完整圖解Node.js的Event Loop(事件迴圈)

NODE.JS 程式語言

Event Loop 事件迴圈

Andy Wu • 12 月 9, 2020 • 有 12 則留言

最近當完兵在面試Backend Engineer的缺，想把主力放在Node.js上，因此又把一些Node.js的核心觀念拿出來啃了一遍。網路上太多把前端瀏覽器的Event Loop與Node.js的Event Loop混著講的文章了，因此趁記憶猶新時把Node.js的Event Loop記錄一下，釐清一下觀念。

在開始前先做個小測驗，如果你都能答對代表你的觀念很清晰，這篇文章的內容你應該都懂了。如果答錯了就請繼續往下閱讀吧。

請寫出各個 `console.log();` 輸出的順序：

```
1  console.log('start');
2
3  process.nextTick(function() {
4    console.log('nextTick1');
5  });
6
7  setTimeout(function() {
8    console.log('setTimeout');
9  }, 0);
10
11 new Promise(function(resolve, reject) {
12   console.log('promise');
13   resolve('resolve');
14 }).then(function(result) {
15   console.log('promise then');
16 });
17
18 (async function() {
19   console.log('async');
```

```
20  })();
21
22  setImmediate(function() {
23    console.log('setImmediate');
24  });
25
26  process.nextTick(function() {
27    console.log('nextTick2');
28  });
29
30  console.log('end');
```

挑戰一下自己先別往下滑

再往下滑就是解答嘍!!

公佈答案

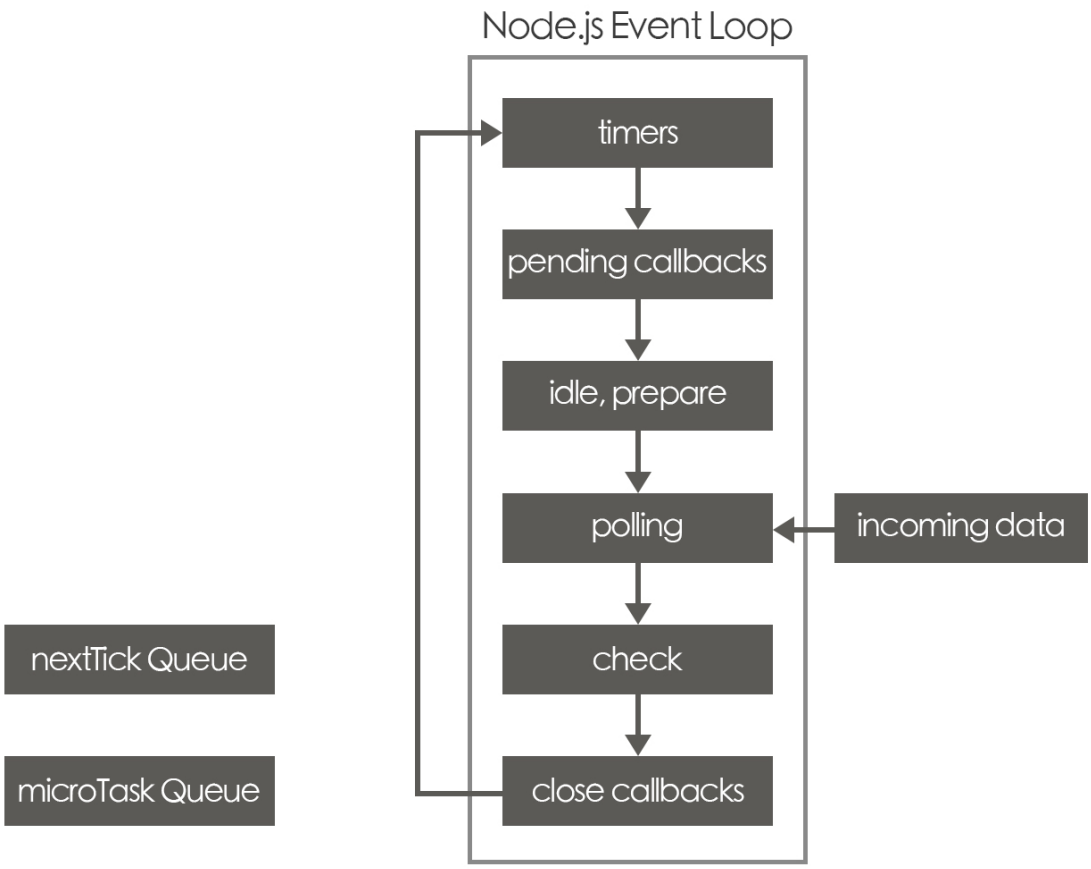
```
start
promise
async
end
nextTick1
nextTick2
promise then
setTimeout
setImmediate
```

你都答對了嗎？

我們都知道Node.js出了名的輕巧高效來自於他只使用單執行緒與Event Loop(事件迴圈)的概念，將凡事有任何需要等待結果的、請求外部資源才能進行的函式，一律丟到Event Loop中等待。

當結果出來或資源載入完畢後，這些callback函式即被觸發成可執行的狀態，等待Node.js的那個單一執行緒有空時來執行他。假設今天Event Loop中有多個可被執行的callback函式存在呢，這就是本篇要介紹Event Loop內部的運作方式啦。看完本篇後保證醍醐灌頂，從此不再搞混了。

