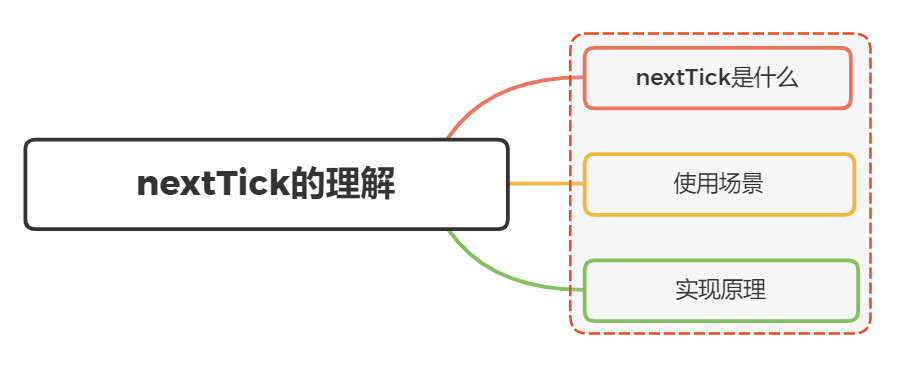
# 面试官：Vue中的$nextTick有什么作用？



## 一、NextTick是什么

官方对其的定义

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

什么意思呢？

我们可以理解成，Vue 在更新 DOM 时是异步执行的。当数据发生变化，Vue将开启一个异步更新队列，视图需要等队列中所有数据变化完成之后，再统一进行更新

举例一下

Html结构

<div id="app"> {{ message }} </div>

构建一个vue实例

const vm = new Vue({  
 el: '#app',  
 data: {  
 message: '原始值'  
 }  
})

修改message

this.message = '修改后的值1'  
this.message = '修改后的值2'  
this.message = '修改后的值3'

这时候想获取页面最新的DOM节点，却发现获取到的是旧值

console.log(vm.$el.textContent) // 原始值

这是因为message数据在发现变化的时候，vue并不会立刻去更新Dom，而是将修改数据的操作放在了一个异步操作队列中

如果我们一直修改相同数据，异步操作队列还会进行去重

等待同一事件循环中的所有数据变化完成之后，会将队列中的事件拿来进行处理，进行DOM的更新

#### 为什么要有nexttick

举个例子

{{num}}  
for(let i=0; i<100000; i++){  
 num = i  
}

如果没有 nextTick 更新机制，那么 num 每次更新值都会触发视图更新(上面这段代码也就是会更新10万次视图)，有了nextTick机制，只需要更新一次，所以nextTick本质是一种优化策略

## 二、使用场景

如果想要在修改数据后立刻得到更新后的DOM结构，可以使用Vue.nextTick()

第一个参数为：回调函数（可以获取最近的DOM结构）

第二个参数为：执行函数上下文

// 修改数据  
vm.message = '修改后的值'  
// DOM 还没有更新  
console.log(vm.$el.textContent) // 原始的值  
Vue.nextTick(function () {  
 // DOM 更新了  
 console.log(vm.$el.textContent) // 修改后的值  
})

组件内使用 vm.$nextTick() 实例方法只需要通过this.$nextTick()，并且回调函数中的 this 将自动绑定到当前的 Vue 实例上

this.message = '修改后的值'  
console.log(this.$el.textContent) // => '原始的值'  
this.$nextTick(function () {  
 console.log(this.$el.textContent) // => '修改后的值'  
})

$nextTick() 会返回一个 Promise 对象，可以是用async/await完成相同作用的事情

this.message = '修改后的值'  
console.log(this.$el.textContent) // => '原始的值'  
await this.$nextTick()  
console.log(this.$el.textContent) // => '修改后的值'

## 三、实现原理

源码位置：/src/core/util/next-tick.js

callbacks也就是异步操作队列

callbacks新增回调函数后又执行了timerFunc函数，pending是用来标识同一个时间只能执行一次

export function nextTick(cb?: Function, ctx?: Object) {  
 let \_resolve;  
  
 // cb 回调函数会经统一处理压入 callbacks 数组  
 callbacks.push(() => {  
 if (cb) {  
 // 给 cb 回调函数执行加上了 try-catch 错误处理  
 try {  
 cb.call(ctx);  
 } catch (e) {  
 handleError(e, ctx, 'nextTick');  
 }  
 } else if (\_resolve) {  
 \_resolve(ctx);  
 }  
 });  
  
 // 执行异步延迟函数 timerFunc  
 if (!pending) {  
 pending = true;  
 timerFunc();  
 }  
  
 // 当 nextTick 没有传入函数参数的时候，返回一个 Promise 化的调用  
 if (!cb && typeof Promise !== 'undefined') {  
 return new Promise(resolve => {  
 \_resolve = resolve;  
 });  
 }  
}

timerFunc函数定义，这里是根据当前环境支持什么方法则确定调用哪个，分别有：

Promise.then、MutationObserver、setImmediate、setTimeout

通过上面任意一种方法，进行降级操作

export let isUsingMicroTask = false  
if (typeof Promise !== 'undefined' && isNative(Promise)) {  
 //判断1：是否原生支持Promise  
 const p = Promise.resolve()  
 timerFunc = () => {  
 p.then(flushCallbacks)  
 if (isIOS) setTimeout(noop)  
 }  
 isUsingMicroTask = true  
} else if (!isIE && typeof MutationObserver !== 'undefined' && (  
 isNative(MutationObserver) ||  
 MutationObserver.toString() === '[object MutationObserverConstructor]'  
)) {  
 //判断2：是否原生支持MutationObserver  
 let counter = 1  
 const observer = new MutationObserver(flushCallbacks)  
 const textNode = document.createTextNode(String(counter))  
 observer.observe(textNode, {  
 characterData: true  
 })  
 timerFunc = () => {  
 counter = (counter + 1) % 2  
 textNode.data = String(counter)  
 }  
 isUsingMicroTask = true  
} else if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {  
 //判断3：是否原生支持setImmediate  
 timerFunc = () => {  
 setImmediate(flushCallbacks)  
 }  
} else {  
 //判断4：上面都不行，直接用setTimeout  
 timerFunc = () => {  
 setTimeout(flushCallbacks, 0)  
 }  
}

无论是微任务还是宏任务，都会放到flushCallbacks使用

这里将callbacks里面的函数复制一份，同时callbacks置空

依次执行callbacks里面的函数

function flushCallbacks () {  
 pending = false  
 const copies = callbacks.slice(0)  
 callbacks.length = 0  
 for (let i = 0; i < copies.length; i++) {  
 copies[i]()  
 }  
}

**小结：**

1. 把回调函数放入callbacks等待执行
2. 将执行函数放到微任务或者宏任务中
3. 事件循环到了微任务或者宏任务，执行函数依次执行callbacks中的回调

## 参考文献

* https://juejin.cn/post/6844904147804749832