JavaScript 事件循环机制（Event Loop）

> 主线程不断从消息队列中获取消息，执行消息，这个过程为事件循环，这种机制叫循环机制，取一次消息并执行的过程叫一次循环。

概念

- **堆（Heap）**
  堆表示一大块非结构化的内存区域，对象、数据被存放在对中。

- **栈（Stack）**
  栈在 JavaScript 中又称执行栈，调用栈，是一种后进先出的数据结构。
  JavaScript 有一个主线程（ main thread ）和调用栈（或执行栈 call-stack ），主线各所有的任务都会被放到调用栈等待主线程执行。

- **队列（Queue）**
  队列及任务队列（ Task Queue ），是一种先进先出的一种数据结构。在队尾中添加新元素，从队头移除元素。

同步任务和异步任务

- **同步任务**是调用立即得到结果，同步任务在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务。

- **异步任务**是调用无法立即得到结果，需要额外的操作才能预期结果的任务，异步任务不进入主线程，而是进入事件任务队列（Event Queue）的任务，只有事件任务队列（Event Queue）通知主线程，某个异步任务可以执行了，该任务才会进入主线程执行。

- 需要理解以下几点：
  - **JavaScript 分为同步任务和异步任务**
  - **同步任务都在主线程上执行，形成一个执行栈**
  - **主线程之外，事件触发线程管理着一个任务队列，只要异步任务（如定时器，http请求）出现，就会在任务队列中放置一个事件**

- **一旦执行栈中的所有同步任务执行完毕（此时JS引擎空闲），系统就读取任务队列，将可运行的异步任务添加到可执行栈中，开始执行**

宏任务（MacroTask）

> 每次执行栈执行的任务就是一个宏任务，这其中也包含每次从任务队列中获取一个事件调用并放入执行栈中执行。

主要包含：
script（整体代码）、setTimeout、setInterval、I/O、UI交互事件、postMessage、MessageChannel、setImmediate（Node.js环境）

注意：浏览器在每执行完一个宏任务之后，下一个宏任务开始执行之前，对页面进行重新的渲染。

微任务（MicroTask）

> 在当前Task执行结束后立即执行的任务。也就是说，在当前Task任务后，下一个Task之前，在渲染之前微任务会被执行

主要包含：
Promise.then、MutationObserver、process.nextTick（Node.js环境）

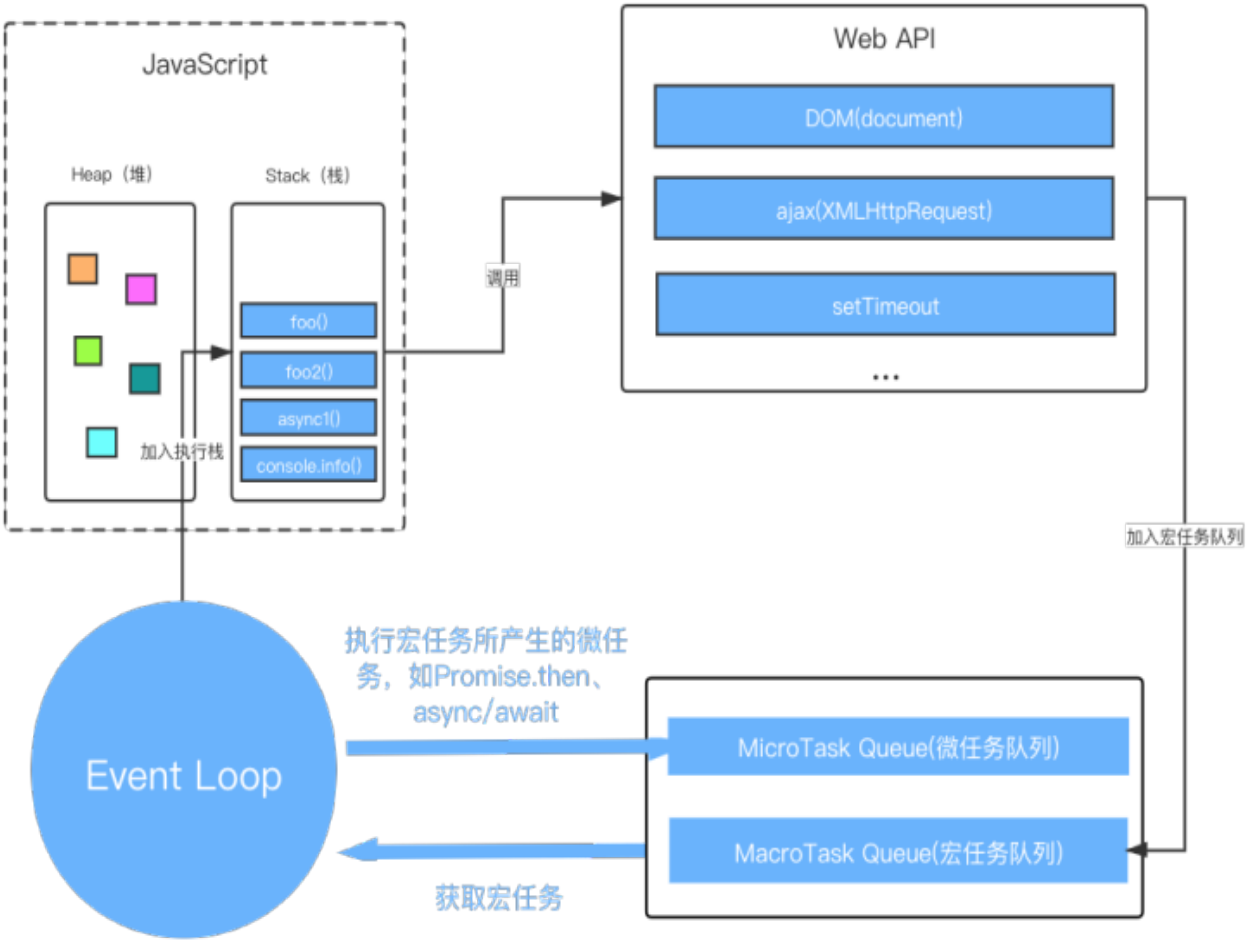
***注意：在某一个 MacroTask 执行完后，就会将在它执行期间产生的所有 MicroTask 都执行完毕（在渲染前）。**

