Event Loop

在前两章节中我们了解了 JS 异步相关的知识。在实践的过程中，你是否遇到过以下场景，为什么 `setTimeout` 会比 `Promise` 后执行，明明代码写在 `Promise` 之前。这其实涉及到了 Event Loop 相关的知识，这一章节我们会来详细地了解 Event Loop 相关知识，知道 JS 异步运行代码的原理，并且这一章节也是面试常考知识点。

进程与线程

> ! 涉及面试题：进程与线程区别？JS 单线程带来的好处？

相信大家经常会听到 JS 是**单线程**执行的，但是你是否疑惑过什么是线程？

讲到线程，那么肯定也得说一下进程。本质上来说，两个名词都是 CPU **工作时间片**的一个描述。

进程描述了 CPU 在**运行指令及加载和保存上下文所需的时间**，放在应用上来说就代表了一个程序。线程是进程中的更小单位，描述了执行一段指令所需的时间。

把这些概念拿到浏览器中来说，当你打开一个 Tab 页时，其实就是创建了一个进程，一个进程中可以有多个线程，比如渲染线程、JS 引擎线程、HTTP 请求线程等等。当你发起一个请求时，其实就是创建了一个线程，当请求结束后，该线程可能就会被销毁。

上文说到了 JS 引擎线程和渲染线程，大家应该都知道，在 JS 运行的时候可能会阻止 UI 渲染，这说明了两个线程是**互斥**的。这其中的原因是因为 JS 可以修改 DOM，如果在 JS 执行的时候 UI 线程还在工作，就可能导致不能安全的渲染 UI。这其实也是一个单线程的好处，得益于 JS 是单线程运行的，可以达到节省内存，节约上下文切换时间，没有锁的问题的好处。当然前面两点在服务端中更容易体现，对于锁的问题，形象的来说就是当我读取一个数字 15 的时候，同时有两个操作对数字进行了加减，这时候结果就出现了错误。解决这个问题也不难，只需要在读取的时候加锁，直到读取完毕之前都不能进行写入操作。

执行栈

> ! 涉及面试题：什么是执行栈？

可以把执行栈认为是一个存储函数调用的**栈结构**，遵循先进后出的原则。

```
1  function foo(b) {
2      var a = 5;
3      return a * b + 10;
4  }
5
6  function bar(x) {
7      var y = 3;
8      return foo(x * y);
9  }
10
11 console.log(bar(6));
```

输出：

执行栈可视化

当开始执行 JS 代码时，首先会执行一个 `main` 函数，然后执行我们的代码。根据先进后出的原则，后执行的函数会先弹出栈，在图中我们也可以发现， `foo` 函数后执行，当执行完毕后就从栈中弹出了。

平时在开发中，大家也可以在报错中找到执行栈的痕迹

```js
function foo() {
  throw new Error('error')
}
function bar() {
  foo()
}
bar()
```

```
Uncaught Error: error                                    VM169:2
    at foo (<anonymous>:2:9)
    at bar (<anonymous>:5:3)
    at <anonymous>:7:1
  foo         @ VM169:2
  bar         @ VM169:5
  (anonymous) @ VM169:7
```

函数执行顺序

大家可以在上图清晰的看到报错在 `foo` 函数， `foo` 函数又是在 `bar` 函数中调用的。

当我们使用递归的时候，因为栈可存放的函数是有**限制**的，一旦存放了过多的函数且没有得到释放的话，就会出现爆栈的问题

```js
function bar() {
  bar()
}
bar()
```

