2.Event Loop机制 (微观)

预习资料

名称	链接	备注
Event Loops标准	https://html.spec.whatwg.org/multipage/webappapis.html#event-loops	标准对Event Loops的说明
Node.js 事件循环，定时器和 process.nextTick()	https://nodejs.org/zh-cn/docs/guides/event-loop-timers-and-nexttick/#what-is-the-event-loop	介绍了nodejs的事件循环和定时器
调用栈	https://juejin.im/post/5d116a9df265da1bb47d717b	介绍js运行调用栈
JS中的栈内存堆内存	https://juejin.im/post/5d116a9df265da1bb47d717b	

异步实现:

1. 宏观:浏览器多线程
2. 微观:EventLoop(事件循环)

一.浏览器的Event Loop 事件循环

先看一段示例:

```
> console.log(1)
  setTimeout(() => {
      //执行后 回调一个宏事件
      console.log('内层宏事件3')
  }, 0)
  console.log('外层宏事件1');

  new Promise((resolve) => {
      console.log('外层宏事件2');
      resolve()
  }).then(() => {
      console.log('微事件1');
  }).then(()=>{
      console.log('微事件2')
  })
```

```
1                                                    VM22116:1
外层宏事件1                                          VM22116:6
外层宏事件2                                          VM22116:9
微事件1                                              VM22116:12
微事件2                                              VM22116:14
◂ ▸ Promise {<fulfilled>: undefined}
内层宏事件3                                          VM22116:4
```

上面打印得结果:1 ，外层宏事件1， 外层宏事件2， 微事件1， 微事件2， 内宏事件3

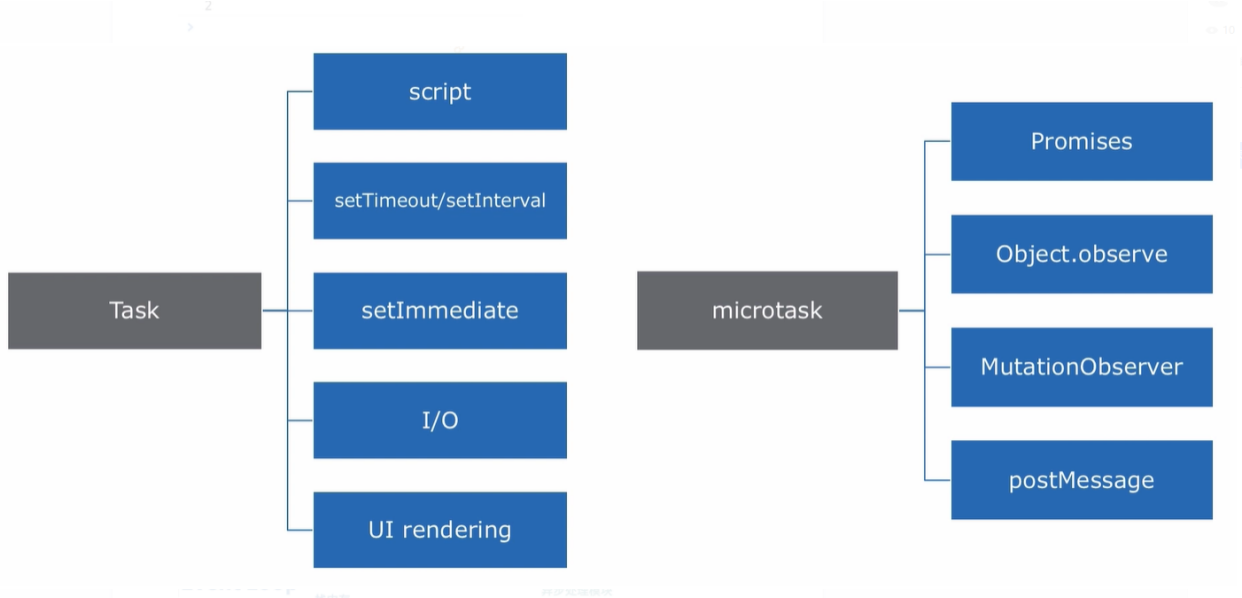
```
1.先执行大的script中的宏任务，打印出1，
2.setTimeout添加到宏任务队列
3.打印出外层宏事件1
4.执行Promise中的同步任务,打印出外层事件2
5.添加then微任务1到微任务队列
6.添加then微任务2到微任务队列
7.清空script宏任务中微任务队列，打印出微事件1 微事件2
8.进入下一轮宏任务，执行宏任务队列中的settimeout 打印出内层宏事件3
```

上述结果我们都知道先执行全部得同步任务，才会去执行异步任务,setTimeout和Promise的优先级谁高呢？按照我们前面学习的异步任务会放入任务队列，promise应该是后放入任务队列后执行才对，为什么promise比定时器先执行呢?