不囉唆了，直接上Node.js內部Event Loop的圖。



注意喔，這是"Node.js"的Event Loop喔，跟瀏覽器跑的JS的Event Loop有一點不太一樣。

Node.js內部的Event Loop總共有六個macrotask queue，或簡稱task queue，各個都有他自己不同的用途。當事件被觸發後，事件的callback會丟到對應的task queue。接下來讓我們來一一講解。

本篇的講解順序：

- 六個**macrotask queue (task queue)**
- 另外兩個高優先層級的**queue**
- **Node.js**解讀程式碼的順序
- 沙盤推演

六個macrotask queue (task queue)

Timers

會被丟到這個queue的事件有 `setTimeout()` 跟 `setInterval()` ，當所設定的時間倒數完畢時，計時器的callback會被丟來這裡等待執行。

Pending callbacks

這個queue主要是給作業系統層級使用的，像是傳輸過程中的TCP errors，socket連線收到了 `ECONNREFUSED` ，他的callback就會被丟來這裡。

Idle, prepare

這個queue連官方的文件都是說給內部使用的，並沒有在文件中多做說明，所以這個queue可以先忽略。

Polling

當Node.js的app在運行時有新的I/O資料進來可以被讀取時，例如會用到串流有提供的 `.on('data', callback)` ，這時他的callback就會被放在這個queue等待被執行。

Check

還記得我們有一個也是有關時間的 `setImmediate()` 還沒講到，他的callback就是排在這裡的。

Close callbacks

當我們今天需要關閉連線、檔案等等的操作時，例如 `socket.on('close', callback)` ，只要是有關"關閉"的動作的callback就會來這裡。

以上這六個task queue是在Event Loop中優先層級比較**低**的，另外還有兩個優先層級比較**高**的queue，我們獨立拉出來講。

另外兩個高優先層級的queue

nextTick Queue

這是優先層級第一高的queue，所有給定的 `process.nextTick()` 的callback都會來這裡。只要這個queue有東西，Event Loop就會優先執行這裡的callback，即便timer到期也一樣。

microTask Queue

這是優先層級第二高的queue，當我們在程式中使用的promise狀態有從pending轉變為resolve或reject時，**resolve或reject所執行的callback**會被排在這個queue。注意是**resolve或reject所執行的callback**喔，例如 `Promise.resolve().then(CALLBACK)` 的那個CALLBACK。

promise在創建時本身帶的函式是同步的，不會進Event Loop。
promise在創建時本身帶的函式是同步的，不會進Event Loop。
promise在創建時本身帶的函式是同步的，不會進Event Loop。

很重要所以說三遍，很多人包括我自己常常不小心掉進這個陷阱。

Node.js解讀程式碼的順序

到目前為止，大家應該都熟悉那8個queue了。我們接著來談談Node.js是依照什麼邏輯順序執行程式碼以及選擇要先執行哪個queue裡面的callback的。

- 首先把整份程式碼跑一遍，遇到同步函式(sync function)當面解決掉，就好像在跑Python、PHP、Java、C那樣。

- 遇到異步函式(async function)就註冊事件，比如說 `setTimeout()` 就會註冊Timer。
- 當整份程式碼掃完一遍後，Node.js就會開始去Event Loop循環了。只要還有註冊的事件的callback尚未被觸發，Node.js就會一直循環，一直循環，一直循環...
- 循環至某個queue時，發現有事件的callback可以被執行了，就把那個callback解決掉。
- 如果在循環的過程中發現發現優先層級第一高的nextTick Queue有東西可以執行了，優先將那個queue的callback清掉。
- 優先層級第二高的microTask Queue也是如此，直到這兩個優先層級比較高的queue都沒有東西可以執行了，再返回原本離開Event Loop的地方繼續循環。

以上就是Node.js核心的執行邏輯，只要照著上面的流程跑，以後再看到有關執行順序的問題都只是清粥小菜了。

沙盤推演

接下來我們拿最一開始的程式碼來模擬看看吧。

第1行是同步函式，當場KO掉。

```
1  console.log('start');
2
3  process.nextTick(function() {
4    console.log('nextTick1');
5  });
6
7  setTimeout(function() {
8    console.log('setTimeout');
9  }, 0);
10
11 new Promise(function(resolve, reject) {
12   console.log('promise');
13   resolve('resolve');
14 }).then(function(result) {
15   console.log('promise then');
```

```
16  });
17
18  (async function() {
19    console.log('async');
20  })();
21
22  setImmediate(function() {
23    console.log('setImmediate');
24  });
25
26  process.nextTick(function() {
27    console.log('nextTick2');
28  });
29
30  console.log('end');
```

