课程目标

前面的课程有通过收集依赖，依赖通知更新后触发更新异步队列，然后再通过virtual dom 虚拟dom diff算法比对，渲染页面。这样达成了一个闭环。这节课我们来说一说闭环之外的。我们都了解了vue3.0 通过proxy代理对象替代了Object.defineProerty()来进行get和set的拦截操作。为什么用proxy来代理Object.defineProerty()呢?

其中一个很重要的原因是Object.defineProerty() 不能监听到对象上添加或者删除属性的变化。除此之外vue3.0还实现了compostion API这个概念。

Vue 3.0 Reactivity APIS

- **reactive**
- **ref**
- **readonly**
- **computed**
- **watchEffect**
- **watch**

下面我们实现 computed和watch 和watchEffect(后面补上，先看Vue3.0)

Vue3.0中reacttive函数的实现

```
/**
  vue3.0中的 reactive函数是通过es6中的proxy，
  vue2.0是用的Object.definedProperty()
  proxy在目标对象的外层搭建了一层拦截，外界对目标对象的某些操作，
  必须通过这层拦截，new Proxy()表示一个Proxy实例,target参数表示所要拦截的目标对象
  handler为一个对象，其属性是当执行一个操作时定义代理的行为的函数(可以理解为某种触发器)。
  Reflect 是一个内置的对象，它提供拦截 JavaScript 操作的方法。Reflect不是一个函数对象，因此它是不可构造的。
*/
```

```
(function (root) {
  //vue响应式源码核心代码实现,封装deps依赖收集的类
  let activeCallback;//定义当前的回调函数
  let queen = [] //存储异步队列
  const queenAdd = add =>{ //添加到异步队列当中
  if(!queen.includes(add)){
  queen.push(add)
  nexTick(executeQueen)
  }else {
  nexTick(executeQueen)
  }
  }
  const executeQueen = ()=>{ //执行异步任务列的所有任务
  if(queen.length>0&&queen[0]){
  queen.forEach(fn=>{
  fn()
  })
  }
  }
  //这里我们需要按照Event Loop那样 执行宏任务后，清空所有的微任务队列，
  // 所有我们把添加任务队列函数,添加到微任务队列中
  const nexTick = callback => Promise.resolve().then(callback)
  //依赖收集的类
  class Dep {
  constructor() {
  this.deps = new Set() //Set存在add方法
  }
  //收集依赖
  depend(){
  if(activeCallback){
  this.deps.add(activeCallback)
  }
  }
  //通知依赖更新
  notify(){
  this.deps.forEach(dep=>queenAdd(dep)) //这里的依赖更新，先把依赖的添加微任务队列当中去。
  }


```

```
48  }
49  const dep = new Dep() //实例化dep类
50  //监听函数
51  const watch = function(callback){
52  activeCallback = callback
53  activeCallback() //初始的调用调用
54  activeCallback = null // 销毁当前的activeCallback
55  }
56  // Object.defineProperty() 创建对象
57  // 创建响应式
58  const reactive =(target = {})=>{
59  if (typeof target !== 'object' || target == null) {
60  return target
61  }
62  const handler= {
63  get(target, key, receiver) {
64  // 只处理本身（非原型的）属性
65  // target是对象，ownKeys是属性的数组
66  // target是数组，ownKeys是索引数组，返回一个包含所有自身属性（不包含继承属性）的数组。
67  // ownKeys(类似于 Object.keys(), 但不会受enumerable影响).
68  const ownKeys = Reflect.ownKeys(target)
69  if (ownKeys.includes(key)) {
70  dep.depend() // 添加监听
71  }
72  const result = Reflect.get(target, key, receiver) //获取对象身上某个属性的值，类似于 target[name]。
73  // 深度监听
74  return reactive(result) // 这里利用递归，进行深拷贝监听(这个是关键)
75  },
76  set(target, key, val, receiver) {
77  // 重复的数据，不处理
78  if (val === target[key]) {
79  return
80  }
81  const result = Reflect.set(target, key, val, receiver) // 将值分配给属性的函数。返回一个Boolean，如果更新成功，则返回true。
82  dep.notify() // 通知监听队列进行更新
83  // 是否设置成功
84  return result
85  },
```

```
  deleteProperty(target, key) {
  const result = Reflect.deleteProperty(target, key)
  // 是否删除成功 暂时不扩展
  return result
  }
  }
  const observed = new Proxy(target, handler)
  return observed
  }
  window.reactive = reactive
  window.watch = watch
})(window)

```