后：

```
with(this){  
  return _c(  
    'p',  
    {  
      attrs:{'title':'Berwin'},  
      on:{'click':c}  
    },  
    [_v("1")]  
  )  
}
```

`_v` 的意思是创建一个文本节点。

`_s` 是返回参数中的字符串。

可能有同学觉得这个格式化后的代码很陌生，其实把with去掉后，就很熟悉了（其实with是用来改变作用域的，去掉也不会影响我们的理解）

```
return _c(
  'p',
  {
    attrs:{'title':'Berwin'},
    on:{'click':c}
  },
  [_v("1")]
)
```

有没有发现这样很熟悉？没错我们去看vue官网的 render渲染函数

[外链图片转存失败,源站可能有防盗链机制,建议将图片保存下来直接上传 (img-sJtun0qJ-1591176301693) (<https://imgkr.cn-bj.ufileos.com/681bf1eb-bc26-429d-8ede-52ddfd99f2e8.png>)]

有没有发现，其实生成的代码字符串就是 vue官网中介绍的render函数里的 createElement函数。

而参数也是对应的

- 第一个参数：标签名
- 第二个参数：节点数据
- 第三个参数：子节点数组

代码生成器的总体逻辑其实就是使用 element ASTs 去递归，然后拼出这样的 `_c('div',[_c('p',[_v(_s(name))])])` 字符串。

那如何拼这个字符串呢？？

请看下面的代码：

```
function genElement (el: ASTElement, state: CodegenState) {
  const data = el.plain ? undefined : genData(el, state)
  const children = genChildren(el, state, true)

  let code = `_c('${el.tag}')${

    data ? `,${data}` : '' // data
  }`
```

```

} ${

  children ? ` ,${children}` : '' // children
})`


return code
}

```

因为 `_c` 的参数需要 `tagName`、`data` 和 `children`。

所以上面这段代码的主要逻辑就是用 `genData` 和 `genChildren` 获取 `data` 和 `children`，然后拼到 `_c` 中去，拼完后把拼好的 `"_c(tagName, data, children)"` 返回。

`el.plain` 为 `true` 该节点没有属性。因此就不需要执行 `genData`。

所以我们现在比较关心的两个问题：

`data` 如何生成的（`genData` 的实现逻辑）？`children` 如何生成的（`genChildren` 的实现逻辑）？我们先看 `genData` 是怎样的实现逻辑：

```

function genData (el: ASTElement, state: CodegenState): string {

  let data = '{'

  // key
  if (el.key) {
    data += `key:${el.key},`
  }

  // ref
  if (el.ref) {
    data += `ref:${el.ref},`
  }

  if (el.refInFor) {
    data += `refInFor:true,`
  }

  // pre
  if (el.pre) {
    data += `pre:true,`
  }

  // ... 类似的还有很多种情况
  data = data.replace(/,$/, '') + '}'

  return data
}

```

可以看到，就是根据 AST 上当前节点上都有什么属性，然后针对不同的属性做一些不同的处理，最后拼出一个字符串~

然后我们在看看 genChildren 是怎样的实现的：

```
function genChildren (
  el: ASTElement,
  state: CodegenState
): string | void {
  const children = el.children
  if (children.length) {
    return `[${
      children.map(c => genNode(c, state)).join(',')
    }]`
  }
}

function genNode (node: ASTNode, state: CodegenState): string {
  if (node.type === 1) {
    return genElement(node, state)
  } if (node.type === 3 && node.isComment) {
    return genComment(node)
  } else {
    return genText(node)
  }
}
```

从上面代码中可以看出，生成 children 的过程其实就是循环 AST 中当前节点的 children，然后把每一项在重新按不同的节点类型去执行 genElement genComment genText。如果 genElement 中又有 children 在循环生成，如此反复递归，最后一圈跑完之后能拿到一个完整的 render 函数代码字符串，就是类似下面这个样子。

```
"_c('div',[_c('p',[_v(_s(name))])])"
```

最后把生成的 code 装到 with 里。

```
export function generate (
  ast: ASTElement | void,
  options: CompilerOptions
): CodegenResult {
```

```
const state = new CodegenState(options)
// 如果ast为空，则创建一个空div
const code = ast ? genElement(ast, state) : '_c("div")'

return {
  render: `with(this){return ${code}}`
}
}
```

关于代码生成器的部分到这里就说完了，其实源码中远不止这么简单，很多细节我都没有去说，我只说了一个大体的流程，对具体细节感兴趣的同学可以自己去看源码了解详情。

本文使用 mdnice 排版

# 学习vue源码（10）手写render渲染函数



compile 部分已经讲完了

（**compile**部分分为 解析器 + 优化器+ 代码生成器），

终于走到了 render，今天就来给自己记录下渲染三部曲的第二部render，

（渲染三部曲= **compile + render**生成Vnode + 将Vnode通过 update 挂载到 页面上），

update里有一系列 diff操作。

咦，render 内容不多的

噔噔噔噔

render 的作用大家应该清楚

就是 执行 compile 生成的 render函数，然后得到返回的 vnode 节点

比如现在存在这个简单的模板

```
<div :data="data">  
  {{data}}  
</div>
```

经过 compile 之后，解析成了对应的 render 函数，如下

```

function render() {
  with(this) {
    return _c('div', {
      attrs: {
        "data": 111
      },
      [_v(111)])
    }
  }
}

```

看着这个莫名其妙的 `render` 函数，里面都是些什么东西？

不怕，主要是出现了两个函数，我们要探索的就是这两个东西

`_c`, `_v`

这两个函数的作用，都是创建 `Vnode`，但是创建的过程不一样

并且 `render` 函数执行的时候，会绑定上 模板对应的实例 为上下文对象

模板是属于哪个实例的，就绑定哪个实例

`render.call(实例)` 再通过 `with` 的作用

调用 `_c` 和 `_v` 就相当于 `vm._c` 和 `vm._v`

## 什么是 `vm._v`

现在就来看看 `vm._v` 是哪里来的

```

function installRenderHelpers(target) {
  target._v = createTextVNode;
}

installRenderHelpers(Vue.prototype);

```

由上面可知，每个Vue 实例都会继承有 `_v` 这个方法，所以可以通过 `vm._v` 直接调用  
再来看看 `_v` 对应的 `createTextVNode` 的作用是什么

创建文本节点！！

看下源码

```
function createTextVNode(val) {  
  
    return new VNode(  
  
        undefined, undefined,  
  
        undefined, String(val)  
  
    )  
}
```

比如这个模板

```
<div>{{data}}</div>
```

`{{data}}` 虽然是字符串，但是也要作为一个子节点存在，所以就当做是 文本节点  
而 `data` 的值是 111

然后 上面的模板就会得到这样的 Vnode 结构如下

```
▼ VNode {tag: "div", children:  
  ▼ children: Array(1)  
    ▼ 0: VNode  
      children: undefined  
      tag: undefined  
      text: "111"  
      child: (...)  
      ► __proto__: Object  
      length: 1  
      ► __proto__: Array(0)  
      tag: "div"  
      text: undefined
```

## 什么是 `vm._c`

`_c` 是一个大头，`render` 的重中之重，先来看看他是怎么来的

```
function initRender(vm) {  
  vm._c = function(a, b, c, d) {  
  
    return createElement(vm, a, b, c, d);  
  
  };  
}  
  
Vue.prototype._init = function(options) {  
  initRender(this)  
}
```

在实例初始化的时候，就会给实例绑定上 \_c 方法

所以，vm 可以直接调用到 \_c

看了上面的源码，看到 \_c 内部调用了 createElement

那就来看看createElement 的源码吧

个人已经简化得非常简单，觉得不偏离我们的主题就可以

```
function createElement(  
  context, tag, data, children  
) {  
  
  return _createElement(  
  
    context, tag, data, children  
  )  
}
```

```
function _createElement(  
  
  context, tag, data, children
```

```
) {  
  
    var vnode;  
  
    if (如果tag是正常html标签) {  
  
        vnode = new VNode(  
            tag, data, children,  
            undefined, undefined,  
            context  
        );  
    }  
    ....如果tag是组件名，就特殊处理，处理流程已经省略  
  
    if (Array.isArray(vnode))  
  
        return vnode  
  
    else {  
  
        // ...动态绑定 style , class，代码已经省略  
  
        return vnode  
    }  
}
```

你一看就可以看到，createElement 主要就是调用了 new VNode，当然了，render 就是为了创建 vnode 的嘛

你在前面也看到了 render 函数，有传了很多参数给 \_c，如下，\_c 再把这些参数传给构造函数 VNode

```
_c('div',
{
    attrs: {"data": 111}

},
[_v(111)]
)
```

上面这些参数都会传给 Vnode，并保存在创建的 Vnode 中

```
function VNode(
    tag, data, children, text
) {
    this.tag = tag;
    this.data = data;
    this.children = children;
    this.text = text;
}
```

然后得到这么一个 Vnode

```
{
    tag:"div",
    data:{

        attrs: {"data": 111}

    },
    children:[{
```

```
    tag:undefined,  
  
    data:undefined,  
  
    text:111  
  
  }]  
}
```

说到这里，已经能很清楚 render 内部是如何创建Vnode 了

但是这里只是其中一种小小的简单 render

要是项目中的render，数据是很多，很复杂的

而我们主要要把握的是主要流程就可以了

不过，还有必要记录其他 render，那就是遍历

## 遍历相关

看下面这个 template

解析成下面的render

```
function render() {  
  
  with(this) {  
  
    return _c('div',  
  
      _l(2,function(item, index) {  
  
        return _c('span')  
  
      })  
    )  
  }  
}
```

看到一个 \_l，他必定就是遍历生成 Vnode 的幕后黑手了

同样的，\_l 和 \_v 在同一个地方 installRenderHelpers 注册的

```
function installRenderHelpers(target) {
  target._l = renderList;
}
```

不客气地搜出 renderList 源码出来

先跳到后面的分析啊，源码有点长了，虽然很简单

```
function renderList(val, _render) {
  var ret, i, l, keys, key;

  // 遍历数组

  if ( Array.isArray(val) ) {

    ret = new Array(val.length);

    // 调用传入的函数，把值传入，数组保存结果

    for (i = 0, l = val.length; i < l; i++) {
      ret[i] = _render(val[i], i);
    }
  }

  // 遍历数字

  else if (typeof val === 'number') {

    ret = new Array(val);
  }
}
```

```

// 调用传入的函数，把值传入，数组保存结果

for (i = 0; i < val; i++) {
    ret[i] = _render(i + 1, i);
}
}

// 遍历对象

else if (typeof val == "object") {

    keys = Object.keys(val);
    ret = new Array(keys.length);

    // 调用传入的函数，把值传入，数组保存结果

    for (i = 0, l = keys.length; i < l; i++) {
        key = keys[i];
        ret[i] = _render(val[key], key, i);
    }
}

// 返回 vnode 数组

return ret
}

```

看到 renderList 接收两个参数，val 和 render，而 \_l 调用的时候，也就是传入的这两个参数，比如下面

```

_l(2,function(item, index) {
    return _c('span')
})

```

val 就是 2，\_render 就是上面的函数

1 遍历的数据 val

遍历的数据分为三种类型，一种是对象，一种是数字，一种是数组

## 2 单个 vnode 渲染回调 \_render

重要是这个回调

1、renderList 每次遍历都会执行回调，并把的每一项 item 和 index 都传入 回调中

2、回调执行完毕，会返回 vnode

3、使用数组保存 vnode，然后 遍历完毕就返回 数组

于是可以看上面的 render 函数，传入了 数字2，和 创建 span 的回调

```
_l(2,function(item, index) {  
  return _c('span')  
})
```

\_l 执行完毕，内部遍历两次，最后返回 两个 span vnode 的数组，然后传给外层的 \_c ，作为 vnode.children 保存

render 执行完毕，得到这样的 vnode

```
{  
  tag:"div",  
  data:undefined,  
  children:[{  
    tag:"span",  
    data:undefined  
  }, {  
    tag:"span",  
    data:undefined  
  }]
```

```
    }]
}
```

都灰常简单啊，没写之前，我还觉得内容应该挺多的，写完发现还可以

当然还有其他的 render

比如要模板含有 filter, 我们来看看

## Filters - 源码版

下面的讲解会以下面例子 作为讲解模板

这里有一个过滤器 all，用来过滤 parentName

```
<div>{{parentName|all }}</div>

new Vue({
  el:document.getElementsByTagName("div")[0],
  data(){
    return {
      parentName:111
    }
  },
  filters:{
    all(){  return "我是过滤器" }
  }
})
```

页面的 filter 解析成什么

首先，上面的例子会被解析成下面的渲染函数

```
(function() {
  with(this) {
    return _c('div',[  
      _v(_s(_f("all"))(parentName))  
    ])
  }
})
```

这段代码继续解释下

1. `_c` 是渲染组件的函数，这里会渲染出根组件
2. 这是匿名自执行函数，后面渲染的时候调用，会绑定当前实例为作用域
3. `with` 的作用是，绑定大括号内代码的 变量访问作用域，所以里面的所有变量都会从 实例上获取

然后，你可以看到 `' parentName | all '` 被解析成 `_f('all')( parentName )`

怎么解析的？

简单说就是，当匹配到 `|` 这个符号，就知道你用过滤器，然后就解析成 `_f` 去获取对应过滤器 并调用，这个过程不赘述

## `_f` 是什么？

`_f` 是获取具体过滤器的函数

`_f` 会在 Vue 初始化的时候，注册到 Vue 的原型上

```
// 已简化

function installRenderHelpers(target) {

  target._s = toString;
  target._f = resolveFilter;
}
```

```
installRenderHelpers(Vue.prototype);
```

所在在上面的渲染函数 with 绑定当前实例vm为作用域之后，\_f 从vm 获取，成了这样 vm.\_f

\_f 是 resolveFilter，一个可以获取具体filter 的函数

使用 \_f("all") 就能获取到 all 这个过滤器，resolveFilter 下面会说

怎么获取下面继续.....

设置的 filter 如何被调用

由上面可以看到，\_f 是 resolveFilter 赋值的，下面是 resolveFilter 源码

```
// 已简化

function resolveFilter(id) {
  return resolveAsset(
    this.$options, 'filters', id, true
  ) || identity
}
```

要是你看过学习vue源码（3）手写Vue.directive、Vue.filter、Vue.component方法，相信这里你很熟悉，其实就是从this.options.filters里找对应的过滤器函数来调用，如图所示

## (5) 实现

```

    ...
    Vue.options['filters'] = Object.create(null);

    Vue.filter = function(id, definition){
        if(!definition){
            return this.options['filters'][id];
        }else{
            this.options['filters'][id] = definition;
            return definition;
        }
    }
}

```

1、在Vue.options中新增了filters属性用于存放过滤器，并在Vue.js上新增了filter方法，它接受两个参数id和definition。

2、Vue.filters方法可以注册或获取过滤器，这取决于definition参数是否存在。

3、如果definition不存在，则使用id从this.options['filters']中读出过滤器并将它返回。

4、如果definition存在，则说明是注册操作，直接将该参数保存到this.options['filters']中。

this.options 会拿到当前组件的所有选项

你问我为什么？

根据上一个问题知道

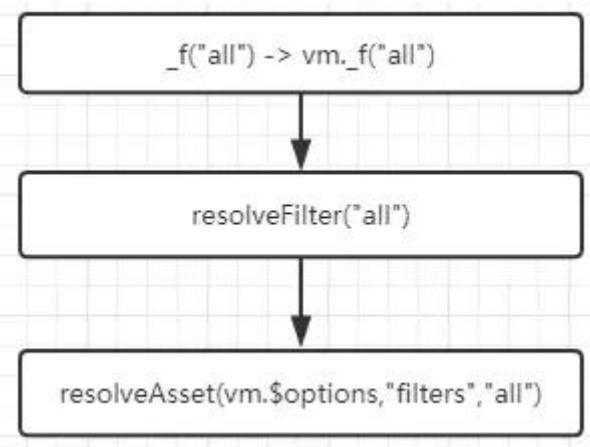
1. \_f 会使用实例去调用，vm.\_f 类似 vm.resolveFilter
2. 所以，resolveFilter 的执行上下文 this 是 vm
3. 所以，this.\$options 就是实例的 options 啊

接着，调用 resolveAsset，目的就是拿到组件选项中的具体 filter

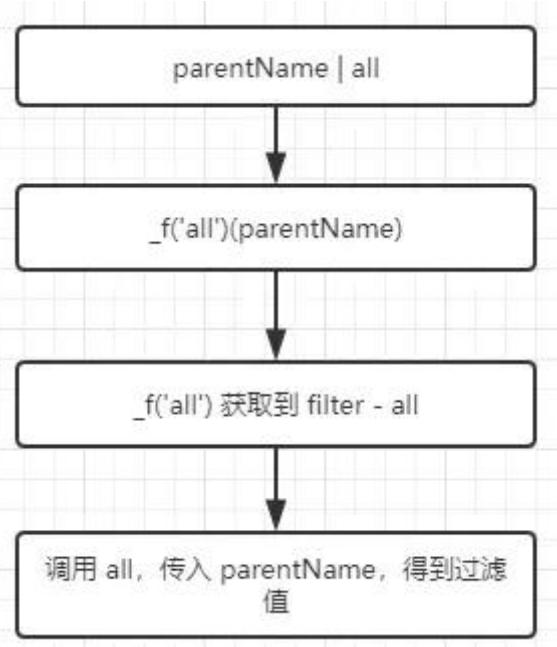
传入当前组件的选项，指定要其选项 filters，指定具体 filter 名

```
function resolveAsset(  
  options, type, id, warnMissing  
) {  
  
  // g: 拿到 filters 选项  
  
  var assets = options[type];  
  
  // g: 返回 调用的 filter  
  
  return assets[id]  
}
```

\_f("all") 流程 就成了下面这样

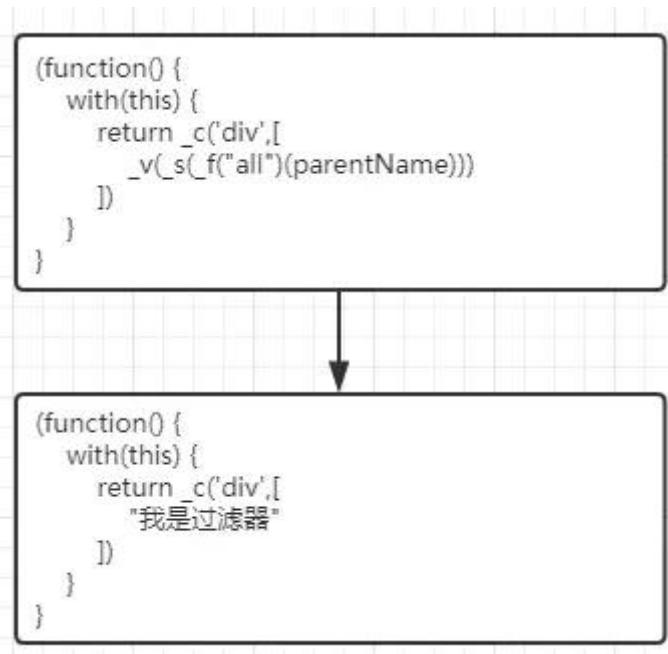


1. 拿到 组件选项 中的 filters
2. 然后再从 filters 中，拿到 all 这个filter
3. 执行返回的 all 过滤函数时，传入需要过滤的值 parentName
4. 得到 返回了 过滤后的值



所以，当渲染函数解析的时候，碰到使用过滤器的地方，按流程拿到过滤值后，就可以渲染到页面上了

`_f("all")(parentName)` 就会变成 "我是过滤器" 放到 渲染函数中，最后，就是渲染到页面了



## 总结

filter 其实就是从组件选项 filters 获取你设置的某个filter，并调用，然后使用你函数执行的返回值渲染

太简单了，总结跟没总结一样.....

render 什么时候开始执行？

如果你看过学习vue源码（4）手写vm.\$mount方法,我相信你已经知道了

```

export function mountComponent(vm, el){
  if(!vm.$options.render){
    vm.$options.render = createEmptyVNode;
    if(process.env.NODE_ENV !== 'production'){
      <!-- 在开发环境发出警告 -->
    }
    <!-- 触发生命周期钩子 -->
    callHook(vm, 'beforeMount');
    <!-- 挂载 -->
    vm._watcher = new Watcher(vm, ()=>{
      vm._update(vm._render())
    }, noop);
    <!-- 触发生命周期钩子 -->
    callHook(vm, 'mounted');
    return vm;
  }
}

```

如图所示，是在挂载阶段执行的。

## 总结

每个模板经过 compile 都会生成一个 render 函数

render 作为 渲染三部曲的第二部，主要作用就是 执行 render，生成 Vnode

把 template 上绑定的数据，都保存到 vnode 中

然后，生成 Vnode，就是为了给 渲染三部曲的 第三部 Diff 提供源动力

从而完成 DOM 挂载

到这里其实基本就已经结束了render的思路，但是源码中有个静态render，这个对渲染性能的提高有极大的帮助，所以必须看下。

没错，就是 静态 render，看过学习vue源码（8）手写优化器的人，应该知道什么是 静态 render

静态 render 就是用于渲染哪些不会变化的节点

大家可以先看看，Vue 是怎么判断某个节点是否是静态节点

好，下面开始我们的正文，想了想，我们还是以几个问题开始吧

1、静态 render 是什么样子的

2、静态 render 是怎么生成和保存

3、静态 render 怎么执行

## 什么是 静态Render

静态 render 其实跟 render 是一样的，都是执行得到 Vnode

只是静态 render，没有绑定动态数据而已，也就是说不会变化

比如说，一个简单 render 是这样的

```
<div>
  {{aa}}
</div>
```

绑定了动态数据，需要从实例去获取

```
_c('div',[_v(_s(aa))])
```

而静态 render 是这样的

```
<div>
  <span>1</span>
</div>
```

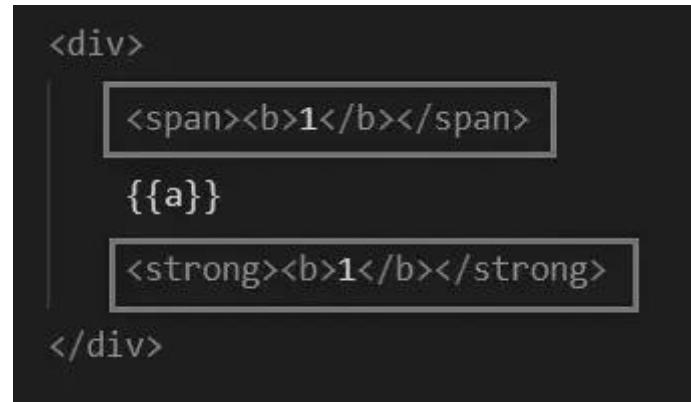
没有动态数据，这个静态render 的执行结果是永远不会变的

```
_c('div',[_c('span',[_v("1")])])
```

## 生成保存静态Render

静态 render 是在 generate 阶段生成的，生成的方式和 render 是一样的

比如在一个模板中，有很多个静态 根节点，像这样



首先，Vue 会在遍历模板的时候，发现 span 和 strong 本身以及其子节点都是静态的

那么就会给 span 和 strong 节点本身设置一个属性 staticRoot，表示他们是静态根节点

然后这两个静态根节点就会生成自己专属的 静态 render

如果你有一直看我的Vue 笔记的话，你应该这里是会有点印象的

之后

静态 render 生成之后是需要保存的，那么保存在哪里呢？

保存在一个数组中，名叫 `staticRenderFns`，就是直接push 进去

当然了，此时的 push 进去的 静态 render 还是字符串，并没有变成函数

以上面的模板为例，这里的 `staticRenderFns` 就是这样，包含了两个字符串

```
staticRenderFns = [
  "_c('span',[_c('b',[_v("1")])])",
  "_c('strong',[_c('b',[_v("1")])])"
]
```

但是在后面会逐个遍历变成可执行的函数

```
staticRenderFns = staticRenderFns.map(code => {

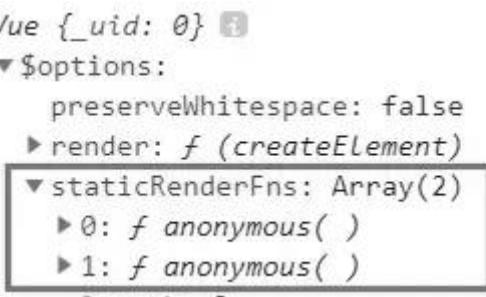
  return new Function(code)

});
```

那么这个 **staticRenderFns** 又是什么啊？

每个 Vue 实例都有一个独立的 `staticRenderFns`，用来保存实例本身的静态 render  
`staticRenderFns` 的位置是

`vm.$options.staticRenderFns`

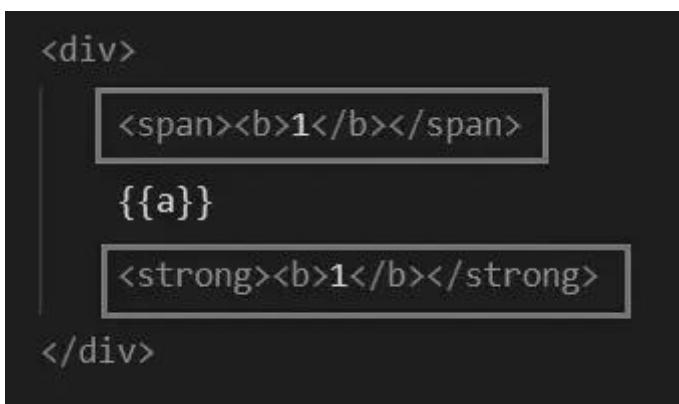


```
▼ Vue {_uid: 0}
  ▼ $options:
    preserveWhitespace: false
    ▶ render: f (createElement)
    ▼ staticRenderFns: Array(2)
      ▶ 0: f anonymous()
      ▶ 1: f anonymous()
```

## 执行静态 Render

静态 render 需要配合 render 使用，怎么说

看个例子



```
<div>
  <span><b>1</b></span>
  {{a}}
  <strong><b>1</b></strong>
</div>
```

这个模板的 render 函数是

```
_c('div',[  
  _m(0),  
  _v(_s(a)),
```

```
_m(1)
})
```

`_m(0), _m(1)` 就是执行的就是静态 render 函数，然后返回 Vnode

于是 render 也可以完成 vnode 树的构建了

那么 `_m` 是什么呢？

在 Vue 初始化时，给 Vue 的原型便注册了这个函数，也就是说每个实例都继承到 `_m`

```
function installRenderHelpers(target) {
  target._m = renderStatic;
}

installRenderHelpers(Vue.prototype);
```

再来看 `renderStatic`

```
function renderStatic(index) {
  var cached = this._staticTrees || (this._staticTrees = []);
  var tree = cached[index];
  // 如果缓存存在，就直接返回
  if (tree) return tree
  // 这里是执行 render 的地方
  tree = cached[index] =
    this.$options.staticRenderFns[index].call(
      this, null, this
```

```
);

// 只是标记静态 和 节点id 而已
markStatic(tree, "__static__" + index, false);

return tree
}
```

这个函数做的事情可以分为几件

- 1、执行静态render
- 2、缓存静态render 结果
- 3、标记 静态 render 执行得到的 Vnode

我们来一个个说

**\*\*1 执行静态render \*\***

上面我们说过了，静态render 保存在 数组 staticRenderFns

所以这个函数接收一个索引值，表示要执行数组内哪个静态render

取出静态render 后，执行并绑定 Vue 实例为上下文对象

然后得到 Vnode

## 2 缓存静态**render** 结果

这一步就是要把上一步得到的 Vnode 缓存起来

那么缓存在哪里呢？

`_staticTrees`

这是一个数组，每个实例都会有一个独立的 `_staticTrees`，用来存在自身的静态 render 执行得到的 Vnode

看一下上个模板中实例保存的 `_staticTrees`

```
▼ Vue { _uid: 0 } ⓘ
  ► $options: {_base: f, preserveWhitespace:
    $vnode: undefined
  ▼ _staticTrees: Array(2)
    ► 0: VNode {tag: "span", data: undefined,
    ► 1: VNode {tag: "strong", data: undefined
      ...
    }
  }
}
```

### 3 标记 静态 render 执行得到的 Vnode

我们已经执行静态render得到了 Vnode，这一步目的是标记

标记什么呢

1、添加标志位 isStatic

2、添加 Vnode 唯一id

renderStatic 中我们看到标记的时候，调用了 markStatic 方法，现在就来看看

```
function markStatic(
  tree, key
) {

  if (Array.isArray(tree)) {

    for (var i = 0; i < tree.length; i++) {

      if ( tree[i] && typeof tree[i] !== 'string' ) {

        var node = tree[i]
      }
    }
  }
}
```

```
        node.isStatic = true;  
  
        node.key = key + "_" + i;  
  
    }  
  
}  
  
}  
  
else {  
  
    tree.isStatic = true;  
  
    tree.key = key  
  
}  
  
}
```

为什么添加标志位 `isStatic` ?

