**你不知道的 Event Loop**

笔者最近忙着做项目之类的，文章输出遗落下了一段时间，这次我们就来聊一个面试中一个比较重要的知识点 —— Event Loop

可能有人会奇怪一个 EventLoop 还能写出什么，且听我慢慢来逼叨，看完这篇文章带你搞定 Event Loop 以及它相关的一些知识点。

# 一、Event Loop 是什么

在开始说 Event Loop 之前，我们先来认识一下它到底是个什么东西。

In computer science, the event loop is a programming construct or design pattern that waits for and dispatches events or messages in a program. The event loop works by making a request to some internal or external "event provider" (that generally blocks the request until an event has arrived), then calls the relevant event handler ("dispatches the event"). The event loop is also sometimes referred to as the message dispatcher, message loop, message pump, or run loop.

上面这段是**Wikipedia[2]**对 Event Loop 的解释，简单的来说就是Event Loop是一个程序结构，用于等待和分派消息和事件我个人的理解是 JS 中的 Event Loop 是浏览器或 Node 的一种协调 JavaScript 单线程运行时不会阻塞的一种机制。

## 为什么要学 Event Loop？

可能有人会比较疑惑前端为什么要学看起来比较底层的 Event Loop，不仅仅是因为这是一道面试的常考题。

1. 作为一个程序员，了解程序的运行机制是很重要的，这样可以帮助你去输出更优质的代码。
2. 前端是一个范围很广的领域，技术一直在更新迭代，掌握了底层的原理可以应对新的技术。
3. 一个优秀的程序员要能让写的代码按照自己想的去运行，如果连代码本身的运行机制都无法掌握的话，就不用谈什么掌控自己的代码了。

# 二、进程和线程

上文我说了 Event Loop 是单线程阻塞问题的一种解决机制，所以在正式开始前还是要先从进程和线程的角度来聊一聊。众所周知的一件事是，JavaScript 是一个单线程机制的语言，那我们先来看看进程和线程的定义：

## 定义

1. 进程：进程是 CPU 资源分配的最小单位
2. 线程：线程是 CPU 调度的最小单位

说实话，光从定义来看你根本感受不到进程和线程到底是什么样的一个东西。简单来说，进程简单理解就是我们平常使用的程序，如 QQ，浏览器，网盘等。进程拥有自己独立的内存空间地址，拥有一个或多个线程，而线程就是对进程粒度的进一步划分。

更通俗的来说，进程就像是一家工厂，多个工厂之间是独立存在的。而线程就像是工厂中的那些工人，共享资源，完成同一个大目标。

## JS 的单线程

很多人都知道的是，JavaScript 是一门**动态的解释型的语言**，具有**跨平台性**。在被问到 JavaScript 为什么是一门单线程的语言，有的人可能会这么回答：“语言特性决定了 JavaScript 是一个单线程语言，JavaScript 天生是一个单线程语言”，这只不过是一层糖衣罢了。

JavaScript 从诞生起就是单线程，原因大概是不想让浏览器变得太复杂，因为多线程需要共享资源、且有可能修改彼此的运行结果，对于一种网页脚本语言来说，这就太复杂了。

准确的来说，我认为 JavaScript 的单线程是指 **JavaScript 引擎是单线程**的，JavaScript 的引擎并不是独立运行的，跨平台意味着 JavaScript 依赖其运行的宿主环境 --- 浏览器(大部分情况下是浏览器)。

浏览器需要渲染 DOM，JavaScript 可以修改 DOM 结构，JavaScript 执行时，浏览器 DOM 渲染停止。如果 JavaScript 引擎线程不是单线程的，那么可以同时执行多段 JavaScript，如果这多段 JavaScript 都操作 DOM，那么就会出现 DOM 冲突。

举个例子来说，在同一时刻执行两个 script 对同一个 DOM 元素进行操作，一个修改 DOM，一个删除 DOM，那这样话浏览器就会懵逼了，它就不知道到底该听谁的，会有资源竞争，这也是 JavaScript 单线程的原因之一。

