## Vue源码解析03



## 复习

https://www.processon.com/view/link/5e830387e4b0a2d87023890a

## 学习目标

- 组件化原理
- 手写vue2

### 组件化机制

组件声明: Vue.component()

initAssetRegisters(Vue) src/core/global-api/assets.js

组件注册使用extend方法将配置转换为构造函数并添加到components选项

- 全局声明
- Vue.component()
- 局部声明
  - o components

### 组件实例创建及挂载

观察生成的渲染函数

```
"with(this){return _c('div',{attrs:{"id":"demo"}},[
    _c('h1',[_v("虚拟DOM")]),_v(" "),
    _c('p',[_v(_s(foo))]),_v(" "),
    _c('comp') // 对于组件的处理并无特殊之处
],1)}"
```

### 整体流程

首先创建的是根实例,首次\_render()时,会得到整棵树的VNode结构,其中自定义组件相关的主要有:

#### createComponent() - src/core/vdom/create-component.js

组件vnode创建

#### createComponent() - src/core/vdom/patch.js

创建组件实例并挂载, vnode转换为dom

#### 整体流程:

new Vue() => \$mount() => vm.\_render() => createElement() => createComponent()
=> vm.\_update() => patch() => createElm => createComponent()

### 创建组件VNode

#### \_createElement - src\core\vdom\create-element.js

\_createElement实际执行VNode创建的函数,由于传入tag是非保留标签,因此判定为自定义组件通过 createComponent去创建

#### createComponent - src/core/vdom/create-component.js

创建组件VNode,保存了上一步处理得到的组件构造函数,props,事件等

#### 创建组件实例

根组件执行更新函数时,会递归创建子元素和子组件,入口createElm

#### createEle() core/vdom/patch.js line751

首次执行\_update()时,patch()会通过createEle()创建根元素,子元素创建研究从这里开始

#### createComponent core/vdom/patch.js line144

自定义组件创建

#### // 组件实例创建、挂载

#### 开课吧web全栈架构师

```
if (isDef(i = i.hook) && isDef(i = i.init)) {
    i(vnode, false /* hydrating */)
}

if (isDef(vnode.componentInstance)) {
    // 元素引用指定vnode.elm, 元素属性创建等
    initComponent(vnode, insertedVnodeQueue)
    // 插入到父元素
    insert(parentElm, vnode.elm, refElm)
    if (isTrue(isReactivated)) {
        reactivateComponent(vnode, insertedVnodeQueue, parentElm, refElm)
    }
    return true
}
```

## 手写实现vue 2

## 起始点

在之前vue1的基础上实现vue2,主要修改点是将compile干掉,替换为vnode方式。

### vue1实现

```
// 实现KVue构造函数
function defineReactive(obj, key, val) {
 // 如果val是对象,需要递归处理之
 observe(val)
 // 管家创建
 const dep = new Dep()
 Object.defineProperty(obj, key, {
   get() {
     console.log('get', key);
     // 依赖收集
     Dep.target && dep.addDep(Dep.target)
     return val
   },
   set(newVal) {
     if (val !== newVal) {
       // 如果newVal是对象, 也要做响应式处理
       observe(newVal)
       val = newVal
       console.log('set', key, newVal);
       // 通知更新
       dep.notify()
```

```
})
}
// 遍历指定数据对象每个key, 拦截他们
function observe(obj) {
 if (typeof obj !== 'object' || obj === null) {
   return obj
 }
 // 每遇到一个对象,就创建一个Observer实例
 // 创建一个Observer实例去做拦截操作
 new Observer(obj)
}
// proxy代理函数: 让用户可以直接访问data中的key
function proxy(vm, key) {
 Object.keys(vm[key]).forEach(k => {
   Object.defineProperty(vm, k, {
     get() {
       return vm[key][k]
     },
     set(v) {
       vm[key][k] = v
     }
   })
 })
}
// 根据传入value类型做不同操作
class Observer {
 constructor(value) {
   this.value = value
   // 判断一下value类型
   // 遍历对象
   this.walk(value)
 }
 walk(obj) {
   Object.keys(obj).forEach(key => {
     defineReactive(obj, key, obj[key])
   })
 }
}
class KVue {
 constructor(options) {
```

```
// 0.保存options
    this. $ options = options
    this.$data = options.data
   // 1.将data做响应式处理
   observe(this.$data)
   // 2.为$data做代理
   proxy(this, '$data')
   // 3.编译模板
   // new Compile('#app', this)
 }
}
// 移除
// class Compile {}
class Watcher {
 constructor(vm, key, updaterFn) {
   this.vm = vm
   this.key = key
   this.updaterFn = updaterFn
   // 依赖收集触发
   Dep.target = this
   this.vm[this.key]
   Dep.target = null
  }
 update() {
   this.updaterFn.call(this.vm, this.vm[this.key])
 }
}
// 管家: 和某个key, ——对应, 管理多个秘书, 数据更新时通知他们做更新工作
class Dep {
 constructor() {
   this.deps = []
  }
 addDep(watcher) {
   this.deps.push(watcher)
  }
 notify() {
   this.deps.forEach(watcher => watcher.update())
  }
}
```

### 测试代码

```
<div id="app"></div>
<script src="kvue2.js"></script>
<script>
  const app = new KVue({
    el: '#app',
    data: {
       counter: 1
    }
})
setInterval(() => {
    app.counter++
}, 1000);
</script>
```

## 总结

Vue2.x降低watcher粒度,引入VNode和Patch算法,大幅提升了vue在大规模应用中的适用性、扩平台的能力和性能表现,是一个里程碑版本。但是同时也存在一定问题:

- 数据响应式实现在性能上存在一些问题,对象和数组处理上还不一致,还引入了额外的API
- 没有充分利用预编译的优势,patch过程还有不少优化空间
- 响应式模块、渲染器模块都内嵌在核心模块中,第三方库扩展不便
- 静态API设计给打包时的摇树优化造成困难
- 选项式的编程方式在业务复杂时不利于维护
- 混入的方式在逻辑复用方面存在命名冲突和来源不明等问题

## 作业

按课上实现手写vue2

要求: 学习中心提交代码**截图**和**代码** 通过标准: 能够正常运转, 完成既定功能

## 思考拓展

探索其他感兴趣的特性,比如: keep-alive、插槽机制、编译器工作原理等。

# 预告

我给大家准备了丰富的vue面试训练营内容,深入讲解常见的面试题解答思路和加分策略。

相关代码可以获取, 戳这里。

后面几节课我们要学习vue3的源码了,如果你还未接触过vue3,最好提前学习一下,我也给大家准备好了,<u>戳这里</u>,三连的同学必定升职加薪!

