

## 29 Node事件循环命题思路剖析

更新时间：2020-05-26 14:21:14



“

完成工作的方法，是爱惜每一分钟。——达尔文

”

### Node事件循环命题思路剖析

Node 事件循环的命题频率并没有浏览器那么高，但每次都能难到一大票人~

不过不要怕，大家结合上面咱们的分析、归纳出自己的一套对 Node、浏览器两套事件循环机制的理解；再掌握下面的三个热门命题点，相信你面对 Event-Loop 是不虚的：

### nextTick 和 Promise.then 的关系

在做这题之前，大家首先要心里默念一遍 Node 中 micro-queue 的这个特征：

**Node** 清空微任务队列的手法比较特别。在浏览器中，我们只有一个微任务队列需要接受处理；但在 **Node** 中，有两类微任务队列：**next-tick** 队列和其它队列。其中这个 **next-tick** 队列，专门用来收敛 `process.nextTick` 派发的异步任务。在清空队列时，优先清空 **next-tick** 队列中的任务，随后才会清空其它微任务。

记住：在 **Node** 异步队列真题中，只要见到 `process.nextTick` 这货，它肯定不是善茬，多半是要和 `Promise.then` 一起出来唬人的。

```
Promise.resolve().then(function() {
  console.log("promise1")
}).then(function() {
  console.log("promise2")
});

process.nextTick(() => {
  console.log('nextTick1')
  process.nextTick(() => {
    console.log('nextTick2')
    process.nextTick(() => {
      console.log('nextTick3')
      process.nextTick(() => {
        console.log('nextTick4')
      })
    })
  })
})
})
```

问：上述代码的输出结果是多少？

思考：我们现在已经知道，不管你整什么微任务过来，只要它不是 `process.nextTick` 派发的，全部都要排队在 `process.nextTick` 后面执行。因此输出顺序是：

```
nextTick1
nextTick2
nextTick3
nextTick4
promise1
promise2
```

## setTimeout 和 setImmediate 的故事

**setImmediate** 是啥？

开始做题之前，先给大家简单介绍一些 **setImmediate**，它的调用形式有以下两种：

```
var immediateID = setImmediate (func, [ param1, param2, ...] );
var immediateID = setImmediate (func) ;
```

注意，**setImmediate** 虽然和 **setTimeout** 类似（它们都用于延迟某个操作），**setImmediate** 可倔多了，它不接受你给它指定执行的时机（没有延时时间作为入参），它只认一个执行时机——离它最近的那一次 **check**。

这也和 **setImmediate** 本身的用意是分不开的：

**setImmediate()**方法用于中断长时间运行的操作，并在完成其他操作（如事件和显示更新）后立即运行回调函数。

了解了 **setImmediate** 的用法和特性，我们一起来看看 **Node** 事件循环面试题中最“诡异”的一道：

```

setTimeout(function() {
  console.log('老铁，我是被 setTimeout 派发的')
}, 0)

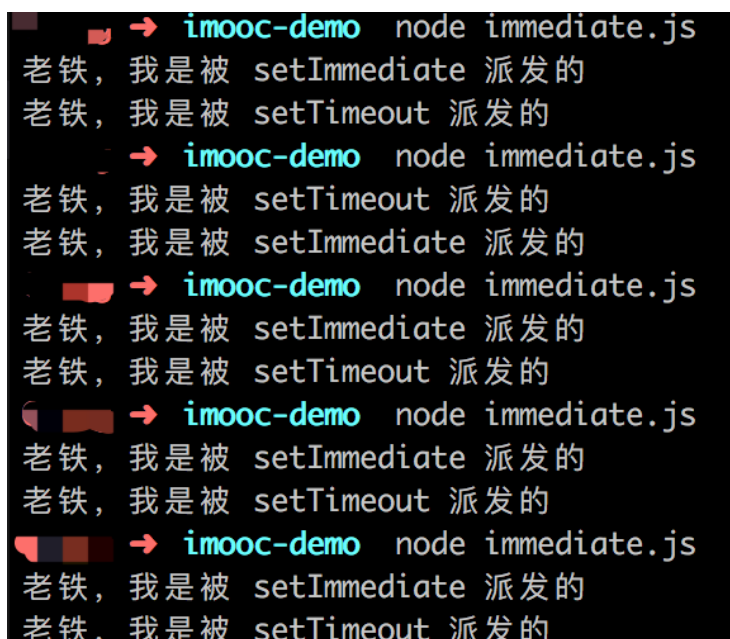
setImmediate(function() {
  console.log('老铁，我是被 setImmediate 派发的')
})