前面我们添加的所有静态标志位都是针对 模板生成的 ast

这里我们是给 Vnode 添加 isStatic，这样才能完成 Vue 的目的。

Vue 目的就是性能优化，在页面改变时，能尽量少的更新节点

于是在页面变化时，当 Vue 检测到该 `Vnode.isStatic = true`，便不会比较这部分内容

从而减少比对时间

Vnode 唯一id

每个静态根Vnode 都会存在的一个属性

```
▼ Vue {_uid: 0} ⓘ
  ► $options: {_base: f, preserveWhitespace: fa}
    $vnode: undefined
  ▼ _staticTrees: Array(2)
    ▼ 0: VNode
      ► children: [VNode]
        isStatic: true
        key: "__static__0"
        tag: "span"
        text: undefined
        child: (...)
```

我也没想到静态Vnode的key有什么作用，毕竟不需要比较，也许是易于区分？？

最后

静态 render 我们就讲完了，是不是很简单，在没看源码之前，我以为很难

现在看完，发现也简单的，不过我也是看了几个月的。。。



鉴于本人能力有限，难免会有疏漏错误的地方，请大家多多包涵，如果有任何描述不当的地方，欢迎后台联系本人，领取红包

本文使用 mdnice 排版

# 学习vue源码 (11) 学习 合并策略



我们之前谈 [学习 vue 源码 \(5\) 手写 Vue.use、Vue.mixin、Vue.compile](#) 的时候 谈到了 `Vue.mixin` 的源码实现，然后谈到了 `mergeOptions`，那时并没有深入解说这个函数的原理。如图所示

## (4) 实现

```
import { mergeOptions } from '../util/index'

export function initMixin(Vue){
  Vue.mixin = function(mixin){
    this.options = mergeOptions(this.options,mixin);
    return this;
  }
}
```

1. `mergeOptions` 会将用户传入的 `mixin` 与 `this.options` 合并成一个新对象，然后将这个生成的新对象覆盖 `this.options` 属性，这里的 `this.options` 其实就是 `Vue.options`。`mergeOptions` 的具体实现，我们后面再讲。

2. 因为 `mixin` 方法修改了 `Vue.options` 属性，而之后创建的每个实例都会用到该属性，所以会影响创建的每个实例。

这次我们就来深入研究下，因此也就离不开Vue中的一个重要思想：合并策略了。

我们有时面试时可能会遇到这样的问题：

- 引入的 mixin 的 data 中有 name 这个属性，自己注册的组件的 data 中也有 name，那哪个会生效？
- 引入的 mixin 中有 created，自己注册的组件中也有 created，那哪个 create 会先执行？
- . . . . .

这样的问题其实还很多，而想要弄清楚这些问题，其实就是 合并策略的问题。

现在先大概地谈下合并策略，是让大家有兴趣去研究源码的时候，可以提前理清一下思路。暂时没时间了解源码的，也可以先了解下内部流程，对解决一些奇奇怪怪的问题也是很有作用的

mixins 我觉得可能大家很少用吧，但是这个东西真的非常有用。相当于封装，提取公共部分。

显然，今天我不是来教大家怎么用的，怎么用看文档就好了，我是讲解生命的真谛 内部的工作原理。如果不懂用的，请去官网看怎么用，兄弟

mixin 不难，就是有点绕，今天我们探索两个问题

1、什么时候合并

2、怎么合并

## 什么时候合并

合并分为两种

### 1. 全局 mixin 和 基础全局 options 合并

这个过程是先于你调用 Vue 时发生的，也是必须是先发生的。这样 mixin 才能合并上你的自定义 options

```
Vue.mixin = function(mixin) {
  this.options = mergeOptions(
    this.options, mixin
  );
  return this
};
```

基础全局 options 是什么？

就是 components , directives , filters 这三个，一开始就给设置在了 Vue.options 上。所以这三个是最先存在全局options。

```
Vue.options = Object.create(null);

['component','directive','filter'].forEach(function(type) {
  Vue.options[type + 's'] = Object.create(null);
});
```

这一步，是调用 Vue.mixin 的时候就马上合并了，然后这一步完成以后，举个栗子

```
Vue.mixin({
  methods: {
    mixins_methods(){}
  }
})

Vue.filter("getName",function(){})
```

全局选项就变成下面这样，然后每个Vue实例都需要和这全局选项合并

```
▼ {components: {}, directives: {},
  ▶ components: {}
  ▶ directives: {}
  ▶ filters: {getName: f}
  ▶ methods: {mixins_methods: f}}
```

可能你会奇怪，我明明没写 `keep-alive` 这些组件，为什么可以用这些组件呢？

其实原因就在这，在我们new Vue，之前，源码 就给 Vue 的 option中的 **components**数组 添加了 `keep-alive` 这些组件。

基础组件形成后 就 和 全局的 mixin 合并生成新的option。

然后，我们 `new Vue()` 时，会传入自己的option，然后就将自己的option和上面那个新的option 合并。

## 2.全局options和自定义options合并

在调用Vue 的时候，首先进行的就是合并

```
function Vue(options){
```

```

vm.$options = mergeOptions(
  { 全局component ,
    全局directive ,
    全局filter 等....},
  options , vm
);

...处理选项，生成模板，挂载DOM 等....  

}

```

options 就是你自己传进去的对象参数，然后跟 全局options 合并，全局options 是哪些，也已经说过了

```

new Vue({
  el: document.getElementsByTagName("div")[0],
  mixins:[{
    data(){
      return {name2:222}
    }
  ],
  data(){
    return {
      name:111
    }
  }
})

```

## 2.怎么合并

下面开始讲解各种合并策略

### 1、函数合并叠加

包括选项：data，provide

把两个函数合并加到一个新函数中，并返回这个函数。在函数里面，会执行那两个方法。

按照这个流程，使用代号

data 中数据有重复的，权重大的优先，比如下面

```

var test_mixins={
  data(){

```

```
        return {name:34}
    }
}

var a=new Vue({
  mixins:[test_mixins],
  data(){
    return {name:12}
  }
})
```

可以看到， mixin 和组件本身的 data 都有 name 这个数据，很显然会以组件本身的为主，因为组件本身权重大

## 2、数组叠加

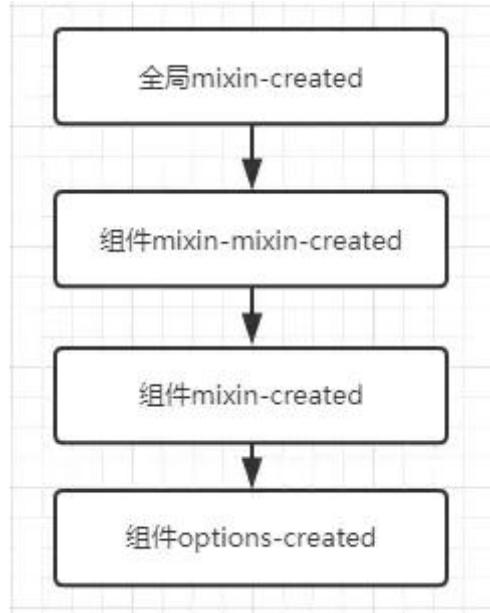
包括生命周期函数和watch

### 生命周期函数

权重越大的越放后面，会合并成一个数组，比如created

```
[  
  全局 mixin - created,  
  组件 mixin-mixin - created,  
  组件 mixin - created,  
  组件 options - created  
]
```

执行流程是



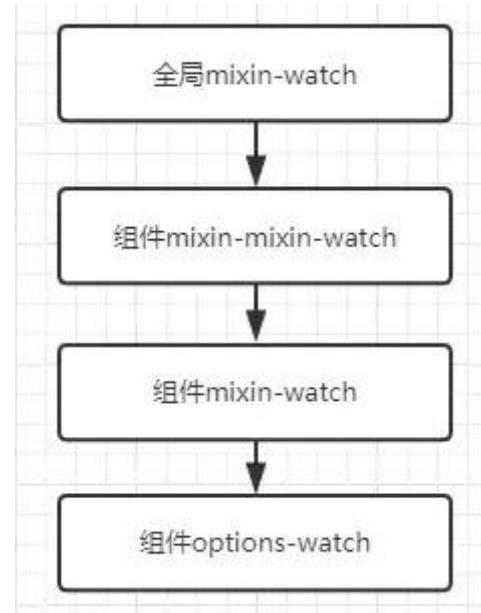
生命周期，权重小的 先执行

## watch

合并成一个下面这样的数组，权重越大的越放后面

```
[  
  全局 mixin - watch,  
  组件 mixin-mixin - watch,  
  组件 mixin - watch,  
  组件 options - watch  
]
```

执行流程是



监听回调，权重小的 先执行

### 3、原型叠加

包括选项：`components`，`filters`，`directives`

两个对象合并的时候，不会相互覆盖，而是 权重小的 被放到 权重大的 的原型上

这样权重大的，访问快些，因为原型链短了

```
A.__proto__ = B  
B.__proto__ = C  
C.__proto__ = D
```

两个对象合并的时候，不会

以 `filter` 为例，下面是四种 `filter` 合并

```
// 全局 filter  
Vue.filter("global_filter",function (params) {})  
  
// mixin 的 mixin  
var mixin_mixin={  
    filters:{  
        mixin_mixin_filter(){  
    }  
}  
  
// mixin filter  
var test_mixins={  
    mixins:[mixin_mixin],  
    filters:{  
        mixin_filter(){  
    }  
}  
  
// 组件 filter  
var a=new Vue({  
    mixins:[test_mixins],  
    filters:{  
        self_filter(){  
    }  
})
```

结果就系这样..

```

▲ filters: Object {self_filter: }
  ↳ self_filter: self_filter(){ ... }
  ▲ __proto__: Object {mixin_filter: }
    ↳ mixin_filter: mixin_filter(){ ... }
  ▲ __proto__: Object {global_filter: }
    ↳ global_filter: function (params) {

```

## 4、覆盖叠加

包括选项：`props`，`methods`，`computed`，`inject`

两个对象合并，如果有重复key，权重大的覆盖权重小的

比如

```

组件的 props: { name:"" }
组件 mixin 的 props: { name:"", age: "" }

```

那么 把两个对象合并，有相同属性，以 权重大的为主，组件的 `name` 会替换 `mixin` 的`name`。

这个其实就是跟第一种（函数合并叠加）是一样的，只不过第一种是函数

## 5、直接替换

这是默认的处理方式，当选项不属于上面的处理方式的时候，就会像这样处理，包含选项：`el`，`template`，`propData` 等

两个数据只替换，不合并，权重大的，会一直替换 权重小的，因为这些属于只允许存在一个，所有只使用权重大的选项

组件 设置 `template`，`mixin` 也设置 `template`，不用怕，组件的肯定优先

这个好像跟 覆盖叠加 很像，其实不一样，覆盖叠加会把两个数据合并，重复的才覆盖。而这个不会合并，直接替换掉整个选项

好了，现在我们结合 源码 来谈谈这五种合并策略。

来看看 `mergeOptions` 函数到底是什么妖魔鬼怪

```
function mergeOptions(parent, child, vm) {  
  
    // 遍历mixins，parent 先和 mixins 合并，然后在和 child 合并  
  
    if (child.mixins) {  
  
        for (var i = 0, l = child.mixins.length; i < l; i++) {  
  
            parent = mergeOptions(parent, child.mixins[i], vm);  
        }  
  
    }  
  
    var options = {}, key;  
  
    // 先处理 parent 的 key，  
  
    for (key in parent) {  
        mergeField(key);  
    }  
  
    // 遍历 child 的key，排除已经处理过的 parent 中的key  
  
    for (key in child) {  
  
        if (!parent.hasOwnProperty(key)) {  
  
            mergeField(key);  
        }  
  
    }  
}
```

// 拿到相应类型的合并函数，进行合并字段，strats 请看下面

```
function mergeField(key) {  
  
    // strats 保存着各种字段的处理函数，否则使用默认处理  
    var strat = strats[key] || defaultStrat;  
  
    // 相应的字段处理完成之后，会完成合并的选项  
  
    options[key] = strat(parent[key], child[key], vm, key);  
}  
  
return options  
  
}
```

这段代码看上去有点绕，其实无非就是

1. 先遍历合并 parent 中的key，保存在变量options
2. 再遍历 child，合并补上 parent 中没有的key，保存在变量options
3. 优先处理 mixins ，但是过程跟上面是一样的，只是递归处理而已

在上面实例初始化时的合并，parent 就是全局选项，child 就是组件自定义选项，因为 parent 权重比 child 低，所以先处理 parent。

“公司开除程序猿，也是先开始作用较低。。。”

重点其实在于各式各样的处理函数 strat，下面将会一一列举

注意，不要对strat这个词迷糊，其实就是 strategy 策略一词的缩写，它就是一个对象，这个对象里保存了上面说的各种策略。如下

```
strat = {  
    // 合并data的策略
```

```

data: function(parentVal, childVal, vm) {
    return mergeDataOrFn(
        parentVal, childVal, vm
    )
}

// 合并生命钩子的策略
created: mergeHook;

mounted: mergeHook;

. . . .
}

```

下面讲各种合并策略

## 1. 默认策略 defaultStrats

该策略不保存在 strat 对象中，也就是说在 strat 对象找不到 key 对应的策略时就会使用该策略，上面 mergeOptions 函数中的

```

function mergeField(key) {
    // strats 保存着各种字段的处理函数，否则使用默认处理
    var strat = strats[key] || defaultStrat;
    // 相应的字段处理完成之后，会完成合并的选项

    options[key] = strat(parent[key], child[key], vm, key);
}

```

这段代码就是这个意思。

好了，看下默认策略的源码实现

```

var defaultStrats= function(parentVal, childVal) {
    return childVal === undefined ?
        parentVal :
        childVal
};

```

这段函数言简意赅，意思就是优先使用组件的options

组件options>组件 mixin options>全局options

## 2.data

```
strats.data = function(parentVal, childVal, vm) {
  return mergeDataOrFn(
    parentVal, childVal, vm
  )
};

function mergeDataOrFn(parentVal, childVal, vm) {
  return function mergedInstanceDataFn() {
    var childData = childVal.call(vm, vm)
    var parentData = parentVal.call(vm, vm)

    if (childData) {
      return mergeData(childData, parentData)
    } else {
      return parentData
    }
  }
}

function mergeData(to, from) {
```

```
if (!from) return to

var key, toVal, fromVal;

var keys = Object.keys(from);

for (var i = 0; i < keys.length; i++) {

    key = keys[i];
    toVal = to[key];

    fromVal = from[key];

    // 如果不存在这个属性，就重新设置

    if (!to.hasOwnProperty(key)) {

        set(to, key, fromVal);
    }
}

// 存在相同属性，合并对象

else if (typeof toVal == "object" && typeof fromVal == "object") {
    mergeData(toVal, fromVal);
}

return to
}
```

我们先默认 data 的值是一个函数，简化下源码，但是其实看上去还是会有些复杂

不过我们主要了解他的工作过程就好了

1、两个data函数 组装成一个函数

2、合并 两个data函数执行返回的 数据对象

### 3.生命钩子

把所有的钩子函数保存进数组，重要的是数组子项的顺序

顺序就是这样

```
[  
  全局 mixin - created,  
  组件 mixin-mixin - created,  
  组件 mixin - created,  
  组件 options - created  
]
```

所以当数组执行的时候，正序遍历，就会先执行全局注册的钩子，最后是组件的钩子

```
function mergeHook(parentVal, childVal) {  
  
  var arr;  
  
  arr = childVal ?  
  
    // concat 不只可以拼接数组，什么都可以拼接  
  
    ( parentVal ?  
  
      // 为什么parentVal 是个数组呢  
  
      // 因为无论怎么样，第一个 parent 都是{ component, filter, directive}  
      // 所以在这里，合并的时候，肯定只有 childVal，然后就变成了数组  
      parentVal.concat(childVal) :  
  
      ( Array.isArray(childVal) ? childVal: [childVal] )  
    ) :  
    parentVal
```

```
        return arr  
  
    }  
  
    strats['created'] = mergeHook;  
  
    strats['mounted'] = mergeHook;  
    ... 等其他钩子
```

## 4. component、directives、filters

我一直觉得这个是比较好玩的，这种类型的合并方式，我是从来没有在项目中使用过的  
原型叠加

两个对象并没有进行遍历合并，而是把一个对象直接当做另一个对象的原型

这种做法的好处，就是为了保留两个相同的字段且能访问，避免被覆盖

学到了学到了.....反正我是学到了

```
strats.components=  
strats.directives=  
  
strats.filters = function mergeAssets(  
  
    parentVal, childVal, vm, key  
) {  
  
    var res = Object.create(parentVal || null);  
  
    if (childVal) {  
  
        for (var key in childVal) {  
  
            res[key] = childVal[key];  
  
        }  
    }  
  
    return res
```

```
}
```

就是下面这种，层层叠加的原型

```
filters: Object {self_filter: }
  ↳ self_filter: self_filter(){ ... }
    ↳ __proto__: Object {mixin_filter: }
      ↳ mixin_filter: mixin_filter(){ ... }
        ↳ __proto__: Object {global_filter: }
          ↳ global_filter: function (params) {
```

## 5. watch

watch 的处理，也是合并成数组，重要的也是合并顺序，跟 生命钩子一样

这样的钩子

```
[  
  全局 mixin - watch,  
  组件 mixin-mixin - watch,  
  组件 mixin - watch,  
  组件 options - watch  
]
```

按照正序执行，最后执行的 必然是组件的 watch

```
strats.watch = function(parentVal, childVal, vm, key) {  
  
  if (!childVal) {  
  
    return Object.create(parentVal || null)  
  
  }  
  
  if (!parentVal)  return childVal
```

```
var ret = {};  
  
// 复制 parentVal 到 ret 中  
  
for (var key in parentVal) {  
    ret[key] = parentVal[key];  
}  
  
for (var key$1 in childVal) {  
  
    var parent = ret[key$1];  
  
    var child = childVal[key$1];  
  
    if (!Array.isArray(parent)) {  
  
        parent = [parent];  
    }  
  
    ret[key$1] = parent ? parent.concat(child) :  
        ( Array.isArray(child) ? child: [child] );  
  
}  
  
return ret  
};
```

## 6. props、computed、methods

这几个东西，是不允许重名的，合并成对象的时候，不是你死就是我活

重要的是，以谁的为主？必然是组件options 为主了

比如

组件mixin 的 props : { name:"", age: "" }

那么 把两个对象合并，有相同属性，组件的 name 会替换 mixin 的name

```
strats.props =
strats.methods =
strats.inject =

strats.computed = function(parentVal, childVal, vm, key) {

    if (!parentVal) return childVal

    // 把 parentVal 的字段 复制到 ret 中

    for (var key in parentVal) {
        ret[key] = parentVal[key];
    }

    if (childVal) {

        for (var key in childVal) {
            ret[key] = childVal[key];
        }
    }

    return ret
};
```

## 最后

合并策略大概就这么多，初看有点复杂，其实很简单。就这几种。

有错误的地方，欢迎更正。

本文使用 mdnice 排版

# 学习vue源码（12）大白话谈响应式原理



## 前言

本文用最简单的方式来解释VUE2 最重点的响应式原理，看不懂算我输！

## 一、响应式原理

什么是响应式原理？

意思就是在改变数据的时候，视图会跟着更新。这意味着你只需要进行数据的管理，给我们搬砖提供了很大的便利。

VUE是利用了Object.defineProperty的方法里面的setter与getter方法的观察者模式来实现。

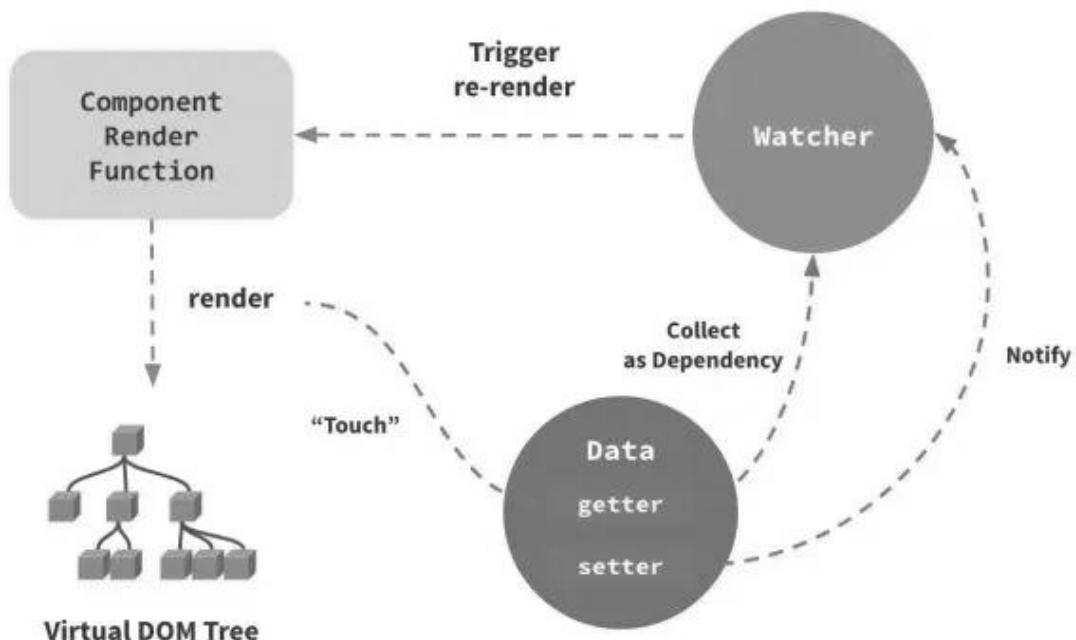
所以在学习VUE的响应式原理之前，先学习两个预备知识：

**Object.defineProperty** 与 观察者模式。

### 三、原理解析

在学完了前面的铺垫之后，我们终于可以开始讲解VUE的响应式原理了。

官网用了一张图来表示这个过程，但是刚开始看可能看不懂，等到文章的最后，我们再来看，应该就能看懂了。



知乎 @daisy

总共分为三步骤：

**1、init 阶段：** VUE 的 data的属性都会被reactive化，也就是加上 setter/getter函数。

```
function defineReactive(obj: Object, key: string, ...) {
  const dep = new Dep()

  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
```

```

get: function reactiveGetter () {
    ...
    dep.depend()
    return value
    ...
},
set: function reactiveSetter (newVal) {
    ...
    val = newVal
    dep.notify()
    ...
}
})
}
}

class Dep {
    static target: ?Watcher;
    subs: Array<Watcher>;
    depend () {
        if (Dep.target) {
            Dep.target.addDep(this)
        }
    }
}

notify () {
    const subs = this.subs.slice()
    for (let i = 0, l = subs.length; i < l; i++) {
        subs[i].update()
    }
}
}

```

其中这里的Dep就是一个观察者类，每一个data的属性都会有一个dep对象。当getter调用的时候，去dep里注册函数，至于注册了什么函数，我们等会再说。

setter的时候，就是去通知执行刚刚注册的函数。

## 2、mount 阶段：

```

mountComponent(vm: Component, el: ?Element, ... ) {
    vm.$el = el
    ...
}

```

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

new Watcher(vm, updateComponent, ...)
}

class Watcher {
  getter: Function;

  // 代码经过简化
  constructor(vm: Component, expOrFn: string | Function, ...) {
    ...
    this.getter = expOrFn
    Dep.target = this           // 注意这里将当前的Watcher赋值给了Dep.target
    this.value = this.getter.call(vm, vm) // 调用组件的更新函数
    ...
  }
}
```

看过我之前这篇 [学习 vue 源码（4）手写 vm.\\$mount 方法文章](#) 的同学，相信会对 `mountComponent` 这个函数很熟悉，如图所示：

### (3) mountComponent具体实现

```

    * * *
    export function mountComponent(vm,el){
      if(!vm.$options.render){
        vm.$options.render = createEmptyVNode;
        if(process.env.NODE_ENV !== 'production'){
          <!-- 在开发环境发出警告 -->
        }
        <!-- 触发生命周期钩子 -->
        callHook(vm,'beforeMount');
        <!-- 挂载 -->
        vm._watcher = new Watcher(vm,()=>{
          vm._update(vm._render());
        },noop);
        <!-- 触发生命周期钩子 -->
        callHook(vm,'mounted');
      }
      return vm;
    }
  }
}

```

1、`vm._update`作用：调用虚拟DOM中的`patch`方法来执行节点的比对与渲染操作。

2、`vm._render`作用：执行渲染函数，得到一份新的`VNode`节点树。

3、`vm._update(vm._render())`作用：先调用渲染函数得到一份最新的`VNode`节点树，然后通过`vm._update`方法对最新的`VNode`和上一次渲染用到的旧`VNode`进行对比并更新DOM节点。简单来说，就是执行了渲染操作。

---

我们已经讲过了。

`mount`阶段的时候，会创建一个`Watcher`类的对象。这个`Watcher`实际上是连接Vue组件与`Dep`的桥梁。

每一个`Watcher`对应一个`vue component`。

这里可以看出`new Watcher`的时候，`constructor`里的`this.getter.call(vm, vm)`函数会被执行。`getter`就是`updateComponent`。这个函数会调用组件的`render`函数来更新重新渲染。

而`render`函数里，会访问`data`的属性，比如

```
render: function (createElement) {
  return createElement('h1', this.blogTitle)
}
```

此时会去调用这个属性blogTitle的getter函数，即：

