# 微任务、宏任务与Event-Loop

首先，JavaScript是一个单线程的脚本语言。

所以就是说在一行代码执行的过程中，必然不会存在同时执行的另一行代码，就像使用`alert()`以后进行疯狂`console.log`，如果没有关闭弹框，控制台是不会显示出一条`log`信息的。

亦或者有些代码执行了大量计算，比方说在前端暴力破解密码之类的鬼操作，这就会导致后续代码一直在等待，页面处于假死状态，因为前边的代码并没有执行完。

所以如果全部代码都是同步执行的，这会引发很严重的问题，比方说我们要从远端获取一些数据，难道要一直循环代码去判断是否拿到了返回结果么？*就像去饭店点餐，肯定不能说点完了以后就去后厨催着人炒菜的，会被揍的。*

于是就有了异步事件的概念，注册一个回调函数，比如说发一个网络请求，我们告诉主程序等到接收到数据后通知我，然后我们就可以去做其他的事情了。

然后在异步完成后，会通知到我们，但是此时可能程序正在做其他的事情，所以即使异步完成了也需要在一旁等待，等到程序空闲下来才有时间去看哪些异步已经完成了，可以去执行。

*比如说打了个车，如果司机先到了，但是你手头还有点儿事情要处理，这时司机是不可能自己先开着车走的，一定要等到你处理完事情上了车才能走。*


## 微任务与宏任务的区别

> 链接：https://juejin.cn/post/6844903657264136200#comment

这个就像去银行办业务一样，先要取号进行排号。 一般上边都会印着类似：“您的号码为XX，前边还有XX人。”之类的字样。

因为柜员同时职能处理一个来办理业务的客户，这时每一个来办理业务的人就可以认为是银行柜员的一个宏任务来存在的，当柜员处理完当前客户的问题以后，选择接待下一位，广播报号，也就是下一个宏任务的开始。 所以多个宏任务合在一起就可以认为说有一个任务队列在这，里边是当前银行中所有排号的客户。 **任务队列中的都是已经完成的异步操作，而不是说注册一个异步任务就会被放在这个任务队列中，就像在银行中排号，如果叫到你的时候你不在，那么你当前的号牌就作废了，柜员会选择直接跳过进行下一个客户的业务处理，等你回来以后还需要重新取号**

而且一个宏任务在执行的过程中，是可以添加一些微任务的，就像在柜台办理业务，你前边的一位老大爷可能在存款，在存款这个业务办理完以后，柜员会问老大爷还有没有其他需要办理的业务，这时老大爷想了一下：“最近P2P爆雷有点儿多，是不是要选择稳一些的理财呢”，然后告诉柜员说，要办一些理财的业务，这时候柜员肯定不能告诉老大爷说：“您再上后边取个号去，重新排队”。 所以本来快轮到你来办理业务，会因为老大爷临时添加的“**理财业务**”而往后推。 也许老大爷在办完理财以后还想 **再办一个信用卡**？或者 **再买点儿纪念币**？ 无论是什么需求，只要是柜员能够帮她办理的，都会在处理你的业务之前来做这些事情，这些都可以认为是微任务。

这就说明：你大爷永远是你大爷 **在当前的微任务没有执行完成时，是不会执行下一个宏任务的。**

所以就有了那个经常在面试题、各种博客中的代码片段：

```
setTimeout(_ => console.log(4))

new Promise(resolve => {
  resolve()
  console.log(1)
}).then(_ => {
  console.log(3)
})

console.log(2)
复制代码
```

`setTimeout` 就是作为宏任务来存在的，而 `Promise.then` 则是具有代表性的微任务，上述代码的执行顺序就是按照序号来输出的。

**所有会进入的异步都是指的事件回调中的那部分代码** 也就是说 `new Promise` 在实例化的过程中所执行的代码都是同步进行的，而 `then` 中注册的回调才是异步执行的。 在同步代码执行完成后才回去检查是否有异步任务完成，并执行对应的回调，而微任务又会在宏任务之前执行。 所以就得到了上述的输出结论 `1、2、3、4` 。

*+部分表示同步执行的代码*

```
+setTimeout(_ => {
-  console.log(4)
+})

+new Promise(resolve => {
+  resolve()
+  console.log(1)
+}).then(_ => {
-  console.log(3)
+})

+console.log(2)
复制代码
```