```

问：上述代码的执行结果是啥？

答：不一定！！

没错，就是不一定。这里我建议大家把这个 demo 拷贝下来，丢进自己的 Node.js 文件里去跑一跑。多跑几次，你就会发现如下的神奇规律：



```

→ imooc-demo node immediate.js
老铁，我是被 setImmediate 派发的
老铁，我是被 setTimeout 派发的
→ imooc-demo node immediate.js
老铁，我是被 setTimeout 派发的
老铁，我是被 setImmediate 派发的
→ imooc-demo node immediate.js
老铁，我是被 setImmediate 派发的
老铁，我是被 setTimeout 派发的
→ imooc-demo node immediate.js
老铁，我是被 setImmediate 派发的
老铁，我是被 setTimeout 派发的
→ imooc-demo node immediate.js
老铁，我是被 setImmediate 派发的
老铁，我是被 setTimeout 派发的

```

没错！正如图中所示一样，这个例子里 `setImmediate` 和 `setTimeout` 谁先执行是个谜啊老铁们——它是随机的！

为啥会这样呢？（注意接下来我们会对这个现象做一个原因分析，这个原因分析至少有一半分，各位坚持住哈）

- 首先，给各位普及一个小知识：`setTimeout` 这个函数的第二个入参，它的取值范围是  $[1, 2^{31}-1]$ 。也就是说，它是不认识 0 这个入参的。不认识咋整呢？强行给你 1 掉！也就是说下面这种写法：

```

setTimeout(function() {
  console.log('老铁，我是被 setTimeout 派发的')
}, 0)

```

其实等价于：

```

setTimeout(function() {
  console.log('老铁，我是被 setTimeout 派发的')
}, 1)

```

也就是说这个回调，其实被延迟了 1ms。

- 然后，各位需要认识到这样一个问题：事件循环的初始化，是需要时间的。

怎么理解这个“需要时间”呢？这意味着初始化事件循环的时间，可能大于 1ms，也可能小于 1ms，这就会带来下面两种可能性：

- 当初始化时间小于 1ms 时：进入了 `timers` 阶段，却发现 `setTimeout` 定时器还没到时间，于是往下走。走到 `check` 阶段，执行了 `setImmediate` 回调；在后面的循环周期里，才会执行 `setTimeout` 回调；
- 当初始化时间大于 1ms 时：进入了 `timers` 阶段，发现 `setTimeout` 定时器已经到时间了，直接执行 `setTimeout` 回调；结束 `timers` 阶段后，走啊走，走到了 `check` 阶段，顺理成章地又执行了 `setImmediate` 回调。

结合咱们上面的分析，相信各位已经认识到：其实每一种输出结果都是对的、都是符合我们事件循环原则的。顺序上的差别是由我们不可控的“事件循环初始化时间”导致的。因此这个“不一定”，咱们到时候一定是回答得理直气壮。

## poll阶段对定时器的处理

前面已经提到过，`poll` 阶段是一个重点阶段，大部分的回调任务都会在这个阶段被处理。重点阶段重点分析，咱们现在就来扒一扒 `poll` 阶段具体有哪些情况：

进入 `poll` 阶段时，我们考虑以下两种场景：

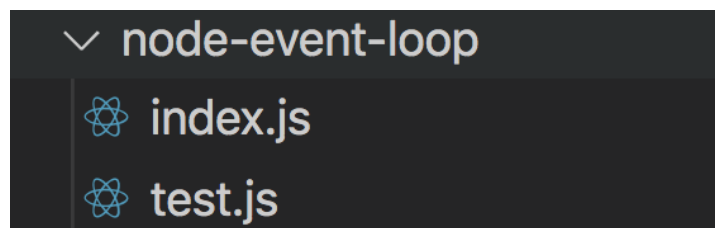
1. `poll` 队列不为空。这种情况好办，直接逐个执行队列内的回调并出队、直到队列被清空（或者到达系统上限）为止；
2. `poll` 队列本来就是空的。没活干，事件循环也不能闲着：它首先会检查有没有待执行的 `setImmediate` 任务，如果有，则往下走、进入到 `check` 阶段开始处理 `setImmediate`；如果没有 `setImmediate` 任务，那么再去检查一下有没有到期的 `setTimeout` 任务需要处理，若有，则跳转到 `timers` 阶段。