事件循环运行机制

在事件循环中，每进行一次循环操作称为tick，每一次tick的任务执行过程如下：

- **执行一个宏任务，如果执行栈没有就从任务队列中获取**
- **执行过程中产生的微任务，就把它添加到微任务队列中**
- **宏任务执行完毕之后（渲染之前），立即依次执行微任务队列中的所有微任务**
- **当宏任务执行完毕之后，开始检查渲染，然后GUI线程接管渲染**
- **渲染完毕之后，检查是否有 `Web Worker` 任务，有则执行**
- **JS线程继续接管，开始下一个宏任务（从事件队列中获取）**

流程图：



每一个宏任务执行完之后渲染之前，检查清空微任务队列，接着交给渲染线程执行渲染，当渲染线程执行完毕，然后交给JavaScript线程执行下一个宏任务

Promise、async/await、setTimeout在事件循环中的执行

- **Promise/then、catch中的任务执行**

  在Promise的代码会被当作同步任务来执行，在主线程的执行中，一旦轮到它执行就会立即执行。而 `then` 和 `catch` 中的代码被加入到为微任务队列中，等同步任务全部执行完毕，才会被加入执行栈中执行。

  ***注意：** `then` **和** `catch` **中代码被加入到为微任务队列而不是任务队列**

- **async/await中的任务执行**

  async中的代码是被当作同步任务来执行，在主线程的执行中，一旦轮到它执行就会立即执行。

await实际上是一个让出线程的标志，await后面的表达式会先执行一遍，将await后面的代码加入到微任务队列中，然后就会跳出整个async函数来执行后面的代码。

- **setTimeout中的任务执行**
  JavaScript引擎线程在解析JavaScript代码时，遇到setTimeout，浏览器渲染进程会新开一个线程定时器触发线程，此时，setTimeout中的回调函数将加入任务队列里面，等主线执行栈中上一个宏任务执行完毕，在下一轮事件循环时回调函数才从任务队列中被添加到执行栈中执行。

例子分析

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

async1();

new Promise(function(resolve) {
  console.log('promise2');
  resolve();
}).then(function() {
  console.log('promise2 then');
})

