\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*代码start\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

new Promise(resolve => {

setTimeout(\_ => console.log(6));

resolve()

console.log(1)

}).then(\_ => {

console.log(3)

fetch('https://api.github.com/', {

method: 'GET'

}).then(res => res.json())

.catch(error => console.error('Error:', error))

.then(response => console.log('Success:', response.emails\_url));

Promise.resolve().then(\_ => {

console.log('before timeout')

}).then(\_ => {

Promise.resolve().then(\_ => {

console.log('also before timeout')

})

})

})

console.log(2)；

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*代码end\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//运行结果

VM1366:7 1

VM1366:26 2

VM1366:9 3

VM1366:18 before timeout

VM1366:21 also before timeout

VM1366:1 4

VM1366:4 6

VM1366:15 Success: <https://api.github.com/user/emails>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*结论如下\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. 我们可以看出，在微任务队列执行时创建的微任务，还是会排在主线程上创建出的宏任务之前执行
2. 在一个事件循环中，异步事件返回结果后（返回结果后！返回结果后！返回结果后！重要的事情说三遍）才会被放到一个任务队列中。然而，根据这个异步事件的类型，这个事件实际上会被对应的宏任务队列或者微任务队列中去。

**什么叫轮询？**

精简版：当第一个异步函数执行完之后，再到异步队列监视。一直不断循环往复，所以叫事件轮询。

详细版：js引擎遇到一个异步事件后并不会一直等待其返回结果，而是会将这个事件挂起，继续执行执行栈中的其他任务。当一个异步事件返回结果后（返回结果后！返回结果后！返回结果后！重要的事情说三遍），js会将这个事件加入与当前执行栈不同的另一个队列，我们称之为事件队列。被放入事件队列不会立刻执行其回调，而是等待当前执行栈中的所有任务都执行完毕， 主线程处于闲置状态时，主线程会去查找事件队列是否有任务。如果有，那么主线程会从中取出排在第一位的事件，并把这个事件对应的回调放入执行栈中，然后执行其中的同步代码…，如此反复，这样就形成了一个无限的循环。这就是这个过程被称为“事件循环（Event Loop）”的原因。

事实上，事件轮询与宏任务和微任务密切相关。