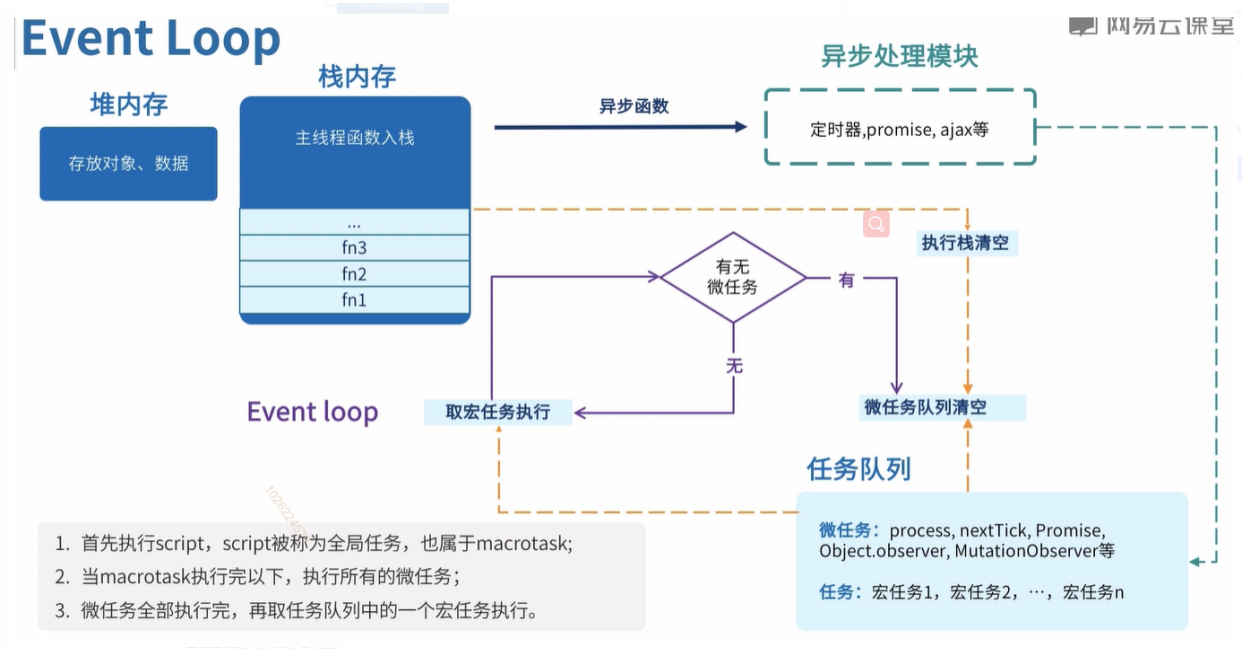
原因是:异步任务分为两类: 宏任务（macrotask ）和微任务（microtask ）

宏任务一般是: script全局执行，setTimeout，setInterval、I/O、setImmediate(nodejs) UI render

微任务主要是：Promise、process.nextTick Object.observe、MutationObserver（是一个类，监听DOM结构的API） postMessage(window之间的通讯)



事件循环示意图:



js之所以能异步，是因为浏览器多线程的关系，线程有自己的存储空间，所以这里就有堆和栈的概念，简单理解就是堆内存的空间比较大，适合存放对象等数据，栈内存空间比较小，适合存放一些基础的数据类型、对象引用和函数的调用

函数调用就会入栈，出栈就是执行，也就是我们所说的调用栈。栈是一个先进后出的数据结构，当最外层的函数出栈的时候(执行完毕之后)栈才会清空。

函数调用可能会调用一些异步函数，这个异步函数就会去找它对应的异步处理模块，主要有定时器、promise、ajax等，异步处理模块会去找它们各自对应的线程，线程会往任务队列

