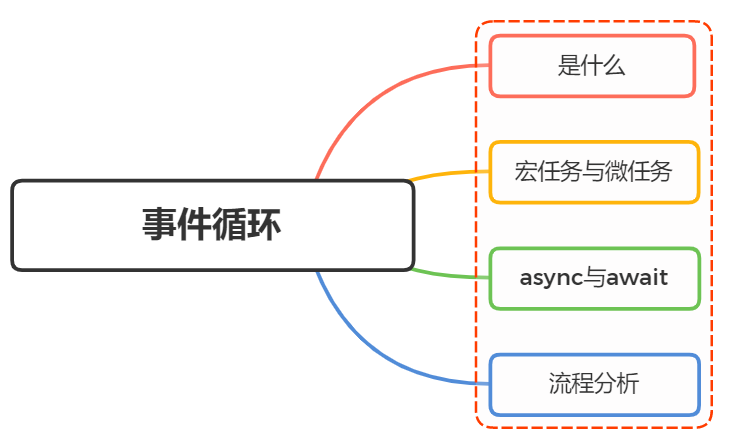
# 面试官：说说你对事件循环的理解



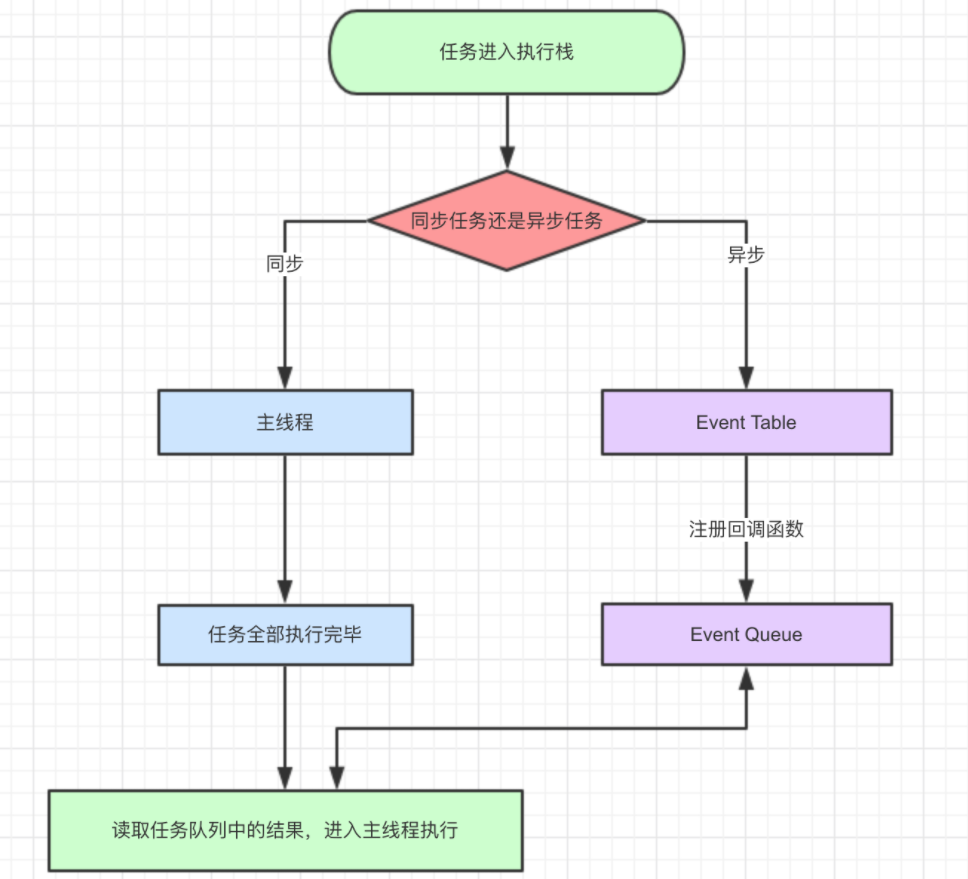
## 一、是什么

首先，JavaScript是一门单线程的语言，意味着同一时间内只能做一件事，但是这并不意味着单线程就是阻塞，而实现单线程非阻塞的方法就是事件循环

在JavaScript中，所有的任务都可以分为

* 同步任务：立即执行的任务，同步任务一般会直接进入到主线程中执行
* 异步任务：异步执行的任务，比如ajax网络请求，setTimeout定时函数等

同步任务与异步任务的运行流程图如下：



从上面我们可以看到，同步任务进入主线程，即主执行栈，异步任务进入任务队列，主线程内的任务执行完毕为空，会去任务队列读取对应的任务，推入主线程执行。上述过程的不断重复就事件循环