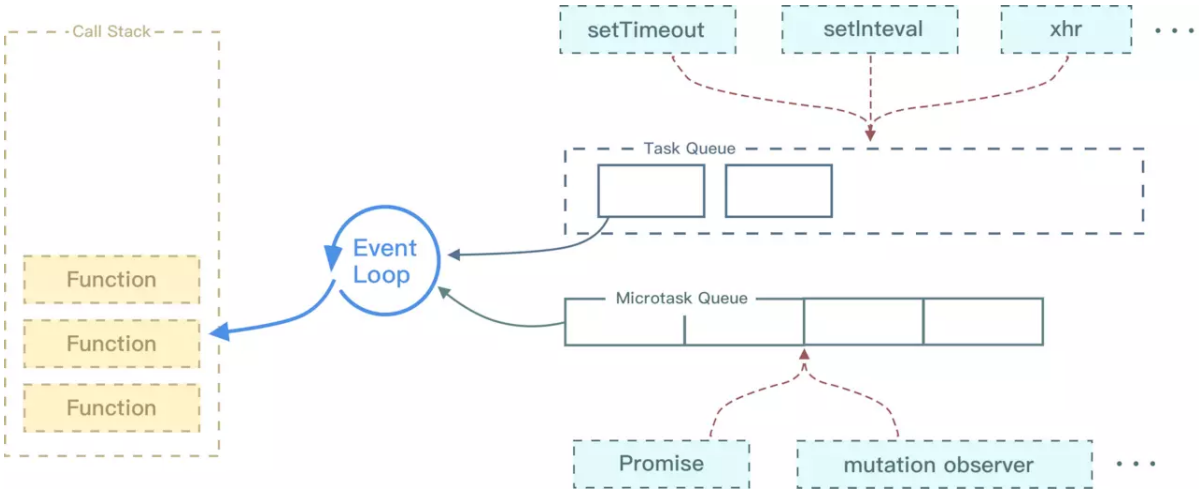
```
▸Uncaught RangeError: Maximum call stack size exceeded
    at bar (<anonymous>:1:13)
    at bar (<anonymous>:2:3)
    at bar (<anonymous>:2:3)
    at bar (<anonymous>:2:3)
    at bar (<anonymous>:2:3)
    at bar (<anonymous>:2:3)
    at bar (<anonymous>:2:3)
    at bar (<anonymous>:2:3)
    at bar (<anonymous>:2:3)
```

爆栈

浏览器中的 Event Loop

> ! 涉及面试题：异步代码执行顺序？解释一下什么是 Event Loop ？

上一小节我们讲到了什么是执行栈，大家也知道了当我们执行 JS 代码的时候其实就是往执行栈中放入函数，那么遇到异步代码的时候该怎么办？其实当遇到异步的代码时，会被**挂起**并在需要执行的时候加入到 Task（有多种 Task） 队列中。一旦执行栈为空，Event Loop 就会从 Task 队列中拿出需要执行的代码并放入执行栈中执行，所以本质上来说 JS 中的异步还是同步行为。



事件循环

不同的任务源会被分配到不同的 Task 队列中，任务源可以分为 **微任务**（microtask）和 **宏任务**（macrotask）。在 ES6 规范中，microtask 称为 `jobs`，macrotask 称为 `task`。下面来看以下代码的执行顺序：

```js
console.log('script start')

async function async1() {
  await async2()
  console.log('async1 end')
}
async function async2() {
  console.log('async2 end')
}
async1()

setTimeout(function() {
  console.log('setTimeout')
}, 0)

new Promise(resolve => {
  console.log('Promise')
  resolve()
})
  .then(function() {
    console.log('promise1')
  })
  .then(function() {
    console.log('promise2')
  })

console.log('script end')
// script start => async2 end => Promise => script end => promise1 => promise2 => async1
```

> ! 注意：新的浏览器中不是如上打印的，因为 await 变快了，具体内容可以往下看

首先先来解释下上述代码的 `async` 和 `await` 的执行顺序。当我们调用 `async1` 函数时，会马上输出 `async2 end`，并且函数返回一个 `Promise`，接下来在遇到 `await` 的时候会就让出线程开始执行 `async1` 外的代码，所以我们完全可以把 `await` 看成是**让出线程**的标志。