里添加事件。

上图中蓝绿色箭头表示往任务队列中添加事件，是一个添加队列项的操作，橘色箭头表示从任务队列中取事件，取出事件并执行这个事件对应的回调函数。任务队列它包括宏任务和微任务

这里有三个点需要注意：

1. 我们整个大的script执行被称为全局任务，属于宏任务范畴
2. 当宏任务执行完成后，会去执行所有的微任务
3. 把微任务执行完直到微任务队列清空后，才会取任务队列中的下一个宏任务进行执行

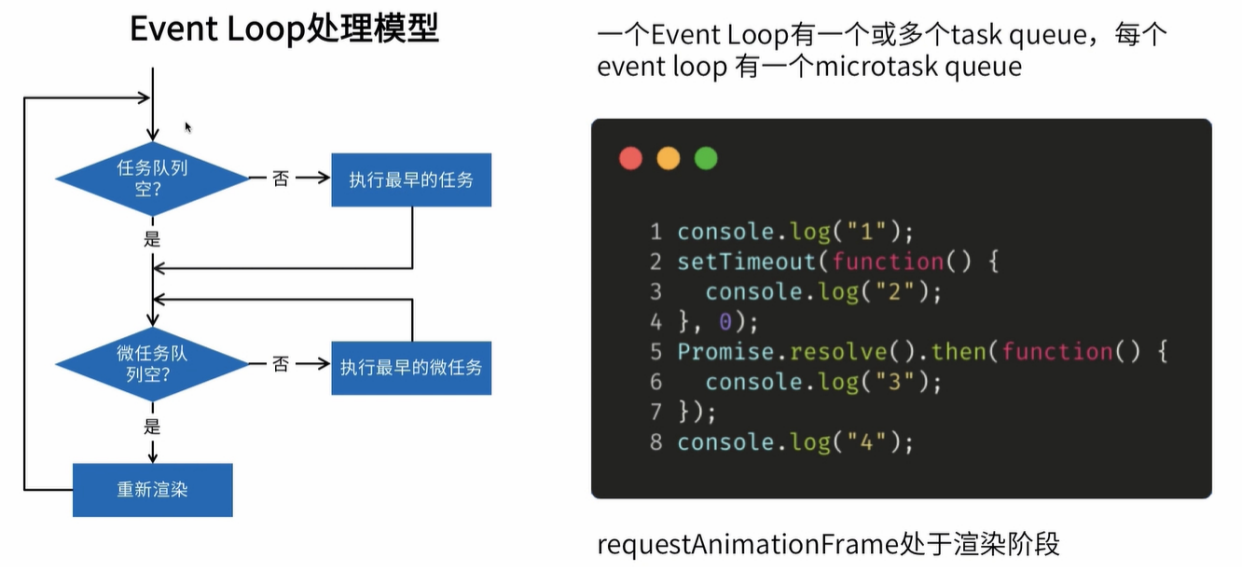
什么时候才会执行微任务？

等到调用栈为空的时候才会执行微任务，调用栈不为空时，任务队列中的微任务一直处于等待状态。

什么是Event Loop？

微任务执行完成后就去取任务队列中的宏任务依次执行，执行每个宏任务的时候都要检查一下当前任务队列中有没有微任务，如果有，就立马执行完所有的微任务再继续执行后续的宏任务。上图中紫色箭头部分的循环就是Event Loop

结合图示模型和一段代码看一下Event Loop的执行过程：



图示模型(左):(这只是对任务队列的描述和代码示例执行顺序是不一样的)

1. 检查任务队列是否为空，如果不为空，拿最早的任务出来执行
2. 如果队列是空的，就去检查微任务对队列
3. 如果微任务队列不为空，执行最早的微任务，直到所有的微任队列清空，才会进行才一个阶段。
4. 微任务队列为空，重新渲染，又重复1.2.3步骤 这就形成的事件循环

代码示例(右):

1. 首先是srcript 全局执行，它属于宏任务范畴，取出script宏任务拿出来执行，输出同步任务 结果1和4
2. 执行后检查该宏任务下的微任务队列，执行所有的微任务，执行 prmoise中的then 输出结果3 (prmoise函数体内部的代码是同步执行的，只有then中的代码才是异步执行)
3. 该宏任务下的微任务队列清空后，重新渲染，再检查一下任务队列(看是否还有宏任务)，队列不为空,任务队列还有一个setTimeout定时器宏任务，执行同步任务，输出2 ,该宏任务中没有微任....

