课程目标

在上一节课程当中，我们知道，监听函数中，每次修改数据，都会重新渲染模板。

```
//初始化函数调用
let x = ref( initValue: 1)
let y = ref( initValue: 2)
let z = ref( initValue: 3)
// 监听函数初始化调用
watch( callback: ()=>{
    let tpl = `hello ${x.value} ${y.value} ${z.value}`
    console.log(tpl);
    document.write(tpl)
})
//修改x的值，触发ref对象中的set函数
//因为添加到微任务队列
x.value = 2
z.value = 2
y.value = 2
```

```
[HMR] Waiting for update signal from WDS...
hello 1 2 3
hello 2 2 3
hello 2 2 2
hello 2 2 2
```

如果监听函数中，不单单有x 还有y z等等监听的变量，模板中有多处变量依赖，每一次变量修改的时候，都是会导致一次渲染，这里渲染了三次，这是否可以优化呢?

在vue当中的，收集依赖，通知依赖更新后会触发vue的异步渲染队列，异步渲染队列就可以解决上述问题。

异步更新队列

vue当中的异步更新队列,当我们在一个函数块中包含多个变量依赖的时候，可以将这些依赖放入一个队列当中去，等到当前函数块执行完毕之后，再进行一次批量的渲染操作。看具体的实现代码

接着上一节课的代码

先定义一个任务队列 添加 执行任务队列的函数，任务队列都定义为微任务

```
let queen = [] //存储异步队列

let queenAdd = add =>{ //添加到异步队列当中
 if(!queen.includes(add)){
 queen.push(add)
 nexTick(executeQueen) //这就是关键 添加任务队列
 }
}

let 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

let nexTick = callback => Promise.resolve().then(callback) //也是关键
```

完整的

```
let activeCallback;//定义当前的回调函数

let queen = [] //存储异步队列

let queenAdd = add =>{ //添加到异步队列当中
 if(!queen.includes(add)){
 queen.push(add)
 nexTick(executeQueen)
 }
}

let 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

let 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。
 }


}
//监听函数
let watch = function(callback){
 activeCallback = callback
 activeCallback() //初始的调用调用
 activeCallback = null // 销毁当前的activeCallback
}
// Object.defineProperty() 创建对象
let ref = (initValue)=>{
 let value = initValue
 let dep = new Dep() //实例化dep类
```

```
 return Object.defineProperty({},'value',{
 get() {
 //依赖收集
 dep.depend()
 return value
 },
 set(newValue) {
 value = newValue
 dep.notify()//当修x属性的时候，也需要进行相应的依赖更新
 }
 })
}

//初始化函数调用
let x = ref(1)
let y = ref(2)
let z = ref(3)
// 监听函数初始化调用
watch(()=>{
 let tpl = `hello ${x.value} ${y.value} ${z.value}`
 console.log(tpl);
 document.write(tpl)
})
//修改x的值，触发ref对象中的set函数
//因为添加到微任务队列
x.value = 2
z.value = 2
y.value = 2
```

修改过后的结果。等所有的宏任务执行完毕后，再清空微任务队列。所有等微任务执行完毕后，才会执行一次watch回调。

```
[HMR] Waiting for update signal from WDS...  log.js?1afd:24
hello 1 2 3                                  main.js?56d7:126
hello 2 2 2                                  main.js?56d7:126
>
```

同时我们可以通过 打印日志可以看出

```
console.log('edit',tpl)
x.value = 2
z.value = 2
y.value = 2
console.log('edit',tpl)
```

```
hello 1 2 3
edit hello 1 2 3
edit hello 1 2 3
hello 2 2 2
```

只有当宏任务执行完毕后,打印出 edit hello 1 2 3 edit hello 1 2 3 然后再去清空的微任务队列，才会执行watch函数，渲染模板。打印出hello 2 2 2

nextTick

在下次DOM更新循环结束之后执行延迟回调。 nextTick 有两处 一个处是全局的，Vue.nexTick([callback,context]) 另外一处是每个实例 vm.$nextTick([callback])。这两个方法通常在修改数据之后使用这个方法，在回调中获取更新后的DOM。

```
mounted: function () {
  this.$nextTick(function () {
    // 这里代码将在当前组件和所有子组件挂载完毕之后执行
  })
}
```