**综述**

**一、总览:**

vue2.0组成:

响应式原理: 监听数据改变, 修改dom

虚拟dom: 最小化修改dom的代价

过程:

将template编译成render函数

实例化vue, 生成oldVnode, 生成oldDom

数据改变 -> 执行render函数生成newVnode -> 对比oldVnode,newVnode -> 修改oldDom为newDom

**响应式**

**一、总览:**

本章着重于响应式的原理, 所以vdom会简写

**1. 分享逻辑(分享后删除)**

0 看看vue2 的实际表现 vue-pricinple里面, 修改数据, 看看表现

0 写在之前

源码经过了数次迭代, 向后兼容, 即使删除了很多无效信息, 也始终让人剪不断理还乱,

与其从终点回到起点, 不如从起点开始, 轻装上阵

省略template编译到->render的过程, 直接使用render函数

弱化虚拟dom的原理

代码中会用闭包来便于切换代码

**1 监听原理**

不想什么 依赖收集, 派发更新, dep, watcher ...之类的名词, 单纯想怎么实现数据改变, 视图更新

总得有个东西监听把

Object.defineProperty 可以监听, 打印输出

**2 监听可以操作dom**

手动给初始值 -> 监听改变 -> 改变更新数据

**3 将修改dom的方式封装一下, 支持多个数据, dom元素**

将响应式封装一下

**4 只让用到的数据改变更新视图**

如果data有很多数据, 改每个都要操作dom吗?? 如何只让用到的数据改变了才更新视图

->

在首次setDom的时候, 因为对数据取值了, 所以可以识别出那些数据使用了

在这个数据使用时, 把setDom方法挂载到这个数据上, 等这个数据改变了, 就执行setDom方法

->

使用一个数组dep保存更新的方法, 当用到数据将setDom推入dep, 修改数据执行dep中的setDom

**5 那就封装成一个vue的样子把**

该有的都有 !!!

.1 把监听相关的封装到observe里面

.2 setDom, data, app的dom元素的处理都放在vue实例里面,

.3 为什么observe没放进vue里面? 这个方法比较公用, 所以没放进去, 后面会证明的

.4 和上次比较

监听数据放在了initData里面, setDom放在了render里面

.5 卡住了....

我们怎么把render方法传到observe里面 ??

已经说了observe不能放进去...

而且如果不执行mount, 实际上不应该有刷新数据的操作的

**6 watcher**

.0 解决上面的没法将render方法传入observe的问题

需要一个桥梁, 只有在执行mount后, 才将render方法传递到数据的dep中

来一个变量控制? 太脏了...

来一个watcher把, new watcher后 其实也是用一个标志位通知数据, dep可以推入render方法了

.1 而且真实情况渲染页面可能是针对不同的数据, 不同的渲染方式, 遵循面向拓展的编程, 最好也来个watcher

.2 最初更新dom时不是整个dom全渲染, 而是渲染执行节点(vue1.0), 所以vue1.0中, 每个数据的更新操作其实是针对不同节点的更新 // 题外话

.3 给用户自定义监听数据的可能性

// 下集预告

6 怎么自定义模版? 数据代理, depIds?, ....

// 下下集预告

数组, 对象, 处理不同 ? $set, $delete ? 男默女泪的官方填坑操作?

参考:

[为什么数组不能监听元素改变](https://segmentfault.com/a/1190000015783546)

[数组依赖收集, 更新过程](https://juejin.im/post/5d579cd36fb9a06aea6190db#heading-3)

[$set, $del 和更新](https://juejin.im/post/5d54133ce51d4561f777e197)

**一、外部表现:**

**1. 使用vue中触发视图渲染的情况**

|  |  |
| --- | --- |
|  | 可修改情况: |
| 改变数据地址 | 是 |
| 模版中使用data未声明数据 | 否, 报错 |
| 改变数组的元素 | 否 |
| 增加数组的元素 | 否 |
| 通过下标删除数组元素 | 否 |
| 改变对象的属性值 | 是 |
| 增加对象的属性值 | 否 |
| 删除对象属性 | 是 |

tip:

1 此处对于数据的操作是通过 vm, 或者在实例内部的mounted中

至于在实例内部的created则不满足上述的表现, 因为此时还没执行render, 还没有依赖收集

**一、知识基础:**

**1. 监听数组/对象 key的读写**

这里核心主要是object.defineProperty

这个方法可以监听对象, 数组中已有属性值, 索引值的变化

但是不能监听对象, 数组中属性值的增添

以数组为例:

var arr = [1, 2, 3, 4]

arr.forEach((item, index) => {

Object.defineProperty(arr, index, {

set: function (val) {

console.log('set')

item = val

},

get: function (val) {

console.log('get')

return item

}

})

})

arr[1]; // get 2

arr[1] = 1; // set 1

arr[10] = 10 // 没有输出!