4.循环检查任务队列.... 直至任务队列清空

再看下下面的代码

```
console.log('start')

setTimeout(() => {
    console.log('setTimeout');
    new Promise(resolve => {
        console.log('promise inner1');
        resolve();
    }).then(() => {
        console.log('promise then1');
    });
},0);

new Promise(resolve => {
    console.log('promise inner2');
    resolve();
}).then(() => {
    console.log('promise then2');
});
```

```
async function async1() {
  console.log("async1 start");
  await async2();
  console.log("async1 end");
}
async function async2() {
  return Promise.resolve().then(_ => {
    console.log("async2 promise");
  });
}

console.log("start");
setTimeout(function() {
  console.log("setTimeout");
}, 0);
async1();
new Promise(function(resolve) {
  console.log("promise1");
  resolve();
}).then(function() {
  console.log("promise2");
});
```

代码一执行过程:

```
1.执行大的script宏任务 打印出start
2.settimeout放入宏任务队列
3.执行同步promise代码 打印出promise inner2
4.微任务1放入微任务队列，调用栈为空，清空微任务队列 promise then2
5.查看宏任务队列，执行下一个宏任务 setimeout 打印出 setTimeout
6.执行同步任务 promise 打印出promise innner1，微任务2放入微任务队列中
7.调用栈为空，清空所有的微任务，打印出promise then1
8.事件循环结束
```

1. srcript 全局执行，取出宏任务执行，执行同步任务 打印出 start， promise inner2，
2. 执行完毕后，检查是否有微任务，执行所有的微任务，promise中的异步then 打印出结果promise then2，
3. 取出下一个宏任务 setTimout 执行同步任务 打印出 setTimout， Promise inner1 检查是否有微任务 执行所有的微任务 promise then1
4. 任务队列为空，事件循环结束

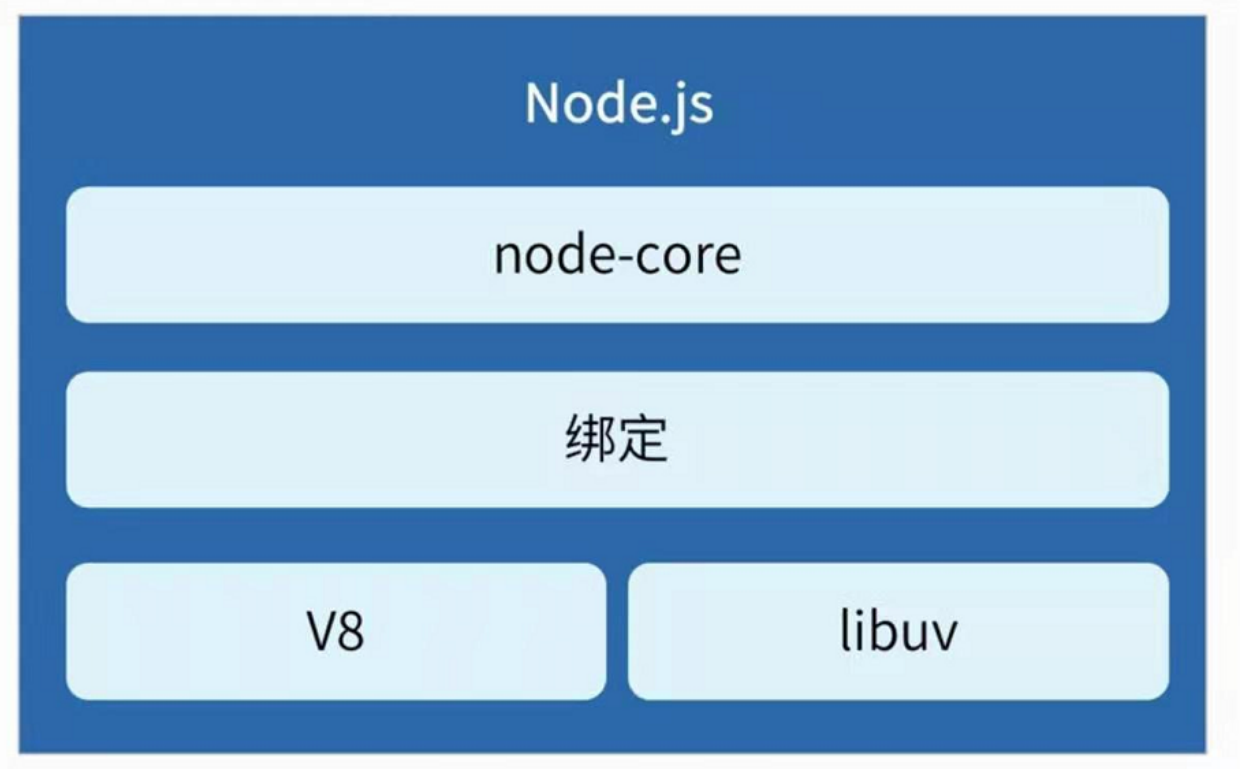
代码二执行过程：

```
1.执行大的script宏任务 打印出start
2.settimeout放入宏任务队列
3.执行async1() 打印出 async1 start1
4.遇到await 先执行async2函数的的调用,然后暂停await后面执行的,这里是执行顺序是从右向左
5.执行async2，把微任务async2 promise添加到微任务队列
6.继续执行后面的同步代码，打印出Promise中的 promise1，把微任务队列的
7.把微任务promise2放入微任务队列
8.调用栈为空后,清空微任务队列 打印出async2 promise promise2
9.继续回到await，打印出的async1 end
10 回到宏任务队列，执行下一个宏任务，打印出setimeOut
11，任务队列清空，事件循环结束
```