```
// getter函数
get: function reactiveGetter () {
  ....
  dep.depend()
  return value
  ....
},

// dep的depend函数
depend () {
  if (Dep.target) {
    Dep.target.addDep(this)
  }
}
```

在depend的函数里，Dep.target就是watcher本身（我们在class Watch里讲过，不记得可以往上第三段代码），这里做的事情就是给blogTitle注册了Watcher这个对象。这样每次render一个vue组件的时候，如果这个组件用到了blogTitle，那么这个组件相对应的Watcher对象都会被注册到blogTitle的Dep中。

这个过程就叫做依赖收集。

收集完所有依赖blogTitle属性的组件所对应的Watcher之后，当它发生改变的时候，就会去通知Watcher更新关联的组件。

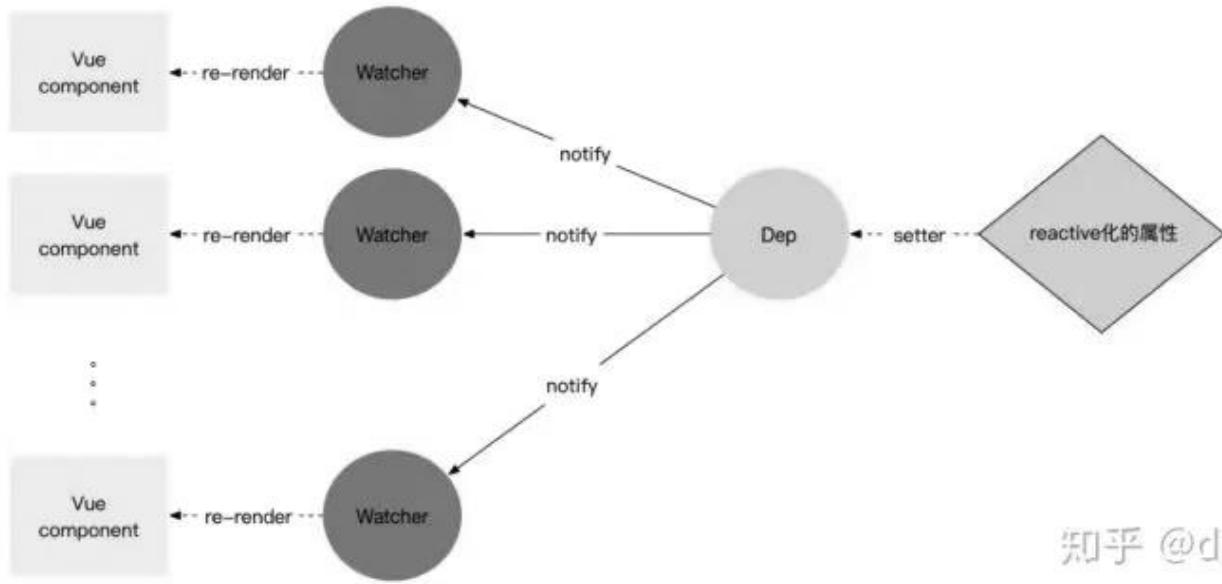
### 3、更新阶段：

当blogTitle发生改变的时候，就去调用Dep的notify函数，然后通知所有的Watcher调用update函数更新。

```
notify () {
  const subs = this.subs.slice()
```

```
for (let i = 0, l = subs.length; i < l; i++) {  
    subs[i].update()  
}  
}
```

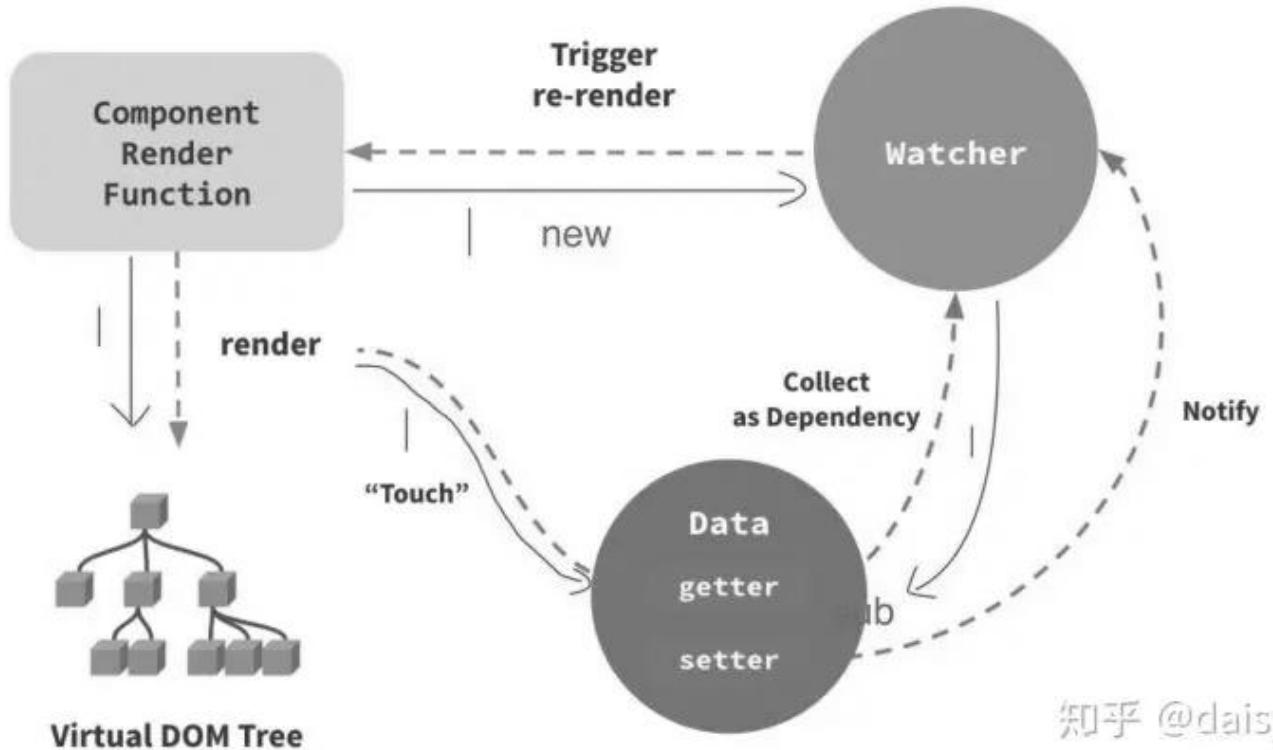
可以用一张图来表示：



由此图我们可以看出Watcher是连接VUE component 跟 data属性的桥梁。

## 总结

最后，我们通过解释官方的图来做个总结。



1、第一步：组件初始化的时候，先给每一个Data属性都注册getter，setter，也就是reactive化。然后再new一个自己的Watcher对象，此时watcher会立即调用组件的render函数去生成虚拟DOM。在调用render的时候，就会需要用到data的属性值，此时会触发getter函数，将当前的Watcher函数注册进sub里。

2、第二步：当data属性发生改变之后，就会遍历sub里所有的watcher对象，通知它们去重新渲染组件。

整个过程就是那么简单啦~

## 附录

### 2.2 观察者模式

什么是观察者模式？它分为注册环节跟发布环节。

以报社订阅报纸来说明：

报社的业务就是出版报纸

1. 向某家报社订阅报纸，只要他们有新报纸出版，就会给你送来。只要你是他们的订阅客户，你就一直会收到新报纸。
2. 当你不想再看报纸的时候，取消订阅，他们就不会再送新报纸来。
3. 只要报社还在运营，就会一直有人（或单位）向他们订阅或取消订阅报纸。

## 出版者+订阅者=观察者模式

观察者模式定义了一系列对象之间的一对多的关系，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。

这里来简单实现一个观察者模式的类

```
// 出版社
function Observer() {
  this.dep = [];

  register(fn) {
    this.dep.push(fn)
  }

  notify() {
    this.dep.forEach(item => item())
  }
}

const reader = new Observer();
// 每来一个订阅者就注册一个想执行的函数
reader.register(() => {'console.log("call daisy")'})
reader.register(() => {'console.log("call anny")'})
reader.register(() => {'console.log("call sunny")'})

// 最后新书籍出版之后，通知所有的客户
wantCake.notify()
```

## 2.1 Object.defineProperty

相信你既然看到源码了，这个也已经是不必介绍了。

VUE给data里所有的属性加上set,get这个过程就叫做Reactive化。

本文使用 mdnice 排版

关注下面的标签，发现更多

# 学习vue源码 (13) 手写 \$nextTick



## 1. 概述

nextTick接收一个回调函数作为参数，它的作用是将回调延迟到下次**DOM**更新周期之后执行。

它与全局Vue.nextTick的原理时一样的。

可能有同学对将回调延迟到下次**DOM**更新周期之后执行不太理解，

其实它的意思就是

在Vue.js中，当状态发生改变时，watcher会得到通知，然后触发虚拟DOM的渲染过程。而watcher触发渲染这个操作并不是同步的，而是异步的。Vue.js中有一个队列（即callbacks），每当需要渲染时，会将watcher推送到这个队列中，在下一次事件循环中再让watcher触发渲染的过程。举个例子

```
mounted: function () {
  this.message = "abc"
  this.$nextTick(function () {
    console.log(this.message)
  })
}
```

上面代码中 `this.message = "abc"` 使得watcher会得到通知，然后触发渲染Dom操作，然后把它放入了队列callback中，

此时callbacks = [渲染Dom操作]，

然后

```
this.$nextTick(function () {  
  console.log(this.message)  
})
```

又把打印message这个操作 放入callbacks中。

此时callbacks= [渲染Dom操作,打印message]

最后再把 依次执行**callbacks**里的操作这个操作 包装成 异步任务。

包装成 异步任务 很简单，作为setTimeout，或promise.then的回调就行

不懂watcher是什么的同学，推荐先看学习vue源码 (12) 大白话谈响应式原理。

在这里 我必须先事先强调 **nextTick**在vue中有两种用途：

- 一种就是 vue内部 使用nextTick，把 渲染**Dom**操作 这个操作 放入到callbacks 中，
- 一种是把nextTick 给开发者使用，也就是 我们经常使用的 `$nextTick`，然后把我们传入 `$nextTick` 的回调函数放到callbacks中。

刚好这两种用途 对应上面我们一开始举的例子中的两个操作。

## 2. 为什么Vue.js使用异步更新队列

Vue.js2.0开始使用虚拟DOM进行渲染，变化侦测的通知只发送到组件，组件内用到的所有状态的变化都会通知到同一个watcher，然后虚拟DOM会对整个组件进行“比对 (diff)”并更改DOM。

也就是说，如果在同一轮事件循环中有两个数据发生了变化，那么组件的watcher会受到两份通知，从而进行两次渲染。事实上，并不需要渲染两次，虚拟DOM会对整个组件进行渲染，所有只需要等所有状态都修改完毕后，一次性将整个组件的DOM渲染到最新即可。

要解决整个问题，Vue.js的实现方式是将收到通知的watcher实例添加到队列中缓存起来，并且在添加到队列之前检查其中是否已经存在相同的watcher，只有不存在时，才将watcher实例添加到队列中。然后在下一次事件循环 (event loop) 中，Vue.js会让队列中的watcher触发渲染流程并清

空队列。这样就可以保证即使在同一事件循环中有两个状态发生改变，watcher最后也只执行一次渲染流程。

举个例子，假如

```
export default {
  data () {
    return {
      msg1: 1,
      msg2: 2,
      msg3: 3
    }
  },
  mounted () {
    this.msg1 = 0
    this.msg2 = 0
    this.msg3 = 0
  },
}
```

如果我们不使用异步队列，那上面的例子中，

- `this.msg1 = 0` 会触发该组件更新视图，
- `this.msg2 = 0` 会触发该组件更新视图
- `this.msg3 = 0` 会触发该组件更新视图

显然，就使得同一个组件更新了三次视图。

注意：同一个组件里的任何一个数据发生改变都会触发整个组件Dom的更新。

当我们使用异步队列后。

- `this.msg1 = 0` 把更新该组件Dom的操作放入callbacks中，
- `this.msg2 = 0` 发现 callbacks 已经有该组件的更新操作，没事了。
- `this.msg3 = 0` 发现 callbacks 已经有该组件的更新操作，没事了。

因此，当callbacks 被包装成异步任务执行时，该组件就只会更新一次DOM，这对性能的提升是多么的高啊啊！！！

了解nextTick原理前，我们需要有是事件循环和执行栈的预备知识，不了解的可以看[这一次，彻底弄懂 JavaScript 执行机制（别还不知道什么是宏任务，什么是微任务）](#)。

## 三 什么是执行栈

当执行一个方法时，Javascript会生成一个与这个方法对应的执行环境，又叫执行上下文。这个执行环境中有这个方法的私有作用域、上层作用域的指向、方法的参数、私有作用域中定义的变量以及this对象。这个执行环境会被添加到一个栈中，这个栈就是执行栈。

如果在这个方法的代码中执行到了一行函数调用语句，那么Javascript会生成这个函数的执行环境并将其添加到执行栈中，然后进入这个执行环境继续执行其中的代码。执行完毕并返回结果后，Javascript会退出执行环境并把这个执行环境从栈中销毁，回到上一个方法的执行环境。这个过程反复进行，直到执行栈中的代码全部执行完毕。这个执行环境的栈就是执行栈。

下次DOM更新周期的意思其实是下次微任务执行时更新DOM。而 `vm.$nextTick` 其实是将回调添加到微任务中。只有在特殊情况下才会降级成宏任务，默认会添加到微任务中。

因此，如果使用 `vm.$nextTick` 来获取更新后的DOM，则需要注意顺序的问题。因为不论是更新DOM的回调还是使用 `vm.$nextTick` 注册的回调，都说向微任务队列中添加任务，所以哪个任务先添加到队列中，就先执行哪个任务。

事实上，更新DOM的回调也是使用 `vm.$nextTick` 来注册到微任务中的。这个我们在上面也强调过

如果想要在 `vm.$nextTick` 中获取更新后的DOM，则一定要在更改数据的后面使用 `vm.$nextTick` 注册回调。

```
new Vue({
  methods:{
    example:function(){
      // 先修改数据 -->
      this.message = 'changed';
      // 然后使用nextTick注册回调 -->
      this.$nextTick(function(){
        // DOM现在更新了 -->
      })
    }
  }
})
```

如果先使用 `vm.$nextTick` 注册回调，然后修改数据，则在微任务队列中先执行使用 `vm.$nextTick` 注册的回调，然后执行更新DOM的回调。所以在回调中得不到最新的DOM，因为此时DOM还没有更新。

在事件循环中，必须当微任务队列中的事件都执行完之后，才会从宏任务队列中取出一个事件执行下一轮，所以添加到微任务队列中的任务的执行时机优先于向宏任务队列中添加的任务。

```
new Vue({
  methods: {
    example:function(){
      // 先使用setTimeout向宏任务中注册回调 -->
      setTimeout(_=>{
        // DOM现在更新了 -->
        },0)
      // 然后修改数据向微任务中注册回调 -->
      this.message = 'changed';
    }
  }
})
```

`setTimeout` 属于宏任务，使用它注册的回调会加入到宏任务中。宏任务的执行要比微任务晚，所以即便是先注册，也是先更新DOM后执行`setTimeout`中设置的回调。

## 4. vm.\$nextTick原理

`vm.$nextTick` 和全局 `Vue.nextTick` 是相同的，所以 `nextTick` 的具体实现并不是在 `Vue` 原型上的 `nextTick` 方法中，而是抽象成了 `nextTick` 方法供两个方法共用。

```
import { nextTick } from '../util/index'

Vue.prototype.$nextTick = function(fn){
  return nextTick(fn, this);
}
```

`Vue` 原型上的 `$nextTick` 方法只是调用了 `nextTick` 方法，具体实现其实在 `nextTick` 中。

由于 `vm.$nextTick` 会将回调添加到任务队列中延迟执行，所以在回调执行前，如果反复调用 `vm.$nextTick`，`Vue.js` 并不会反复将回调添加到任务队列中，只会向任务队列中添加一个任务。(这个我们在上面说过了)

Vue.js内部有一个列表用来存储 `vm.$nextTick` 参数中提供的回调。在一轮事件循环中，`vm.$nextTick` 只会向任务队列添加一个任务，多次使用 `vm.$nextTick` 只会将回调添加到回调列表中缓存起来。当任务触发时，依次执行列表中的所有回调并清空列表。

### (5) nextTick方法的实现方式

```
<!-- 回调列表 -->
const callbacks = [];
let pending = false;

<!-- 回调列表 -->
const callbacks = [];
let pending = false;

<!-- 执行所有回调并清空列表 -->
function flushCallbacks(){
  pending = false;
  const copies = callbacks.slice(0);
  callbacks.length = 0;
  for(let i = 0;i<copies.length;i++){
    copies[i]();
  }
}

let microTimerFunc;
const p = Promise.resolve();
<!-- 添加微任务 -->
microTimerFunc = () =>{
  p.then(flushCallbacks)
}

export function nextTick(cb,ctx){
  <!-- 将回调加入回调队列 -->
  callbacks.push(()=>{
    if(cb){
      cb.call(ctx);
    }
  })
  <!-- 第一次进入，添加微任务 -->
  if(!pending){
    pending = true;
    microTimerFunc();
  }
}
<!-- 测试一下 -->
nextTick(function(){
```

```
console.log(this.name); // Berwin
}, {name: 'Berwin'})
```

解释上面代码

1、通过数组 callbacks 来存储用户注册的回调。

2、声明了变量 pending 来标记是否已经向队列中添加一个任务了。每当向任务队列中插入任务时，将 pending 设置为 true，每当任务被执行时将 pending 设置为 false，这样就可以通过 pending 的值来判断是否需要向任务队列中添加任务。

3、函数 flushCallbacks，即被注册的那个任务。当这个函数被触发时，会将 callbacks 中的所有函数依次执行，然后清空 callbacks，并将 pending 设置为 false。即一轮事件循环中， flushCallbacks 只会执行一次。

4、microTimerFunc 函数，它的作用是使用 Promise.then 将 flushCallbacks 添加到微任务队列中。这个其实是我们所说的包装成异步。

5、执行 nextTick 函数注册回调时，首先将回调函数添加到 callbacks 中，然后使用 pending 判断是否需要向任务队列中新增任务。

在 Vue.js 2.4 版本之前，nextTick 方法在任何地方都使用微任务，但是微任务的优先级太高，在某些场景下可能会出现问题。所以 Vue.js 提供了在特殊场合下可以强制使用宏任务的方法。

```
<!-- 回调列表 -->
const callbacks = [];
let pending = false;

<!-- 执行所有回调并清空列表 -->
function flushCallbacks(){
  pending = false;
  const copies = callbacks.slice(0);
  callbacks.length = 0;
  for(let i = 0; i < copies.length; i++){
    copies[i]();
  }
}

<!-- 微任务 -->
let microTimerFunc;
<!-- 宏任务 -->
let macroTimerFunc = function(){...}
```

```

<!-- 使用宏任务标识 -->
let userMacroTask = false;
const p = Promise.resolve();
<!-- 添加微任务 -->
macroTimerFunc = () =>{
  p.then(flushCallbacks)
}

export function withMacroTask(fn){
  return fn._withTask || (fn._withTask = function(){
    userMacroTask = true;
    const res = fn.apply(null,arguments);
    userMacroTask = false;
    return res;
  )));
}

export function nextTick(cb,ctx){
  <!-- 将回调加入回调队列 -->
  callbacks.push(()=>{
    if(cb){
      cb.call(ctx);
    }
  })
  <!-- 第一次进入，添加微任务 -->
  if(!pending){
    pending = true;
    <!-- 添加宏任务代码 -->
    if(userMacroTask){
      macroTimerFunc();
    }else{
      microTimerFunc();
    }
  }
}

```

1、新增了withMacroTask函数，它的作用是给回调函数做一层包装，保证在整个回调函数执行过程中，如果修改了状态（数据），那么更新DOM的操作会被推到宏任务队列中，也就是说，更新DOM的执行时间会晚于回调函数的执行时间。

2、withMacroTask先将变量userMacroTask设置为true，然后执行回调，如果这时候回调中修改了数据（触发了更新DOM的操作），而userMacroTask是true，那么更新DOM的操作会被推送到宏任务队列中。当回调执行完毕后，将userMacroTask恢复为false。

3、被withMacroTask包裹的函数所使用的所有 `vm.$nextTick` 方法都会将回调添加到宏任务队列中，其中包括状态被修改后触发的更新DOM的回调和用户自己使用 `vm.$nextTick` 注册的回调

等。

## macroTimerFunc如何将回调添加到宏任务队列中

Vue.js 优先使用 `setImmediate`，但是它存在兼容性问题，只能在 IE 中使用，所以使用 `MessageChannel` 作为备选方案。如果浏览器也不支持 `MessageChannel`，那么最后会使用 `setTimeout` 来将回调添加到宏任务队列中。

```
<!-- setImmediate -->
if(typeof setImmediate !=='undefined' && isNative(setImmediate)){
  macroTimerFunc = () =>{
    setImmediate(flushCallbacks);
  }
<!-- MessageChannel -->
}else if(typeof MessageChannel !== 'undefined' &&(isNative(MessageChannel)||

  MessageChannel.toString()=='[Object MessageChannelConstructor]')){

  const channel = new MessageChannel();
  const port = channel.port2;
  channel.port1.onmessage = flushCallbacks;
  macroTimerFunc = () =>{
    port.postMessage(1);
  }
}else{
<!-- setTimeout -->
  macroTimerFunc = () =>{
    setTimeout(flushCallbacks,0);
  }
}
```

`microTimerFunc` 的实现原理是使用 `Promise.then`，但并不是所有浏览器都支持 `Promise`，当不支持时，会被降级成 `macroTimerFunc`。

```
if(typeof Promise !== 'undefined' && isNative(Promise)){

  const p = Promise.resolve();
  microTimerFunc = () =>{
    p.then(flushCallbacks);
  }
}else{
```

```
microTimerFunc = macroTimerFunc;
}
```

官方文档中有一句话：如果没有提供回调且在支持Promise的环境中，则返回一个Promise。

```
this.$nextTick()
.then(function(){
//DOM更新了
}))
```

要实现这个功能，需要在nextTick中进行判断，如果没有提供回调且当前环境支持Promise，那么返回Promise，并且在callbacks中添加一个函数，当这个函数执行时，执行Promise的resolve即可。

```
export function nextTick(cb,ctx){
let _resolve;
<!-- 将回调加入回调队列 -->
callbacks.push(()=>{
if(cb){
cb.call(ctx);
}else if(_resolve){
_resolve(ctx);
}
})
<!-- 第一次进入，添加微微任务 -->
if(!pending){
pending = true;
<!-- 添加宏任务代码 -->
if(userMacroTask){
macroTimerFunc();
}else{
microTimerFunc();
}
}
if(!cb && typeof Promise !== 'undefined'){
return new Promise(resolve =>{
_resolve = resolve;
})
```

```
    })  
}  
}
```

## 5. 完整的代码

```
<!-- 回调列表 -->  
const callbacks = [];  
let pending = false;  
  
<!-- 执行所有回调并清空列表 -->  
function flushCallbacks(){  
  pending = false;  
  const copies = callbacks.slice(0);  
  callbacks.length = 0;  
  for(let i = 0;i<copies.length;i++){  
    copies[i]();  
  }  
}  
<!-- 添加微任务的函数 -->  
let microTimerFunc;  
<!-- 添加宏任务的函数 -->  
let macroTimerFunc;  
<!-- 使用宏任务标识 -->  
let userMacroTask = false;  
<!-- 添加宏任务macroTimerFunc实现 -->  
<!-- setImmediate -->  
if(typeof setImmediate !=='undefined' && isNative(setImmediate)){  
  macroTimerFunc = () =>{  
    setImmediate(flushCallbacks);  
  }  
<!-- MessageChannel -->  
}else if(typeof MessageChannel !== 'undefined' &&(isNative(MessageChannel)||  
  MessageChannel.toString()=='[Object MessageChannelConstructor]')){  
  const channel = new MessageChannel();  
  const port = channel.port2;  
  channel.port1.onmessage = flushCallbacks;  
  macroTimerFunc = () =>{  
    port.postMessage(1);  
  }  
}else{  
<!-- setTimeout -->  
  macroTimerFunc = () =>{  
    setTimeout(flushCallbacks,0);  
  }  
}
```

```
}

<!-- 添加微任务microTimerFunc实现 -->
<!-- 支持Promise -->
if(typeof Promise !== 'undefined' && isNative(Promise)){
  const p = Promise.resolve();
  microTimerFunc = () =>{
    p.then(flushCallbacks);
  }
} else{
  <!-- 不支持Promise降级为宏任务 -->
  microTimerFunc = macroTimerFunc;
}