## 二、宏任务与微任务

如果将任务划分为同步任务和异步任务并不是那么的准确，举个例子：

console.log(1)  
  
setTimeout(()=>{  
 console.log(2)  
}, 0)  
  
new Promise((resolve, reject)=>{  
 console.log('new Promise')  
 resolve()  
}).then(()=>{  
 console.log('then')  
})  
  
console.log(3)

如果按照上面流程图来分析代码，我们会得到下面的执行步骤：

* console.log(1)，同步任务，主线程中执行
* setTimeout() ，异步任务，放到 Event Table，0 毫秒后console.log(2)回调推入 Event Queue 中
* new Promise ，同步任务，主线程直接执行
* .then ，异步任务，放到 Event Table
* console.log(3)，同步任务，主线程执行

所以按照分析，它的结果应该是 1 => 'new Promise' => 3 => 2 => 'then'

但是实际结果是：1=>'new Promise'=> 3 => 'then' => 2

出现分歧的原因在于异步任务执行顺序，事件队列其实是一个“先进先出”的数据结构，排在前面的事件会优先被主线程读取

例子中 setTimeout回调事件是先进入队列中的，按理说应该先于 .then 中的执行，但是结果却偏偏相反

原因在于异步任务还可以细分为微任务与宏任务

### 微任务

一个需要异步执行的函数，执行时机是在主函数执行结束之后、当前宏任务结束之前

常见的微任务有：

* Promise.then
* MutaionObserver
* Object.observe（已废弃；Proxy 对象替代）
* process.nextTick（Node.js）

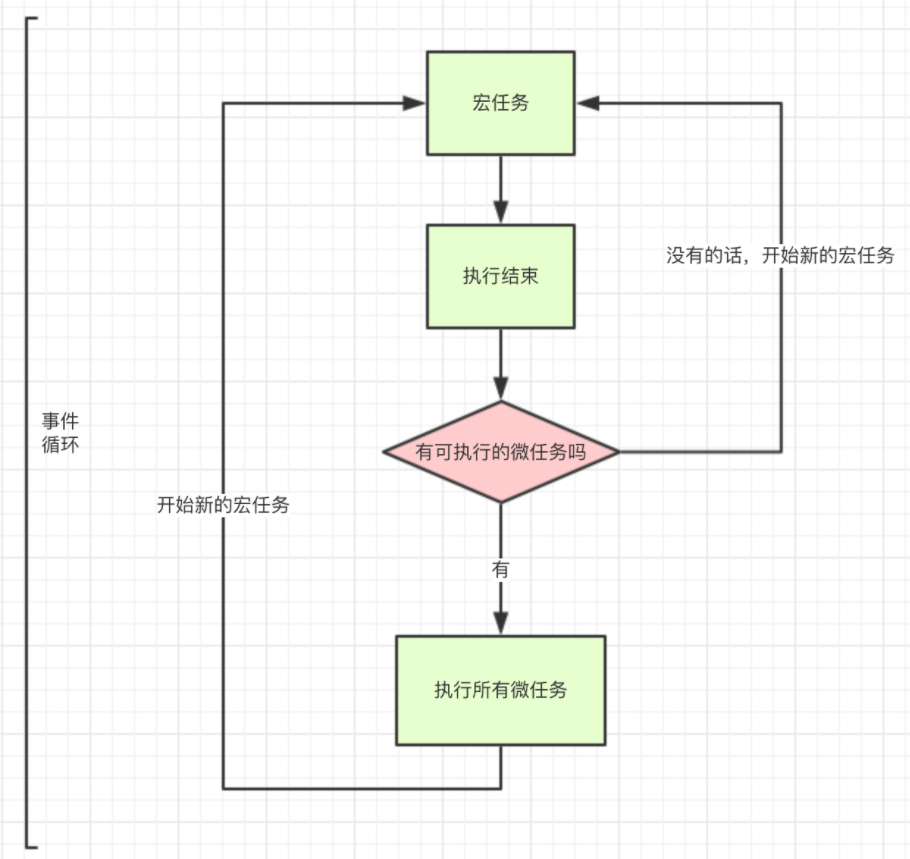
### 宏任务

宏任务的时间粒度比较大，执行的时间间隔是不能精确控制的，对一些高实时性的需求就不太符合

常见的宏任务有：

* script (可以理解为外层同步代码)
* setTimeout/setInterval
* UI rendering/UI事件
* postMessage、MessageChannel
* setImmediate、I/O（Node.js）