本来`setTimeout`已经先设置了定时器（相当于取号），然后在当前进程中又添加了一些`Promise`的处理（临时添加业务）。

所以进阶的，即便我们继续在`Promise`中实例化`Promise`，其输出依然会早于`setTimeout`的宏任务：

```
setTimeout(_ => console.log(4))

new Promise(resolve => {
  resolve()
  console.log(1)
}).then(_ => {
  console.log(3)
  Promise.resolve().then(_ => {
    console.log('before timeout')
  }).then(_ => {
    Promise.resolve().then(_ => {
      console.log('also before timeout')
    })
  })
})

console.log(2)
复制代码
```

当然了，实际情况下很少会有简单的这么调用`Promise`的，一般都会在里边有其他的异步操作，比如`fetch`、`fs.readFile`之类的操作。 而这些其实就相当于注册了一个宏任务，而非是微任务。

*P.S. 在Promise/A+的规范中，Promise的实现可以是微任务，也可以是宏任务，但是普遍的共识表示(至少Chrome是这么做的)，Promise应该是属于微任务阵营的*

所以，明白哪些操作是宏任务、哪些是微任务就变得很关键，这是目前业界比较流行的说法：

## 宏任务

| # | 浏览器 | Node |
| --- | --- | --- |
| `I/O` | ✅ | ✅ |
| `setTimeout` | ✅ | ✅ |
| `setInterval` | ✅ | ✅ |
| `setImmediate` | ❌ | ✅ |
| `requestAnimationFrame` | ✅ | ❌ |

*有些地方会列出来UI Rendering，说这个也是宏任务，可是在读了HTML规范文档以后，发现这很显然是和微任务平行的一个操作步骤 requestAnimationFrame姑且也算是宏任务吧，requestAnimationFrame在MDN的定义为，下次页面重绘前所执行的操作，而重绘也是作为宏任务的一个步骤来存在的，且该步骤晚于微任务的执行*

## 微任务

| # | 浏览器 | Node |
| --- | --- | --- |
| `process.nextTick` | ❌ | ✅ |
| `MutationObserver` | ✅ | ❌ |
| `Promise.then catch finally` | ✅ | ✅ |

# Event-Loop是个啥

上边一直在讨论 宏任务、微任务，各种任务的执行。 但是回到现实，`JavaScript`是一个单进程的语言，同一时间不能处理多个任务，所以何时执行宏任务，何时执行微任务？我们需要有这样的一个判断逻辑存在。

每办理完一个业务，柜员就会问当前的客户，是否还有其他需要办理的业务。**（检查还有没有微任务需要处理）*** 而客户明确告知说没有事情以后，柜员就去查看后边还有没有等着办理业务的人。**（结束本次宏任务、检查还有没有宏任务需要处理）*** 这个检查的过程是持续进行的，每完成一个任务都会进行一次，而这样的操作就被称为`Event Loop`。*(这是个非常简易的描述了，实际上会复杂很多)*

而且就如同上边所说的，一个柜员同一时间只能处理一件事情，即便这些事情是一个客户所提出的，所以可以认为微任务也存在一个队列，大致是这样的一个逻辑：

```
const macroTaskList = [
  ['task1'],
  ['task2', 'task3'],
  ['task4'],
]

for (let macroIndex = 0; macroIndex < macroTaskList.length; macroIndex++) {
  const microTaskList = macroTaskList[macroIndex]

  for (let microIndex = 0; microIndex < microTaskList.length; microIndex++) {
    const microTask = microTaskList[microIndex]
```

```
    // 添加一个微任务
    if (microIndex === 1) microTaskList.push('special micro task')

    // 执行任务
    console.log(microTask)
  }

  // 添加一个宏任务
  if (macroIndex === 2) macroTaskList.push(['special macro task'])
}

// > task1
// > task2
// > task3
// > special micro task
// > task4
// > special macro task
复制代码
```

*之所以使用两个for循环来表示，是因为在循环内部可以很方便的进行push之类的操作（添加一些任务），从而使迭代的次数动态的增加。*

以及还要明确的是，`Event Loop`只是负责告诉你该执行那些任务，或者说哪些回调被触发了，真正的逻辑还是在进程中执行的。

# 在浏览器中的表现

在上边简单的说明了两种任务的差别，以及`Event Loop`的作用，那么在真实的浏览器中是什么表现呢？ 首先要明确的一点是，宏任务必然是在微任务之后才执行的（因为微任务实际上是宏任务的其中一个步骤）