我的output窗

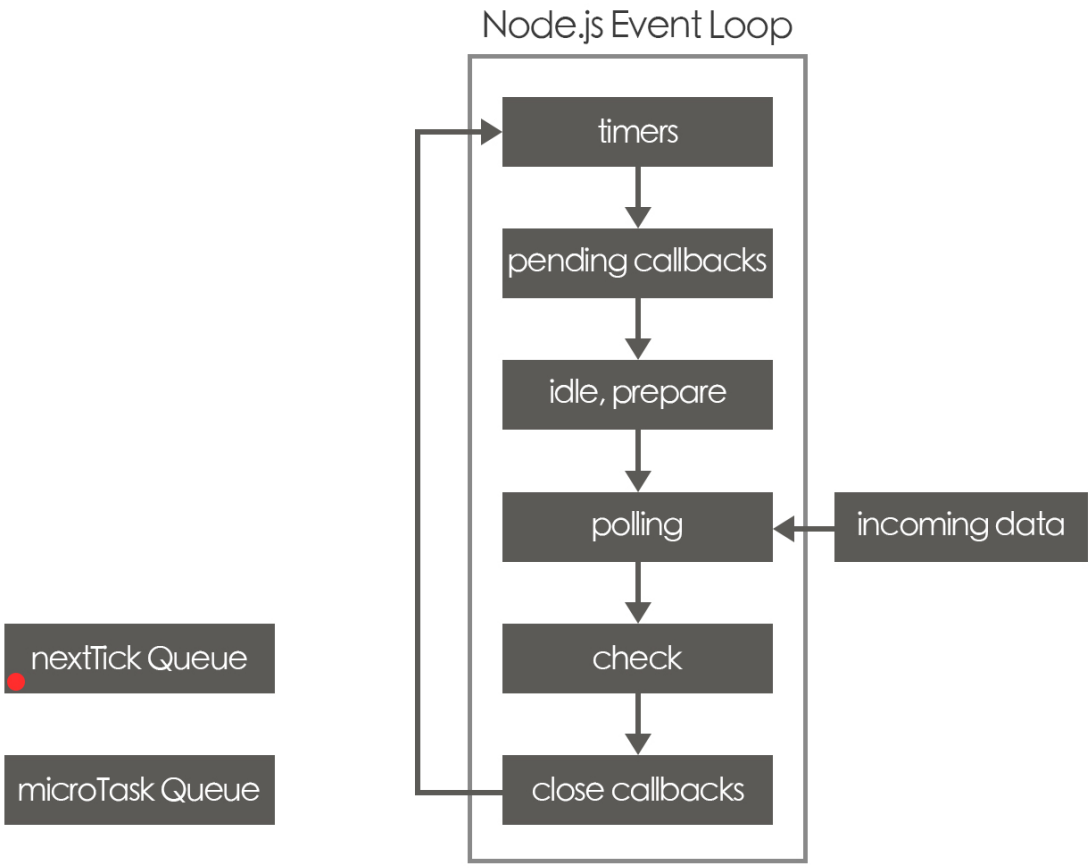
```
start
```

第3-5行遇到了 `nextTick()` ，將他的callback排至最高優先層級的nextTick Queue。此時，雖然最高優先層級的queue有東西了，但是我們的整份程式碼還沒掃完，還不能確保之後還有沒有同步函式，因此該callback還不會被Node.js執行。

```
1   console.log('start');
2
3   process.nextTick(function() {
4     console.log('nextTick1');
5   });
6
7   setTimeout(function() {
8     console.log('setTimeout');
9   }, 0);
10
11  new Promise(function(resolve, reject) {
12    console.log('promise');
13    resolve('resolve');
14  }).then(function(result) {
15    console.log('promise then');
16  });
17
18  (async function() {
19    console.log('async');
20  })();
21
```

```
22  setImmediate(function() {
23    console.log('setImmediate');
24  });
25
26  process.nextTick(function() {
27    console.log('nextTick2');
28  });
29
30  console.log('end');
```

(丟進queue內的callback以紅色圓圈圈表示)