<!-- 将回调包在这个函数中，将任务加入到宏任务中 -->
export function withMacroTask(fn){
  return fn._withTask || (fn_withTask = function(){
    userMacroTask = true;
    const res = fn.apply(null,arguments);
    userMacroTask = false;
    return res;
  })
}

export function nextTick(cb,ctx){
  let _resolve;
  <!-- 将回调加入回调队列 -->
  callbacks.push(()=>{
    if(cb){
      cb.call(ctx);
    } else if(_resolve){
      _resolve(ctx);
    }
  })
  <!-- 第一次进入，添加微任务 -->
  if(!pending){
    pending = true;
    <!-- 添加宏任务代码 -->
    if(userMacroTask){
      macroTimerFunc();
    } else{
      microTimerFunc();
    }
  }
  <!-- nextTick无回调且支持Promise返回Promise -->
  if(!cb && typeof Promise !== 'undefined'){
    return new Promise(resolve =>{
      _resolve = resolve;
    })
  }
}
```

本文使用 mdnice 排版

# 学习vue源码（14）深入学习diff



## 大白话简述

这一节，先对diff进行简单的描述，不会出现任何的源码，只是为了帮助大家建立一种思路，了解下 Diff 的大概内容。

### 1、Diff 的作用

2、Diff 的做法

3、Diff 的比较逻辑

4、简单的例子

## 1. Diff 作用

Diff 的出现，就会为了减少更新量，找到最小差异部分DOM，只更新差异部分DOM就好了

这样消耗就会小一些

数据变化一下，没必要把其他没有涉及的没有变化的DOM 也替换了

## 2. Diff 做法

Vue 只会对新旧节点中 父节点是相同节点 的 那一层子节点 进行比较

也可以说成是

只有两个新旧节点是相同节点的时候，才会去比较他们各自的子节点

最大的根节点一开始可以直接比较

这也叫做 同层级比较，并不需要递归，虽然好像降低了一些复用性，也是为了避免过度优化，是一种很高效的 Diff 算法

## 新旧节点是什么？

所有的 新旧节点 指的都是 Vnode 节点，Vue 只会比较 Vnode 节点，而不是比较 DOM

因为 Vnode 是 JS 对象，不受平台限制，所以以它作为比较基础，代码逻辑后期不需要改动

拿到比较结果后，根据不同平台调用相应的方法进行处理就好了

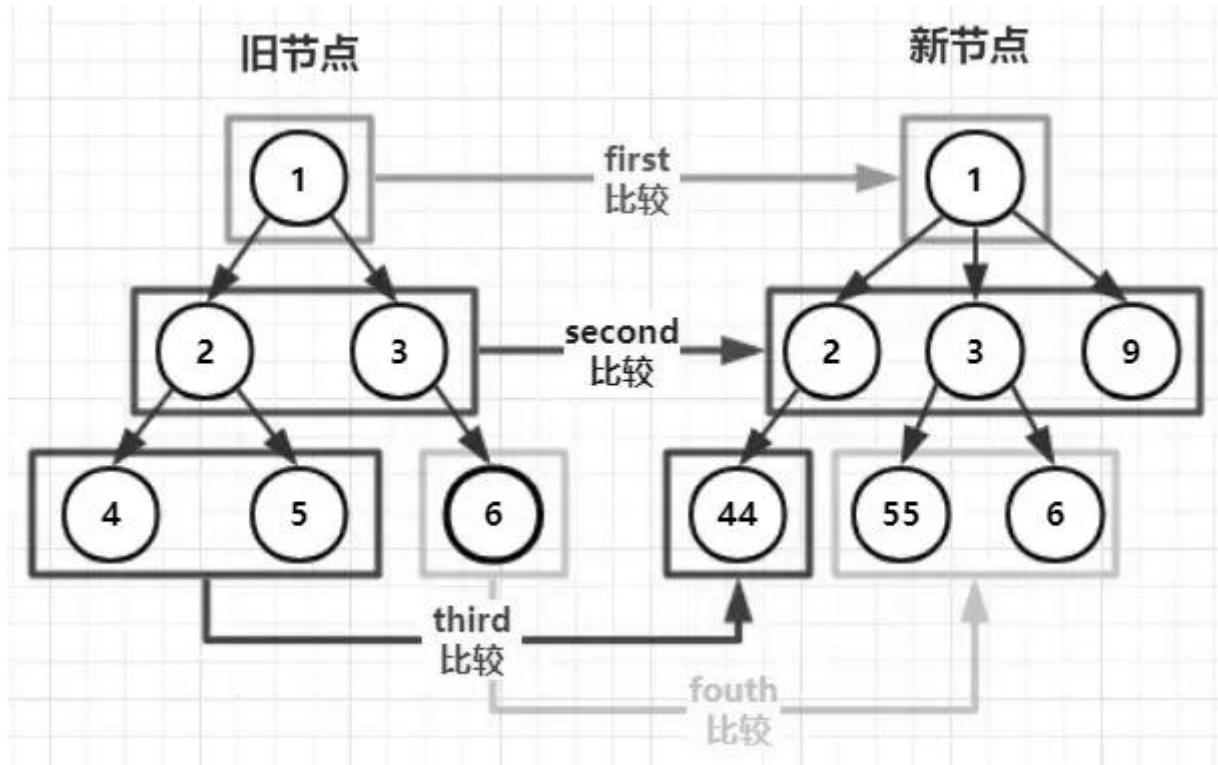
父节点是相同节点是什么意思？

比如下图出现的 四次比较（从 first 到 fouth），他们的共同特点都是有 相同的父节点

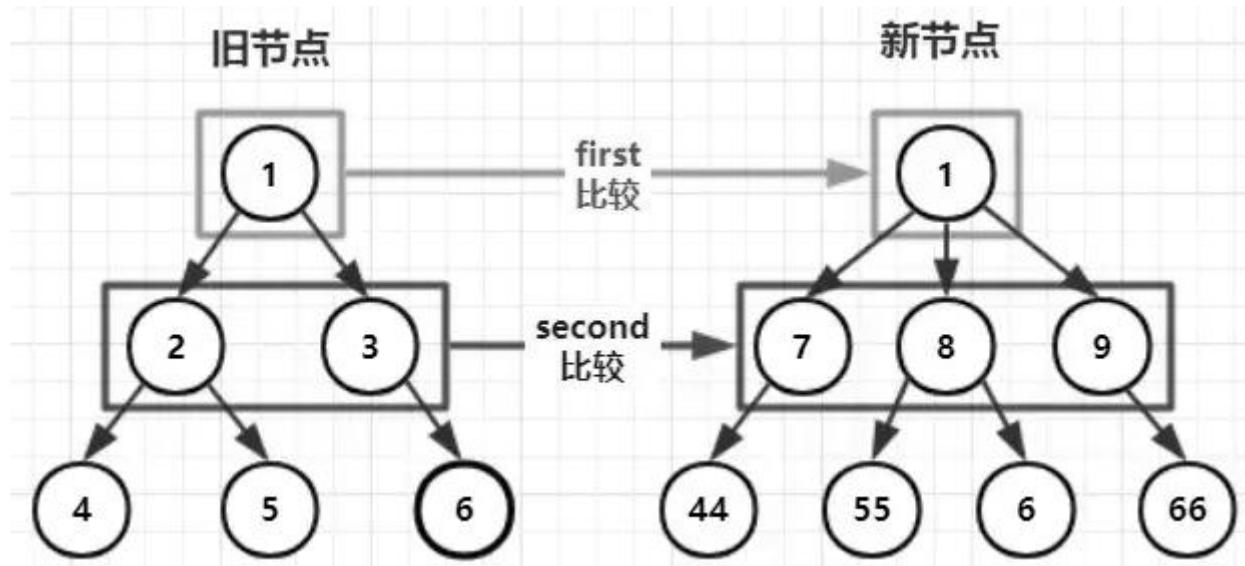
比如 蓝色方的比较，新旧子节点的父节点是相同节点 1

比如 红色方的比较，新旧子节点的父节点都是 2

所以他们才有比较的机会



而下图中，只有两次比较，就是在 蓝色方 比较中，并没有相同节点，所以不会再进行下级子节点比较



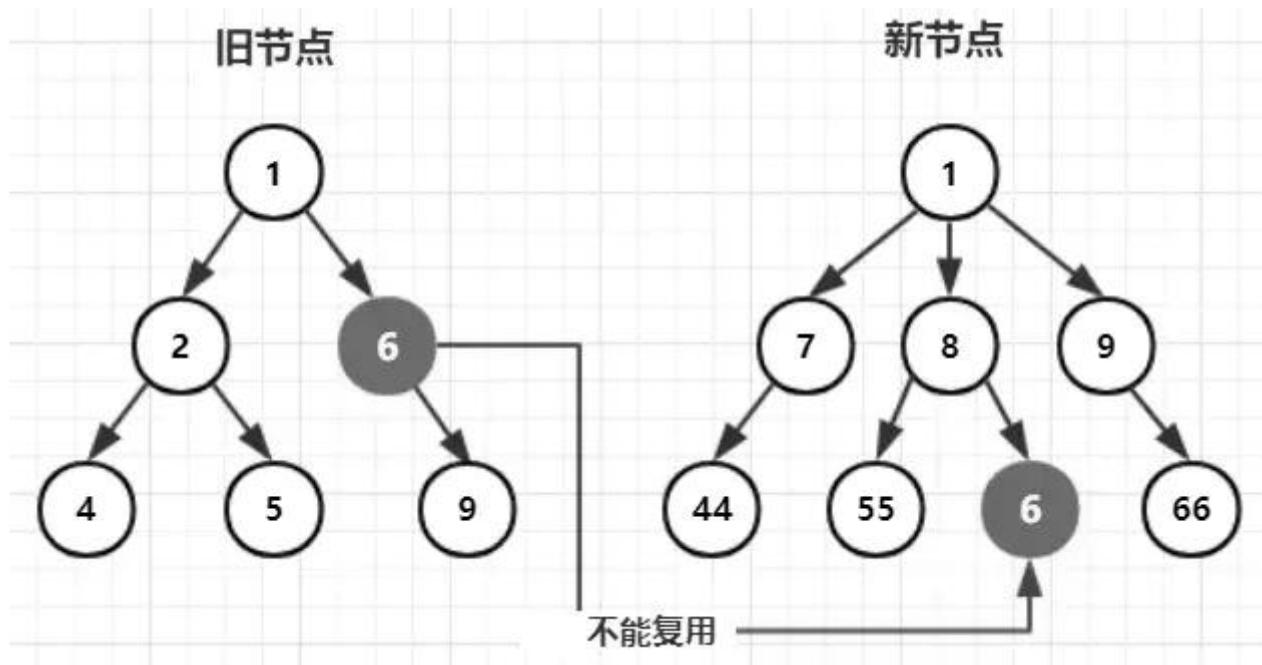
### 3. Diff 比较逻辑

Diff 比较的内核是 节点复用，所以 Diff 比较就是为了在 新旧节点中 找到 相同的节点

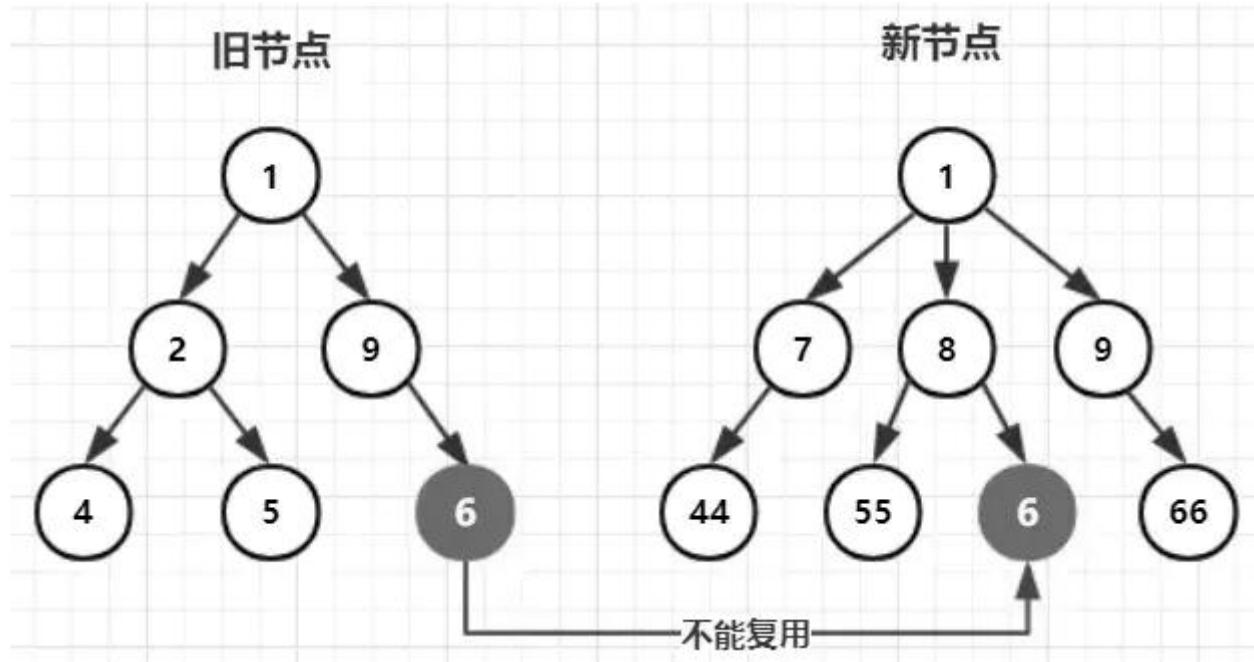
这个的比较逻辑是建立在上一步说过的同层比较基础之上的

所以说，节点复用，找到相同节点并不是无限制递归查找

比如下图中，的确 旧节点树 和 新节点树 中有相同节点 6，但是然并卵，旧节点6并不会被复用

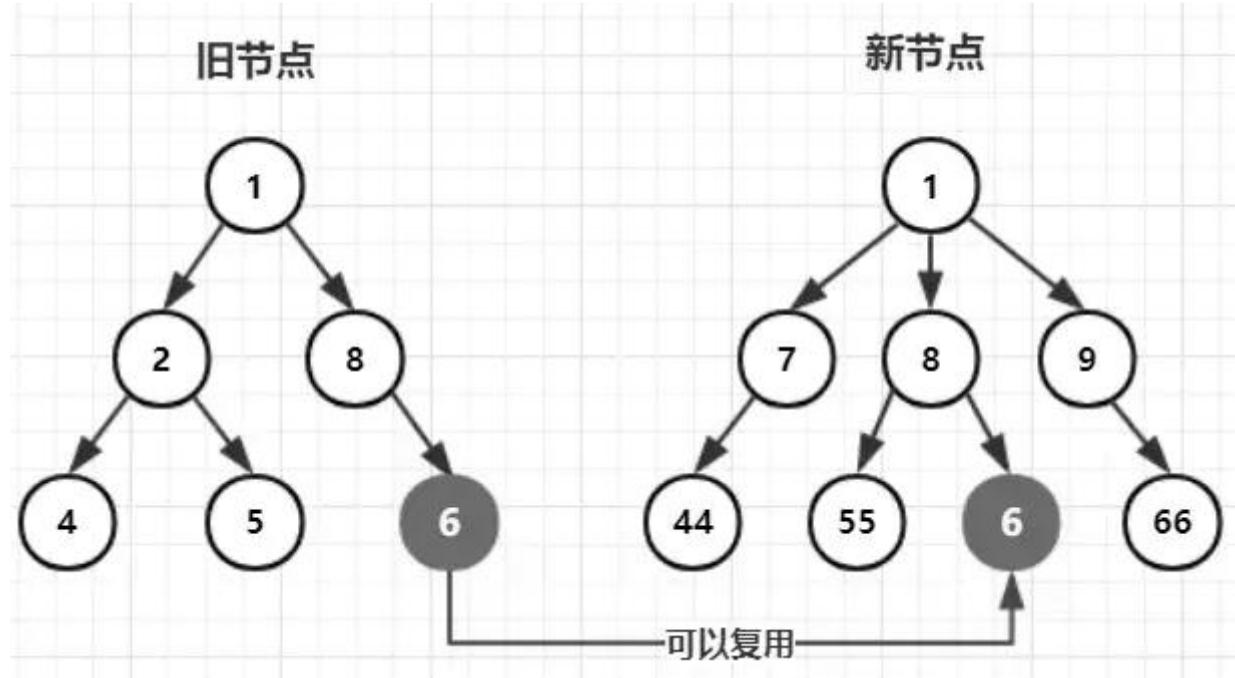


就算在同一层级，然而父节点不一样，依旧然并卵



只有这种情况的节点会被复用，相同父节点 8

下面说说 Diff 的比较逻辑



1、能不移动，尽量不移动

2、没得办法，只好移动

### 3、实在不行，新建或删除

比较处理流程是下面这样

在新旧节点中

1、先找到 不需要移动的相同节点，消耗最小

2、再找相同但是需要移动的节点，消耗第二小

3、最后找不到，才会去新建删除节点，保底处理

比较是为了修改**DOM** 树

其实这里存在 三种树，一个是 页面**DOM** 树，一个是 旧**VNode** 树，一个是 新 **Vnode** 树

页面**DOM** 树 和 旧**VNode** 树 节点一一对应的

而 新**Vnode** 树则是表示更新后 页面**DOM** 树 该有的样子

这里把 旧**Vnode** 树 和 新**Vnode**树 进行比较的过程中

不会对这两棵**Vode**树进行修改，而是以比较的结果直接对 真实**DOM** 进行修改

比如说，在 旧 **Vnode** 树同一层中，找到 和 新**Vnode** 树 中一样但位置不一样节点

此时需要移动这个节点，但是不是移动 旧 **Vnode** 树 中的节点

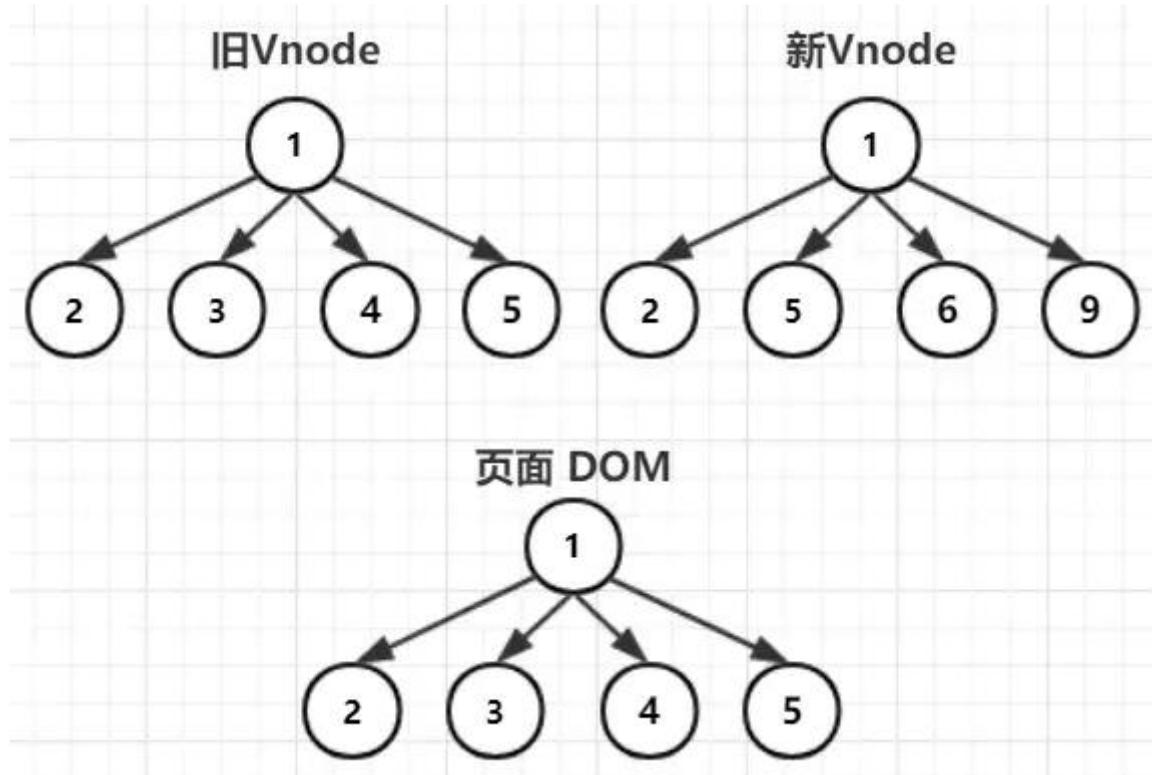
而是 直接移动 **DOM**

总的来说，新旧 **Vnode** 树是拿来比较的，页面**DOM** 树是拿来根据比较结果修改的

如果你有点懵，我们就来就简单说个例子

## 4. Diff 简单例子

比如下图存在这两棵 需要比较的新旧节点树 和一棵 需要修改的页面 DOM树

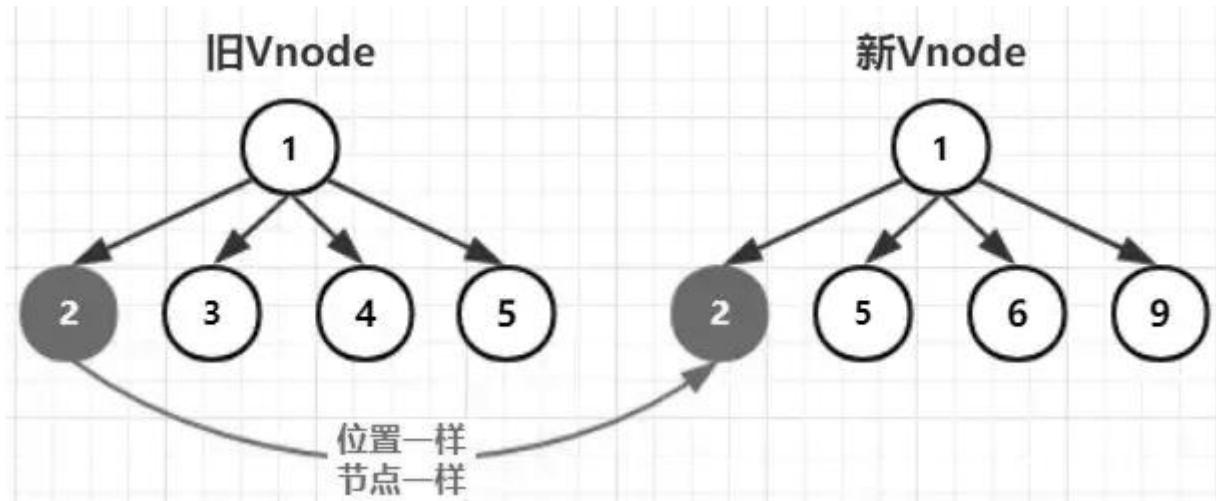


第一轮比较开始

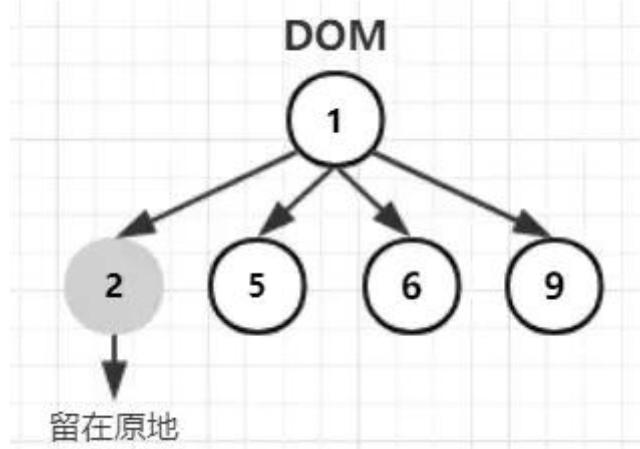
因为父节点都是 1，所以开始比较他们的子节点

按照我们上面的比较逻辑，所以先找 相同 && 不需移动 的点

毫无疑问，找到 2



拿到比较结果，这里不用修改DOM，所以 DOM 保留在原地

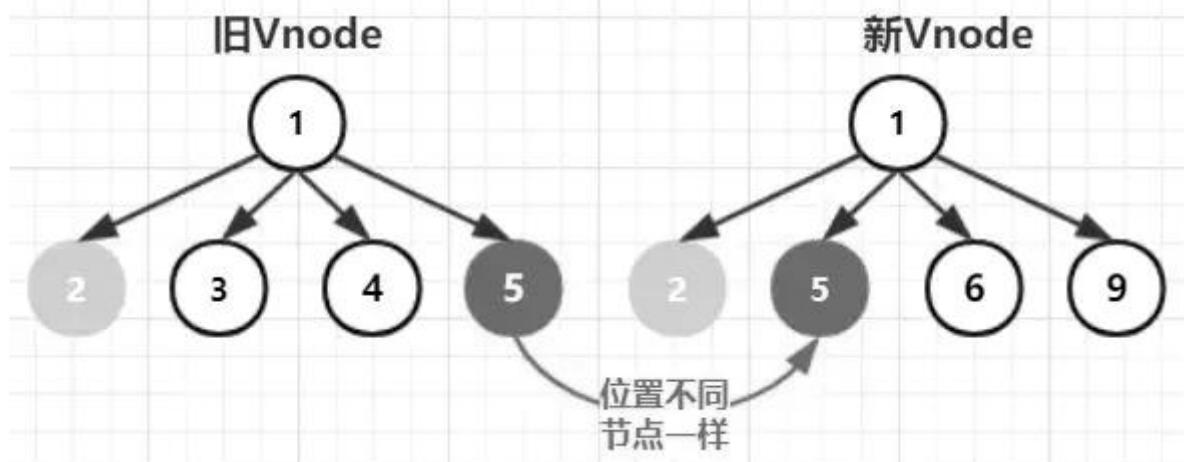


第二轮比较开始

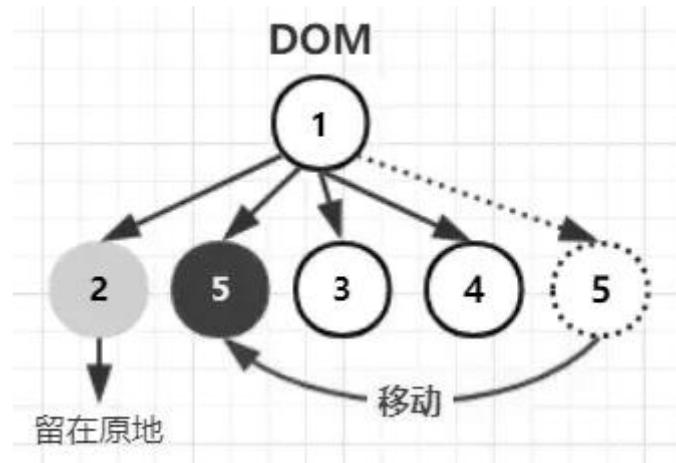
然后，没有相同 && 不需移动 的节点了

只能第二个方案，开始找相同的点

找到 节点5，相同但是位置不同，所以需要移动



拿到比较结果，页面 DOM 树需要移动DOM 了，不修改，原样移动

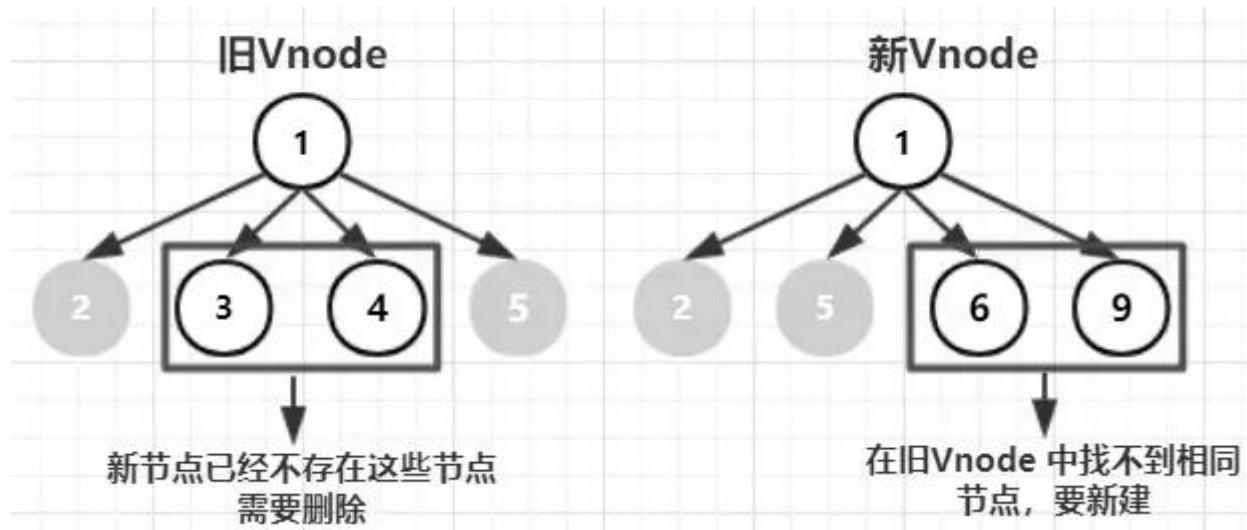


第三轮比较开始

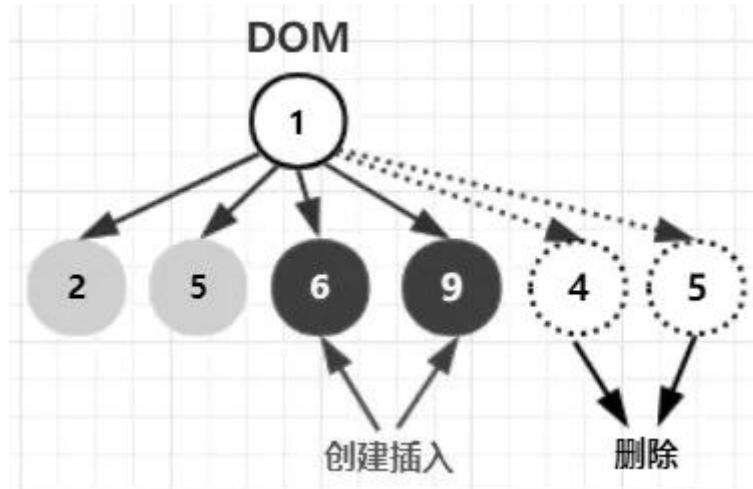
继续，哦吼，相同节点也没得了，没得办法了，只能创建了

所以要根据 新Vnode 中没找到的节点去创建并且插入

然后旧Vnode 中有些节点不存在 新VNode 中，所以要删除



于是开始创建节点 6 和 9，并且删除节点 3 和 4



然后页面就完成更新啦

怎么样，有没有感兴趣，感兴趣就看源码吧哈哈

## 2. 从新建实例到开始diff

这一节不会对源码深入研究，而是跟着例子走一遍流程。

本节很短

先对整个流程有个把握，再仔细去探索 Diff 的思想



首先，当你新建实例的时候，比如这样