`I/O`这一项感觉有点儿笼统，有太多的东西都可以称之为`I/O`，点击一次`button`，上传一个文件，与程序产生交互的这些都可以称之为`I/O`。

假设有这样的一些`DOM`结构：

```
<style>
  #outer {
    padding: 20px;
    background: #616161;
  }

  #inner {
    width: 100px;
    height: 100px;
    background: #757575;
  }
</style>
<div id="outer">
  <div id="inner"></div>
</div>
复制代码
const $inner = document.querySelector('#inner')
const $outer = document.querySelector('#outer')
```

```
function handler () {
  console.log('click') // 直接输出

  Promise.resolve().then(_ => console.log('promise')) // 注册微任务

  setTimeout(_ => console.log('timeout')) // 注册宏任务

  requestAnimationFrame(_ => console.log('animationFrame')) // 注册宏任务

  $outer.setAttribute('data-random', Math.random()) // DOM属性修改，触发微任务
}

new MutationObserver(_ => {
  console.log('observer')
}).observe($outer, {
  attributes: true
})

$inner.addEventListener('click', handler)
$outer.addEventListener('click', handler)
复制代码
```

如果点击`#inner`，其执行顺序一定是：`click` -> `promise` -> `observer` -> `click` -> `promise` -> `observer` -> `animationFrame` -> `animationFrame` -> `timeout` -> `timeout`。

因为一次`I/O`创建了一个宏任务，也就是说在这次任务中会去触发`handler`。 按照代码中的注释，在同步的代码已经执行完以后，这时就会去查看是否有微任务可以执行，然后发现了`Promise`和`MutationObserver`两个微任务，遂执行之。 因为`click`事件会冒泡，所以对应的这次`I/O`会触发两次`handler`函数(*一次在inner、一次在outer*)，所以会优先执行冒泡的事件(*早于其他的宏任务*)，也就是说会重复上述的逻辑。 在执行完同步代码与微任务以后，这时继续向后查找有木有宏任务。 需要注意的一点是，因为我们触发了`setAttribute`，实际上修改了`DOM`的属性，这会导致页面的重绘，而这个`set`的操作是同步执行的，也就是说`requestAnimationFrame`的回调会早于`setTimeout`所执行。

## 一些小惊喜

使用上述的示例代码，如果将手动点击`DOM`元素的触发方式变为`$inner.click()`，那么会得到不一样的结果。在`Chrome`下的输出顺序大致是这样的：`click` -> `click` -> `promise` -> `observer` -> `promise` -> `animationFrame` -> `animationFrame` -> `timeout` -> `timeout`。

与我们手动触发`click`的执行顺序不一样的原因是这样的，因为并不是用户通过点击元素实现的触发事件，而是类似`dispatchEvent`这样的方式，我个人觉得并不能算是一个有效的`I/O`，在执行了一次`handler`回调注册了微任务、注册了宏任务以后，实际上外边的`$inner.click()`并没有执行完。 所以在微任务执行之前，还要继续冒泡执行下一次事件，也就是说触发了第二次的`handler`。 所以输出了第二次`click`，等到这两次`handler`都执行完毕后才会去检查有没有微任务、有没有宏任务。

两点需要注意的：

1. `.click()`的这种触发事件的方式个人认为是类似`dispatchEvent`，可以理解为同步执行的代码

```
document.body.addEventListener('click', _ => console.log('click'))

document.body.click()
document.body.dispatchEvent(new Event('click'))
console.log('done')

// > click
// > click
// > done
复制代码
```

1. `MutationObserver`的监听不会说同时触发多次，多次修改只会有一次回调被触发。

```
new MutationObserver(_ => {
  console.log('observer')
  // 如果在这输出DOM的data-random属性，必然是最后一次的值，不解释了
}).observe(document.body, {
  attributes: true
})

document.body.setAttribute('data-random', Math.random())
document.body.setAttribute('data-random', Math.random())
document.body.setAttribute('data-random', Math.random())

// 只会输出一次 ovserver
复制代码
```

*这就像去饭店点餐，服务员喊了三次，XX号的牛肉面，不代表她会给你三碗牛肉面。 上述观点参阅自Tasks, microtasks, queues and schedules，文中有动画版的讲解*