1. 首先大的script是一个宏任务，宏任务执行完输出start
2. 调用async1函数，输出async1 start
3. 遇到async1函数体内部的await时没有等待async2执行，输出promise1（promise1是同步执行的）
4. 执行async2函数，输出async2 promise
5. 检查是否有微任务，输出promise2（盲猜await会让执行栈优先读取微任务队列循环执行，直至清空微任务队列后再去执行其他任务）
6. 执行await之后的代码，输出async1 end
7. 取出下一个宏任务,执行输出 setTimeout
8. 任务列队为空，事件循环结束

二 .Node.js的Event Loop

node的事件循环时处理非阻塞 I/O 操作的机制 它的架构图



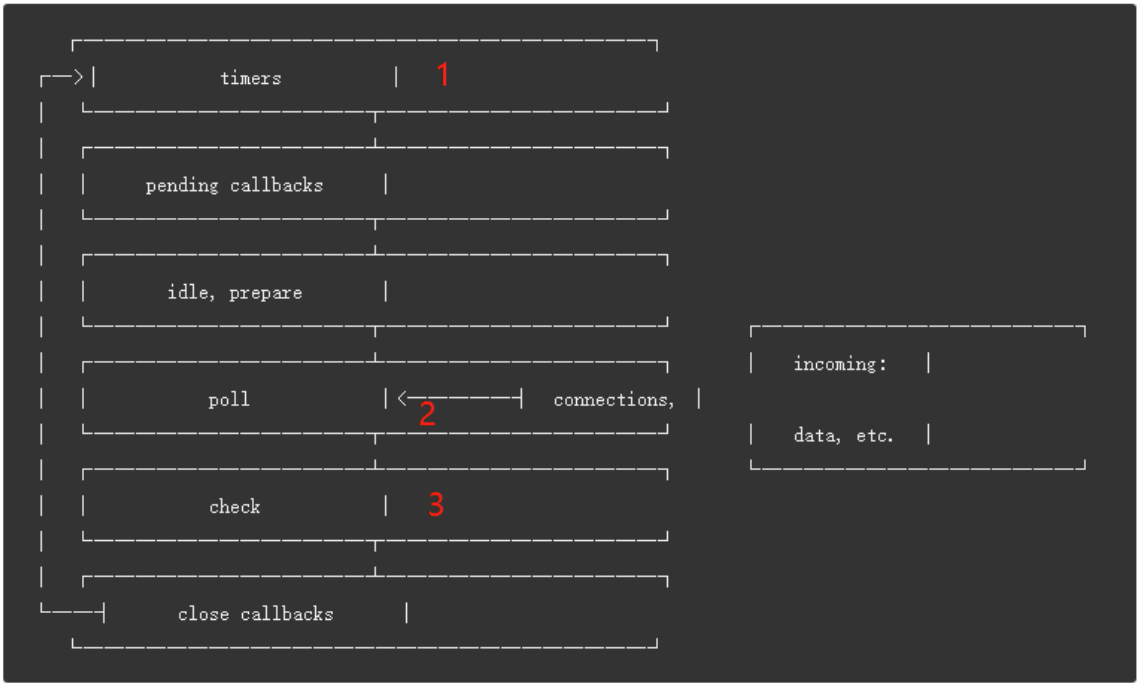
架构图一共分三层

1. 第一层 node-core --- Node的API的核心js库
2. 第二层 绑定 --- 包装和暴露 js，libuv和其他低级功能
3. V8 chrome开源的引擎,libuv第三方库，node底层的I/O引擎，C语言编写的事件驱动的一个库，它主要负责nodeApi的执行，它会分配不同的任务给不同的线程，从而形成一个EventLoop,它以异步的方式将任务的执行结果返回给V8引擎。node.js是非阻塞的I/O单线程，它实现非阻塞的原因就在libuv库这里(node实现异步的原因)
浏览器实现异步的是浏览器内核的多线程

node.js的Event Loop分为哪几个阶段?

1. timers：执行timer的回调，比如setTimeout、setInterval在此阶段执行
2. pending callbacks：系统操作的回调，暂时不用关心这个阶段
3. idle，pepare：内部使用，暂时不用关心这个阶段
4. poll：等待新I/O事件进来,也可以称作轮询,主要是控制何时定时器执行
5. check：执行setImmediate回调
6. close callbacks：内部使用，这个阶段会执行一些socket.on('close', ...)操作，也不太需要关心。
7.

这6个阶段我们暂时需要关注三个阶段,一个是timers 阶段 一个poll阶段 一个是check阶段



每一个阶段都有一个callbacks的先进先出的队列需要执行，比如定时器的阶段它有定时器的队列，check的阶段有check的队列。每个阶段都有一个 FIFO 队列来执行回调,当Event Loop运行到一个指定阶段时，该阶段的FIFO 队列将会被执行，当队列callback执行完或者执行的callbacks数量超过该阶段的上限时，Event Loop会转入下一个阶段。（callbacks数量有上限，如果超过这个上限就不会执行完，直接进入下一个阶段）

什么是I/O操作?