```
new Vue({  
  el: document.getElementsByTagName("div")[0],  
  data(){  
    return {  
      obj:{ name:1111 },  
    }  
  }  
})
```

你调用一个 Vue 函数，所以来看下 Vue 函数

```
function Vue() {  
  ... 已省略其他  
  
  new Watcher(function() {  
  
    vm._update(vm._render());  
  })  
  
  ... 已省略其他  
}
```

函数中做了两件事

1、为实例新建一个 watcher

2、为 watcher 绑定更新回调（就是 new Watcher 传入的 function ）

每个实例都会有一个专属的 watcher，而绑定的回调，在页面更新时会调用

我们现在来看下简化的 Watcher 的源码

```
function Watcher(expOrFn){  
  
    this.getter = expOrFn;  
  
    this.get();  
  
}  
  
Watcher.prototype.get = function () {  
  
    this.getter()  
  
}
```

watcher 会保存更新回调，并且在新建 watcher 的时候就会立刻调用一遍更新回调

现在我们继续看 更新回调的内容

```
vm._update(vm._render());
```

如果你看到之前的文章应该知道这两个函数的作用

现在就来简单说一下

### vm.\_render

生成页面模板对应的 Vnode 树，比如

```
<div>  
  <span></span>  
  {{num}}  
</div>
```

生成的 Vnode 树是（其中num的值是111）

```
{  
  tag: "div",  
  
  children:[{  
    tag: "span"  
  
    },{  
      tag: undefined,  
  
      text: "111"  
  
    }]  
}
```

这一步是通过 compile 生成的，具体的话可以简单看下 [学习vue源码（6）熟悉模板编译原理](#)

### vm.\_update

比较旧Vnode 树 和 vm.\_render 生成的新 Vnode 树 进行比较

比较完后，更新页面的DOM，从而完成更新

ok，我们看下源码

```
Vue.prototype._update = function(vnode) {  
  
  var vm = this;  
  
  var prevEl = vm.$el;  
  
  var prevVnode = vm._vnode;
```

```
vm._vnode = vnode;

// 不存在旧节点

if (!prevVnode) {

  vm.$el = vm.__patch__(
    vm.$el, vnode,
    vm.$options._parentElm,
    vm.$options._refElm
  );
}

else {

  vm.$el = vm.__patch__(
    prevVnode, vnode
  );
}
```

解释其中几个点

## 1 vm.\_vnode

这个属性保存的就是当前 Vnode 树

当页面开始更新，而生成了新的 Vnode 树之后

这个属性则会替换成新的Vnode

所以保存在这里，是为了方便拿到 旧 Vnode 树

## 2 vm.patch

是的，没有错，你在两处地方看到这个东西

这个东西就是 Diff 的主要内容，内有乾坤，内容很多，不会在这里说，毕竟今天只探索流程

但是要看看这个东西怎么来的

```
var patch = createPatchFunction();

Vue.prototype.__patch__ = patch ;
```

嗯，是经过一个 `createPatchFunciton` 生成的

然后赋值到 `Vue` 的原型上，所以可以 `vm.patch` 调用喽

我们再来说说 `_update` 函数中出现的那两处 `patch`

### (1) 不存在旧节点

不需要进行比较，直接全部创建

`vm.$el` 保存的是 DOM 节点，如果不存在旧节点，那么 `vm.$el` 此时也是不存在的

而传入 `vm.$el` 为空的时候，`patch` 拿到这个值判断为空的时候，就直接创建DOM，不会去做其他操作了

### (2) 存在旧节点

需要把旧节点和新节点比较，尽量找到最小差异部分，然后进行更新，这部分内容就是 Diff 的重点了，需要花费不少精力的。

好了，这一节就结束了。



### 3. 相关辅助函数

在开始我们的 Diff 主要内容之前，我们先来了解下其中会用的一些辅助函数

本来可以放到 Diff 那里写，但是全部合起来内容又太多

而且这些函数比较具有公用性，就是抽出来用

所以打算独立一节，先预热一下，内容也不多，也挺简单，光看下也会对我们的思维有所帮助

#### 1. 节点操作函数

下面是 Diff 在比较节点时，更新DOM 会使用到的一些函数

本来会有更多，为了看得方便，我把一些合并了

下面会介绍三个函数

`insert`，`createElm`，`createChildren`

简单介绍

`insert` 作用是把节点插入到某个位置

`createElm` 作用是创建DOM 节点

`createChildren` 作用也是创建DOM 节点，但是处理的是一个数组，并且会创建 DOM 节点 和文本节点

下面就来仔细说说这三个方法

### **insert**

这个函数的作用就是 插入节点

但是插入也会分两种情况

1、没有参考兄弟节点，直接插入父节点的子节点末尾

2、有参考兄弟节点，则插在兄弟节点前面

可以说这个函数是 Diff 的基石哈哈

```
function insert(parent, elm, ref) {  
  if (parent) {
```

```
if (ref) {  
  
    if (ref.parentNode === parent) {  
  
        parent.insertBefore(elm, ref);  
    }  
} else {  
    parent.appendChild(elm);  
}  
}  
}  
}
```

## createElm

这个函数的作用跟它的名字一样，就是创建节点的意思

创建完节点之后，会调用 `insert` 去插入节点

你可以看一下，不难

```
function createElm(vnode, parentElm, refElm) {  
  
    var children = vnode.children;  
  
    var tag = vnode.tag;  
  
    if (tag) {  
  
        vnode.elm = document.createElement(tag)  
  
        // 先把 子节点插入 vnode.elm，然后再把 vnode.elm 插入parent  
  
        createChildren(vnode, children);  
  
        // 插入DOM 节点  
    }  
}
```

```
insert(parentElm, vnode.elm, refElm);  
}  
  
else {  
  
    vnode.elm = document.createTextNode(vnode.text);  
  
    insert(parentElm, vnode.elm, refElm);  
}  
}
```

createElm 需要根据 Vnode 来判断需要创建什么节点

## 1、文本节点

当 vnode.tag 为 undefined 的时候，创建文本节点，看下 真实文本vnode

并且，文本节点是没有子节点的

```
▼ 0: VNode  
  children: undefined  
  isStatic: false  
  key: undefined  
  tag: undefined  
  text: "222"  
  child: undefined
```

## 2、普通节点

vnode.tag 有值，那就创建相应的 DOM

但是 该 DOM 可能存在子节点，所以子节点甚至子孙节点，也都是要创建的

所以会调用一个 createChildren 去完成所有子孙节点的创建

### createChildren

这个方法处理子节点，必然是用遍历递归的方法逐个处理的

1 如果子节点是数组，则遍历执行 `createElm` 逐个处理

2 如果子节点的 `text` 属性有数据，则表示这个 `vnode` 是个文本节点，直接创建文本节点，然后插入到父节点中

```
function createChildren(vnode, children) {
  if (Array.isArray(children)) {

    for (var i = 0; i < children.length; ++i) {
      createElm(children[i], vnode.elm, null);
    }
  } else if (
    typeof vnode.text === 'string' ||
    typeof vnode.text === 'number' ||
    typeof vnode.text === 'boolean'
  ) {
    vnode.elm.appendChild(
      document.createTextNode(vnode.text)
    )
  }
}
```

## 服务Diff工具函数

下面的函数是 Vue 专门用来服务 Diff 的，介绍两个

`createKeyToOldIdx`，`sameVnode`

### `createKeyToOldIdx`

接收一个 children 数组，生成 key 与 index 索引对应的一个 map 表

```
function createKeyToOldIdx(
  children, beginIdx, endIdx
) {
  var i, key;
  var map = {};
  for (i = beginIdx; i <= endIdx; ++i) {
    key = children[i].key;
    if (key) {
      map[key] = i;
    }
  }
  return map
}
```

比如你的旧子节点数组是

```
[{
  tag:"div",  key: "key_1"
}, {
  tag:"strong", key:"key_2"
}, {
  tag:"span",   key:"key_4"
}]
```

经过 createKeyToOldIdx 生成一个 map 表 oldKeyToIdx，是下面这样

```
{
  "key_1":0,
  "key_2":1,
  "key_4":2
}
```

把 vnode 的 key 作为属性名，而该 vnode 在 children 的位置 作为 属性值

这个函数在 Diff 中的作用是

判断某个新 vnode 是否在这个旧的 Vnode 数组中，并且拿到它的位置。就是拿到 新 Vnode 的 key，然后去这个 map 表中去匹配，是否有相应的节点，有的话，就返回这个节点的位置

比如

现在我有一个 新子节点数组，一个 旧子节点数组

我拿到 新子节点数组 中的某一个 newVnode，我要判断他是否 和 旧子节点数组 中某个vnode 相同

要怎么判断？？？难道是双重遍历数组，逐个判断 `newVnode.key==vnode.key` ？？

Vue 用了更聪明的办法，使用 旧 Vnode 数组生成一个 map 对象 obj

当 `obj[ newVnode.key ]` 存在的时候，说明 新旧子节点数组都存在这个节点

并且我能拿到该节点在 旧子节点数组 中的位置（属性值）

反之，则不存在

这个方法也给我们提供了在项目中相似场景的一个解决思路，以对象索引查找替代数组遍历

希望大家记住哦

### **sameVnode**

这个函数在 Diff 中也起到非常大的作用，大家务必要记住啊

它的作用是判断两个节点是否相同

这里说的相同，并不是完全一毛一样，而是关键属性一样，可以先看下源码

```
function sameVnode(a, b) {
  return (
    a.key === b.key &&
    a.tag === b.tag &&
    !!a.data === !!b.data &&
    sameInputType(a, b)
  )
}

function sameInputType(a, b) {
  if (a.tag !== 'input') return true
  var i;
  var types = [
    'text', 'number', 'password',
    'search', 'email', 'tel', 'url'
  ]
  var typeA = (i = a.data) && (i = i.attrs) && i.type;
  var typeB = (i = b.data) && (i = i.attrs) && i.type;
  // input 的类型一样，或者都属于基本input类型
  return (
    typeA === typeB ||
    types.indexOf(typeA) > -1 &&
    types.indexOf(typeB) > -1
  )
}
```

判断的依据主要是 三点，key，tag，是否存在 data

这里判断的节点是只是相对于 节点本身，并不包括 children 在内

也就是说，就算data不一样，children 不一样，两个节点还是可能一样

比如下面这两个

```
▼ VNode {tag: "div", data: {...}, ...}
  ▶ children: [VNode]
  ▼ data:
    ▶ attrs: {name: "1", a: 111}
    ▶ __proto__: Object
    key: undefined
    tag: "div"
    text: undefined

  ▼ VNode {tag: "div", data: {...}, ...}
  ▶ children: (2) [VNode, VNode]
  ▼ data:
    ▶ attrs: {dd: "1", b: 111}
    ▶ __proto__: Object
    key: undefined
    tag: "div"
    text: undefined
```

有一种特殊情况，就是 input 节点

input 需要额外判断，两个节点的 type 是否相同

或者

两个节点的类型可以不同，但是必须属于那些 input 类型

sameVnode 的内容就到这里了，但是我不禁又开始思考一个问题

为什么 sameVnode 会这么判断？？

下面纯属个人意淫想法，仅供参考

sameVnode 应用在 Diff，作用是为了判断节点是否需要新建

当两个 新旧vnode 进行 sameVnode 得到 false 的时候，说明两个vnode 不一样，会新建DOM 插入

也就是两个节点从根本上不一样时才会创建

其中会比较 唯一标识符 key 和 标签名 tag，从而得到 vnode 是否一样，这些是毫无疑问的了

但是这里不需要判断 data 是否一样，我开始不太明白

后面想到 data 是包含有一些 dom 上的属性的，所以 data 不一样没有关系

因为就算不一样，他们还是基于同一个 DOM

因为DOM属性的值是可能是动态绑定动态更新变化的，所以变化前后的两个 vnode，相应的 data 肯定不一样，但是其实他们是同一个 Vnode，所以 data 不在判断范畴

但是 data 在新旧节点中，必须都定义，或者都不定义

不存在一个定义，而一个没定义，但是会相同的 Vnode

比如，下面这个就会存在data

```
<div :b="num">  
  </div>
```

这个就不会存在data

```
<div>  
  </div>
```

他们在模板中，肯定是不属于同一个节点

总结

涉及的函数主要分为两类

一类是专门负责操作 DOM 的，insert，createElm，createChildren

这类函数比较通用，就算在我们自己的项目中也可以用得上

一类是专门特殊服务 Diff 的，`createKeyToOldIdx`，`sameVnode`

其中会包含一些项目的解决思路

大家务必先记住一下这几个函数，在下节内容的源码中会频繁出现

到时不会仔细介绍

## 4. Diff 流程

---

Diff 的内容不算多，但是如果要讲得很详细的话，就要说很多了，而且要配很多图

这是 Diff 的最后一节，最重要也是最详细的一节了

所以本节内容很多，先提个内容概览

1、分析 Diff 源码比较步骤

2、个人思考为什么如此比较

3、写个例子，一步步走个Diff 流程

文章很长，也非常详细，如果你对这内容有兴趣的话，也推荐边阅读源码边看

下面开始我们的正文



在之前，我们已经探索了 Vue 是如何从新建实例到开始 diff 的

你应该还有印象，其中 Diff 涉及的一个重要函数就是 `createPatchFunction`

```
var patch = createPatchFunction();
Vue.prototype.__patch__ = patch
```

那么我们就来看下这个函数

### **createPatchFunction**

```
function createPatchFunction() {
  return function patch(
```

```
oldVnode, vnode, parentElm, refElm

) {

    // 没有旧节点，直接生成新节点

    if (!oldVnode) {

        createElement(vnode, parentElm, refElm);

    }

    else {

        // 且是一样 Vnode

        if (sameVnode(oldVnode, vnode)) {

            // 比较存在的根节点

            patchVnode(oldVnode, vnode);

        }

        else {

            // 替换存在的元素

            var oldElm = oldVnode.elm;

            var _parentElm = oldElm.parentNode

            // 创建新节点

            createElement(vnode, _parentElm, oldElm.nextSibling);

            // 销毁旧节点

            if (_parentElm) {

                removeVnodes([oldVnode], 0, 0);

            }

        }

    }

    return vnode.elm

}

}
```

这个函数的作用就是

比较 新节点 和 旧节点 有什么不同，然后完成更新

所以你看到接收一个 oldVnode 和 vnode

处理的流程分为

- 1、没有旧节点
- 2、旧节点 和 新节点 自身一样（不包括其子节点）
- 3、旧节点 和 新节点自身不一样

速度来看下这三个流程了

### 1 .没有旧节点

没有旧节点，说明是页面刚开始初始化的时候，此时，根本不需要比较了

直接全部都是新建，所以只调用 `createElm`

### 2 .旧节点 和 新节点 自身一样

通过 `sameVnode` 判断节点是否一样，这个函数在上节中说过了

旧节点 和 新节点自身一样时，直接调用 `patchVnode` 去处理这两个节点

`patchVnode` 下面会讲到这个函数

在讲 `patchVnode` 之前，我们先思考这个函数的作用是什么？

当两个Vnode自身一样的时候，我们需要做什么？

首先，自身一样，我们可以先简单理解，是 Vnode 的两个属性 `tag` 和 `key` 一样

那么，我们是不知道其子节点是否一样的，所以肯定需要比较子节点

所以，`patchVnode` 其中的一个作用，就是比较子节点

### 3 .旧节点 和 新节点 自身不一样

当两个节点不一样的时候，不难理解，直接创建新节点，删除旧节点

## patchVnode

在上一个函数 `createPatchFunction` 中，有出现一个函数 `patchVnode`

我们思考了这个函数的其中的一个作用是 比较两个Vnode 的子节点

是不是我们想的呢，可以先来过一下源码

```
function patchVnode(oldVnode, vnode) {  
  
  if (oldVnode === vnode) return  
  
  var elm = vnode.elm = oldVnode.elm;  
  
  var oldCh = oldVnode.children;  
  
  var ch = vnode.children;  
  
  // 更新children  
  
  if (!vnode.text) {  
  
    // 存在 oldCh 和 ch 时  
  
    if (oldCh && ch) {  
  
      if (oldCh !== ch)  
  
        updateChildren(elm, oldCh, ch);  
  
    }  
  
    // 存在 newCh 时，oldCh 只能是不存在，如果存在，就跳到上面的条件了  
  
    else if (ch) {  
  
      if (oldVnode.text) elm.textContent = '';  
  
      for (var i = 0; i <= ch.length - 1; ++i) {  
  
    }  
  }  
}
```

```
        createElm(  
            ch[i], elm, null  
        );  
    }  
  
    else if (oldCh) {  
  
        for (var i = 0; i <= oldCh.length - 1; ++i) {  
  
            oldCh[i].parentNode.removeChild(el);  
        }  
  
    }  
  
    else if (oldVnode.text) {  
        elm.textContent = '';  
    }  
}  
  
else if (oldVnode.text !== vnode.text) {  
    elm.textContent = vnode.text;  
}  
}
```

我们现在就来分析这个函数

没错，正如我们所想，这个函数的确会去比较处理子节点

总的来说，这个函数的作用是

1、Vnode 是文本节点，则更新文本（文本节点不存在子节点）

2、Vnode 有子节点，则处理比较更新子节点

更进一步的总结就是，这个函数主要做了两种判断的处理

1、Vnode 是否是文本节点

2、Vnode 是否有子节点

下面我们来看看这些步骤的详细分析

## 1 Vnode是文本节点

当 VNode 存在 text 这个属性的时候，就证明了 Vnode 是文本节点

我们可以先来看看 文本类型的 Vnode 是什么样子

```
▼ 0: VNode
  children: undefined
  isStatic: false
  key: undefined
  tag: undefined
  text: "111"
  child: (...)
```

所以当 Vnode 是文本节点的时候，需要做的就是，更新文本

同样有两种处理

1、当 新Vnode.text 存在，而且和 旧 VNode.text 不一样时

直接更新这个 DOM 的 文本内容

```
elm.textContent = vnode.text;
```

注：textContent 是 真实DOM 的一个属性， 保存的是 dom 的文本，所以直接更新这个属性

2、新Vnode 的 text 为空，直接把 文本DOM 赋值给空

```
elm.textContent = '';
```

## 2 Vnode存在子节点

当 Vnode 存在子节点的时候，因为不知道 新旧节点的子节点是否一样，所以需要比较，才能完成更新

这里有三种处理

1、新旧节点 都有子节点，而且不一样

2、只有新节点

3、只有旧节点

后面两个节点，相信大家都能想通，但是我们还是说一下

## 1 只有新节点

只有新节点，不存在旧节点，那么没得比较了，所有节点都是全新的

所以直接全部新建就好了，新建是指创建出所有新DOM，并且添加进父节点的

## 2 只有旧节点

只有旧节点而没有新节点，说明更新后的页面，旧节点全部都不见了

那么要做的，就是把所有的旧节点删除

也就是直接把DOM 删除

## 3 新旧节点 都有子节点，而且不一样

咦惹，又出现了一个新函数，那就是 updateChildren

预告一下，这个函数非常的重要，是 Diff 的核心模块，蕴含着 Diff 的思想

可能会有点绕，但是不用怕，相信在我的探索之下，可以稍微明白些

同样的，我们先来思考下 updateChildren 的作用

记得条件，当新节点 和 旧节点 都存在，要怎么去比较才能知道有什么不一样呢？

哦没错，使用遍历，新子节点和旧子节点一个个比较

如果一样，就不更新，如果不样，就更新

下面就来验证下我们的想法，来探索一下 updateChildren 的源码

## updateChildren

这个函数非常的长，但是其实不难，就是分了几种处理流程而已，但是一开始看可能有点懵

或者可以先跳过源码，看下分析，或者便看分析边看源码

```
function updateChildren(parentElm, oldCh, newCh) {  
  
    var oldStartIdx = 0;  
  
    var oldEndIdx = oldCh.length - 1;  
  
    var oldStartVnode = oldCh[0];  
  
    var oldEndVnode = oldCh[oldEndIdx];  
  
    var newStartIdx = 0;  
  
    var newEndIdx = newCh.length - 1;  
  
    var newStartVnode = newCh[0];  
  
    var newEndVnode = newCh[newEndIdx];  
  
    var oldKeyToIdx, idxInOld, vnodeToMove, refElm;
```

```
// 不断地更新 OldIndex 和 OldVnode , newIndex 和 newVnode

while (

    oldStartIdx <= oldEndIdx &&

    newStartIdx <= newEndIdx

) {

    if (!oldStartVnode) {

        oldStartVnode = oldCh[++oldStartIdx];

    }

    else if (!oldEndVnode) {

        oldEndVnode = oldCh[--oldEndIdx];

    }

    // 旧头 和新头 比较

    else if (sameVnode(oldStartVnode, newStartVnode)) {

        patchVnode(oldStartVnode, newStartVnode);

        oldStartVnode = oldCh[++oldStartIdx];
        newStartVnode = newCh[++newStartIdx];

    }

    // 旧尾 和新尾 比较

    else if (sameVnode(oldEndVnode, newEndVnode)) {

        patchVnode(oldEndVnode, newEndVnode);

        oldEndVnode = oldCh[--oldEndIdx];
        newEndVnode = newCh[--newEndIdx];

    }

    // 旧头 和 新尾 比较

    else if (sameVnode(oldStartVnode, newEndVnode)) {

        patchVnode(oldStartVnode, newEndVnode);

        // oldStartVnode 放到 oldEndVnode 后面，还要找到 oldEndValue 后面的节点
    }
}
```

```
parentElm.insertBefore(  
    oldStartVnode.elm,  
    oldEndVnode.elm.nextSibling  
);  
  
oldStartVnode = oldCh[++oldStartIdx];  
newEndVnode = newCh[--newEndIdx];  
}  
  
// 旧尾 和新头 比较  
  
else if (sameVnode(oldEndVnode, newStartVnode)) {  
  
    patchVnode(oldEndVnode, newStartVnode);  
  
    // oldEndVnode 放到 oldStartVnode 前面  
  
    parentElm.insertBefore(oldEndVnode.elm, oldStartVnode.elm);  
  
    oldEndVnode = oldCh[--oldEndIdx];  
    newStartVnode = newCh[++newStartIdx];  
}  
  
// 单个新子节点 在 旧子节点数组中 查找位置  
  
else {  
  
    // oldKeyToIdx 是一个 把 Vnode 的 key 和 index 转换的 map  
  
    if (!oldKeyToIdx) {  
        oldKeyToIdx = createKeyToOldIdx(  
            oldCh, oldStartIdx, oldEndIdx  
        );  
    }  
  
    // 使用 newStartVnode 去 OldMap 中寻找 相同节点，默认key存在  
  
    idxInOld = oldKeyToIdx[newStartVnode.key]
```

```
// 新孩子中，存在一个新节点，老节点中没有，需要新建

if (!idxInOld) {

    // 把 newStartVnode 插入 oldStartVnode 的前面

    createElement(
        newStartVnode,
        parentElm,
        oldStartVnode.elm
    );
}

else {

    // 找到 oldCh 中 和 newStartVnode 一样的节点

    vnodeToMove = oldCh[idxInOld];
    if (sameVnode(vnodeToMove, newStartVnode)) {

        patchVnode(vnodeToMove, newStartVnode);

        // 删除这个 index

        oldCh[idxInOld] = undefined;
        // 把 vnodeToMove 移动到 oldStartVnode 前面

        parentElm.insertBefore(
            vnodeToMove.elm,
            oldStartVnode.elm
        );
    }

    // 只能创建一个新节点插入到 parentElm 的子节点中

    else {

```

```
// same key but different element. treat as new element

createElm(
    newStartVnode,
    parentElm,
    oldStartVnode.elm
);

}

}

// 这个新子节点更新完毕，更新 newStartIdx，开始比较下一个

newStartVnode = newCh[++newStartIdx];
}

}

// 处理剩下的节点

if (oldStartIdx > oldEndIdx) {

    var newEnd = newCh[newEndIdx + 1]

    refElm = newEnd ? newEnd.elm : null;
    for (; newStartIdx <= newEndIdx; ++newStartIdx) {

        createElm(
            newCh[newStartIdx], parentElm, refElm
        );
    }
}

// 说明新节点比对完了，老节点可能还有，需要删除剩余的老节点

else if (newStartIdx > newEndIdx) {

    for (; oldStartIdx<=oldEndIdx; ++oldStartIdx) {

        oldCh[oldStartIdx].parentNode.removeChild(el);
    }
}
```

```
}
```

首先要明确这个函数处理的是什么

处理的是 新子节点 和 旧子节点，循环遍历逐个比较

如何 循环遍历？

1、使用 while

2、新旧节点数组都配置首尾两个索引

新节点的两个索引：newStartIdx , newEndIdx

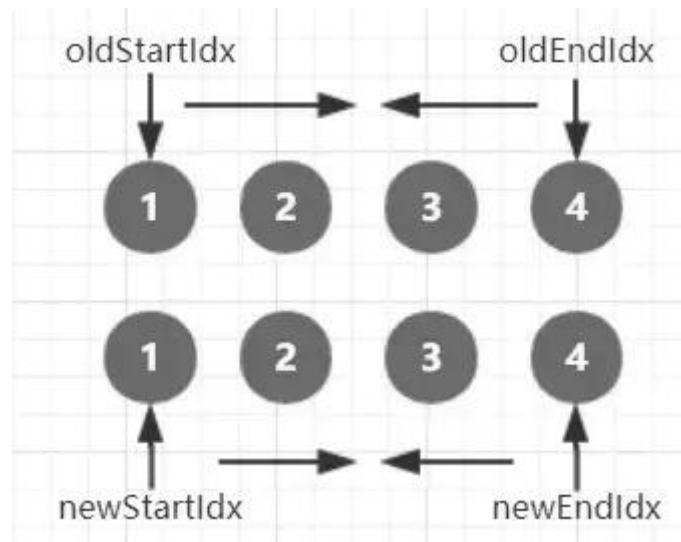
旧节点的两个索引：oldStartIdx , oldEndIdx

以两边向中间包围的形式 来进行遍历

头部的子节点比较完毕，startIdx 就加1

尾部的子节点比较完毕，endIndex 就减1

只要其中一个数组遍历完（ $startIdx < endIdx$ ），则结束遍历



源码处理的流程分为两个

1、比较新旧子节点

2、比较完毕，处理剩下的节点

我们来逐个说明这两个流程

## 1 比较新旧子节点

注：这里有两个数组，一个是 新子Vnode数组，一个旧子Vnode数组

在比较过程中，不会对两个数组进行改变（比如不会插入，不会删除其子项）

而所有比较过程中都是直接 插入删除 真实页面DOM

我们明确一点，比较的目的是什么？

找到 新旧子节点中的 相同的子节点，尽量以 移动 替代 新建 去更新DOM

只有在实在不同的情况下，才会新建

比较更新计划步骤：

首先考虑，不移动DOM

其次考虑，移动DOM

最后考虑，新建 / 删除 DOM

能不移动，尽量不移动。不行就移动，实在不行就新建

下面开始说源码中的比较逻辑

五种比较逻辑如下

1、旧头 == 新头

2、旧尾 == 新尾

3、旧头 == 新尾

4、旧尾 == 新头

5、单个查找

来分析下这五种比较逻辑

**1 旧头 == 新头**