# 在Node中的表现

Node也是单线程，但是在处理`Event Loop`上与浏览器稍微有些不同，这里是Node官方文档的地址。

就单从API层面上来理解，Node新增了两个方法可以用来使用：微任务的`process.nextTick`以及宏任务的`setImmediate`。

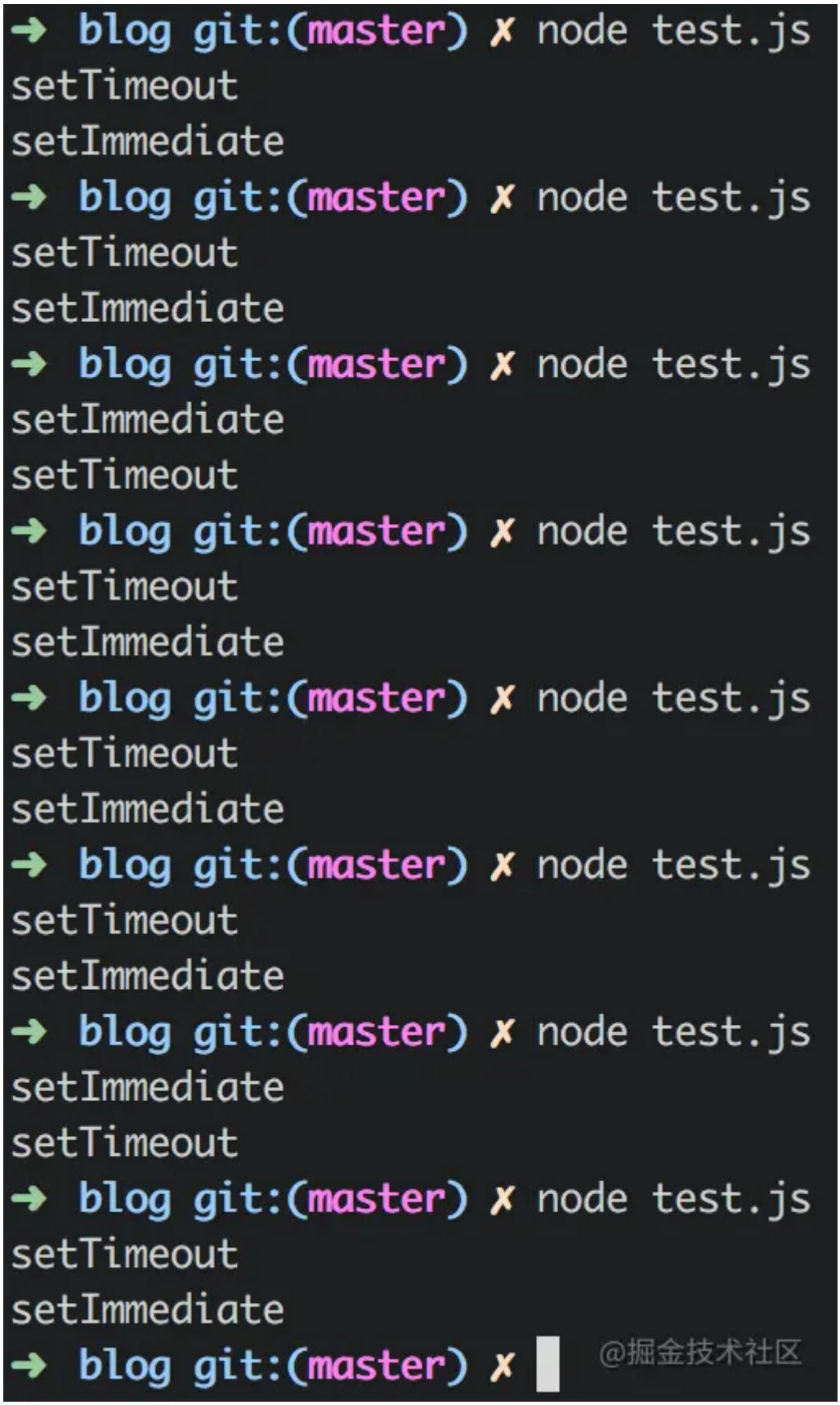
## setImmediate与setTimeout的区别

在官方文档中的定义，`setImmediate`为一次`Event Loop`执行完毕后调用。`setTimeout`则是通过计算一个延迟时间后进行执行。

但是同时还提到了如果在主进程中直接执行这两个操作，很难保证哪个会先触发。 因为如果主进程中先注册了两个任务，然后执行的代码耗时超过`XXs`，而这时定时器已经处于可执行回调的状态了。 所以会先执行定时器，而执行完定时器以后才是结束了一次`Event Loop`，这时才会执行`setImmediate`。

```
setTimeout(_ => console.log('setTimeout'))
setImmediate(_ => console.log('setImmediate'))
复制代码
```