console.log('script end');
```

```
33
34 /**
35 输出内容如下：
36 script start
37 async1 start
38 async2
39 promise2
40 script end
41 async1 end
42 promise2 then
43 setTimeout
44 promise1
45 promise1 then
46 **/
```

- 上面是一段JavaScript代码，JavaScript引擎线程开始解析，这是，宏任务队列中只有一个script（整体代码）任务

- 然后我们看到首先定义了两个async函数，接着往下看，然后遇到了 `console` 语句，直接输出 `script start` 。输出之后，script任务继续往下执行，遇到 `setTimeout` ，其作为一个宏任务源，则会先将其任务分发到对应的队列中

- script任务继续往下执行，执行了async1()函数，async函数中在await之前的代码是立即执行的，所以会立即输出 `async1 start` 。

  遇到await时，会将await后面的表达式执行一遍，所以就紧接着输出 `async2` ，然后将await后面的代码（ `console.log('async1 end');` ）添加到微任务队列（MicroTask Queue）中，接着跳出async1函数来执行后面的代码。

- script任务继续往下执行，这时遇到Promise实例。由于Promise中的函数当作同步任务是立即执行的，而后续 `.then` 则会被分发到微任务队列（MicroTask Queue）中去。所以会先输出 `promise2` ，然后执行 `resolve` ，将 `promise2 end` 分配到对应队列。

- script任务继续往下执行，最后只有一句输出 `script end` ，到这里，这个宏任务（script（整体代码））就执行完毕了。

  根据上述，每次执行完一个宏任务（MacroTask）之后，会去检查是否存在微任务（MicroTask）；如果存在，则执行微任务（MicroTask）直至清空微任务队列（MicroTask Queue）。

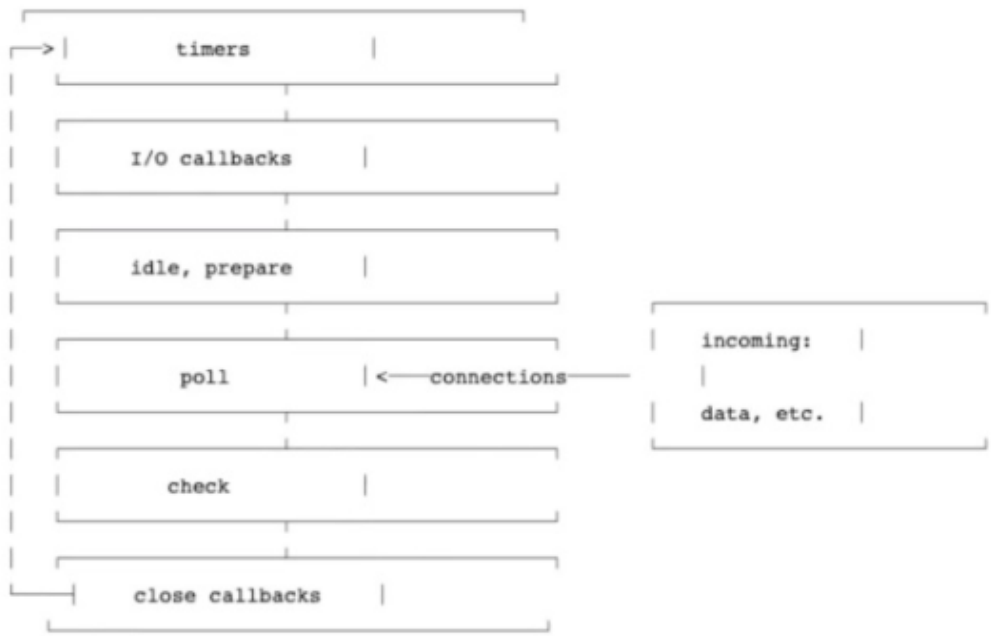
- 在script任务执行完毕之后，开始检查清空微任务队列。此时，微任务队列中有`console.log('async1 end');` 和 `console.log('promise2 then');` ，因此按先后顺序输出 `async1 end` `promise2 then` 。当所有的微任务（MicroTask）执行完毕之后，表示第一轮的循环就结束了。

- 第二轮循环依旧从宏任务（MacroTask）开始。此时宏任务队列中只有一个 `setTimeout` 回调函数内容，立即执行，遇到`console` 语句，直接输出`setTimeout` 。

- 接着往下执行，这时遇到Promise实例，同样由于Promise中的函数当作同步任务是立即执行的，而后续 `.then` 则会被分发到微任务队列（MicroTask Queue）中去。所以会先输出 `promise1` ，然后执行 `resolve` ，将 `promise1 end` 分配到对应队列。

- 这时，第二轮循环的 `setTimeout` 回调函数宏任务（MacroTask）已执行完毕，接着检查清空微任务队列（MicroTask Queue）。此时队列中只有`console.log('promise1 then');` ，按先后顺序直接输出`promise1 then` ，清空完微任务队列（MicroTask Queue）之后，表示第二轮的循环就结束了，至此整个流程结束。

拓展知识 - Node端



该图来自官网，这里展示了在node的事件循环的6个阶段。

- timers:该阶段执行定时器的回调,如setTimeout() 和 setInterval()。
- I/O callbacks:该阶段执行除了close事件，定时器和setImmediate()的回调外的所有回调
- idle, prepare:内部使用
- poll:等待新的I/O事件，node在一些特殊情况下会阻塞在这里

- check: setImmediate()的回调会在这个阶段执行
- close callbacks: 例如socket.on('close', ...)这种close事件的回调

***注意：Node中的6个阶段每个阶段执行完都会伴随着执行微任务,同个MicroTask队列下process.tick()会优于Promise。**