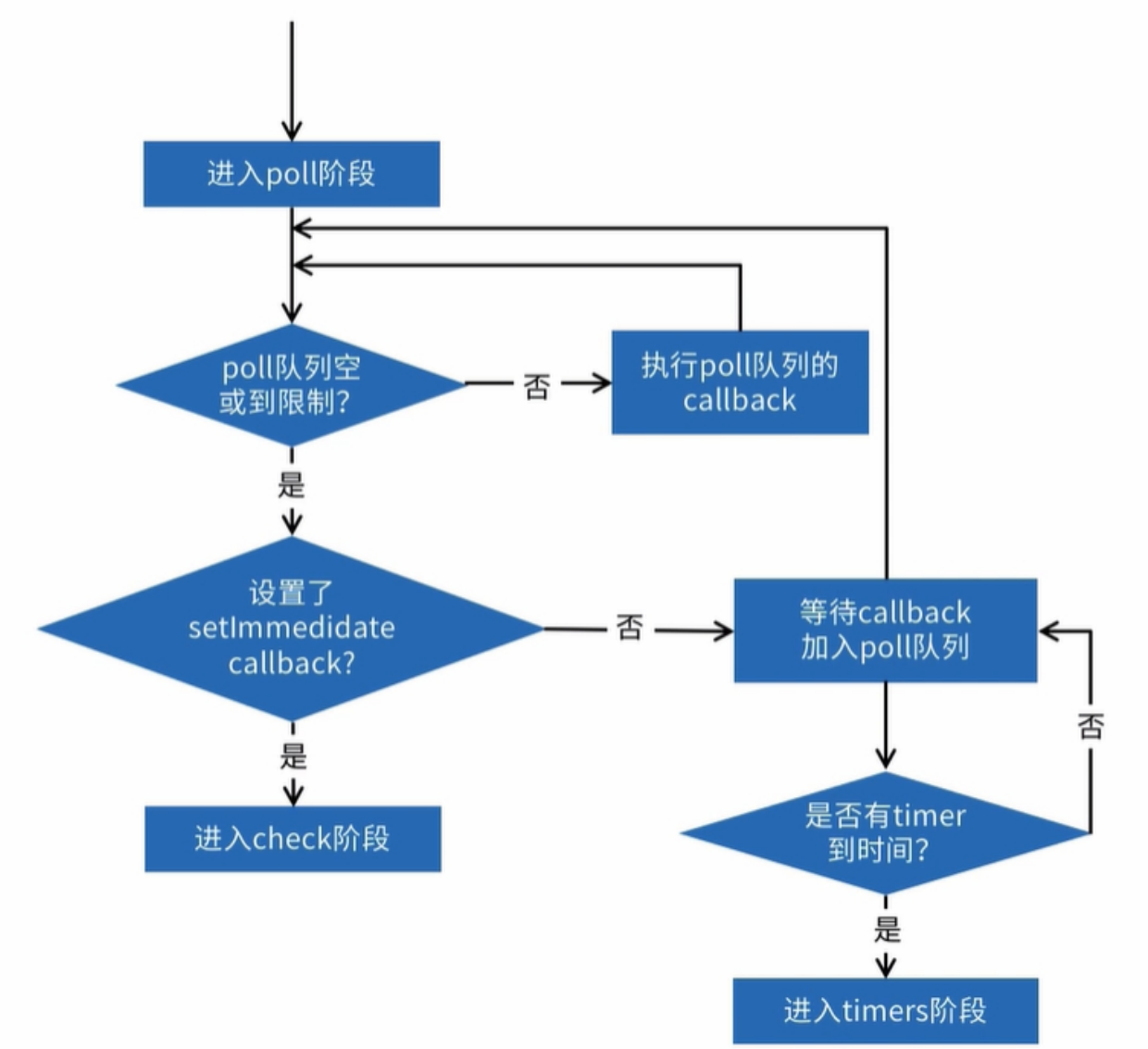
数据在内部存储器和外部存储器或其他周边设备之间的输入和输出。

同步的I/O　例如:文件一次性的全部读取,然后一次性的输出

异步的I/O　例如:文件分布读取,一边读出,一边输出

poll阶段主要有两个功能:

1. 计算应该被block多久，为什么是block多久呢？因为poll阶段会等待I/O的操作(读取文件就一种I/O的操作）另外一种说法:计算应该阻塞和轮询I/O的时间
2. 处理poll队列的事件



- 上图首先Event Loop进入poll阶段，进入poll阶段后查看poll阶段队列中是不是空的或者callbacks数量到了上限
- 如果不是空的callbacks也没有到上限，就执行poll队列里面的callback，循环这个过程直到poll队列空了或者到了限制
- 这个时候看一下setImmedidate有没有设置callback，如果有就进入check阶段
- 如果没有设置就会有一个等待状态，会等待callback加入poll队列里面，此时如果有新的callback，就会再次进入poll队列去检查，然后循环上面的步骤
- 在等待callback加入poll队列空闲的时候，会去检查定时器有没有到时间，如果定时器到时间了又有对应的callback，它就会进入timers定时器阶段
- 如果定时器没有到时间，就会继续等待

代码示例一:

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

function someAsyncOperation(callback) {
```

```
  // Assume this takes 18ms to complete
  fs.readFile('./test.txt','utf-8', callback);
}

const timeoutScheduled = Date.now();

setTimeout(() => {
  const delay = Date.now() - timeoutScheduled;

  console.log(`${delay}ms have passed since I was scheduled`);
}, 100);

// do someAsyncOperation which takes 95 ms to complete
someAsyncOperation(() => {
  const startCallback = Date.now();
  console.log(startCallback -timeoutScheduled)