这时候，事件循环，宏任务，微任务的关系如图所示



按照这个流程，它的执行机制是：

* 执行一个宏任务，如果遇到微任务就将它放到微任务的事件队列中
* 当前宏任务执行完成后，会查看微任务的事件队列，然后将里面的所有微任务依次执行完

回到上面的题目

console.log(1)  
setTimeout(()=>{  
 console.log(2)  
}, 0)  
new Promise((resolve, reject)=>{  
 console.log('new Promise')  
 resolve()  
}).then(()=>{  
 console.log('then')  
})  
console.log(3)

流程如下

// 遇到 console.log(1) ，直接打印 1  
// 遇到定时器，属于新的宏任务，留着后面执行  
// 遇到 new Promise，这个是直接执行的，打印 'new Promise'  
// .then 属于微任务，放入微任务队列，后面再执行  
// 遇到 console.log(3) 直接打印 3  
// 好了本轮宏任务执行完毕，现在去微任务列表查看是否有微任务，发现 .then 的回调，执行它，打印 'then'  
// 当一次宏任务执行完，再去执行新的宏任务，这里就剩一个定时器的宏任务了，执行它，打印 2

## 三、async与await

async 是异步的意思，await则可以理解为 async wait。所以可以理解async就是用来声明一个异步方法，而 await是用来等待异步方法执行

### async

async函数返回一个promise对象，下面两种方法是等效的

function f() {  
 return Promise.resolve('TEST');  
}  
  
// asyncF is equivalent to f!  
async function asyncF() {  
 return 'TEST';  
}

### await

正常情况下，await命令后面是一个 Promise对象，返回该对象的结果。如果不是 Promise对象，就直接返回对应的值

async function f(){  
 // 等同于  
 // return 123  
 return await 123  
}  
f().then(v => console.log(v)) // 123

不管await后面跟着的是什么，await都会阻塞后面的代码

async function fn1 (){  
 console.log(1)  
 await fn2()  
 console.log(2) // 阻塞  
}  
  
async function fn2 (){  
 console.log('fn2')  
}  
  
fn1()  
console.log(3)

上面的例子中，await 会阻塞下面的代码（即加入微任务队列），先执行 async外面的同步代码，同步代码执行完，再回到 async 函数中，再执行之前阻塞的代码

所以上述输出结果为：1，fn2，3，2

## 四、流程分析

通过对上面的了解，我们对JavaScript对各种场景的执行顺序有了大致的了解

这里直接上代码：

async function async1() {  
 console.log('async1 start')  
 await async2()  
 console.log('async1 end')  
}  
async function async2() {  
 console.log('async2')  
}  
console.log('script start')  
setTimeout(function () {  
 console.log('settimeout')  
})  
async1()  
new Promise(function (resolve) {  
 console.log('promise1')  
 resolve()  
}).then(function () {  
 console.log('promise2')  
})  
console.log('script end')

分析过程：

1. 执行整段代码，遇到 console.log('script start') 直接打印结果，输出 script start
2. 遇到定时器了，它是宏任务，先放着不执行
3. 遇到 async1()，执行 async1 函数，先打印 async1 start，下面遇到await怎么办？先执行 async2，打印 async2，然后阻塞下面代码（即加入微任务列表），跳出去执行同步代码
4. 跳到 new Promise 这里，直接执行，打印 promise1，下面遇到 .then()，它是微任务，放到微任务列表等待执行
5. 最后一行直接打印 script end，现在同步代码执行完了，开始执行微任务，即 await下面的代码，打印 async1 end
6. 继续执行下一个微任务，即执行 then 的回调，打印 promise2
7. 上一个宏任务所有事都做完了，开始下一个宏任务，就是定时器，打印 settimeout

所以最后的结果是：script start、async1 start、async2、promise1、script end、async1 end、promise2、settimeout