37 Vue核心——nextTick原理源码级解析

更新时间：2020-08-28 10:54:25

> 学习从来无捷径，循序渐进登高峰。 —— 高永祚

nextTick 是 Vue 异步更新策略的核心，同时涉及到 Event-Loop 这样的优质考点，是 Vue 面试中不可多得的综合性出题方向。

对于 nextTick，我的建议是直接读源码——读完整的源码。不同于响应式原理，nextTick 所涉及的 Vue 源码内容非常集中、代码量短小精悍，阅读源码的时间成本相比之下非常低。这种情况下，比起大量阅读各路面经、了解其作者对 nextTick 个人化的理解，不如直接从源码中吸收思路、形成属于自己的一套理解。

本节，我们就带大家来分模块理解 nextTick 源码。

nextTick 初相见

在深入其源码之前，首先要建立起一个感性的认知——nextTick 到底是个啥东西？

我们知道，Vue 是异步更新，也就是说，假如你触发了下面这样一个事件：

```
methods: {
  update() {
    for (let i = 0; i < 10; i++) {
      this.testNum = this.testNum + i;
    }
  },
},
```

在你的 Vue 视图中， testNum 会发生变化。不过需要注意的是这个变化的过程，虽然我们把 firstNum 循环修改了 10 次，但是实际上它只会把最后一次的值更新到视图上——这也是非常合理的，比如说我们这个 demo 里，每一次循环给 testNum 的赋值只不过是一个过程，最终的目的是拿到 10 次循环的计算结果而已。如果我们硬去算 10 次，那么不必要的性能开销必然是令人肉疼的。

Vue 很聪明，它知道“心急吃不了热豆腐”。当数据更新发生时，它不会立刻给你执行视图层的更新动作。而是先把这个更新给“存起来”，等到“时机成熟”再执行它；这个“存起来”的地方，叫做异步更新队列；即便一个 watcher 被多次触发，它也只会被推进异步更新队列一次。在同步逻辑执行完之后，watcher 对应的就是其依赖属性的最新的值。最后，Vue 会把异步更新队列的动作集体出队，批量更新。

这个实现异步任务派发的接口，就叫做 “nextTick”。

不得不说的 Event-Loop

说到“异步更新队列”，相信不少同学会联想起一些有趣的知识点——没错，异步队列，最最典型的莫过于咱们前面讲过的 macro-task-queue 和 micro-task-queue。实际上，Vue 非常“懒”，它并没有自己去实现和维护一套异步队列逻辑，而是完全依赖于浏览器暴露的 api 接口来实现异步任务的派发。

因此，在阅读源码之前，大家务必确保自己对 浏览器中的 event-loop 这节有了扎实的掌握。最好是能回去再通读一遍原文，确保自己接下来读源码的思路不要被打断。

Vue-nextTick 源码概览

说是逐行解析，咱们一个字儿都不少，这里我直接把 vue/src/core/util/next-tick.js 这个文件里的代码祭出来：

（注释里有我简化的解析，大家在阅读的过程中，可以先根据解析的提示自己理解一下。如果实在理解不动，再看我后面的详细解析）

```
import { noop } from 'shared/util'
import { handleError } from './error'
import { isIE, isIOS, isNative } from './env'

export let isUsingMicroTask = false

const callbacks = []
let pending = false

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

// 用来派发异步任务的函数
let timerFunc
```

```
// 下面这一段逻辑，是根据浏览器的不同，选择不同的 api 来派发异步任务
if (typeof Promise !== 'undefined' && isNative(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]'
)) {
  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)) {
  timerFunc = () => {
    setImmediate(flushCallbacks)
  }
} else {
  // Fallback to setTimeout.
  timerFunc = () => {
    setTimeout(flushCallbacks, 0)
  }
}

// 暴露 nextTick 方法
export function nextTick (cb?: Function, ctx?: Object) {
  let _resolve
  // 维护一个异步更新队列
  callbacks.push(() => {
    if (cb) {
      try {
        cb.call(ctx)
      } catch (e) {
        handleError(e, ctx, 'nextTick')
      }
    } else if (_resolve) {
      _resolve(ctx)
    }
  })
  // pending 是一个锁，确保任务执行的有序性
  if (!pending) {
    pending = true
    timerFunc()
  }
  // 兜底逻辑，处理入参不是回调的情况
  if (!cb && typeof Promise !== 'undefined') {
    return new Promise(resolve => {
      _resolve = resolve
    })
  }
}
```