**2. 监听数组调用了api**

通过Object.create创建一个全新的数组原型对象arrayProto, 在这上面对执行数组变异方法添加额外处理, 然后将arrayProto作为想监听数组的原型

tip: 不能直接重写Array.prototype, 会污染全局所有数组

// 获得原型上的方法

const arrayProto = Array.prototype;

// 创建一个隔离的新对象，使用现有的对象来提供新创建的对象的\_\_proto\_\_

const arrayMethods = Object.create(arrayProto);

// 做一些拦截的操作, 此处只处理push方法一个

Object.defineProperty(arrayMethods, 'push', {

value(...args) {

console.log('用户传进来的参数', args);

// 真正的push 保证数据如用户期望

arrayProto.push.apply(this, args);

},

enumerable: true,

writable: true,

configurable: true,

});

let list = [1];

list.\_\_proto\_\_ = arrayMethods; // 重置原型

list.push(2, 3);

console.log('用户得到的list:', list);

**响应式原理:**

**1. 流程总览**

名词解析:

收集器: dep实例, 可以收集依赖

依赖: 就是一个watcher, 实际上就是一次render函数更新页面的操作

更新: 执行收集上来的依赖

**核心流程:**

**observe(vm.$data)**

观测数据, 定义了一些:

可以收集依赖的情况, 如何收集依赖,

可以更新的条件, 如何派发更新

**new Watcher(vm, render)**

触发可以收集依赖的条件, 收集依赖

**this.msg = 1/items.push() /$set**

触发派发更新的条件, 派发更新

tip:

new Vue时执行observer, 调用$mount时执行new Watcher, 改变数据时就是改变数据

**依赖收集, 派发更新的定义:**

真实情况是, vm.$data中的数据可能是 对象,数组相互嵌套

从$data开始, 递归向下观测, observer方法

观测每个元素value时, 都会new一个observer实例, 并value.\_\_ob\_\_ = observer

observer有个dep (手动收集器)

数组:

观测自身之后, 再观测子元素

重写数组的变异方法, 只要触发这些方法, 就会让数组手动收集器派发更新

对象:

观测自身之后, 观测每个val

每个key都会有自己的dep (自动收集器), 只要对key读取操作, 就会让自动收集器收集依赖, 同

时会让val的手动收集器手机依赖

**触发依赖收集:**

new watcher之后, Dep.target = watcher, 执行了一次render, 触发了对象中key的get钩子

在get钩子中, 收集了key.dep对应的依赖, 同时也收集了val.\_\_ob\_\_.dep对应的依赖

? 数组的依赖能收集吗 ?

例如: vm.$data = {items: arr}, 只要render函数中有this.items的操作, items的自动收集器就会收集, 同时arr的手动收集器也会被收集

**派发更新:**

对象: 修改value, 直接触发set钩子, 执行key对相应的dep的notify方法

数组: 数组变异方法执行时, 在对变异方法的劫持中, 通过\_\_ob\_\_获取数组的observer, 手动

observer.dep.notify

**虚拟dom:**

dom操作由虚拟dom接管, 此章节不是重点

**官方api派发更新:**

即使数组变异方法可以迫使数组更新, 而且修改数组元素的属性也可以更新, 但是只修改元素则无法更新

**set**

数组: 调用splice方法, 触发更新

对象: defineReactive手动设置响应式, \_\_ob\_\_.dep.notify 手动更新

**delete**

数组: 调用splice方法, 触发更新

对象: delete obj[key]删除属性, \_\_ob\_\_.dep.notify 手动更新

**2. 总结**

|  |  |  |
| --- | --- | --- |
|  | 数组 | 对象 |
| 收集依赖 | get钩子的:  \_\_ob\_\_.dep | get钩子的:  key.dep  \_\_ob\_\_.dep |
| 触发数组变异方法 | 对变异方法劫持中手动更新 | 无 |
| 改变key/index | $set调数组变异方法, 自动更新 | set钩子: key.dep.notify |
| 增加key/index | $set调数组变异方法, 自动更新 | $set增加新key响应式  手动更新 |
| 删除key/index | $set调数组变异方法, 自动更新 | $set删除key  手动更新 |

**话题2**

**一、话题:**

**一、话题:**

**1.话题**