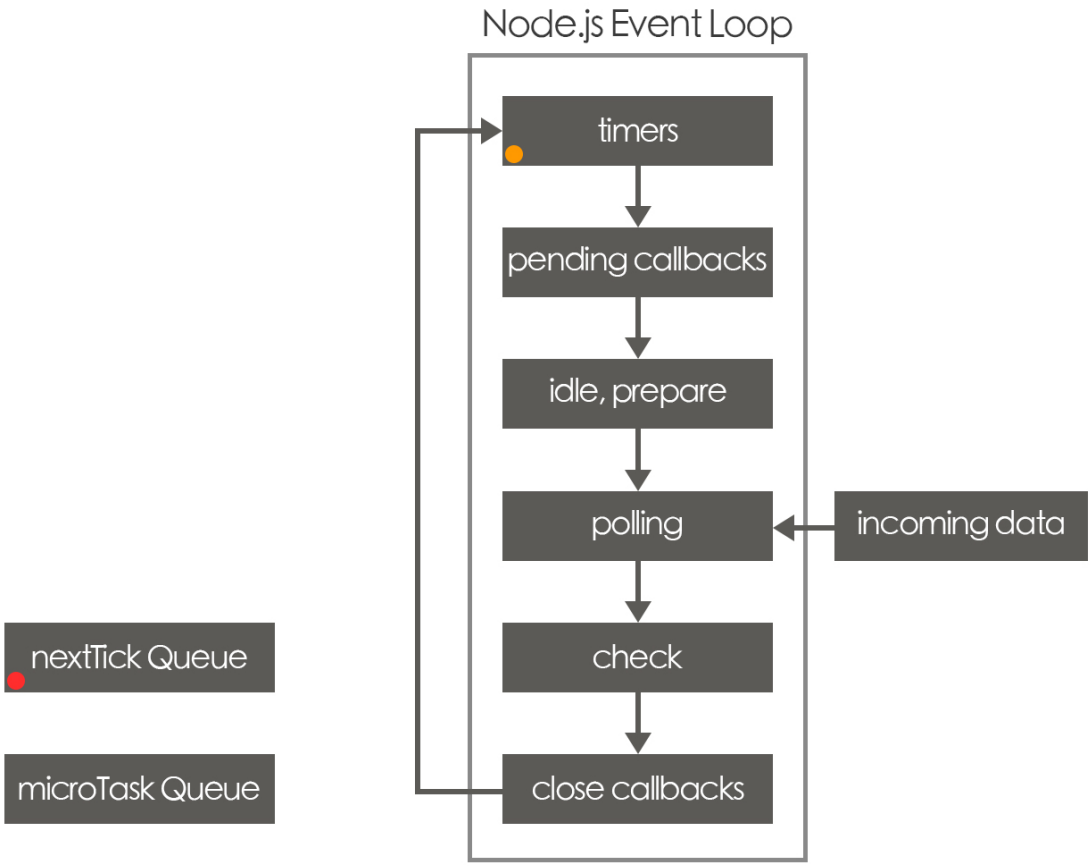


再往下走到第7-9行遇到了 `setTimeout()` ，將Timer倒數計時並註冊他的callback。

```
1  console.log('start');
2
3  process.nextTick(function() {
4    console.log('nextTick1');
5  });
6
```

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

new Promise(function(resolve, reject) {
  console.log('promise');
  resolve('resolve');
}).then(function(result) {
  console.log('promise then');
});

(async function() {
  console.log('async');
})();

setImmediate(function() {
  console.log('setImmediate');
});

process.nextTick(function() {
  console.log('nextTick2');
});

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

(callback以橘色圓圈圈表示)



下一步第11-13行創建了一個新的promise，還記得上面說的嗎：

promise在創建時本身帶的函式是同步的，不會進Event Loop。

或者是換個角度想，那個創建時帶的函式其實就是類似promise的建構子，負責告訴Node.js我這個promise什麼情況會reslove，什麼情況會reject。因此在創建這個promise物件時的初始化肯定是同步的呀，不然這個物件要等到哪時才會被建立。

如果上面的解釋方法有說到你的話，第11-13行你肯定不會再錯了，是個同步函式，會被當場解決掉。

```
1  console.log('start');
2
3  process.nextTick(function() {
4    console.log('nextTick1');
5  });
6
7  setTimeout(function() {
8    console.log('setTimeout');
9  }, 0);
```

```
10
11  new Promise(function(resolve, reject) {
12    console.log('promise');
13    resolve('resolve');
14  }).then(function(result) {
15    console.log('promise then');
16  });
17
18  (async function() {
19    console.log('async');
20  })();
21
22  setImmediate(function() {
23    console.log('setImmediate');
24  });
25
26  process.nextTick(function() {
27    console.log('nextTick2');
28  });
29
30  console.log('end');
```