# 三、浏览器

## 浏览器的多线程

之前说过，JavaScript 运行的宿主环境浏览器是多线程的。

以 Chrome 来说，我们可以通过 Chrome 的任务管理器来看看。



Chrome任务管理器

当你打开一个 Tab 页面的时候，就创建了一个进程。如果从一个页面打开了另一个页面，打开的页面和当前的页面属于同一站点的话，那么这个页面会复用父页面的渲染进程。

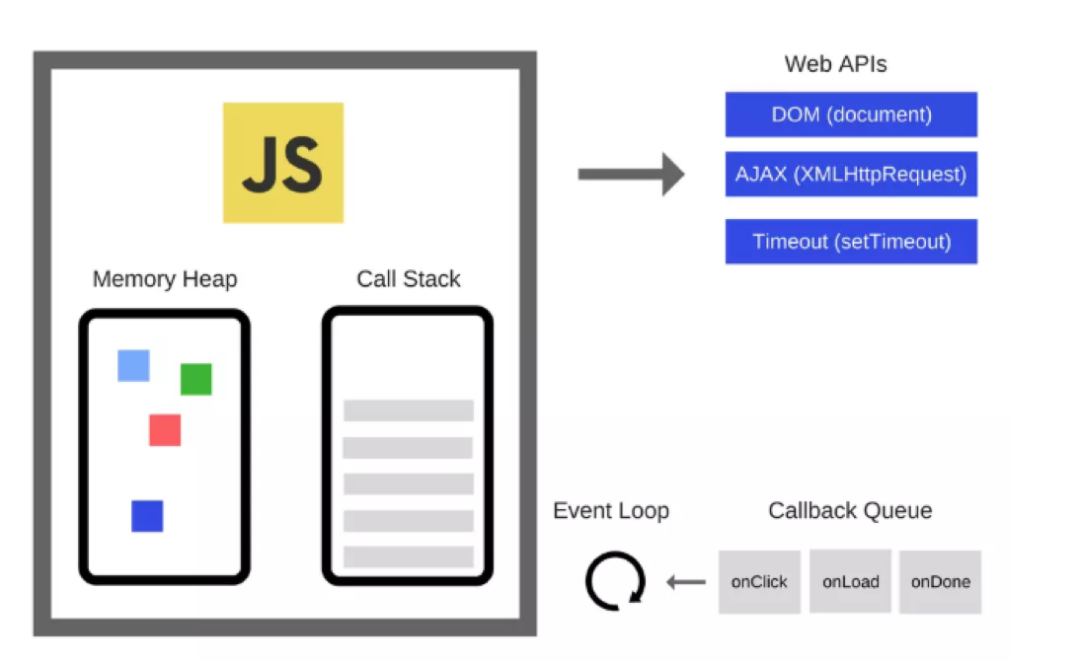
## 浏览器主线程常驻线程

1. GUI 渲染线程
   * 绘制页面，解析 HTML、CSS，构建 DOM 树，布局和绘制等
   * 页面重绘和回流
   * 与 JS 引擎线程互斥，也就是所谓的 JS 执行阻塞页面更新
2. JS 引擎线程
   * 负责 JS 脚本代码的执行
   * 负责准执行准备好待执行的事件，即定时器计数结束，或异步请求成功并正确返回的事件
   * 与 GUI 渲染线程互斥，执行时间过长将阻塞页面的渲染
3. 事件触发线程
   * 负责将准备好的事件交给 JS 引擎线程执行
   * 多个事件加入任务队列的时候需要排队等待(JS 的单线程)
4. 定时器触发线程
   * 负责执行异步的定时器类的事件，如 setTimeout、setInterval
   * 定时器到时间之后把注册的回调加到任务队列的队尾
5. HTTP 请求线程
   * 负责执行异步请求
   * 主线程执行代码遇到异步请求的时候会把函数交给该线程处理，当监听到状态变更事件，如果有回调函数，该线程会把回调函数加入到任务队列的队尾等待执行