```
sameVnode(oldStartVnode, newStartVnode)
```

当两个新旧的两个头一样的时候，并不用做什么处理

符合我们的步骤第一条，不移动DOM完成更新

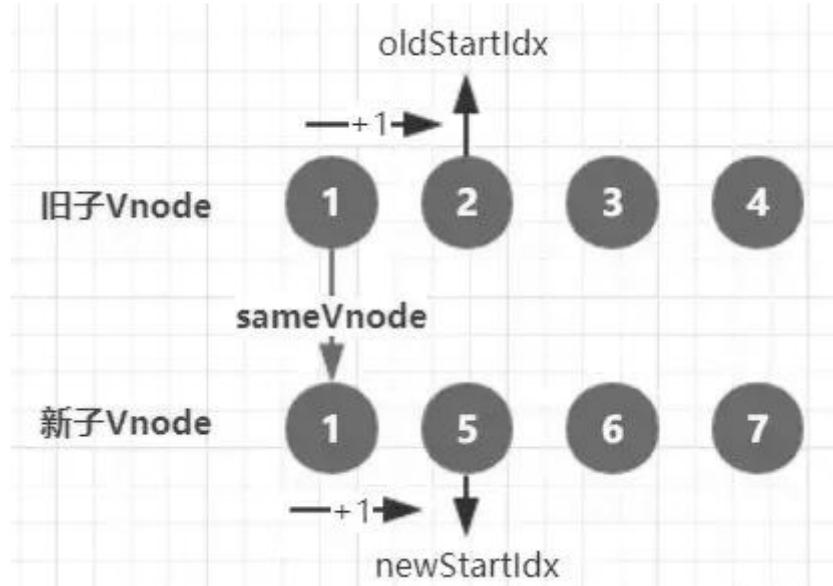
但是看到一句，patchVnode

就是为了继续处理这两个相同节点的子节点，或者更新文本

因为我们不考虑多层DOM 结构，所以 新旧两个头一样的话，这里就算结束了

可以直接进行下一轮循环

```
newStartIdx ++ , oldStartIdx ++
```



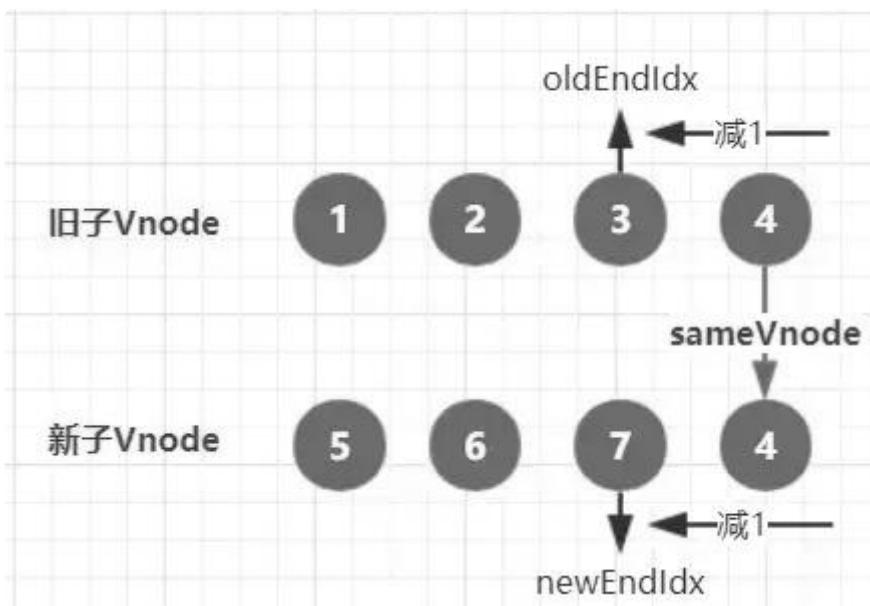
## 2 旧尾 == 新尾

```
sameVnode(oldEndVnode, newEndVnode)
```

和 头头 相同的处理是一样的

尾尾相同，直接跳入下个循环

```
newEndIdx ++ , oldEndIdx ++
```



### 3 旧头 == 新尾

```
sameVnode(oldStartVnode, newEndVnode)
```

这步不符合 不移动DOM，所以只能 移动DOM 了

怎么移动？

源码是这样的

```
parentElm.insertBefore(  
    oldStartVnode.elm,  
    oldEndVnode.elm.nextSibling  
)
```

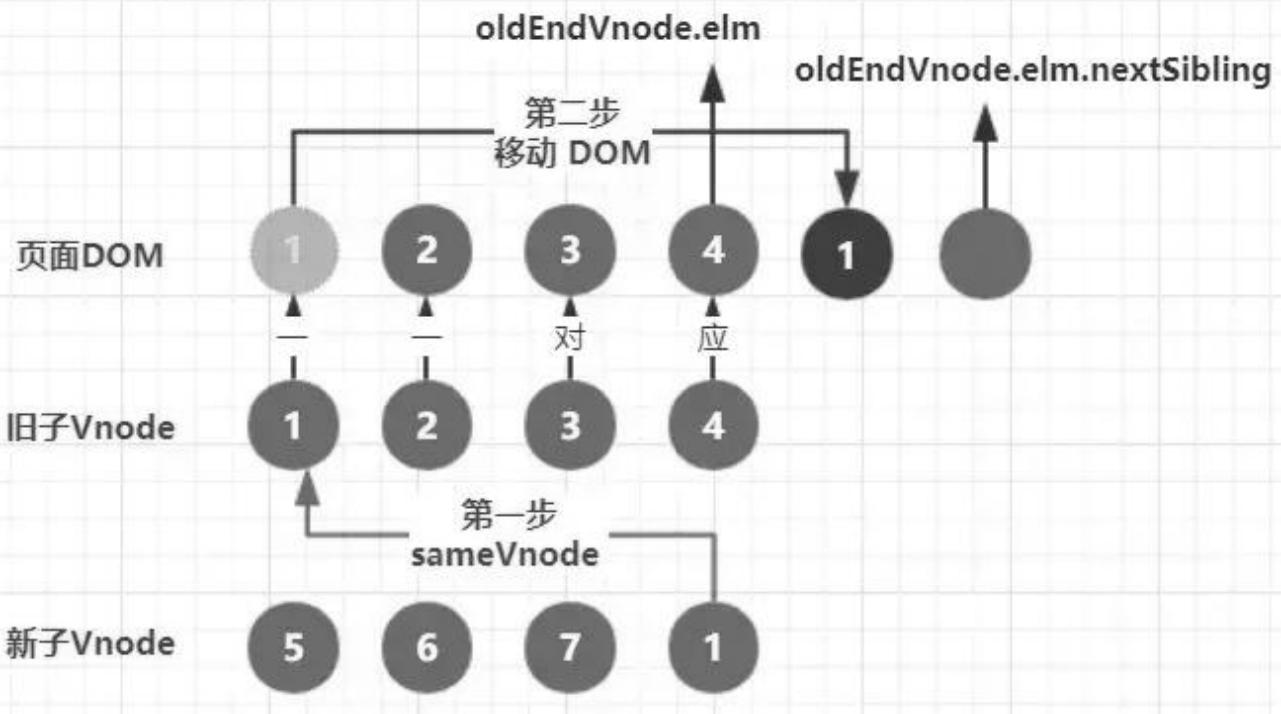
以 新子节点的位置 来移动的，旧头 在新子节点的 末尾

所以把 oldStartVnode 的 dom 放到 oldEndVnode 的后面

但是因为没有把dom 放到谁后面的方法，所以只能使用 insertBefore

即放在 oldEndVnode 后一个节点的前面

图示是这样的



然后更新两个索引

```
oldStartIdx++, newEndIdx--
```

4 旧尾 == 新头

```
sameVnode(oldEndVnode, newStartVnode)
```

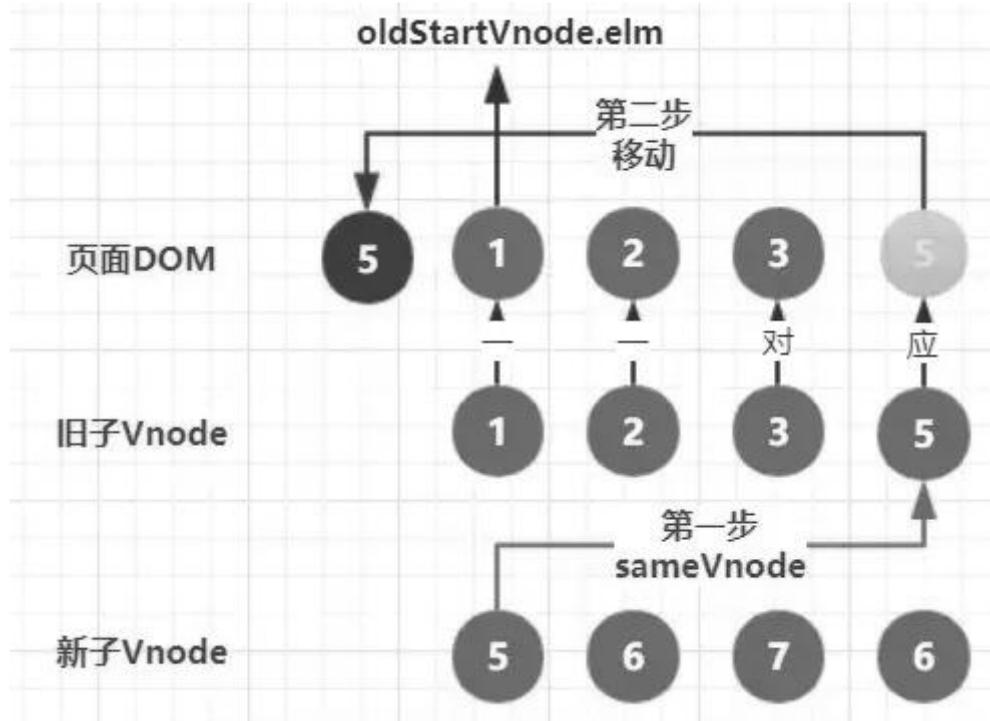
同样不符合 不移动DOM，也只能 移动DOM 了

怎么移动？

```
parentElm.insertBefore(  
  oldEndVnode.elm,  
  oldStartVnode.elm  
)
```

把 oldEndVnode DOM 直接放到当前 oldStartVnode.elm 的前面

图示是这样的



然后更新两个索引

```
oldEndIdx--, newStartIdx++
```

## 5 单个遍历查找

当前面四种比较逻辑都不行的时候，这是最后一种处理方法

拿 新子节点的子项，直接去 旧子节点数组中遍历，找一样的节点出来

流程大概是

- 1、生成旧子节点数组以 `vnode.key` 为key 的 `map` 表
- 2、拿到新子节点数组中 一个子项，判断它的`key`是否在上面的`map` 中
- 3、不存在，则新建DOM
- 4、存在，继续判断是否 `sameVnode`

下面就详细说一下

## 1 生成map 表

这个map 表的作用，就主要是判断存在什么旧子节点

比如你的旧子节点数组是

```
[{  
  tag:"div",  key:1  
, {  
  
  tag:"strong",  key:2  
, {  
  
  tag:"span",  key:4  
}]
```

经过 createKeyToOldIdx 生成一个 map 表 oldKeyToIdx

```
{ vnodeKey: 数组Index }
```

属性名是 vnode.key，属性值是 该 vnode 在children 的位置

是这样（具体源码看上篇文章 Diff - 源码版 之 相关辅助函数）

```
oldKeyToIdx = {  
  1:0,  
  2:1,  
  4:2  
}
```

## 2 判断 新子节点是否存在旧子节点数组中

拿到新子节点中的 子项Vnode，然后拿到它的 key

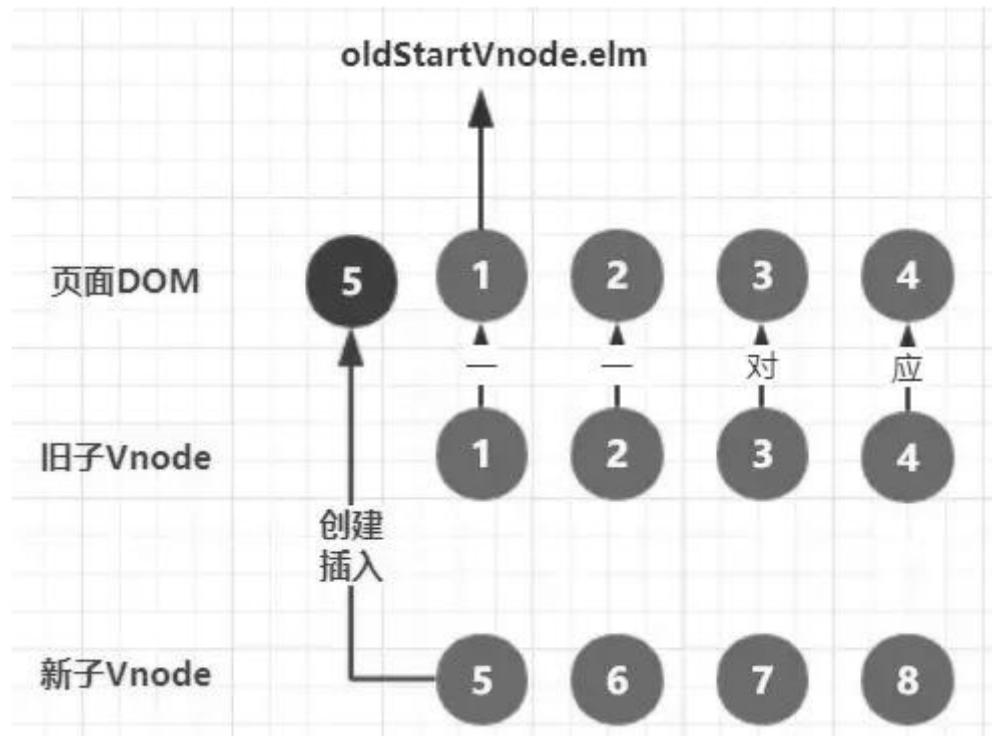
去匹配map 表，判断是否有相同节点

```
oldKeyToIdx[newStartVnode.key]
```

3 不存在旧子节点数组中

直接创建DOM，并插入oldStartVnode 前面

```
createElm(newStartVnode, parentElm, oldStartVnode.elm);
```



4 存在旧子节点数组中

找到这个旧子节点，然后判断和新子节点是否 sameVnode

如果相同，直接移动到 oldStartVnode 前面

如果不同，直接创建插入 `oldStartVnode` 前面

我们上面说了比较子节点的处理的流程分为两个

1、比较新旧子节点

2、比较完毕，处理剩下的节点

比较新旧子节点上面已经说完了，下面就到了另一个流程，比较剩余的节点，详情看下面

处理可能剩下的节点

在`updateChildren` 中，比较完新旧两个数组之后，可能某个数组会剩下部分节点没有被处理过，所以这里需要统一处理

1 新子节点遍历完了

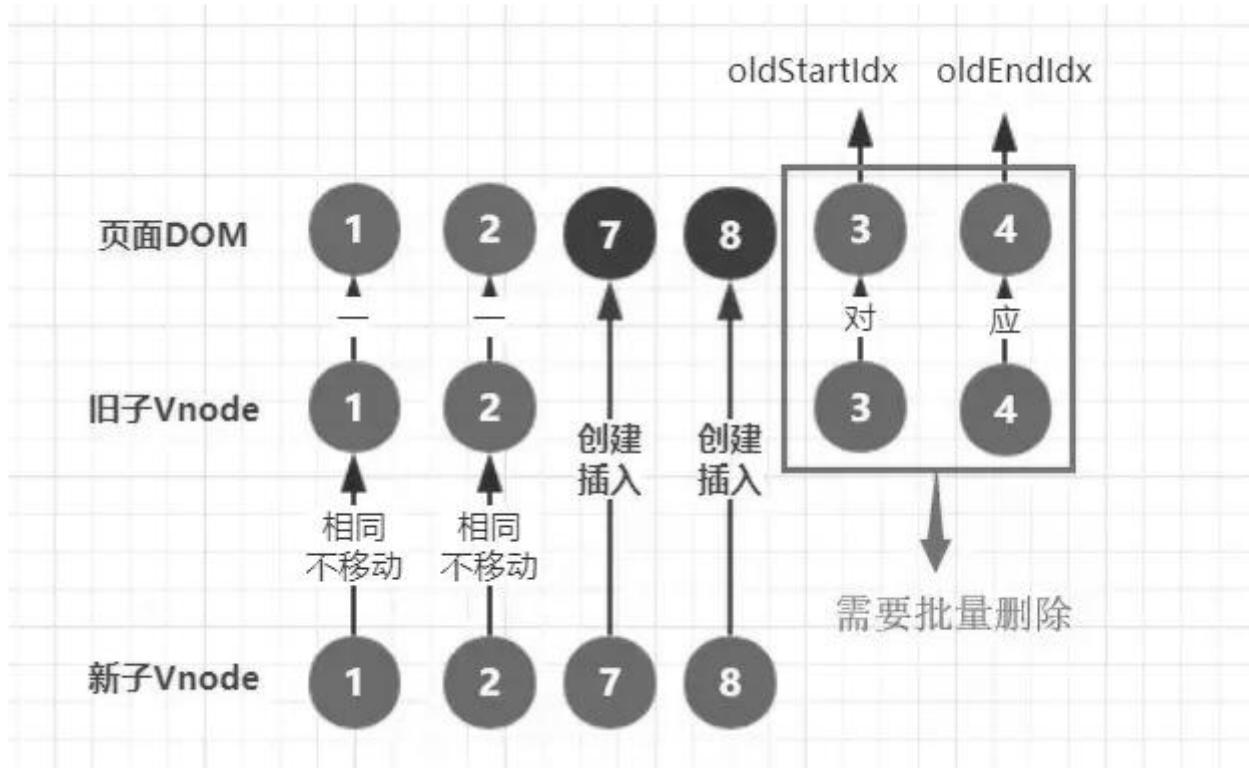
```
newStartIdx > newEndIdx
```

新子节点遍历完毕，旧子节点可能还有剩

所以我们要对可能剩下的旧节点进行 批量删除！

就是遍历剩下的节点，逐个删除DOM

```
for ( ; oldStartIdx <= oldEndIdx; ++oldStartIdx) {  
    oldCh[oldStartIdx]  
  
    .parentNode  
  
    .removeChild(el);  
}
```



## 2. 旧子节点遍历完了

`oldStartIdx > oldEndIdx`

旧子节点遍历完毕，新子节点可能有剩

所以要对剩余的新子节点处理

很明显，剩余的新子节点不存在 旧子节点中，所以全部新建

```

for (; newStartIdx <= newEndIdx; ++newStartIdx) {
  createElm(
    newCh[newStartIdx],
    parentElm,
    refElm
  );
}

```

但是新建有一个问题，就是插在哪里？

所以其中的 refElm 就成了疑点，看下源码

```
var newEnd = newCh[newEndIdx + 1]  
  
refElm = newEnd ? newEnd.elm : null;
```

refElm 获取的是 newEndIdx 后一位的节点

当前没有处理的节点是 newEndIdx

也就是说 newEndIdx+1 的节点如果存在的话，肯定被处理过了

如果 newEndIdx 没有移动过，一直是最后一位，那么就不存在 newCh[newEndIdx + 1]

那么 refElm 就是空，那么剩余的新节点就全部添加进父节点孩子的末尾，相当于

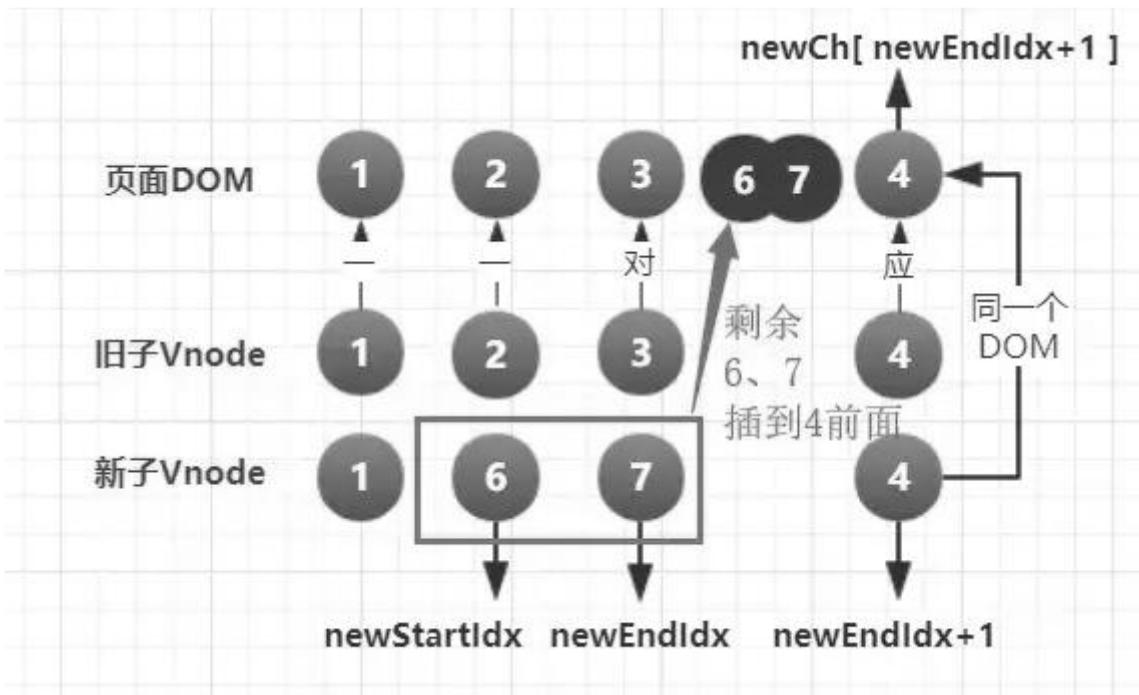
```
for ( ; newStartIdx <= newEndIdx; ++newStartIdx) {  
    parentElm.appendChild(  
  
        newCh[newStartIdx]  
  
    );  
}
```

如果 newEndIdx 移动过，那么就逐个添加在 refElm 的前面，相当于

```
for ( ; newStartIdx <= newEndIdx; ++newStartIdx) {  
    parentElm.insertBefore(  
  
        newCh[newStartIdx] ,  
  
        refElm  
    )
```

```
)  
}
```

如图



思考为什么这么比较

我们已经讲完了所有 Diff 的内容，大家也应该能领悟到 Diff 的思想

但是我强迫自己去思考一个问题，就是

为什么会这样去比较？

以下纯属个人意淫想法，没有权威认证，仅供参考

我们所有的比较，都是为了找到 新子节点 和 旧子节点 一样的子节点

而且我们的比较处理的宗旨是

1、能不移动，尽量不移动

2、没得办法，只好移动

3、实在不行，新建或删除

首先，一开始比较，肯定是按照我们的第一宗旨 不移动 ，找到可以不移动的节点

而 头头，尾尾比较 符合我们的第一宗旨，所以出现在最开始，嗯，这个可以想通

然后就到我们的第二宗旨 移动，按照 updateChildren 的做法有

旧头新尾比较，旧尾新头比较，单个查找比较

我开始疑惑了，咦？头尾比较为了移动我知道，但是为什么要出现这种比较？

明明我可以单个查找的方式，完成所有的移动操作啊？

我思考了很久，头和尾的关系，觉得可能是为了避免极端情况的消耗？？

怎么说？

比如当我们去掉头尾比较，全部使用单个查找的方式

如果出现头 和 尾 节点一样的时候，一个节点需要遍历 从头找到尾 才能找到相同节点

这样实在是太消耗了，所以这里加入了 头尾比较 就是为了排除 极端情况造成的消耗操作

当然，这只是我个人的想法，仅供参考，虽然这么说，我也的确做了个例子测试

子节点中加入了出现两个头尾比较情况的子项 b div

```
oldCh = ['header', 'span', 'div', 'b']
newCh = ['sub', 'b', 'div', 'strong']
```

使用 Vue 去更新，比较更新速度，然后更新十次，计算平均值

1、全用 单个查找，用时 **0.91ms**

2、加入头尾比较，用时 **0.853ms**

的确是快一些喔

本文使用 **mdnice** 排版

## 关注下面的标签，发现更多相似文章

Vue.js

阳光是sunny Lv1

前端工程师 @ 菜鸟

获得点赞 1,076 · 获得阅读 49,111

# 学习vue源码（15）手写 \$forceUpdate,vm.\$destroy方法



vm.\$forceUpdate

## （1）作用

迫使Vue.js实例重新渲染。注意它仅仅影响实例本身以及插入插槽内容的子组件，而不是所有子组件。

## （2）实现

只需要执行watcher的update方法，就可以让实例重新渲染。

Vue.js的每一个实例都有一个**watcher**。当状态发生改变时，会通知到组件级别，然后组件内部使用虚拟DOM进行更详细的重新渲染操作。

事实上，组件就是Vue.js实例，所以组件几倍的watcher和Vue.js实例上的watcher说的是同一个watcher。

手动执行实例watcher的update方法，就可以使Vue.js实例重新渲染。

```
Vue.prototype.$forceUpdate = function(){
  const vm = this;
  if(vm._watcher){
    vm._watcher.update();
  }
}
```

vm.\_watcher就是Vue.js实例的watcher，每当组件内依赖的数据发生变化时，都会自动触发Vue.js实例中 watcher 的 update 方法。

重新渲染的实现原理并不难，Vue.js的自动渲染通过变化侦测来侦测数据，即当数据发生变化时，Vue.js实例重新渲染。而vm.\$forceUpdate是手动通知Vue.js实例重新渲染。

## vm.\$destroy

### （1）作用

完全销毁一个实例，它会清理该实例与其他实例的连接，并解绑其全部指令及监听器，同时会触发beforeDestory和destroyed的钩子函数。

（2）这个方法并不是很常用，大部分场景下并不需要销毁组件，只需要使用v-if或则v-for等指令以数据驱动的方式控制子组件的生命周期即可。

### （3）实现原理