那如果连 `setTimeout` 任务也没有呢？那咱 `poll` 阶段也不钻这个牛角尖了——没活干就算了，我等着！此时 `poll` 阶段会进入到等待状态，等待回调任务的到来。一旦有回调进入，`poll` 就会“立刻出击”。

结合这一连串的分析，各位需要记住这样一个结论——在 `poll` 阶段处理的回调中，如果既派发了 `setImmediate`、又派发了 `setTimeout`，那么这个顺序是板上钉钉的——一定是先执行 `setImmediate`，再执行 `setTimeout`。

接下来我们一起来看一道真题：

首先给出如下的目录结构：



- `test.js` 文件中是任意代码
- `index.js` 文件中，写入下面内容：

```
const fs = require('fs')
const path = require('path')
const filePath = path.join(__dirname, 'test.js')

console.log(filePath)

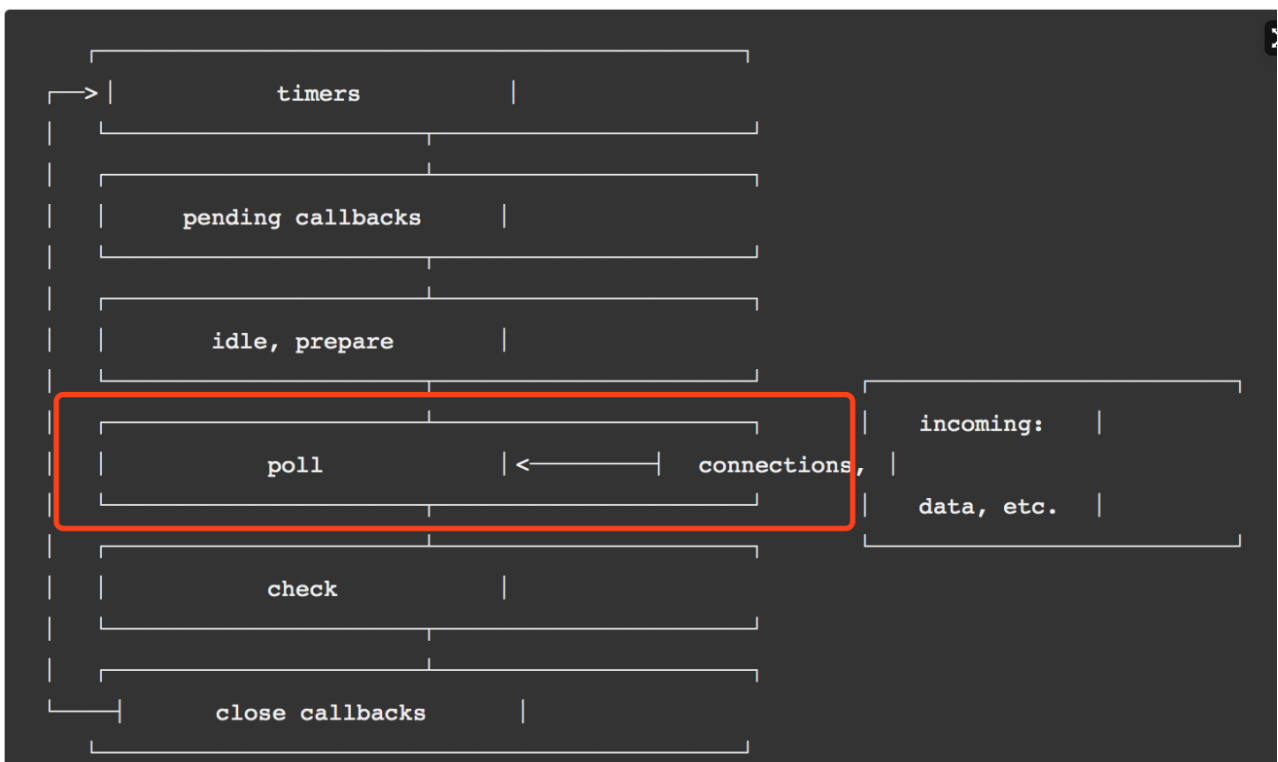
// -- 异步读取文件
fs.readFile(filePath, 'utf8', function(err, data){
  setTimeout(function() {
    console.log("老铁，我是被 setTimeout 派发的")
  }, 0)

  setImmediate(function() {
    console.log("老铁，我是被 setImmediate 派发的")
  })
});
```