这里没看懂没关系，后面我会再说。

# 四、浏览器端的 Event Loop

看到这里，总算是进入正题了，先讲讲浏览器端的 Event Loop 是什么样的。

JS运行机制图

上图是一张 JS 的运行机制图，Js 运行时大致会分为几个部分：

1. Call Stack：调用栈(执行栈)，所有同步任务在主线程上执行，形成一个执行栈，因为 JS 单线程的原因，所以调用栈中每次只能执行一个任务，当遇到的同步任务执行完之后，由任务队列提供任务给调用栈执行。
2. Task Queue：任务队列，存放着异步任务，当异步任务可以执行的时候，任务队列会通知主线程，然后该任务会进入主线程执行。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。

说到这里，Event Loop 也可以理解为：不断地从任务队列中取出任务执行的一个过程。

## 同步任务和异步任务

上文已经说过了 JavaScript 是一门单线程的语言，一次只能执行一个任务，如果所有的任务都是同步任务，那么程序可能因为等待会出现假死状态，这对于一个用户体验很强的语言来说是非常不友好的。

比如说向服务端请求资源，你不可能一直不停的循环判断有没有拿到数据，就好像你点了个外卖，点完之后就开始一直打电话问外卖有没有送到，外卖小哥都会抄着锅铲来打你(狗头)。因此，在 JavaScript 中任务有了同步任务和异步任务，异步任务通过注册回调函数，等到数据来了就通知主程序。

### 概念

简单的介绍一下同步任务和异步任务的概念。

1. 同步任务：必须等到结果来了之后才能做其他的事情，举例来说就是你烧水的时候一直等在水壶旁边等水烧开，期间不做其他的任何事情。
2. 异步任务：不需要等到结果来了才能继续往下走，等结果期间可以做其他的事情，结果来了会收到通知。举例来说就是你烧水的时候可以去做自己想做的事情，听到水烧开的声音之后再去处理。

从概念就可以看出来，异步任务从一定程度上来看比同步任务更高效一些，核心是提高了用户体验。

## Event Loop

Event Loop 很好的调度了任务的运行，宏任务和微任务也知道了，现在我们就来看看它的调度运行机制。

JavaScript 的代码执行时，主线程会从上到下一步步的执行代码，同步任务会被依次加入执行栈中先执行，异步任务会在拿到结果的时候将注册的回调函数放入任务队列，当执行栈中的没有任务在执行的时候，引擎会从任务队列中读取任务压入执行栈(Call Stack)中处理执行。

### 宏任务和微任务

现在就有一个问题了，任务队列是一个消息队列，先进先出，那就是说，后来的事件都是被加在队尾等到前面的事件执行完了才会被执行。如果在执行的过程中突然有重要的数据需要获取，或是说有事件突然需要处理一下，按照队列的先进先出顺序这些是无法得到及时处理的。这个时候就催生了宏任务和微任务，微任务使得一些异步任务得到及时的处理。

曾经看到的一个例子很好，宏任务和微任务形象的来说就是：你去营业厅办一个业务会有一个排队号码，当叫到你的号码的时候你去窗口办充值业务(宏任务执行)，在你办理充值的时候你又想改个套餐(微任务)，这个时候工作人员会直接帮你办，不可能让你重新排队。

所以上文说过的异步任务又分为宏任务和微任务，JS 运行时任务队列会分为宏任务队列和微任务队列，分别对应宏任务和微任务。

先介绍一下(浏览器环境的)宏任务和微任务大致有哪些：

* 宏任务：
  1. script(整体的代码)
  2. setTimeout
  3. setInterval
  4. I/O 操作
  5. UI 渲染 (对这个笔者持保留意见)
* 微任务：
  1. Promise.then
  2. MutationObserver