我的output窗

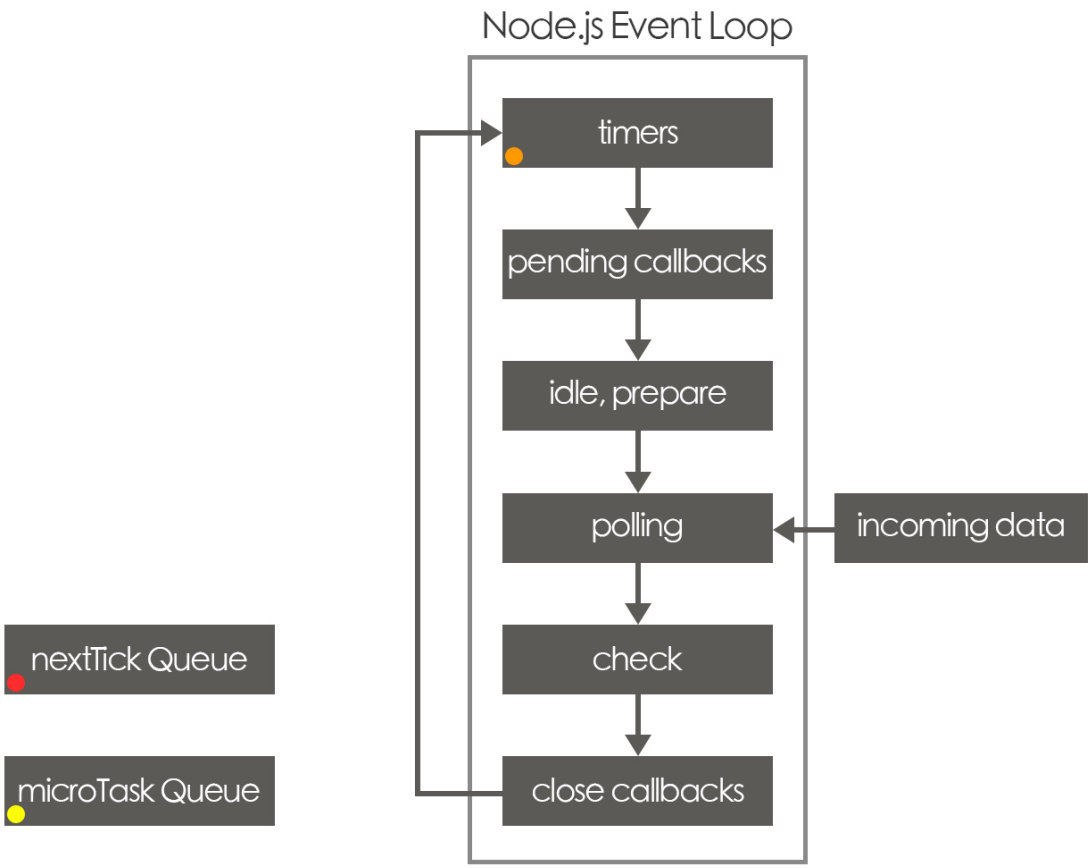
```
start
promise
```

至於14-16行呢，他的執行時機是依賴於promise的結果，也就是promise的狀態從pending轉變為resolve後。因此是個異步函式，丟進優先層級第二高的microTask Queue。

```
1   console.log('start');
2
3   process.nextTick(function() {
4     console.log('nextTick1');
5   });
6
7   setTimeout(function() {
8     console.log('setTimeout');
9   }, 0);
10
11  new Promise(function(resolve, reject) {
12    console.log('promise');
13    resolve('resolve');
14  }).then(function(result) {
15    console.log('promise then');
```

```
16  });
17
18  (async function() {
19    console.log('async');
20  })();
21
22  setImmediate(function() {
23    console.log('setImmediate');
24  });
25
26  process.nextTick(function() {
27    console.log('nextTick2');
28  });
29
30  console.log('end');
```

(callback以黃色圓圈圈表示)



繼續往下，我們遇到了一個很詭異的async function。

```
1  console.log('start');
2
```

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

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

new Promise(function(resolve, reject) {
  console.log('promise');
  resolve('resolve');
}).then(function(result) {
  console.log('promise then');
});

(async function() {
  console.log('async');
})();

setImmediate(function() {
  console.log('setImmediate');
});

process.nextTick(function() {
  console.log('nextTick2');
});

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

根據JS的文件說明，如果在function前面加async關鍵字宣告的異步函式，在執行時會被自動轉換成promise。

> Async functions always return a promise. If the return value of an async function is not explicitly a promise, it will be implicitly wrapped in a promise.

也就是說

```
(async function() {
  console.log('async');
})();
```

實際上會被轉換成

```
(function() {
  return new Promise(function(resolve, reject) {
```

```
    console.log('async');
    resolve();
  });
})();
```