  // do something that will take 10ms...
  while (Date.now() - startCallback < 100) {
  // do nothing
  }
})
```

代码执行过程:
先调度一个100毫秒的超时的定时器,然后脚本异步读取 test.txt文件.
首先timer阶段调度一个100的定时器,callbacks回调函数是打印出一段时间消耗文字
事件循环进入轮询 poll阶段 等待I/O操作,fs.readFile尚未完成,18ms后文件读取完毕,再执行回调需要118ms,所以总共延迟了118ms,poll队列执行完毕后,检查是否有timer是不是有到时间settimeout,发现settimeout早就到时间了,此时打印出结果,时间一共延迟了118ms.

也就是说，并不是我们设置了定时器多少毫秒之后这个定时器就会在多少毫秒之后执行，它之前的操作可能会阻塞它的执行，这个跟浏览器端的定时器是一样的，都存在这个问题。

代码示例二:

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

fs.readFile(__filename, () => {
```

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

代码二也是一个读文件的操作，事件循环进入poll阶段,poll队列执行,readFile里面传入的异步回调执行一个setTimeout，然后再执行一个setImmediate，根据前面的流程图我们知道，setImmediate设置了callback会直接进入check阶段，如果没有设置回调，才会等待callback加入poll队列，在这个等待的过程中才会检查timer，所以说check阶段会比timers阶段先执行，这里的运行结果就是先打印Immediate，再打印setTimeout。
打印出来的结果是: Immediate setTimeout

注意:上述setImmediate优先于setTimeout的前提是 两个函数是在I/O循环内调用,意思就是poll阶段回调内有setImmediate回调,和setTimeout回调.如果两个函数不在poll回调内,两个计时器的顺序是不确定性的.比如:

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

fs.readFile(__filename, () => {
 console.log(121)
});

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


setImmediate(() => {
 console.log('immediate');
 process.nextTick(()=>{
 console.log('nextTick2');
 });
})

process.nextTick(()=>{
 console.log('nextTick1');
});
```

它结果的先后顺序是　nextTick1　timeout　immediate　nextTick2　121

node.js中有一个process模块，它代表进程的意思，它有一个nextTick方法。

process.nextTick()

- 它是一个异步的node API，但不属于Event Loop阶段，它的作用是调用这个方法时，Event Loop就会停下来，先去执行这个方法的回调。

关于nextTick的代码示例:

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

fs.readFile(__filename, () => {
 process.nextTick(()=>{
 console.log('nextTick2');
 process.nextTick(()=>{
 console.log('nextTick1');
 setImmediate(() => {
 console.log('immediate');

 })

 });
 });
 setTimeout(() => {
 console.log('timeout');
 }, 0);
});
```

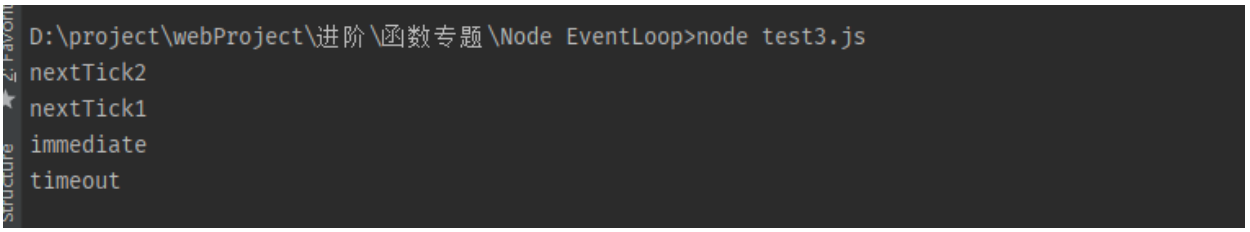
上面的代码也是读文件，读文件的callback中包含一个process和settimeOut ,process里面又包含process和setImmediate.....

执行过程如下：

1. 首先readFile进入poll阶段，将callback加入poll队列
2. poll队列执行readFile的callback
3. 有process,暂停事件循环,打印出nextTick2

4. 然后里面又有process暂停事件循环,打印出nextTick1
5. 接着出现setImmediate,有callback,进入check阶段,打印出immediate
6. 然后再去检查有没有到时间的定时器，这里有一个setTimeout到时间了，进入timers阶段打印出setTimeout

运行代码可以看到结果是符合预期的



课后练习,发表一篇源于EventLoop的文章