### 事件运行顺序

1. 执行同步任务，同步任务不需要做特殊处理，直接执行(下面的步骤中遇到同步任务都是一样处理) --- 第一轮从 script开始
2. 从宏任务队列中取出队头任务执行
3. 如果产生了宏任务，将宏任务放入宏任务队列，下次轮循的时候执行
4. 如果产生了微任务，将微任务放入微任务队列
5. 执行完当前宏任务之后，取出微任务队列中的所有任务依次执行
6. 如果微任务执行过程中产生了新的微任务，则继续执行微任务，直到微任务的队列为空
7. 轮循，循环以上 2 - 6

总的来说就是：同步任务/宏任务 -> 执行产生的所有微任务(包括微任务产生的微任务) -> 同步任务/宏任务 -> 执行产生的所有微任务(包括微任务产生的微任务) -> 循环......

注意：微任务队列

### 举个栗子

光说不练假把式，现在就来看一个例子：

放图的原因是为了让大家在看解析之前可以先自己按照运行顺序走一遍，写好答案之后再来看解析。  
解析：  
(用绿色的表示同步任务和宏任务，红色表示微任务)

+ console.log('script start')  
+ setTimeout(function() {  
+ console.log('setTimeout')  
+ }, 0)  
+ new Promise((resolve, reject)=>{  
+ console.log("promise1")  
+ resolve()  
+ })  
- .then(()=>{  
- console.log("then11")  
+ new Promise((resolve, reject)=>{  
+ console.log("promise2")  
+ resolve();  
+ })  
- .then(() => {  
- console.log("then2-1")  
- })  
- .then(() => {  
- console.log("then2-2")  
- })  
- })  
- .then(()=>{  
- console.log("then12")  
- })  
+ console.log('script end')

1. 首先遇到 console.log()，输出 script start
2. 遇到 setTimeout 产生宏任务，注册到**宏任务队列[setTimeout]**，下一轮 Event Loop 的时候在执行
3. 然后遇到 new Promise 构造声明(同步)，log 输出 promise1，然后 resolve
4. resolve 匹配到 **promise1 的第一个 then**，把这个 then 注册到**微任务队列[then11]中**，继续当前整体脚本的执行
5. 遇到最后的一个 log，输出 script end，**当前执行栈清空**
6. **从微任务队列中取出队头任务'then11'** 进行执行，其中有一个 log，输出 then11
7. 往下遇到 new Promise 构造声明(同步)，log 输出 promise2，然后 resolve
8. resolve 匹配到 **promise2 的第一个 then**，把这个 then 注册到**微任务队列[then2-1]**，当前 then11 可执行部分结束，然后产生了 **promise1 的第二个 then**，把这个 then 注册到**微任务队列[then2-1, then12]**
9. **拿出微任务队头任务'then2-1'** 执行，log 输出 then2-1，触发 **promise2 的第二个 then**，注册到**微任务队列[then12, then2-2]**
10. **拿出微任务队头任务'then12'**，log 输出 then12
11. **拿出微任务队头任务'then2-2'**，log 输出 then2-2
12. 微任务队列执行完毕，别忘了宏任务队列中的 setTimeout，log 输出 setTimeout

经过以上一番缜(xia)密(gao)分析，希望没有绕晕你，最后的输出结果就是：  
script start -> promise1 -> script end -> then11 -> promise2 -> then2-1 -> then12 -> then2-2 -> setTimeout