於是又回到了11-13行的概念，只是這次是會在同一行被call的匿名函式中創建了一個新的promise，並且跟剛剛一樣創建promise時帶的函式是同步的。

我的output窗

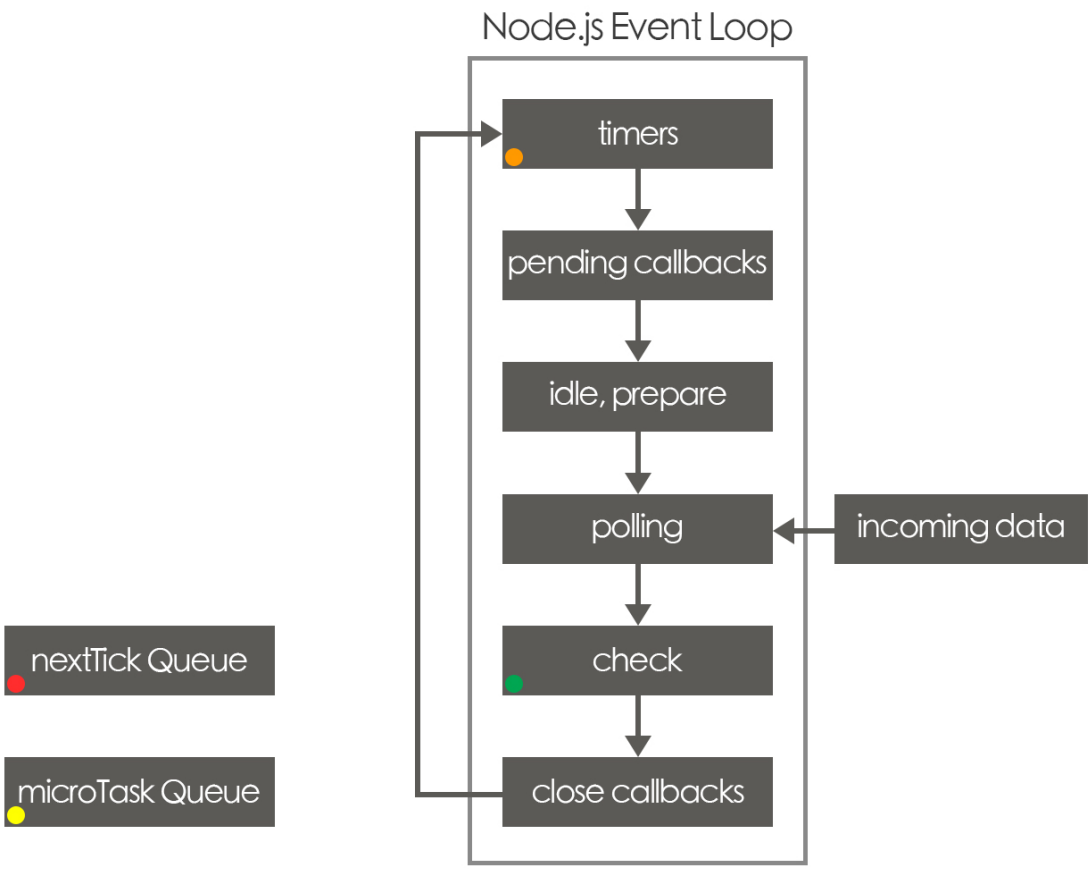
```
start
promise
async
```

大家應該都有跟上，那我們繼續往下走吧。第22-24行遇到了 `setImmediate()` ，這個很單純，直接把callback往Event Loop的check queue丟就好。

```
console.log('start');

process.nextTick(function() {
  console.log('nextTick1');
});

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

new Promise(function(resolve, reject) {
  console.log('promise');
  resolve('resolve');
}).then(function(result) {
  console.log('promise then');
});

(async function() {
  console.log('async');
})();

setImmediate(function() {
  console.log('setImmediate');
});

process.nextTick(function() {
  console.log('nextTick2');
});
```

```
29
30  console.log('end');
```

(callback以綠色圓圈圈表示)

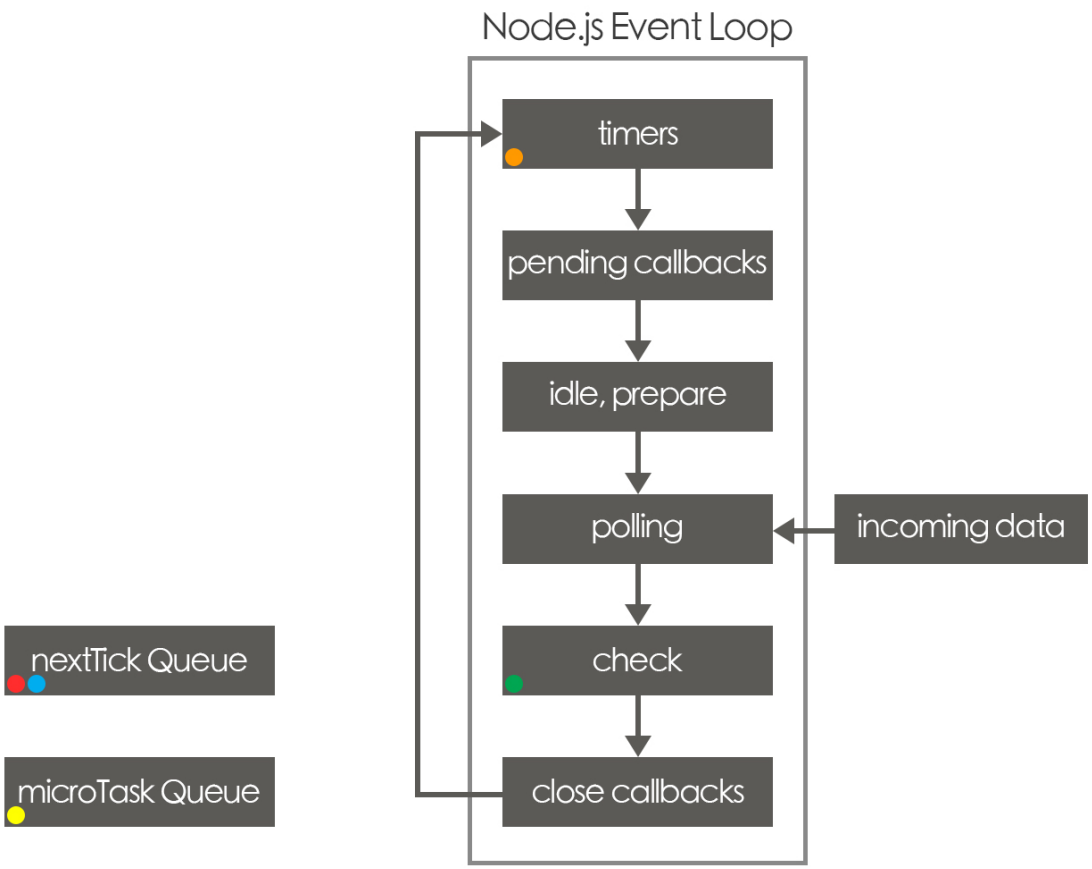


再往下，又是一個 `nextTick()` 。老樣子我們將他的callback排至最高優先層級的nextTick Queue。

```
1   console.log('start');
2
3   process.nextTick(function() {
4     console.log('nextTick1');
5   });
6
7   setTimeout(function() {
8     console.log('setTimeout');
9   }, 0);
10
11  new Promise(function(resolve, reject) {
12    console.log('promise');
13    resolve('resolve');
```

```
14  }).then(function(result) {
15    console.log('promise then');
16  });
17
18  (async function() {
19    console.log('async');
20  })();
21
22  setImmediate(function() {
23    console.log('setImmediate');
24  });
25
26  process.nextTick(function() {
27    console.log('nextTick2');
28  });
29
30  console.log('end');
```

(callback以藍色圓圈圈表示)