```
Vue.prototype.$destroy = function(){
  const vm = this;
  if(vm._isBeingDestroyed){
    return;
  }
  callHook(vm,"beforeDestroy");
```

```
    vm._isBeingDestroyed = true;
}
```

1、为了防止vm.\$destroy被反复执行，先对属性\_isBeingDestroyed进行判断，如果它为true，说明Vue.js实例正在被销毁，直接使用return语句退出函数执行逻辑。因为销毁只需要销毁一次即可，不需要反复销毁。

2、然后调用callHook函数触发beforeDestroy的钩子函数（callHook会触发参数中提供的钩子函数）。

(4) 销毁实例的逻辑1 首先，需要清理当前组件与父组件之间的连接。组件就是Vue.js实例，所以要清理当前组件与父组件之间的连接，只需要将当前组件实例从父组件实例的\$children属性中删除即可。

说明：Vue.js实例的\$children属性存储了所有子组件

```
const parent = vm.$parent;
if(parent && !parent._isBeingDestroyed && !vm.$options.abstract){
  remove(parent.$children,vm)
}
```

1、如果当前实例有父级，同时父级没有被销毁且不是抽象组件，那么将自己从父级的子列表中删除，也就是将自己的实例从父级的\$children属性中删除。

2、事实上，子组件在不同父组件中是不同的Vue.js实例，所以一个子组件实例的父级只有一个，销毁操作也只需要从父级的子组件列表中销毁当前这个Vue.js实例。

```
export function remove(arr,item){
  if(arr.length){
    const index = arr.indexOf(item);
    if(index>-1){
      return arr.splice(index,1);
    }
  }
}
```

## （5）销毁实例的逻辑2

1、父子组件间的链接断掉之后，需要销毁实例上的所有watcher，也就是说需要将实例上所有的依赖追踪断掉。

2、状态会收集一些依赖，当状态发生改变时会向这些依赖发送通知，而被收集的依赖就是watcher实例。因此，当Vue.js实例被销毁时，应该将实例所监听的状态都取消掉，也就是从状态的依赖列表中将watcher移除。

3、watcher的teardown方法，它的作用是从所有依赖项的Dep列表中将自己移除。即只要执行这个方法，就可以断掉这个watcher所监听的所有状态。

4、断掉Vue.js实例自身的watcher实例监听的所有状态。

```
if(vm._watcher){  
  vm._watcher.teardown();  
}
```

5、执行了组件自身的watcher实例的teardown方法，从所有依赖项的订阅列表中删除watcher实例。删除之后，当状态发生变化时，watcher实例就不会再得到通知。

## 6、vm.\_watcher来源

当执行new Vue()时，会执行一系列初始化操作并渲染组件到实体上，其中就包括vm.\_watcher的处理

7、从Vue.js2.0开始，变化侦测的粒度调整为中等粒度，它只会发送通知到组件级别，然后组件使用虚拟DOM进行重新渲染。组件其实就是Vue.js实例。

## 8、怎么通知到组件级别

在Vue.js实例上，有一个watcher，也就是vm.\_watcher，它会监听这个组件中用到的所有状态，即这个组件内用到的所有状态的依赖列表中都会收集到vm.\_watcher。当这些状态发生变

化时，也都会通知`vm._watcher`，然后这个`watcher`再调用虚拟DOM进行重新渲染。

## （6）销毁实例的逻辑

1、只从状态的依赖列表中删除Vue.js实例上的`watcher`实例是不够的。Vue.js提供了`vm.watch`方法，它允许用户监听某个状态。因此，还需要销毁用户使用`vm.watch`所创建的`watcher`实例。

2、从状态的依赖列表中销毁用户创建的`watcher`实例和销毁Vue实例上的`watcher`实例相同，只需要执行`watcher`的`teardown`方法。

3、问题：如何知道用户创建了多少个`watcher`？

1) Vue.js的解决方案是执行`new Vue()`时，在初始化的流程中，在`this`上添加一个`_watchers`属性

```
vm._watchers = [];
```

2) 每当创建`watcher`实例时，都会将`watcher`实例添加到`vm._watchers`中

```
export default class Watcher{
  constructor(vm, expOrFn, cb){
    <!-- 每当创建watcher实例时，都将watcher实例添加到vm._watchers中 -->
    vm._watchers.push(this);
  }
}
```

4、只需要遍历`vm._watchers`并依次执行每一项`watcher`实例的`teardown`方法，就可以将`watcher`实例从它所监听的状态的依赖列表中移除。

```
let i = vm._watchers.length;
while(i--){
```

```
    vm._watchers[i].teardown();
}
```

(7) 向Vue.js实例添加\_isDestroyed属性来表示Vue.js实例已经被销毁。

```
vm._isDestroyed = true;
```

(8) 当vm.\$destroy执行时，Vue.js不会将已经渲染到页面中的DOM节点移除，但会将模板中的所有指令解绑。

```
vm._patch_(vm._vnode,null)
```

(9) 触发destroyed钩子函数

```
callHook(vm,'destroyed')
```

(10) 最后，移除实例上的所有事件监听器。

```
vm.$off()
```

vm.\$off() 我们在学习vue源码（1）手写与事件相关的实例方法已经谈过其实现，实现也挺简单，感兴趣可以看一看。

(11) 完整代码

```
Vue.prototype.$destroy = function(){
  const vm = this;
  <!-- 防止重复销毁 -->
  if(vm._isBeingDestroyed){
```

```
return;
}

<!-- 调用钩子函数beforeDestroy -->
callHook(vm,"beforeDestroy");

vm._isBeingDestroyed = true;

<!-- 删除自己与父级之间的连接 -->
const parent = vm.$parent;

if(parent && !parent._isBeingDestroyed && !vm.$options.abstract){
    remove(parent.$children,vm);
}

<!-- 从watcher监听的所有状态的依赖列表中移除watcher -->
if(vm._watcher){
    vm._watcher.teardown();
}

let i = vm._watchers.length;
<-- 将从vm.$watcher创建的watcher实例从它所监听的状态的依赖列表中移除 -->
while(i--){
    vm._watchers[i].teardown();
}

<!-- 表示实例已经被销毁 -->
vm._isDestroyed = true;

<!-- 将模板中的所有指令解绑 -->
vm._patch_(vm._vnode,null)
<-- 触发destroyed钩子函数 -->
callHook(vm,'destroyed')
<-- 移除实例上的所有事件监听器 -->
vm.$off();
}
```

本文使用 mdn nice 排版

# 学习vue源码（16）初探生命周期之各阶段都在干嘛



## 一、概述

每个Vue.js实例在创建时都要经过一系列初始化，例如设置数据监听、编译模板、将实例挂载到DOM并在数据变化时更新DOM等。

同时，也会运行一些叫作生命周期钩子的函数，给在不同阶段添加自定义代码的机会。

## 二、生命周期

Vue.js生命周期可以分为4个阶段：初始化阶段、模板编译阶段、挂载阶段、卸载阶段。

### 初始化阶段

new Vue（）到created之间的阶段叫作初始化阶段。

这个阶段的主要目的是在Vue.js实例上初始化一些属性、事件以及响应式数据，如props、methods、data、computed、watch、provide和inject等。

### 模板编译阶段

在created钩子函数与beforeMount钩子函数之间的阶段是模板编译阶段。

这个阶段的主要目的是将模板编译为渲染函数，只存在于完整版中。如果在只包含运行时的构建版本中执行new Vue（），则不会存在这个阶段。

当使用vue-loader或vueify时，\*.vue文件内部的模板会在构建时预编译成Javascript，所以最终打好的包里是不需要编译器的，用运行时版本即可。由于模板这时已经预编译成了渲染函数，所以在生命周期中并不存在模板编译阶段，初始化阶段的下一个生命周期直接是挂载阶段。

### 挂载阶段

beforeMount钩子函数到mounted钩子函数之间的是挂载阶段。

在这个阶段，Vue.js会将其实例挂载到DOM元素上，通俗地讲，就是讲模板渲染到指定的DOM元素中。

在挂载的过程中，Vue.js会开启Watcher来持续追踪依赖的变化。

在已挂载状态下，Vue.js仍会持续追踪状态的变化。当数据（状态）发生变化时，Watcher会通知虚拟DOM重新渲染视图，并且会在渲染视图前出发`beforeUpdate`钩子函数，渲染完毕后触发`updated`钩子函数。

通常，在运行时的大部分时间下，Vue.js处于已挂载状态，每当状态发生变化时，Vue.js都会通知组件使用虚拟DOM重新渲染，也就是常说的响应式。这个状态会持续到组件被销毁。

## 卸载阶段

应用调用 `vm.$destroy` 方法后，Vue.js的生命周期会进入卸载阶段。

`** vm.$destroy **`我们在上一篇文章中实现过：[学习vue源码（15）手写`forceUpdate, vm.`](#)

### destroy方法

在这个阶段，Vue.js会将自身从父组件中删除，取消实例上所有依赖的追踪并且移除所有事件监听器。

## 小结

生命周期可以在整体上分为两部分

1、第一部分是初始化阶段、模板编译阶段与挂载阶段。

2、第二部分是卸载阶段。

## 三、从源码角度了解生命周期

卸载阶段的内部原理就是 `vm.$destroy` 方法的内部原理。模板编译阶段和挂载阶段（mount）的原理已述过，见前面文章。现在主要介绍初始化阶段的内部原理。

## new Vue()被调用时发生了什么

当new Vue()被调用时，会首先进行一些初始化操作，然后进入模板编译阶段，最后进入挂载阶段。

```
function Vue(optipons){  
  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);  
}  
export default Vue;
```

1、首先进行安全检查。在非生产环境下，如果没有使用new调用Vue，则会在控制台抛出错误警告：Vue是构造函数，应该使用new关键字来调用。

2、然后调用 `this._init(options)` 来执行生命周期的初始化流程。即生命周期的初始化流程在`this._init`中实现。

## 四、\_init方法的定义

(1) Vue.js通过调用initMixin方法将\_init挂载到Vue构造函数的原型上。

```
import { initMixin } from './init'  
function Vue(optipons){  
  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);
}
initMixin(Vue);
export default Vue;

```

(2) 将init.js文件导出的initMixin函数导入后，通过调用initMixin函数向Vue构造函数的原型中挂载一些方法。

```

export function initMixin(Vue){
  Vue.prototype._init = function(options){
    <!-- 做些什么 -->
  }
}

```

在Vue构造函数的prototype属性上添加了一个\_init方法。即\_init方法方法的定义与前面介绍的Vue.js实例方法的挂载方式是相同的。

## 五、\_init方法的内部原理

当new Vue()执行后，触发的一系列初始化流程都是在\_init方法中启动的。

(1) 实现

```

Vue.prototype._init = function(options){
  vm.$options = mergeOptions(
    resolveConstructorOptions(vm.constructor),
    options || {},
    vm
  )

  initLifecycle(vm);
  initEvents(vm);
  initRender(vm);
  callHook(vm, 'beforeCreate');
}

```

```
initInjections(vm); //在data/props前初始化inject  
initState(vm);  
initProvide(vm); //在data/props前初始化provide  
callHook(vm, 'created');  
  
//如果用户在实例化Vue.js时传递了el选项，则自动开启模板编译阶段与挂载阶段  
//如果没有传递el选项，则不进入下一个生命周期流程  
//用户需要执行vm.$mount方法，手动开启模板编译阶段与挂载阶段  
  
if(vm.$options.el){  
  vm.$mount(vm.$options.el);  
}  
}
```

1、Vue.js会在初始化流程的不同时期通过callHook函数触发生命周期钩子。

2、在执行初始化流程之前，实例上挂载了 \$options 属性。目的是将用户传递的options选项与当前构造函数的options属性及其父级实例构造函数的options属性，合并生成一个新的options并赋值给 \$options 属性。

3、resolveConstructorOptions函数的作用就是获取当前实例中构造函数的options选项及其所有父级的构造函数的options。之所以会有父级，是因为当前Vue.js实例可能是一个子组件，它的父组件就是它的父级。

4、在生命周期钩子**beforeCreate**被触发之前执行了initLifecycle、initEvents和initRender。

5、在初始化的过程中，首先初始化事件与属性，然后触发生命周期钩子**beforeCreate**。

6、随后初始化provide/inject和状态，这里的状态指的是props、methods、data、computed以及watch。

7、解这触发生命周期钩子**created**。

8、最后，判断用户是否在参数中提供了el选项，如果是，则调用 `vm.$mount` 方法，进入后面的生命周期阶段。

## 六、callHook函数的内部原理

- (1) Vue.js通过callHook函数来触发生命周期钩子。
- (2) callHook的作用是触发用户设置的生命周期钩子，而用户设置的生命周期钩子会在执行new Vue()时通过参数传递给Vue.js。也就是说，可以在Vue.js的构造函数中通过options参数得到用户设置的生命周期钩子。
- (3) 用户传入的options参数最终会与构造函数的options属性合并并生成新的options并赋值到`vm.$options`属性中，所以可以通过`vm.$options`得到用户设置的生命周期函数。例如，通过`vm.$options.created`得到用户设置的created钩子函数。
- (4) Vue.js在合并options的过程中会找出options中所有key是钩子函数的名字，并将它转换成数组。
- (5) 所有生命周期钩子的函数名

```
beforeCreate
created
beforeMount
mounted
beforeUpdate
updated
beforeDestroy
destroyed
activated
deactivated
errorCaptured
```

- (6) 通过`vm.$options.created`获取的是一个数组，数组中包含了钩子函数。

```
console.log(vm.$options.created)//[fn]
```

数组原因：可能存在多个钩子函数，例如mixin混入的和用户自己设置的。转换成数组后，可以在同一个生命周期钩子列表中保存多个生命周期钩子。

## （7）实现原理

只需要从 `vm.$options` 中获取生命周期钩子列表，遍历列表，执行每一个生命周期钩子，就可以触发钩子函数。

```
export function callHook(vm, hook){  
  const handlers = vm.$options[hook];  
  if(handlers){  
    for(let i = 0,j = handlers.length ; i<j;i++){  
      try{  
        handlers[i].call(vm);  
      }catch(e){  
        handleError(e,vm,'${hook}hook')  
      }  
    }  
  }  
}
```

1、`callHook`接收`vm`和`hook`两个参数，其中前者是Vue.js实例的`this`，后者是生命周期钩子的名称。

2、使用`hook`从 `vm.$options` 中获取钩子函数列表后赋值给`handlers`，随后遍历`handlers`，执行每一个钩子函数。

3、使用`try...catch`语句捕获钩子函数发生的错误，并使用`handleError`处理错误。`handleError`会依次执行父组件的`errorCaptured`钩子函数与全局的`config.errorHandler`，这也是为什么生命周期钩子`errorCaptured`可以捕获子孙组件的错误。

## 七、`errorCaptured`与错误处理

## （1）作用

捕获来自子孙组件的错误，此钩子函数会收到三个参数：错误对象、发生错误的组件实例以及一个包含错误来源信息的字符串。此钩子函数可以返回false，阻止该错误继续向上传播。

## （2）传播规则

默认情况下，如果全局的config.errorHandler被定义，那么所有的错误都会发送给它，这样这些错误可以在单个位置报告给分析服务。

如果一个组件继承的链路或其父级从属链路中存在多个errorCaptured钩子，则它们将会被相同的错误逐个唤起。

如果errorCaptured钩子函数自身抛出了一个错误，则这个新错误和原本被捕获的错误都会发送给全局的config.errorHandler。

一个errorCaptured钩子函数能够返回false来阻止错误继续向上传播。这本质上是说“这个错误已经被搞定，应该被忽略”。它会阻止其他被这个错误唤起的errorCaptured钩子函数和全局的config.errorHandler。

（3）errorCaptured钩子函数与Vue.js的错误处理有着千丝万缕的关系。Vue.js会捕获所有用户代码抛出的错误，然后使用一个名叫handleError的函数来处理这些错误。

（4）用户编写的所有函数都是Vue.js调用的，例如用户在代码中注册的事件、生命周期钩子、渲染函数、函数类型的data属性、`vm.$watch`的第一个参数（函数类型）、nextTick和指令等。

（5）而Vue.js在调用这些函数时，会使用try...catch语句来捕获有可能发生的错误。当错误发生并且被try...catch语句捕获后，Vue.js会使用handleError函数来处理错误，该函数会依次触发父组件链路上的每一个父组件中定义的errorCaptured钩子函数。如果全局的config.errorHandler被定义，那么所有的错误也会同时发送给config.errorHandler。也就是说，错误的传播规则是在handleError函数中实现的。

## （6）handleError原理

将所有错误发送给config.errorHandler

```
export function handleError (err,vm,info){
  <!-- 这里的config.errorHandler就是Vue.config.errorHandler -->
  if(config.errorHandler){
    try{
      return config.errorHandler.call(null,err,vm,info);
    }catch(e){
      logError(e);
    }
  }
  logError(e);
}

function logError(err){
  console.log(err);
}
```

1、先判断Vue.config.errorHandler是否存在，如果存在，则调用它，并将错误对象、发生错误的组件实例以及一个包含错误来源信息的字符串通过参数的方式传递给它，并且使用try...catch语句捕获错误

2、如果全局错误处理函数也发生错误，则在控制台打印其中抛出的错误。

3、不论用户是否使用Vue.config.errorHandler捕获错误，Vue.js都会将错误信息打印在控制台。

如果一个组件继承的链路或其父级从属链路中存在多个errorCaptured钩子函数，则它们将会被相同的错误逐个唤起。

```
export function handleError (err,vm,info){
  if(vm){
    let cur = vm;
    while((cur = cur.$parent)){
      const hooks = cur.$options.errorCaptured;
      if(hooks){
        for(let i = 0;i<hooks.length;i++){
          hooks[i](err,cur,info);
        }
      }
    }
  }
}
```

```

        hooks[i].call(cur,err,vm,info);
    }
}
}
}
globalHandleError(err,vm,info);
}

function globalHandleError(err,vm,info){
<!-- 这里的config.errorHandler就是Vue.config.errorHandler -->
if(config.errorHandler){
try{
    return config.errorHandler.call(null,err,vm,info);
}catch(e){
    logError(e);
}
}
logError(e);
}

function logError(err){
console.log(err);
}

```

1、新增globalHandleError函数，将全局错误处理相关的代码放到这个函数中。

2、通过while语句自底向上不停地循环获取父组件，直到根组件。

3、在循环中，通过 `cur.$options.errorCaptured` 属性独出errorCaptured钩子函数列表，遍历钩子函数列表并依次执行列表中的每一个errorCaptured钩子函数。

如果errorCaptured钩子函数自身抛出了一个错误，那么这个新错误和原本被捕获的错误都会发送给全局的config.errorHandler。

```

export function handleError (err,vm,info){
if(vm){
let cur = vm;
while((cur = cur.$parent)){
const hooks = cur.$options.errorCaptured;
if(hooks){

```

```

for(let i = 0;i<hooks.length;i++){
  try{
    hooks[i].call(cur,err,vm,info);
  }catch(e){
    globalHandleError(e,cur,"errorCaptured hook");
  }
}
}
}
}
globalHandleError(err,vm,info);
}

```

1、只需要使用try...catch语句捕获钩子函数可能发生的错误，并通过执行globalHandleError将捕获到的错误发送给全局错误处理函数config.errorHandler即可。

2、因为这个错误是钩子函数自身抛出的新错误，所以不影响自底向上执行钩子函数的流程。而原有的错误则会在自底向上这个循环结束后，将错误传递给全局错误处理钩子函数。

一个errorCaptured钩子函数能够返回false来阻止错误继续向上传播 1、它会阻止其他被这个错误唤起的errorCaptured钩子函数和全局的config.errorHandler。

```

export function handleError (err,vm,info){
  if(vm){
    let cur = vm;
    while((cur = cur.$parent)){
      const hooks = cur.$options.errorCaptured;
      if(hooks){
        for(let i = 0;i<hooks.length;i++){
          try{
            const capture = hooks[i].call(cur,err,vm,info) === false;
            if(capture) return;
          }catch(e){
            globalHandleError(e,cur,"errorCaptured hook");
          }
        }
      }
    }
  }
}

```

```
    }
}
globalHandleError(err,vm,info);
}
```

2、使用capture保存钩子函数执行后的返回值，如果返回值false，则使用return语句停止程序继续执行。

3、其巧妙地地方在于代码中地逻辑是先自底向上传递错误，之后再执行globalHandleError将错误发送给全局错误处理钩子函数。所以只要再自底向上这个循环中地某一层执行了return语句，程序机会立即停止执行，从而是实现功能。因为一旦钩子函数返回了false，handleError函数将会执行return语句终止程序执行，所以错误向上传递和全局的config.errorHandler都会被停止。

本文使用 mdnice 排版

# 学习vue源码（17）再探生命周期之初始化实例属性及事件



在前一篇文章学习vue源码（16）初探生命周期各阶段都在干嘛

Vue.js生命周期可以分为4个阶段：初始化阶段、模板编译阶段、挂载阶段、卸载阶段。

而初始化阶段又可分为

在Vue.js实例上初始化一些属性、事件以及响应式数据，如props、methods、data、computed、watch、provide和inject等。

这一次，我们就来探究第一阶段：初始化阶段的属性、事件，如代码所示，研究initLifecycle，initEvents，initRender。都干了什么。这件这三个初始化都在beforeCreate钩子函数触发前初始化的。

```
Vue.prototype._init = function(options){  
  vm.$options = mergeOptions(  
    resolveConstructorOptions(vm.constructor),  
    options || {},  
    vm  
)  
  
  initLifecycle(vm);  
  initEvents(vm);  
  initRender(vm);  
  callHook(VM, 'beforeCreate');  
  initInjections(vm); //在data/props前初始化inject  
  initState(vm);  
  initProvide(vm); //在data/props前初始化provide  
  callHook(vm, 'created');  
  
  if(vm.$options.el){  
    vm.$mount(vm.$options.el);  
  }  
}
```

## 一、初始化实例属性

在Vue.js的整个生命周期中，初始化实例属性是第一步。

需要实例化的属性既有Vue.js内部需要用到的属性（如vm.\_watcher），也有提供给外部使用的属性（例如vm.\$parent）。

以\$开发的属性是提供给用户使用的外部属性，以\_开头的属性是提供给内部使用的内部属性。

Vue.js通过initLifecycle函数向实例中挂载属性，该函数接收Vue.js实例作为参数。所以在函数中，只需要向Vue.js实例设置属性即可达到向Vue.js实例挂载属性的目的。

### （5）实现

```

export function initLifecycle(vm) {
  const options = vm.$options;
  // 找出第一个非抽象父类
  let parent = options.parent;
  if (parent && !options.abstract) {
    while (parent.$options.abstract && parent.$parent) {
      parent = parent.$parent;
    }
    parent.$children.push(vm);
  }
  vm.$parent = parent;
  vm.$root = parent ? parent.$root : vm;

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

  vm._watcher = null;
  vm._isDestroyed = false;
  vm._isBeingDestroyed = false;
}

```

在Vue.js实例上设置一些属性并提供一个默认值。

`vm.$parent` 属性，它需要找到第一个非抽象类型的父级，所以代码中会进行判断：如果当前组件不是抽象组件并且存在父级，那么需要通过while来自底向上循环。如果父级是抽象类，那么继续向上，直到遇到第一个非抽象类的父级时，将它赋值给 `vm.$parent` 属性。(抽象类指的是transition，keepAlive这些)

`vm.$children` 属性，它会包含当前实例的直接子组件。该属性的值是从子组件中主动添加到父组件中的。`parent.$children.push(vm)`，就是将当前实例添加到父组件实例的 `$children` 属性中。

`vm.$root`，它标识当前组件树的根Vue.js实例。如果当前组件没有父组件，那么它自己其实就是根组件，它的 `$root` 属性是它自己，而它的子组件的 `vm.$root` 属性是沿用父级的 `$root`，所以其直接子组件的 `$root` 属性还是它，其孙组件的 `$root` 属性沿用其直接子组件中的 `$root` 属性，以此类推。因此，这其实是自顶向下将根组件的 `$root` 依次传递给每一个子组件的过程。

## 二、初始化事件

- (1) 初始化事件是指将父组件在模板中使用的v-on注册的事件添加到子组件的事件系统（Vue.js的事件系统）中。
- (2) Vue.js中，父组件可以在使用子组件的地方用v-on来监听子组件触发的事件。
- (3) 在模板编译阶段，可以得到某个标签上的所有属性，其中就包括使用v-on或@注册的事件。
- (4) 在模板编译阶段，我们会将整个模板编译成渲染函数，而渲染函数其实就是一些嵌套在一起的创建元素节点的函数。
- (5) 创建元素节点的函数是这样的：

```
_c(tagName, data, children)
```

- 。这是渲染函数，不理解的可以看[学习vue源码（10）学习render渲染函数](#)
- (6) 当渲染流程启动时，渲染函数会被执行并生成一份VNode，随后虚拟DOM会使用VNode进行对比与渲染。
- (7) 在这个过程中会创建一些元素，但此时会判断当前这个标签究竟是真的标签还是一个组件。
- (8) 如果是组件标签，那么会将子组件实例化并给它传递一些参数，其中就包括父组件在模板中使用v-on注册在子组件标签上的事件；
- (9) 如果是平台标签，则创建元素并插入到DOM中，同时会将标签上使用v-on注册的事件注册到浏览器事件中。

(10) 即，如果v-on写在组件标签上，那么这个事件会注册到子组件Vue.js事件系统中；如果是写在平台标签上，例如div，那么事件会被注册到浏览器事件中。

(11) 子组件在初始化时，也就是初始化Vue.js实例时，有可能会接收父组件向子组件注册的事件。而子组件自身在模板中注册的事件，只有在渲染的时候才会根据虚拟DOM的对比结果来确定是注册事件还是解绑事件。所以在实例初始化阶段，被初始化的事件指的是父组件在模板中使用v-on监听子组件内触发的事件。

(12) Vue.js通过initEvents函数执行初始化事件相关的逻辑。

```
export function initEvents(vm){
  vm._events = Object.create(null);
  <!-- 初始化父组件附加的事件 -->
  const listeners = vm.$options._parentListeners;
  if(listeners){
    updateComponentListeners(vm, listeners);
  }
}
```

1、在vm上新增\_events属性并将它初始化为空对象，用来存储事件。事实上，所有使用vm.\$on注册的事件监听器都会保存到vm.\_events属性中。

2、在模板编译阶段，当模板解析到组件标签时，会实例化子组件，同时将标签上注册的事件解析成object并通过参数传递给子组件。所以当子组件被实例化时，可以在参数中获取父组件向自己注册的事件，这些事件最终会被保存在vm.\$options.\_parentListeners中。

html

```
<button-counter v-on:increment="incrementTotal"></button-counter>
```

vm.\$options.\_parentListeners：