然后当同步代码全部执行完毕以后，就会去执行所有的异步代码，那么又会回到 `await` 的位置执行返回的 `Promise` 的 `resolve` 函数，这又会把 `resolve` 丢到微任务队列中，接下来去执行 `then` 中的回调，当两个 `then` 中的回调全部执行完毕以后，又会回到 `await`

的位置处理返回值，这时候你可以看成是 `Promise.resolve(返回值).then()` ，然后 `await` 后的代码全部被包裹进了 `then` 的回调中，所以 `console.log('async1 end')` 会优先执行于 `setTimeout` 。

如果你觉得上面这段解释还是有点绕，那么我把 `async` 的这两个函数改造成你一定能理解的代码

```js
new Promise((resolve, reject) => {
  console.log('async2 end')
  // Promise.resolve() 将代码插入微任务队列尾部
  // resolve 再次插入微任务队列尾部
  resolve(Promise.resolve())
}).then(() => {
  console.log('async1 end')
})
```

也就是说，如果 `await` 后面跟着 `Promise` 的话， `async1 end` 需要等待三个 tick 才能执行到。那么其实这个性能相对来说还是略慢的，所以 V8 团队借鉴了 Node 8 中的一个 Bug，在引擎底层将三次 tick 减少到了二次 tick。但是这种做法其实是违法了规范的，当然规范也是可以更改的，这是 V8 团队的一个 PR，目前已被同意这种做法。

所以 Event Loop 执行顺序如下所示：

- 首先执行同步代码，这属于宏任务
- 当执行完所有同步代码后，执行栈为空，查询是否有异步代码需要执行
- 执行所有微任务
- 当执行完所有微任务后，如有必要会渲染页面
- 然后开始下一轮 Event Loop，执行宏任务中的异步代码，也就是 `setTimeout` 中的回调函数

所以以上代码虽然 `setTimeout` 写在 `Promise` 之前，但是因为 `Promise` 属于微任务而 `setTimeout` 属于宏任务，所以会有以上的打印。

微任务包括 `process.nextTick` ， `promise` ， `MutationObserver` 。

宏任务包括 `script` ， `setTimeout` ， `setInterval` ， `setImmediate` ， `I/O` ， `UI rendering` 。

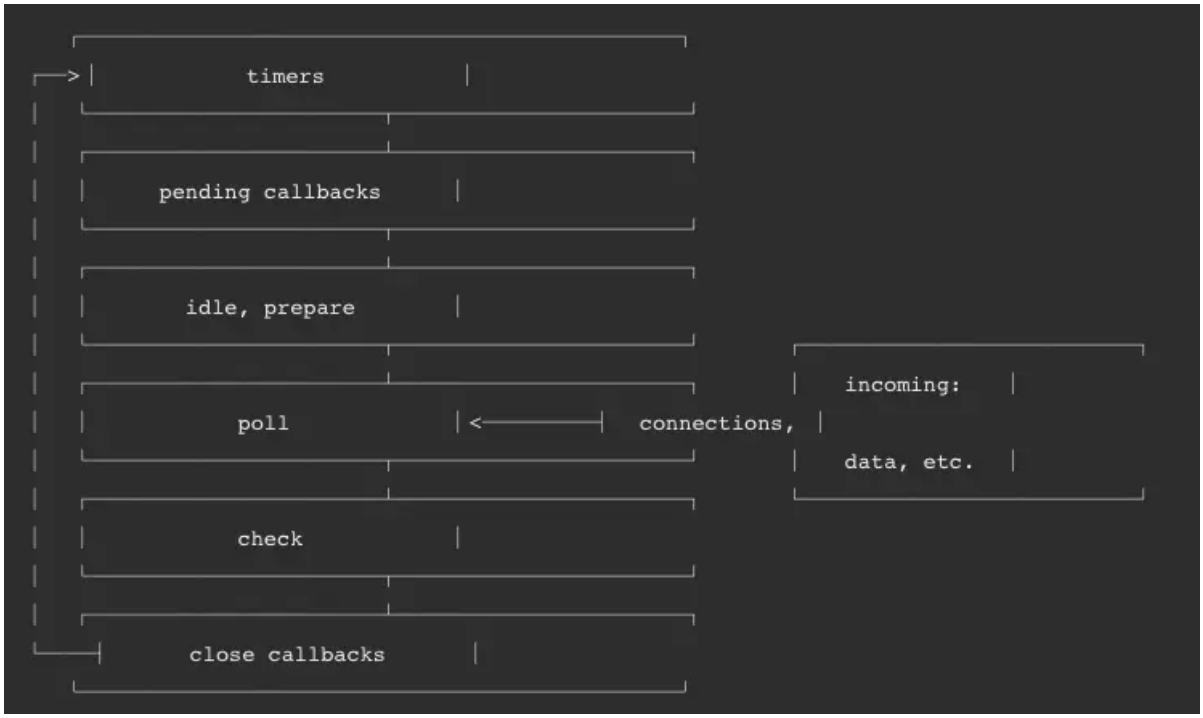
这里很多人会有个误区，认为微任务快于宏任务，其实是错误的。因为宏任务中包括了 `script` ，浏览器会**先执行一个宏任务**，接下来有异步代码的话才会先执行微任务。