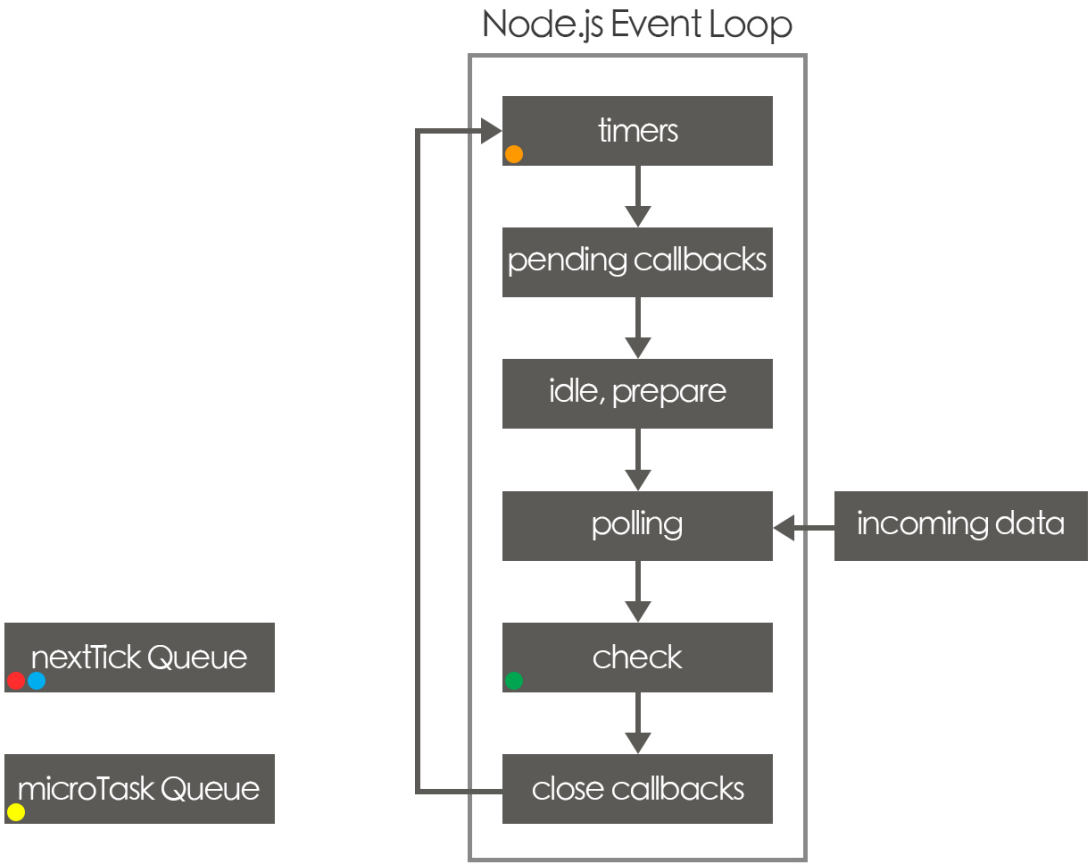
YA! 來到了最後一行是 `console.log()` 同步函式，想都不用想當場KO掉。

```
console.log('start');

process.nextTick(function() {
  console.log('nextTick1');
});

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

new Promise(function(resolve, reject) {
  console.log('promise');
  resolve('resolve');
}).then(function(result) {
  console.log('promise then');
});

(async function() {
  console.log('async');
})();

setImmediate(function() {
  console.log('setImmediate');
});

process.nextTick(function() {
  console.log('nextTick2');
});

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

我的output窗

```
start
promise
async
end
```

到目前為止我們終於把整份程式碼掃過一遍了，不過Node.js並不會因此就在這邊終止程式。還記得我們註冊的事件callback都還沒處理吧，我們現在要做的就是把那些積在queue裡的callback清空。



先從優先序最高的nextTick Queue開始清，把紅色圓圈圈那個callback取出來執行。

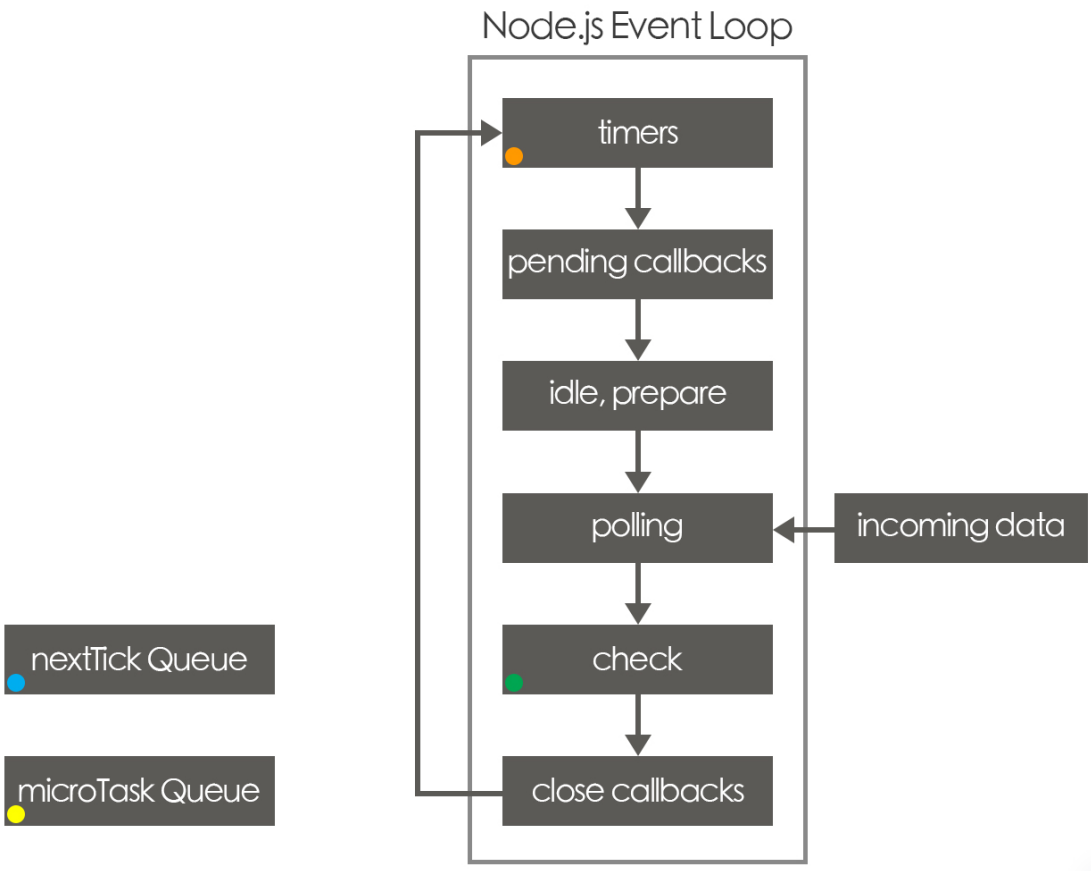
```
console.log('start');