### 宏任务？微任务？

不知道大家看了宏任务和微任务之后会不会有一个疑惑，宏任务和微任务都是异步任务，微任务之前说过了是为了及时解决一些必要事件而产生的。

* 为什么要有微任务？  
  为什么要有微任务的原因前面已经说了，这里就不再赘述，简单说一下就是为了及时处理一些任务，不然等到最后再执行的时候拿到的数据可能已经是被污染的数据达不到预期目标了。
* 是什么宏任务？什么是微任务？  
  相信大家在学习 Event Loop 查找资料的时候，肯定各种资料里面都会讲到宏任务和微任务，但是不知道你有没有灵魂拷问过你自己：什么是宏任务？什么是微任务？怎么区分宏任务和微任务？不能只是默许接受这个概念，在这里，我根据我的个人理解进行一番说(hu)明(che)
* 宏任务和微任务的真面目  
  其实在 Chrome 的源码中并没有什么宏任务和微任务的代码或是说明，在 **JS 大会[3]**上提到过微任务这个名词，但是也没有说到底什么是微任务。

宏任务  
文章最开始的时候说过，在 chrome 里，每个页面都对应一个进程。而该进程又有多个线程，比如 JS 线程、渲染线程、IO 线程、网络线程、定时器线程等等，这些线程之间的通信是通过向对象的任务队列中添加一个任务（postTask）来实现的。**宏任务的本质可以认为是多线程事件循环或消息循环，也就是线程间通信的一个消息队列。**

就拿 setTimeout 举例来说，当遇到它的时候，浏览器就会对 Event Loop 说：嘿，我有一个任务交给你，Event Loop 就会说：好的，我会把它加到我的 todoList 中，之后我会执行它，它是需要调用 API 的。

**宏任务的真面目是浏览器派发，与 JS 引擎无关的，参与了 Event Loop 调度的任务**

微任务  
微任务是在运行宏任务/同步任务的时候产生的，是属于当前任务的，所以它不需要浏览器的支持，内置在 JS 当中，直接在 JS 的引擎中就被执行掉了。

### 特殊的点

1. async 隐式返回 Promise 作为结果
2. 执行完 await 之后直接跳出 async 函数，让出执行的所有权
3. 当前任务的其他代码执行完之后再次获得执行权进行执行
4. 立即 resolve 的 Promise 对象，是在本轮"事件循环"的结束时执行，而不是在下一轮"事件循环"的开始时

### 再举个栗子

console.log('script start')  
  
 asyncfunction async1() {  
 await async2()  
 console.log('async1 end')  
 }  
 asyncfunction async2() {  
 console.log('async2 end')  
 }  
 async1()  
  
 setTimeout(function() {  
 console.log('setTimeout')  
 }, 0)  
  
 newPromise(resolve => {  
 console.log('Promise')  
 resolve()  
 })  
 .then(function() {  
 console.log('promise1')  
 })  
 .then(function() {  
 console.log('promise2')  
 })  
  
 console.log('script end')

按照之前的分析方法去分析之后就会得出一个结果：  
script start => async2 end => Promise => script end => promise1 => promise2 => async1 end => setTimeout

可以看出 async1 函数获取执行权是作为微任务的队尾，但是，在 Chrome73(金丝雀) 版本之后，async 的执行优化了，它会在 promise1 和 promise2 的输出之前执行。笔者大概了解了一下应该是用 PromiseResolve 对 await 进行了优化，减少了 Promise 的再次创建，有兴趣的小伙伴可以看看 Chrome 的源码。

# 五、Node 中的 Event Loop

Node 中也有宏任务和微任务，与浏览器中的事件循环类似。Node 与浏览器事件循环不同，其中有多个宏任务队列，而浏览器是只有一个宏任务队列。

Node 的架构底层是有 libuv，它是 Node 自身的动力来源之一，通过它可以去调用一些底层操作，Node 中的 Event Loop 功能就是在 libuv 中封装实现的。