Node 中的 Event Loop

> 涉及面试题：Node 中的 Event Loop 和浏览器中的有什么区别？process.nexttick 执行顺序？

Node 中的 Event Loop 和浏览器中的是完全不相同的东西。

Node 的 Event Loop 分为 6 个阶段，它们会按照**顺序**反复运行。每当进入某一个阶段的时候，都会从对应的回调队列中取出函数去执行。当队列为空或者执行的回调函数数量到达系统设定的阈值，就会进入下一阶段。



timer

timers 阶段会执行 `setTimeout` 和 `setInterval` 回调，并且是由 poll 阶段控制的。

同样，在 Node 中定时器指定的时间也不是准确时间，只能是**尽快**执行。

I/O

I/O 阶段会处理一些上一轮循环中的**少数未执行**的 I/O 回调

idle, prepare

idle, prepare 阶段内部实现，这里就忽略不讲了。

poll

poll 是一个至关重要的阶段，这一阶段中，系统会做两件事情

1. 回到 timer 阶段执行回调
2. 执行 I/O 回调

并且在进入该阶段时如果没有设定了 timer 的话，会发生以下两件事情

- 如果 poll 队列不为空，会遍历回调队列并同步执行，直到队列为空或者达到系统限制
- 如果 poll 队列为空时，会有两件事发生
  - 如果有 `setImmediate` 回调需要执行，poll 阶段会停止并且进入到 check 阶段执行回调
  - 如果没有 `setImmediate` 回调需要执行，会等待回调被加入到队列中并立即执行回调，这里同样会有个超时时间设置防止一直等待下去

当然设定了 timer 的话且 poll 队列为空，则会判断是否有 timer 超时，如果有的话会回到 timer 阶段执行回调。

check

check 阶段执行 `setImmediate`

close callbacks

close callbacks 阶段执行 close 事件

在以上的内容中，我们了解了 Node 中的 Event Loop 的执行顺序，接下来我们将会通过代码的方式来深入理解这块内容。

首先在有些情况下，定时器的执行顺序其实是**随机**的

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

对于以上代码来说， `setTimeout` 可能执行在前，也可能执行在后

- 首先 `setTimeout(fn, 0) === setTimeout(fn, 1)` ，这是由源码决定的
- 进入事件循环也是需要成本的，如果在准备时候花费了大于 1ms 的时间，那么在 timer 阶段就会直接执行 `setTimeout` 回调
- 那么如果准备时间花费小于 1ms，那么就是 `setImmediate` 回调先执行了