Vue-nextTick 源码分模块解析

在上面这段代码里，有三个关键角色：nextTick、timerFunc 和 flushCallbacks，我们逐个来看。首先，nextTick 是入口函数，也是主角，我们看看它做了什么：

逻辑统筹者——nextTick

```
// 暴露 nextTick 方法
export function nextTick (cb?: Function, ctx?: Object) {
  let _resolve
  // 维护一个异步更新队列
  callbacks.push(() => {
    if (cb) {
      try {
        cb.call(ctx)
      } catch (e) {
        handleError(e, ctx, 'nextTick')
      }
    } else if (_resolve) {
      _resolve(ctx)
    }
  })
  // pending 是一个锁，确保任务执行有序、不重复
  if (!pending) {
    pending = true
    timerFunc()
  }
  // 兜底逻辑，处理入参不是回调的情况
  if (!cb && typeof Promise !== 'undefined') {
    return new Promise(resolve => {
      _resolve = resolve
    })
  }
}
```

这里面需要引起大家重视的有三个变量：

- callbacks - 异步更新队列
- pending - “锁”
- timerFunc - 异步任务的派发函数

我们顺着代码来捋一下这个逻辑：

首先，**nextTick 的入参是一个回调函数，这个回调函数就是一个“任务”。**每次 nextTick 接收一个任务，它不会立刻去执行它，而是把它 push 进 callbacks 这个异步更新队列里（也就是存起来）。接着，去检查 pending 的值。这个 pending 有何妙用呢？大家知道，我这个异步更新队列肯定不能一直往里塞东西，我得找个时机把它派发出去对不对？那么如何决定啥时候派发它呢？

如果说这个 pending 为 false，意味着啥？ 意味着“现在还没有一个异步更新队列被派发出去”，那么就调用 timerFunc，把当前维护的这个异步队列给派发出去；那如果 pending 为 true 呢？意味着现在异步更新队列（callbacks）已经被派发出去了，此时 callbacks 已经呆在浏览器的异步任务队列里、确保会被执行了，因此没有必要再执行一遍 timerFunc 去重复派发这个队列，只需要往里面添加任务就可以了。

异步任务派发器——timerFunc

在上面这个过程里，我们提及了两个关键的动作——“派发”和“执行”。

大家注意，派发和执行细说的话，是两个概念。

“派发”是说我把你的 callbacks 队列丢进浏览器整体的异步队列里等待执行——注意是“等待执行”，派发完成后，任务队列并没有被执行，只是进入到了等待被执行的状态里。

要想细致理解派发动作，我们要仔细瞅瞅 timerFunc 做了啥：

```
// 用来派发异步任务的函数
let timerFunc


// 下面这一段逻辑，是根据浏览器的不同，选择不同的 api 来派发异步任务
if (typeof Promise !== 'undefined' && isNative(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]'
)) {
  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)) {
  timerFunc = () => {
    setImmediate(flushCallbacks)
  }
} else {
  // Fallback to setTimeout.
  timerFunc = () => {
    setTimeout(flushCallbacks, 0)
  }
}
```

这段代码看似内容不少，实际逻辑很清晰。大家先不用管太多，直接看每一个 if-else 分支里 timerFunc 的定义，这里我给大家抽出来：

```
timerFunc = () => {
  p.then(flushCallbacks)
  if (isIOS) setTimeout(noop)
}

timerFunc = () => {
  counter = (counter + 1) % 2
  textNode.data = String(counter)
}

timerFunc = () => {
  setImmediate(flushCallbacks)
}

timerFunc = () => {
  setTimeout(flushCallbacks, 0)
}
```