## 宏任务和微任务

Node 中的宏任务和微任务在浏览器端的 JS 相比增加了一些，这里只列出浏览器端没有的：

* 宏任务
  1. setImmediate
* 微任务
  1. process.nextTick

## 事件循环机制的六个阶段

### 六个阶段

Node 的事件循环分成了六个阶段，每个阶段对应一个宏任务队列，相当于是宏任务进行了一个分类。

1. timers(计时器)  
   执行 setTimeout 以及 setInterval 的回调
2. I/O callbacks  
   处理网络、流、TCP 的错误回调
3. idel, prepare --- 闲置阶段  
   node 内部使用
4. poll(轮循)  
   执行 poll 中的 I/O 队列，检查定时器是否到时间
5. check(检查)  
   存放 setImmediate 回调
6. close callbacks  
   关闭回调，例如 sockect.on('close')

### 轮循顺序

执行的轮循顺序 --- 每个阶段都要等对应的宏任务队列执行完毕才会进入到下一个阶段的宏任务队列

1. timers
2. I/O callbacks
3. poll
4. setImmediate
5. close events

每两个阶段之间执行微任务队列

## Event Loop 过程

1. 执行全局的 script 同步代码
2. 执行微任务队列，先执行所有 Next Tick 队列中的所有任务，再执行其他的微任务队列中的所有任务
3. 开始执行宏任务，共六个阶段，从第一个阶段开始执行自己宏任务队列中的所有任务(浏览器是从宏任务队列中取第一个执行！！)
4. 每个阶段的宏任务执行完毕之后，开始执行微任务
5. TimersQueue -> 步骤2 -> I/O Queue -> 步骤2 -> Check Queue -> 步骤2 -> Close Callback Queue -> 步骤2 -> TimersQueue ...

这里要注意的是，nextTick 事件是一个单独的队列，它的优先级会高于微任务，所以在当前宏任务/同步任务执行完成之后，会先执行 nextTick 队列中的所有任务，再去执行微任务队列中的所有任务。

## setTimeout 和 setImmediate

在这里要单独说一下 setTimeout 和 setImmediate，setTimeout 定时器很熟悉，那就说说 setImmediate

setImmediate() 方法用于把一些需要长时间运行的操作放在一个回调函数里，并在浏览器完成其他操作（如事件和显示更新）后立即运行回调函数。从定义来看就是为了防止一些耗时长的操作阻塞后面的操作，这也是为什么 check 阶段运行顺序排的比较后。

### 举个栗子

我们来看这样的一个例子：

setTimeout(() => {  
 console.log('setTimeout')  
}, 0)  
  
setImmediate(() => {  
 console.log('setImmediate')  
})

这里涉及 timers 阶段和 check 阶段，按照上面的运行顺序来说，timers 阶段是在第一个执行的，会早于 check 阶段。运行这段程序可以看到如下的结果：

可是再多运行几次，你就会看到如下的结果：

setImmediate 的输出跑到 setTimeout 前面去了，这时候就是：小朋友你是否有很多的问号❓

### 分析

我们来分析一下原因，timers 阶段确实是在 check 阶段之前，但是在 timers 阶段时候，这里的 setTimeout 真的到了执行的时间吗？

这里就要先看看 setTiemout(fn, 0)，这个语句的意思不是指不延迟的执行，而是指在可以执行 setTimeout 的时候就立即执行它的回调，也就是处理完当前事件的时候立即执行回调。

在 Node 中 setTimeout 第二个时间参数的最小值是 1ms，小于 1ms 会被初始化为 1(浏览器中最小值是 4ms)，所以在这里 setTimeout(fn, 0) === setTimeout(fn, 1)

setTimeout 的回调函数在 timers 阶段执行，setImmediate 的回调函数在 check 阶段执行，Event Loop 的开始会先检查 timers 阶段，但是在代码开始运行之前到 timers 阶段(代码的启动、运行)会消耗一定的时间，所以会出现两种情况：

1. timers 前的准备时间超过 1ms，满足 loop -> timers >= 1，setTimeout 的时钟周期到了，则执行 timers 阶段(setTimeout)的回调函数
2. timers 前的准备时间小于 1ms，还没到 setTimeout 预设的时间，则先执行 check 阶段(setImmediate)的回调函数，下一次 Event Loop 再进入 timers 阶段执行 timer 阶段(setTimeout)的回调函数