```
{ increment :function(){}}
```

3、如果 `vm.$options._parentListeners` 不为空，则调用`updateComponentListeners`方法，将父组件向子组件注册的事件注册到子组件实例中。

### (13) `updateComponentListeners`逻辑

只需要循环 `vm.$options._parentListeners` 并使用 `vm.$on` 把事件都注册到`this._events`中即可。

```
let target;

function add(event,fn,once){
  if(once){
    target.$once(event,fn);
  }else{
    target.$on(event,fn);
  }
}

function remove(event,fn){
  target.$off(event,fn);
}

export function updateComponentListeners(vm,listeners,oldListeners){
  target = vm;
  updateListeners(listeners,oldListeners || {},add,remove,vm);
}
```

`vm.$on,$once,$off` 我们在 [学习vue源码（1）手写与事件相关的实例方法](#)讲过，不理解的可以看一下。

1、封装了`add`和`remove`这两个函数，用来新增和删除事件。

2、`updateListeners`函数对比`listeners`和`oldListeners`的不同，并调用参数中提供的`add`和`remove`进行相应的注册事件和卸载事件的操作。

### (14) updateListeners函数

1、实现思路：如果listeners对象中存在某个key（也就是事件名）在oldListeners中不存在，那么说明这个事件是需要新增的事件；反过来，如果oldListeners中存在某些key（事件名）在listeners中不存在，那么说明这个事件是需要从事件系统中移除的。

```
export function updateListeners(on, oldOn, add, remove, vm){
  let name, cur, old, event;
  for(name in on){
    cur = on[name];
    old = oldOn[name];
    event = normalizeEvent(name);
    if(isUndef(cur)){
      process.env.NODE_ENV!=='production' && warn(
        'Invalid handler for event "${event.name}":got'+String(cur),
        vm
      )
    }else if(isUndef(old)){
      if(isUndef(cur.fns)){
        cur = on[name] = createFnInvoker(cur);
      }
      add(event.name, cur, event.once, event.capture, event.passive);
    }else if(cur!==old){
      old.fns = cur;
      on[name] = old;
    }
  }
  for(name in oldOn){
    if(isUndef(on[name])){
      event = normalizeEvent(name);
      remove(event.name, oldOn[name], event.capture);
    }
  }
}
```

2、该函数接收5个参数，分别是on、oldOn、add、remove和vm。

3、其主要逻辑是比对on和oldOn来分辨哪些事件需要执行add注册事件，哪些事件需要执行remove删除事件。

4、可以分为两部分呢，第一部分是循环on，第二部分是循环oldOn。第一部分的主要作用是判断哪些事件在oldOn中不存在，调用add注册这些事件。第二部分的作用是循环oldOn，判断哪些事件在on中不存在，调用remove移除这些事件。

5、在循环on的过程中，有如下三个判断

- 判断事件名对应的值是否是undefined或null，如果是，则在控制台触发警告。
- 判断该事件名在oldOn中是否存在，如果不存在，则调用add注册事件。
- 如果事件名在on和oldOn中都存在，但是它们并不相同，则将事件回调替换成on中的回调，并且把on中的回调引用指向真实的事件系统中注册的事件，也就是oldOn中对应的事件。

6、isUndef函数用于判断传入的参数是否为undefined或null。

#### (15) normalizeEvent函数

1、Vue.js的模板中支持事件修饰符，例如capture、once和passive等，如果我们在模板中注册事件时使用了事件修饰符，那么在模板编译阶段解析标签上的属性时，会将这些修饰符改成对应的符号加载事件名的前面，例如

```
<child v-on:increment.once="a"></child>
```

vm.\$options.\_parentListeners为：

```
{~increment:function(){}}
```

事件名的前面新增了一个~符号，说明该事件的事件修饰符是once，通过这样的方式来分辨当前事件是否使用了事件修饰符。

2、normalizeEvent的作用是将事件修饰符解析出来。

```

const normalizeEvent = name =>{
  const passive = name.charAt(0) === '&';
  name = passive ? name.slice(1) : name;
  const once = name.charAt(0) === '~';
  name = once ? name.slice(1) : name;
  const capture = name.charAt(0) === '!';
  name = capture ? name.slice(1) : name;
  return{
    name,
    once,
    capture,
    passive
  }
}

```

3、如果事件有修饰符，则会将它截取出来。最终输出的对象中保存了事件名以及一些事件修饰符，这些修饰符为true说明事件使用了此事件修饰符。

### 三、初始化属性方法

这个比较简单，

我们直接看源代码

```

export function initRender (vm: Component) {
  vm._vnode = null
  vm._staticTrees = null
  const options = vm.$options
  const parentVnode = vm.$vnode = options._parentVnode node in parent tree
  const renderContext = parentVnode && parentVnode.context
  vm.$slots = resolveSlots(options._renderChildren, renderContext)
  vm.$scopedSlots = emptyObject

  vm._c = (a, b, c, d) => createElement(vm, a, b, c, d, false)
}

```

```

vm.$createElement = (a, b, c, d) => createElement(vm, a, b, c, d, true)

const parentData = parentVnode && parentVnode.data

defineReactive(vm, '$attrs', parentData && parentData.attrs || emptyObject, null, true)
defineReactive(vm, '$listeners', options._parentListeners || emptyObject, null, true)

}

```

可见，这也是给vue实例 初始化一些属性和方法，其中包括

- `_vnode`：表示这个实例的节点
- `_staticTrees`：表示是否是静态节点

什么是静态节点我们在学习vue源码（8）手写优化器说过

- `$slots`、`$scopedSlots` 都属该组件的插槽
- `_c`与`$createElement`都是用于创建VNode节点，

`_c`我们在学习vue源码（9）手写代码生成器里用到。

`createElement`如果我们自己写`render`函数的话，就会用到。如图所示



```

Vue.component('anchored-heading', {
  render: function (createElement) {
    return createElement(
      'h' + this.level, // 标签名称
      this.$slots.default // 子节点数组
    )
  },
  props: {
    level: {
      type: Number,
      required: true
    }
  }
})

```

- 然后又添加了 `$attrs` 、 `$listeners`

总结一下：

这篇文章讲了 生命周期中的初始化的一部分

这一部分是属于**beforeCreate**钩子函数触发前初始化的。

包括

```
initLifecycle(vm);
initEvents(vm);
initRender(vm);
```

其中`initLifecycle`给实例初始化了这些属性

```
$parent
$root

$children
$refs

Watcher
_isDestroyed
_isBeingDestroyed
```

`initEvents` 则是初始化了 写在子组件上的事件。其事件都保存在`vm._events`中

`initRender` 则是初始化了这些属性

```
_vnode
_staticTrees
```

`\$slots`、`\$scopedSlots`

\_c与 `$createElement`

`\$attrs`、`\$listeners`

本文使用 **mdnice** 排版

# 学习vue源码（18）三探生命周期之初始化 provide与inject



上篇文章学习vue源码（17）再探生命周期之初始化实例属性及事件讲解了初始化阶段的

```
initLifecycle(vm)
initEvents(vm)
initRender(vm)
```

即beforeCreate钩子函数触发前对实例 属性和事件的初始化。

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

这一次来讲解 created钩子函数触发前，beforeCreate触发后 的initInjections和initProvide，可能你会问为什么.initState为什么不一起讲呢，因为 initState也是一个大块头，放到后面单独讲一下。

## 概念解析

成对出现：provide和inject是成对出现的

作用：用于父组件向子孙组件传递数据

使用方法：provide在父组件中返回要传给下级的数据，inject在需要使用这个数据的子辈组件或者孙辈等下级组件中注入数据。

使用场景：由于vue有\$parent属性可以让子组件访问父组件。但孙组件想要访问祖先组件就比较困难。通过provide/inject可以轻松实现跨级访问父组件的数据

## 使用

provide和inject需要配合使用，是多层级组件的通信方式。相信，面试问vue组件间有哪些通信方式，就会提到这个。（什么？你说你不知道。。。, 那正好可以了解下，下次有得说了）

在此之前，在我们描述访问父级组件实例的时候，展示过一个类似这样的例子：

```
<google-map>
  <google-map-region v-bind:shape="cityBoundaries">
    <google-map-markers v-bind:places="iceCreamShops"></google-map-markers>
  </google-map-region>
</google-map>
```

HTML

在这个组件里，所有 `<google-map>` 的后代都需要访问一个 `getMap` 方法，以便知道要跟哪个地图进行交互。不幸的是，使用 `$parent` 属性无法很好的扩展到更深层级的嵌套组件上。这也是依赖注入的用武之地，它用到了两个新的实例选项： `provide` 和 `inject`。

简单的说，当组件的引入层次过多，我们的子孙组件想要获取祖先组件的资源，那么怎么办呢，总不能一直取父级往上吧，而且这样代码结构容易混乱。这个就是这对选项要干的事情

`provide`和`inject`需要配合使用，它们的含义如下：

`provide`:一个对象或返回一个对象的函数，该对象包含可注入起子孙的属性，可以使用ES6的 `Symbols`作为key(只有原生支持Symbol才可以)

`inject`:`inject` 选项应该是一个字符串数组或一个对象，该对象的 `key` 代表了本地绑定的名称，`value` 为其 `key` (字符串或 `Symbol`) 以在可用的注入中搜索。

来个例子理解一下

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <script src="https://cdn.jsdelivr.net/npm/vue@2.6.10/dist/vue.js"></script>
  <title>Document</title>
```

```

</head>
<body>
  <div id="app"><child></child></div>
  <script>
    Vue.component('child',{
      inject:['message'],
      template:'<p>{{message}}</p>'
    })
    new Vue({
      el:'#app',
      provide:{message:'Hello Vue!'}
    })
  </script>
</body>
</html>

```

输出:Hello Vue!,对应的DOM节点渲染为:

```

-----+
  ▼ <div id="app">
    <p>Hello Vue!</p>
  </div>

```

是不是感觉和props的传值差不多，当然这不是 provide/inject该干的事情，我们在中间再嵌套一层组件就知道他的用处了，例如:

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <script src="https://cdn.jsdelivr.net/npm/vue@2.6.10/dist/vue.js"></script>
</head>
<body>
  <div id="app"><test></test></div>
  <script>
    Vue.component('child',{
      inject:['message'],
      template:'<p>{{message}}</p>'
    })
  </script>

```

```

        })
      Vue.component('test',{
        template:`<div><child></child></div>`
      })
    new Vue({
      el:'#app',
      provide:{message:'Hello Vue!'}
    })
  </script>
</body>
</html>

```

```

▼<div id="app">
  ▼<div>
    <p>Hello Vue!</p>
  </div>
</div>

```

provide/inject就是做这个用的，多层嵌套时用起来不要太爽噢

## 源码分析

### provide

```

// src/core/instance/inject.js
export function initProvide (vm: Component) {
  const provide = vm.$options.provide
  if (provide) {
    vm._provided = typeof provide === 'function'
      ? provide.call(vm)
      : provide
  }
}

```

provide 是向下传递数据的选项。这里先拿到 provide 选项中的内容，如果有 provide 选项，将 provide 选项传递给 vm.\_provided 变为 Vue 实例全局数据。

这里看一下例子更清楚，下例中传入数据 foo，数据内容为 bar。

```
var Provider = {
  provide: {
    foo: 'bar'
  },
  // ...
}
```

### ⚠ inject

```
// src/core/instance/inject.js
export function initInjections (vm: Component) {
  const result = resolveInject(vm.$options.inject, vm)
  if (result) {
    observerState.shouldConvert = false
    Object.keys(result).forEach(key => {
      defineReactive(vm, key, result[key])
    })
    observerState.shouldConvert = true
  }
}
```

简化后的源码可以看到，首先通过 `resolveInject` 方法获取 `inject` 选项搜索结果，如果有搜索结果，遍历搜索结果并为其中的数据添加 `setter` 和 `getter`。接着来看下 `resolveInject` 方法：

```
export function resolveInject (inject: any, vm: Component): ?Object {
  if (inject) {
    const result = Object.create(null)
    // 获取 inject 选项的 key 数组
    const keys = hasSymbol
      ? Reflect.ownKeys(inject).filter(key => {
        /* istanbul ignore next */
        return Object.getOwnPropertyDescriptor(inject, key).enumerable
      })
      : Object.keys(inject)
    for (let i = 0; i < keys.length; i++) {
      const key = keys[i]
      const value = inject[key]
      if (value) {
        if (Array.isArray(value)) {
          result[key] = value.map(item => {
            if (isPlainObject(item)) {
              return item
            } else if (isRef(item)) {
              return item.value
            }
            return item
          })
        } else if (isPlainObject(value)) {
          result[key] = Object.create(null)
          for (const k in value) {
            if (value.hasOwnProperty(k)) {
              result[key][k] = value[k]
            }
          }
        } else if (isRef(value)) {
          result[key] = value.value
        } else {
          result[key] = value
        }
      }
    }
  }
}
```

```
: Object.keys(inject)

for (let i = 0; i < keys.length; i++) {
  const key = keys[i]
  const provideKey = inject[key].from
  let source = vm
  while (source) {
    if (source._provided && provideKey in source._provided) {
      result[key] = source._provided[provideKey]
      break
    }
    source = source.$parent
  }
  if (!source) {
    if ('default' in inject[key]) {
      const provideDefault = inject[key].default
      result[key] = typeof provideDefault === 'function'
        ? provideDefault.call(vm)
        : provideDefault
    } else if (process.env.NODE_ENV !== 'production') {
      warn(`Injection "${key}" not found`, vm)
    }
  }
}
return result
}
```

获取 inject 选项的 key 数组

遍历 key 数组，通过向上冒泡来查找 provide 中是否有 key 与 inject 选项中 from 属性同名的

如果有，则将这个数据传递给 result；

如果没有，检查 inject 是否有 default 选项设定默认值或者默认方法，如果有则将默认值返传给 result，最终返回 result 对象。

所以，inject 的写法应该是有 default 默认值的：

```
const Child = {  
  inject: {  
    foo: { default: 'foo' }  
  }  
}
```

或者是有 from 查找键和 default 默认值的：

```
const Child = {  
  inject: {  
    foo: {  
      from: 'bar',  
      default: 'foo'  
    }  
  }  
}
```

或者为 default 默认值设定一个工厂方法：

```
const Child = {  
  inject: {  
    foo: {  
      from: 'bar',  
      default: () => [1, 2, 3]  
    }  
  }  
}
```

好吧，我承认这就是引用的官网的三个例子~ 不过意思到就好啦。

本文使用 mdnice 排版

# 学习vue源码 (19) 四探生命周期之初始化props



前面文章已经把 created钩子函数触发前，beforeCreate触发后的initInjections和initProvide讲完了，现在开始讲 initState的props部分。

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

如代码所示，这一部分，也是created钩子函数触发前，beforeCreate触发后的最后一部分了。

这一部分也叫做初始化状态

当我们使用Vue.js开发应用时，经常会使用一些状态，例如props、methods、data、computed和watch。在Vue.js内部，这些玩状态在使用之前需要进行初始化。

## initState函数

```
export function initState(vm){  
  vm._watchers = [];  
  const opts = vm.$options;  
  if(opts.props) initProps(vm,opts.props);  
  if(opts.methods) initMethods(vm,opts.methods);  
  if(opts.data){  
    initData(vm);  
  }else{  
    <!-- 不存在，则直接使用observe函数观察空对象 -->  
    <!-- observe函数的作用是将数据转换为响应式的 -->  
    observe(vm._data = {},true/*asRootData*/);  
  }  
  if(opts.computed) initComputed(vm,opts.computed);  
  if(opts.watch && opts.watch !== nativeWatch){  
    initWatch(vm,opts.watch);  
  }  
}
```

1、首先在vm上新增一个属性\_watchers，用来保存当前组件中所有的watcher实例。无论是使用vm.\$watch注册的watcher还是使用watch选项添加的watcher实例，都会添加到vm.\_watchers中。

2、可以通过vm.\_watchers得到当前Vue.js实例中所注册的所有watcher实例，并将它们一次卸载。

3、用户在实例化Vue.js时使用了哪些状态，哪些状态就需要被初始化，没有用到的状态则不用初始化。例如，用户只使用了data，那么只需要初始化data即可。

4、初始化的顺序其实是精心安排的。先初始化props，后初始化data，这样就可以在data中使用props中的数据了。在watch中既可以观察props，也可以观察data，因为它是最后被初始化的。

5、初始化状态可以分为5个子项，分别是初始化props、初始化methods、初始化data、初始化computed和初始化watch。

## 初始化props

开始之前，我提出问题

- 1、父组件 怎么传值给 子组件的 props
- 2、子组件如何读取props

这一节，我们带着问题去开始我们的讲解

明白这三个问题，那么相信你也就理解了 props 的工作原理

### 场景设置

现在我们有一个这样的 根组件 A 和 他的 子组件 testb 根组件A 会把 `parentName` 传给 子组件 testb 的 props 根组件A 也是 组件testb 的 父组件

```
<div class="a" >
  <testb :child-name="parentName" ></testb>
</div>
new Vue({
  el:".a",
  name:"A",
  components:{
    testb:{
```

```

    props:{
        childName:""
    },
    template: '<p>父组件传入的 props 的值 {{childName}}</p>',
}
),
data(){
    return {
        parentName:"我是父组件"
    }
},
})
)

```

按照上面的例子，开始我们的问题解析

## 父组件怎么传值给子组件的 **props**

这部分内容是 **props** 的重中之重，必须理解

### 1 props 传值的设置

根据上面的场景设置，`testb` 是一个子组件，接收一个 `props (child-name)` 然后 根组件 A 把自身的 `parentName` 绑定到 子组件的属性 `child-name` 上

```

<div class="a" >
  <testb :child-name="parentName" ></testb>
</div>

```

### 2 props 父传子前

父组件的模板 会被解析成一个 模板渲染函数

```

(function() {
  with(this){
    return _c('div',{staticClass:"a"},[
      _c('testb',{attrs:{'child-name':parentName}})
    ],1)
  }
})

```

```
}
```

```
)
```

这段代码需要解释下

其实，这是属于代码生成器的部分，感兴趣的可以看[学习vue源码 \(9\) 手写代码生成器](#)

1、`_c` 是渲染组件的函数，`_c('testb')` 表示渲染 `testb` 这个子组件

2、因为 `with` 的作用是，绑定大括号内代码的 变量访问作用域

3、这是一个匿名自执行函数，会在后面执行

简化上面的函数，做个例子测试一下

### 3 .props 开始赋值

之后，模板函数会被执行，执行时会绑定父组件为作用域

所以渲染函数内部所有的变量，都会从父组件对象 上去获取

绑定了父作用域之后，`parentName` 自然会从父组件获取，类似这样

```
{ attrs: { child-name: parentVm.parentName } }
```

函数执行了，内部的 `_c('testb')` 第一个执行，然后传入了 赋值后的 attrs

父组件赋值之后的 attrs 就是下面这样

```
{ attrs: { child-name: "我是父组件" } }
```

此时，父组件就正式 利用 props 把 `parentName` 传给了 子组件的props `child-name`

## 4. 子组件保存 props

```
_c('testb',{attrs:{'child-name':parentName}})
```

子组件拿到父组件赋值过后的 attr

而 attrs 包含 普通属性 和 props，所以需要 筛选出 props，然后保存起来

## 5. 子组件 设置响应式 props

props 会被 保存到 实例的 \_props 中，并且 会逐一复制到 实例上，并且每一个属性会被设置为 响应式的

```
▼ VueComponent {_uid: 1, _renderProxy: Proxy,
  ► $root: Vue {_uid: 0, _isVue: true, $options: Object, $parent: null, $children: Array(1), $refs: Object, $el: null, $vnode: VNode<div>, $slots: Object, $deactivated: false, $destroyed: false, $finalized: false, $isMounted: false, $isDestroyed: false, $isDeactivated: false, $isBeingDestroyed: false}, $scopedSlots: {}, $slots: {}}
  ► childName: "我是父组件"
  ► _events: {}
  ▼ _props:
    ► childName: "我是父组件"
      ► get childName: f reactiveGetter()
      ► set childName: f reactiveSetter(newVal)
      ► __proto__: Object
```

你看到的，每一个 实例都会有一个 \_props 的同时，也会把属性直接放在 实例上。

对于props还有个问题：就是我们写的时候可是是数组形式，也是对象形式，这是为什么呢？

### 1. 数组形式

```
props: ['name', 'value']
```

### 2. 对象形式

对象形式内部也提供了三种写法：

```

props: {
  // 基础的类型检查
  name: String,
  // 多个可能的类型
  value: [String, Number],
  // 对象形式
  id: {
    type: Number,
    required: true
  }
}

```

其实 vue 初始化时，会把 props 统一规格化成对象形式

```

function normalizeProps (options: Object, vm: ?Component) {
  const props = options.props
  if (!props) return
  const res = {}
  let i, val, name
  if (Array.isArray(props)) {
    ...
  } else if (isPlainObject(props)) {
    ...
  } else if (process.env.NODE_ENV !== 'production') {
    ...
  }
  options.props = res
}

```

normalizeProps 函数就是 vue 实际处理 props 的地方，从函数名的翻译我们可以看出该函数的功能就是标准化 props 的值。该函数主要分成 3 部分：① 从 options 对象中获取 props 的值并且定义一个 res 空对象；② 几个 if ... else，分别根据 props 值的不同类型来处理 res 对象；③ 用处理后的 res 对象覆盖原来 options 对象的 props 属性的值。

接下来看看那几个 if ... else 的代码：

```

if (Array.isArray(props)) {
  i = props.length
  while (i--) {
    val = props[i]
    if (typeof val === 'string') {
      name = camelize(val)
      res[name] = { type: null }
    } else if (process.env.NODE_ENV !== 'production') {
      warn('props must be strings when using array syntax.')
    }
  }
}

```

这个代码实际就是处理props的值为数组的情况，例如：

```
props: ['name', 'value']
```

。使用while遍历该数组，如果数组内元素的类型不是字符串并且不是生产环境，那么就抛错：‘props的值类型为数组时，数组里面的元素的类型就必须是字符串’。如果是字符串的情况下，使用camelize函数处理一下val的值，并且赋值给name变量。这里的camelize函数的实际作用就是将'-'转换为驼峰。camelize函数具体的实现方式在后面分析。然后在res对象上面添加一个为name变量的属性，该属性的值为空对象 { type: null }。

```
props: ['name', 'value']
```

这种写法经过上面的处理后就会变成了下面这样：

```

props: {
  name: {
    type: null
  },
  value: {
    type: null
  }
}

```

接下来看看下面这个else if(isPlainObject(props))，这里的isPlainObject函数实际就是返回props的值是否为object，isPlainObject函数的具体实现我们也在后面分析。

```
else if (isPlainObject(props)) {
  for (const key in props) {
    val = props[key]
    name = camelize(key)
    res[name] = isPlainObject(val)
      ? val
      : { type: val }
  }
}
```

使用for...in遍历props对象，和上面一样使用camelize函数将'-'转换为驼峰。这里有个三目运算：

`res[name] = isPlainObject(val) ? val : { type: val }` 判断了一下val如果是object，那么在res对象上面添加一个为name变量的属性，并且将该属性的值设置为val。这个其实就是处理下面这种props的写法：

```
props: {
  // 对象形式
  id: {
    type: Number,
    required: true
  }
}
```

如果val不是object，那么也在res对象上面添加一个为name变量的属性，并且将该属性的值设置为`{ type: val }`。这个其实就是处理下面这种props的写法：

```
props: {
  // 基础的类型检查
  name: String,
  // 多个可能的类型
}
```

```
value: [String, Number],  
}
```

经过处理后props会变成了下面这样：

```
props: {  
  name: {  
    type: String  
  },  
  value: {  
    type: [String, Number]  
  }  
}
```

所以不管我们使用vue提供的props哪种写法，最终vue都会帮我们转换成下面这种类型：

```
props: {  
  name: {  
    ...  
    type: '类型'  
  }  
}
```

明白了这些之后，看开头我们提出的initProps函数，就容易理解的多了

## initProps

如果觉得有点难理解，可以先跟着代码后面的解释，逐行明白每一行代码的作用。

```
function initProps (vm: Component, propsOptions: Object) {  
  
  const propsData = vm.$options.propsData || {}  
  
  const props = vm._props = {}
```

```
const keys = vm.$options._propKeys = []

const isRoot = !vm.$parent

if (!isRoot) {
  toggleObserving(false)
}

for (const key in propsOptions) {
  //
  keys.push(key)

  const value = validateProp(key, propsOptions, propsData, vm)

  // 非生产环境下进行检查和提示
  if (process.env.NODE_ENV !== 'production') {
    //
    const hyphenatedKey = hyphenate(key)
    //

    if (isReservedAttribute(hyphenatedKey) ||
        config.isReservedAttr(hyphenatedKey)) {
      warn(
        `${hyphenatedKey}` + ` is a reserved attribute and cannot be used as component prop.`,
        vm
      )
    }
    defineReactive(props, key, value, () => {
      if (vm.$parent && !isUpdatingChildComponent) {
        warn(
          `Avoid mutating a prop directly since the value will be ` +
          `overwritten whenever the parent component re-renders. ` +
          `Instead, use a data or computed property based on the prop's ` +
          `value. Prop being mutated: "${key}"``,
          vm
        )
      }
    })
  } else {
    defineReactive(props, key, value)
  }

  if (!(key in vm)) {
    proxy(vm, `_props`, key)
  }
}
```

```

    }
    // 开启观察状态标识
    toggleObserving(true)
}

```

1. 接收vm，propsOptions两个参数，propsOptions是我们开发者写的组件里的props。
2. 赋值propsData，propsData父组件传入的props
3. 定义实例的\_props私有属性，并赋值给props
4. key用来缓存prop键，以便将来props更新可以使用Array而不是动态对象键枚举进行迭代。
5. isRoot表示是否是根实例，对于非根实例，关闭观察标识即toggleObserving(false)
6. 遍历props配置对象，向缓存键值数组key中添加键名。
7. 验证prop的值：validateProp执行对初始化定义的props的类型检查和默认赋值，如果有定义类型检查，布尔值没有默认值时会被赋予false，字符串默认undefined。意思就是如果你在子组件写的props的类型是布尔值的话例如：

```

props : {
  flag : {
    type : Boolean
  }
}

```

那么即使父组件没有传入这个flag，那flag这个变量能用，且值是false，如果有默认值，例如

```

props : {
  flag : {
    type : Boolean ,
    default : true
  }
}

```

那么即使父组件没有传入这个flag，那flag这个变量能用，且值是默认值。

如果父组件传入，也就是说，下面这些情况，子组件中flag的值都是true

```
<child flag></child>  
  
<child flag="flag"></child>
```

如果prop是其他类型，父组件没传入的话，那就看有没有设置默认值，有的话就用默认值。

validateProp完成的就是这个功能，可以说是对props的值进行加工生成value

8. hyphenate 进行键名的转换，将驼峰式转换成连字符式的键名。例如user-name就会转化成userName
9. isReservedAttribute用于判断prop是不是保留变量名，是的话报错提示
10. prop和data一样都要通过defineReactive建立数据监听，成为响应式。
11. proxy则是代理了。

ok,props的初始化就讲到这里啦。

本文使用 mdnice 排版