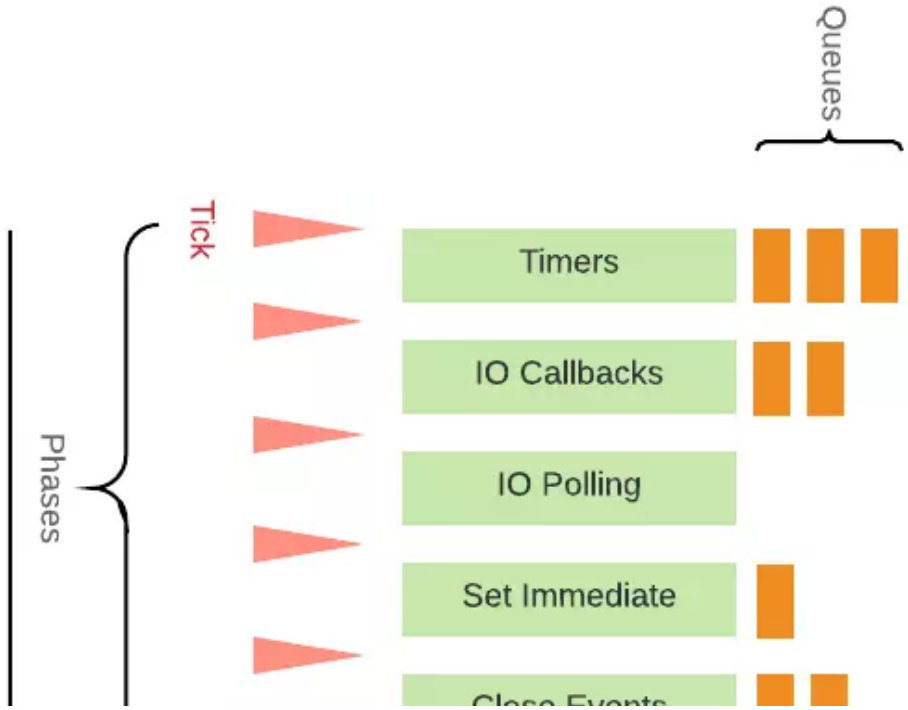
当然在某些情况下，他们的执行顺序一定是固定的，比如以下代码：

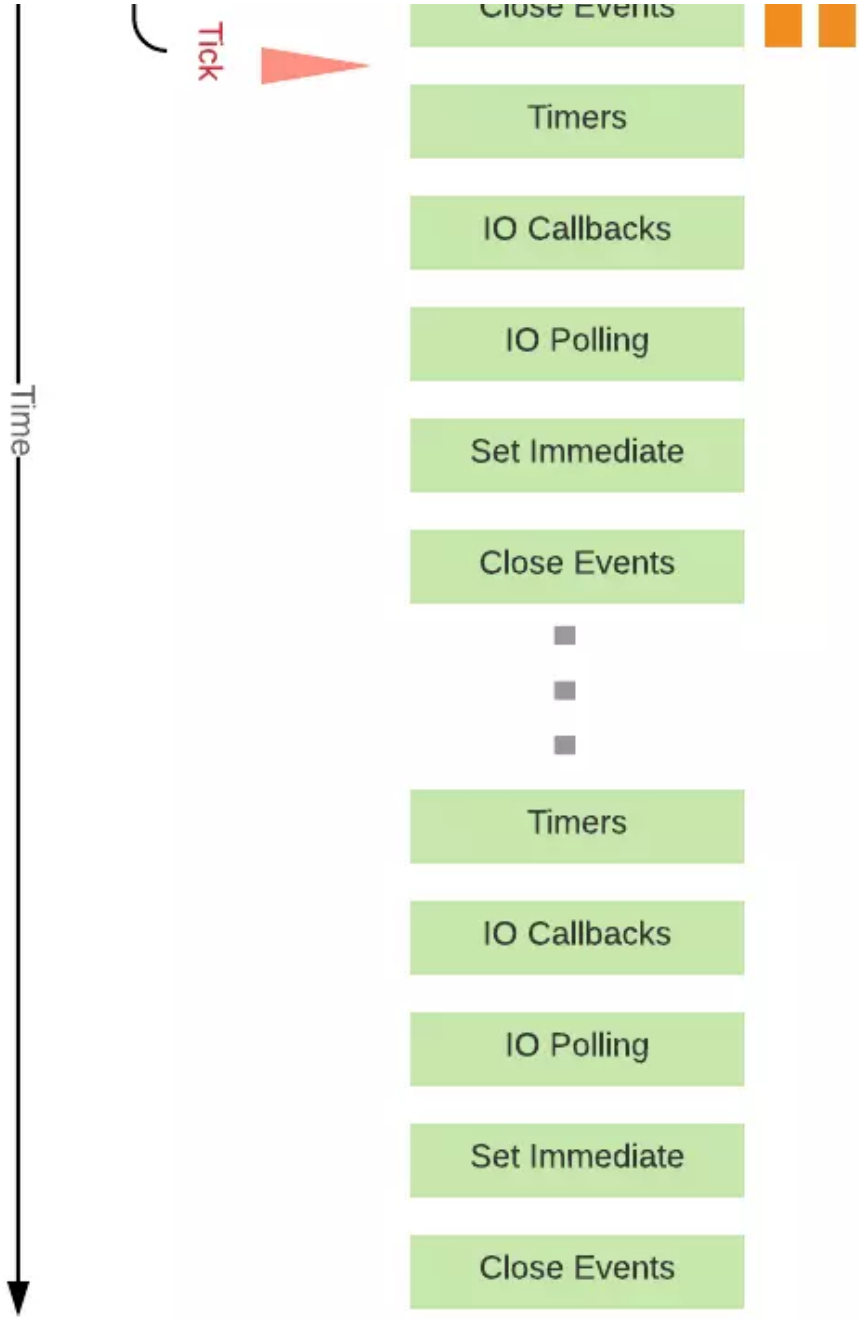
```js
const fs = require('fs')

fs.readFile(__filename, () => {
    setTimeout(() => {
        console.log('timeout');
    }, 0)
    setImmediate(() => {
        console.log('immediate')
    })
})
```

在上述代码中， `setImmediate` 永远**先执行**。因为两个代码写在 IO 回调中，IO 回调是在 poll 阶段执行，当回调执行完毕后队列为空，发现存在 `setImmediate` 回调，所以就直接跳转到 check 阶段去执行回调了。

上面介绍的都是 macrotask 的执行情况，对于 microtask 来说，它会在以上每个阶段完成前**清空** microtask 队列，下图中的 Tick 就代表了 microtask






```js
setTimeout(() => {
  console.log('timer21')
}, 0)

Promise.resolve().then(function() {
  console.log('promise1')
})
```

对于以上代码来说，其实和浏览器中的输出是一样的，microtask 永远执行在 macrotask 前面。

最后我们来讲讲 Node 中的 `process.nextTick` ，这个函数其实是独立于 Event Loop 之外的，它有一个自己的队列，当每个阶段完成后，如果存在 nextTick 队列，就会**清空队列中的所有回调函数**，并且优先于其他 microtask 执行。

```js
setTimeout(() => {
 console.log('timer1')

 Promise.resolve().then(function() {
   console.log('promise1')
 })
}, 0)

process.nextTick(() => {
 console.log('nextTick')
 process.nextTick(() => {
   console.log('nextTick')
   process.nextTick(() => {
     console.log('nextTick')
     process.nextTick(() => {
       console.log('nextTick')
     })
   })
 })
})
```

对于以上代码，大家可以发现无论如何，永远都是先把 nextTick 全部打印出来。

小结

这一章节我们学习了 JS 实现异步的原理，并且了解了在浏览器和 Node 中 Event Loop 其实是不相同的。Event Loop 这个知识点对于我们理解 JS 是如何执行的至关重要，同时也是常考题。如果大家对于这个章节的内容存在疑问，欢迎在评论区与我互动。