你会发现，不同的 timerFunc 之间有一个共性——它们都在派发 flushCallbacks 这个函数（这个函数我们下面会讲）。那么不同的 timerFunc 间有啥区别呢？我们看到区别在于派发 flushCallbacks 的方式不同，这里一共有四种派发方式：

```
- Promise.then
- MutationObserver
- setImmediate
- setTimeout
```

这一整坨的代码里，其实有不少是在处理浏览器的兼容性，细节上不必深究。这里我给大家把关键逻辑抽离为伪代码，其实事情很简单：

```
if(当前环境支持 Promise){
    Promise.then 派发 timerFunc
} else if (当前环境支持 MutationObserver) {
        MutationObserver 派发 timerFunc
} else if (当前环境支持 setImmediate) {
    setImmediate 派发 timerFunc
} else {
    setTimeout 派发 timer Func
}
```

我们看到，timerFunc 按照优先级分别可能通过：Promise.then、MutationObserver、setImmediate、setTimeout 四种途径派发。这个优先级比较有看头，大家会发现它是优先派发 micro-task、次选 macro-task。注意注意，这里有个出题点：

为什么 Vue 优先派发的是 micro-task?

这里就考到你对 Event-Loop 执行过程的理解了。我们前面讲过 Event-Loop 的执行流程是这样的：

1. 执行并出队一个 macro-task。
2. 全局上下文（script 标签）被推入调用栈，同步代码执行。在执行的过程中，通过对一些接口的调用，可以产生新的 macro-task 与 micro-task，它们会分别被推入各自的任务队列里。**这个过程本质上是队列的 macro-task 的执行和出队的过程**。
3. 上一步我们出队的是一个 macro-task，这一步我们处理的是 micro-task。但需要注意的是：当 macro-task 出队时，任务是一个一个执行的；而 micro-task 出队时，任务是一队一队执行的。因此，我们处理 micro 队列这一步，会逐个执行队列中的任务并把它出队，直到队列被清空。
4. 执行渲染操作，更新界面；
5. 检查是否存在 Web worker 任务，如果有，则对其进行处理；

现在需要大家特别关注的是 **1、2、3、4** 之间的关系！大家想，如果我在 2 中派发的是一个 macro-task，那么这个任务会在什么时候被执行？

仔细想想，它是不是会被推移到下一个循环里的 1 中被执行？这就有问题了——如果我的任务是用来更新 UI 界面的，那么它在我当前循环的 4 中并不会被感知；你只能等它在下一个循环中的 1 中生效后，才能在下一个循环的 4 中被感知、进而更新到界面上。

问题出现了：macro-task 形式的派发，会导致我们的界面更新**延迟一个事件循环**。在当前循环的 4，这个渲染时机一定程度上被“浪费”了，它并不能及时渲染出我们的更新。

此外，**micro-task** 是一队一队来更新，而 **macro-task** 是一个一个来更新。从更新效率上来说，micro-task 也会更优秀。

任务执行器——flushCallbacks

flushCallbacks 的逻辑相对上面简单不少：

```
function flushCallbacks () {
  // 把“锁”打开
  pending = false
  // 创造 callbacks 副本，避免副作用
  const copies = callbacks.slice(0)
  // callbacks 队列置空
  callbacks.length = 0
  // 逐个执行异步任务
  for (let i = 0; i < copies.length; i++) {
    copies[i]()
  }
}
```

它负责把 callbacks 里的任务逐个取出，依次执行。

注意，进入 flushCallbacks 后，做的第一件事情就是把 pending 置为 false。因为 flushCallbacks 执行完毕后，callbacks 将被清空、浏览器的异步任务队列中也就没有 Vue 的异步任务了。此时必须把 pending 置空，确保下一个 Vue 异步任务队列进来时，可以及时被派发。

}

36 Vue核心——响应式原理源码级解析

38 Vue 优质真题深度解读