问：输出顺序如何？为什么？

解答：经过我们前面的分析，这道题的答案是毫无疑问的：先输出 `setImmediate` 中的 `console`，然后再输出 `setTimeout` 中的 `console`。

分析：各位首先要注意到——`readFile`这个接口，是一个异步的文件I/O接口。在 Node 中，I/O 回调是在 `poll` 阶段处理的。因此，当 `readFile` 回调被执行时，实际上走到了这个阶段：



在 `poll` 阶段，派出去两个任务：一个是在 `check` 阶段被处理的 `setImmediate` 回调，一个是在 `timers` 阶段被处理的 `setTimeout` 回调。

结合上图以及我们前面的分析，可以很明显地看出：`check` 阶段永远比 `timers` 阶段离 `poll` 更近，因此 `setImmediate` 总是比 `setTimeout` 先执行。

所以说大家记住这个结论：在 `poll` 阶段处理的回调中，如果既派发了 `setImmediate`、又派发了 `setTimeout`，那么这个顺序是板上钉钉的——一定是先执行 `setImmediate`，再执行 `setTimeout`。

## 注意！！！Node11事件循环已与浏览器事件循环机制趋同！

这是个大坑啊同学们！

一道阴险的考题

我们一起来看看这样一道题：

```
setTimeout(() => {  
  console.log('timeout1');  
}, 0);  
  
setTimeout(() => {  
  console.log('timeout2');  
  Promise.resolve().then(function() {  
    console.log('promise1');  
  });  
}, 0);  
  
setTimeout(() => {  
  console.log('timeout3')  
}, 0)
```

问：上述代码在浏览器、Node中的执行结果各是什么？

这题很险，但各位只要记住一句话：Node11开始，Node的事件循环已经和浏览器趋同。注意是“趋同”而不是一毛一样。其中最明显的改变是：

Node11开始，timers 阶段的setTimeout、setInterval等函数派发的任务、包括 setImmediate 派发的任务，都被修改为：一旦执行完当前阶段的一个任务，就立刻执行微任务队列。

这就意味着，上面这道题，在浏览器和在 Node11 中跑出来的结果一毛一样——不信各位切换到高版本跑跑看。

我这里可以给大家看一下我用 Node9.3.0 和用 Node12.4.1 分别跑上面demo代码的结果：

以下是 v9.3.0 版本下的执行结果：

```
imooc-demo node -v  
v9.3.0  
imooc-demo node specialNode.js  
timeout1  
timeout2  
timeout3  
promise1
```

我们看到在 timers 阶段，依次执行了所有的 setTimeout 回调、清空了队列——这符合我们前面对 Node 事件循环机制的描述。

以下是 v12.4.1 版本下的执行结果：

```
imooc-demo node -v
v12.14.1
imooc-demo node specialNode.js
timeout1
timeout2
promise1
timeout3
```

同时我们再看一下浏览器跑上面代码的结果：

---

timeout1

---

timeout2

---

promise1

---

timeout3

很明显，**Node11**及以上的版本，对这段代码的执行结果和浏览器一毛一样。

明确了高低版本间的区别，这里需要大家采取的对策是：只要遇到 **Node** 事件循环相关的编码类题目，都在答题结束后补充上咱们前面对 **Node11** 版本的这部分讲解。让面试官知道，你不是一个死背教条、一万年不更新一次知识库的憨憨；而是一个紧跟时代变化，头脑灵活的好程序员。

}