最开始就说了，一个优秀的程序员要让自己的代码按照自己想要的顺序运行，下面我们就来控制一下 setTimeout 和 setImediate 的运行。

* 让 setTimeout 先执行  
  上面代码运行顺序不同无非就是因为 Node 准备时间的不确定性，我们可以直接手动延长准备时间👇

const start = Date.now()  
 while (Date.now() - start < 10)  
 setTimeout(() => {  
 console.log('setTimeout')  
 }, 0)  
  
 setImmediate(() => {  
 console.log('setImmediate')  
 })

* 让 setImmediate 先执行  
  setImmediate 是在 check 阶段执行，相对于 setTimeout 来说是在 timers 阶段之后，只需要想办法把程序的运行环境控制在 timers 阶段之后就可以了。

让程序至少从 I/O callbacks 阶段开始 --- 可以套一层文件读写把把程序控制在 I/O callbacks 阶段的运行环境中👇

const fs = require('fs')  
  
fs.readFile(\_\_dirname, () => {  
 setTimeout(() => {  
 console.log('setTimeout')  
 }, 0)  
   
 setImmediate(() => {  
 console.log('setImmediate')  
 })  
})

## Node 11.x 的变化

timers 阶段的执行有所变化

setTimeout(() =>console.log('timeout1'))  
setTimeout(() => {  
 console.log('timeout2')  
 Promise.resolve().then(() =>console.log('promise resolve'))  
})

1. node 10 及之前的版本：  
   要考虑上一个定时器执行完成时，下一个定时器是否到时间加入了任务队列中，如果未到时间，先执行其他的代码。  
   比如：  
   timer1 执行完之后 timer2 到了任务队列中，顺序为 timer1 -> timer2 -> promise resolve  
   timer2 执行完之后 timer2 还没到任务队列中，顺序为 timer1 -> promise resolve -> timer2
2. node 11 及其之后的版本：  
   timeout1 -> timeout2 -> promise resolve  
   一旦执行某个阶段里的一个宏任务之后就立刻执行微任务队列，这和浏览器端运行是一致的。

## 小结

Node 端和浏览器端有什么不同

1. 浏览器端的 Event Loop 和 Node.js 中的 Event Loop 是不同的，实现机制也不一样
2. Node.js 可以理解成有4个宏任务队列和2个微任务队列，但是执行宏任务时有6个阶段
3. Node.js 中限制性全局 script 代码，执行完同步代码后，先从微任务队列 Next Tick Queue 中取出所有任务放入调用栈执行，再从其他微任务队列中取出所有任务放入调用栈中执行，然后开始宏任务的6个阶段，每个阶段都将其宏任务队列中的所有任务都取出来执行(浏览器是只取第一个执行)，每个宏任务阶段执行完毕之后开始执行微任务，再开始执行下一阶段宏任务，以此构成事件循环
4. 宏任务包括 ....
5. 微任务包括 ....

看到这里，你应该对浏览器端和 Node 端的 Event Loop 有了一定的了解，那就留一个题目。

不直接放代码是想让大家先自己思考然后在敲代码运行一遍~

### 最后多一嘴

本文到这里算是结束了，还是那句话，做一个程序员要知其然更要知其所以然。我写些文章也是想把知识输出，检验自己是不是真的学懂了。文章中可能还存在一些没有说清楚的地方或者是有错的地方，欢迎直接指出~

我把我的学习记录都记录在了我的 github 并且会持续的更新下去，有兴趣的小伙伴可以看看~  
**github[4]**

### 参考资料

[1]博客链接: https://tearill.github.io/

[2]Wikipedia: https://en.wikipedia.org/wiki/Event\_loop

[3]JS 大会: https://www.bilibili.com/video/BV1bE411B7ez?t=478

[4]github: https://github.com/tearill/Reading\_Record