有兴趣的可以自己试验一下，执行多次真的会得到不同的结果。



但是如果后续添加一些代码以后，就可以保证`setTimeout`一定会在`setImmediate`之前触发了：

```
setTimeout(_ => console.log('setTimeout'))
setImmediate(_ => console.log('setImmediate'))

let countdown = 1e9

while(countdown--) { } // 我们确保这个循环的执行速度会超过定时器的倒计时，导致这轮循环没有结束时，
setTimeout已经可以执行回调了，所以会先执行`setTimeout`再结束这一轮循环，也就是说开始执行
`setImmediate`
复制代码
```

如果在另一个宏任务中，必然是`setImmediate`先执行：

```
require('fs').readFile(__dirname, _ => {
  setTimeout(_ => console.log('timeout'))
  setImmediate(_ => console.log('immediate'))
})

// 如果使用一个设置了延迟的setTimeout也可以实现相同的效果
复制代码
```

## process.nextTick

就像上边说的，这个可以认为是一个类似于`Promise`和`MutationObserver`的微任务实现，在代码执行的过程中可以随时插入`nextTick`，并且会保证在下一个宏任务开始之前所执行。

在使用方面的一个最常见的例子就是一些事件绑定类的操作：

```
class Lib extends require('events').EventEmitter {
  constructor () {
    super()

    this.emit('init')
  }
}

const lib = new Lib()

lib.on('init', _ => {
  // 这里将永远不会执行
  console.log('init!')
})
复制代码
```

因为上述的代码在实例化`Lib`对象时是同步执行的，在实例化完成以后就立马发送了`init`事件。 而这时在外层的主程序还没有开始执行到`lib.on('init')`监听事件的这一步。 所以会导致发送事件时没有回调，回调注册后事件不会再次发送。

我们可以很轻松的使用`process.nextTick`来解决这个问题：

```
class Lib extends require('events').EventEmitter {
  constructor () {
    super()

    process.nextTick(_ => {
      this.emit('init')
    })

    // 同理使用其他的微任务
    // 比如Promise.resolve().then(_ => this.emit('init'))
    // 也可以实现相同的效果
  }
}
复制代码
```

这样会在主进程的代码执行完毕后，程序空闲时触发`Event Loop`流程查找有没有微任务，然后再发送`init`事件。

*关于有些文章中提到的，循环调用process.nextTick会导致报警，后续的代码永远不会被执行，这是对的，参见上边使用的双重循环实现的loop即可，相当于在每次for循环执行中都对数组进行了push操作，这样循环永远也不会结束*

# 多提一嘴async/await函数

因为，`async/await`本质上还是基于`Promise`的一些封装，而`Promise`是属于微任务的一种。所以在使用`await`关键字与`Promise.then`效果类似：

```
setTimeout(_ => console.log(4))

async function main() {
  console.log(1)
  await Promise.resolve()
  console.log(3)
}

main()

console.log(2)
复制代码
```

**async函数在await之前的代码都是同步执行的，可以理解为await之前的代码属于new Promise时传入的代码，await之后的所有代码都是在Promise.then中的回调**

# 小节

JavaScript的代码运行机制在网上有好多文章都写，本人道行太浅，只能简单的说一下自己对其的理解。 并没有去生抠文档，一步一步的列出来，像什么查看当前栈、执行选中的任务队列，各种balabala。 感觉对实际写代码没有太大帮助，不如简单的入个门，扫个盲，大致了解一下这是个什么东西就好了。

推荐几篇参阅的文章：

- tasks-microtasks-queues-and-schedules
- understanding-js-the-event-loop
- 理解Node.js里的process.nextTick()
- 浏览器中的EventLoop说明文档
- Node中的EventLoop说明文档
- requestAnimationFrame | MDN
- MutationObserver | MDN