process.nextTick(function() {
  console.log('nextTick1');
});

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

new Promise(function(resolve, reject) {
  console.log('promise');
  resolve('resolve');
}).then(function(result) {
  console.log('promise then');
});

(async function() {
  console.log('async');
})();

setImmediate(function() {
  console.log('setImmediate');
```

```
24  });
25
26  process.nextTick(function() {
27    console.log('nextTick2');
28  });
29
30  console.log('end');
```

我的output窗

```
start
promise
async
end
nextTick1
```

執行完後的8個queue的狀態



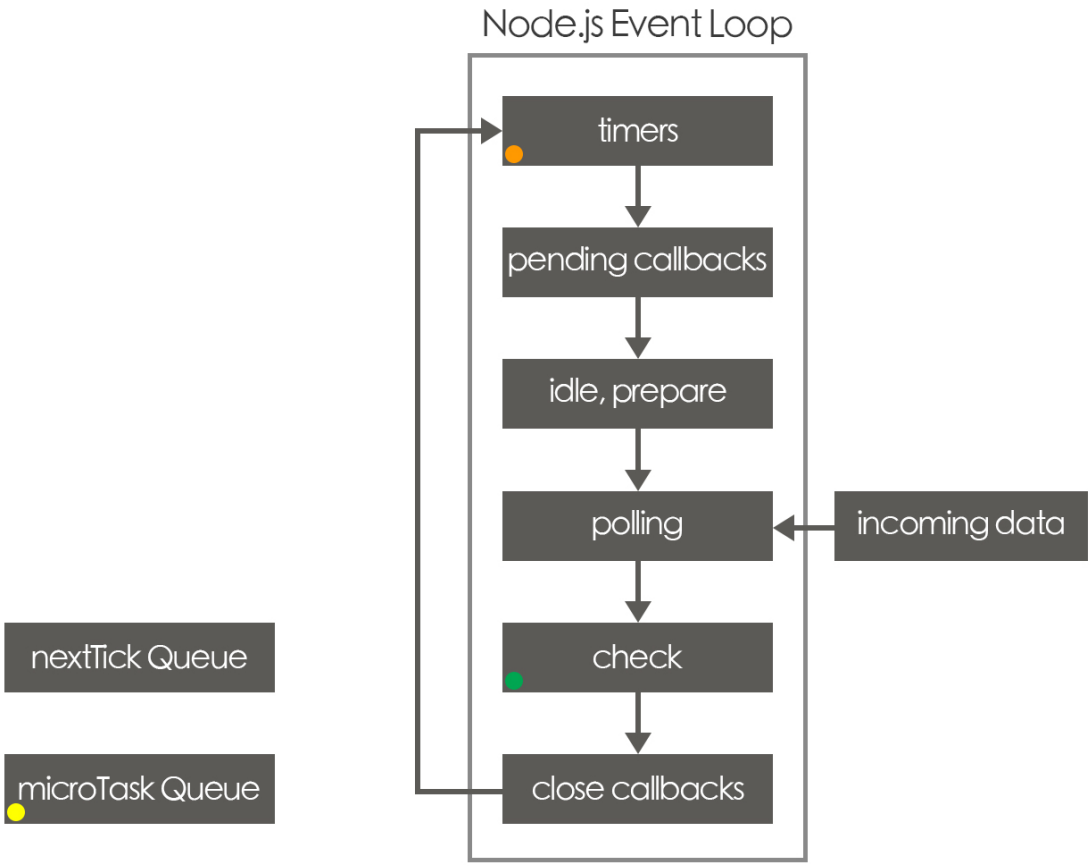
nextTick Queue還有東西，繼續取。

```
1  console.log('start');
2
3  process.nextTick(function() {
4    console.log('nextTick1');
5  });
6
7  setTimeout(function() {
8    console.log('setTimeout');
9  }, 0);
10
11 new Promise(function(resolve, reject) {
12   console.log('promise');
13   resolve('resolve');
14 }).then(function(result) {
15   console.log('promise then');
16 });
17
18 (async function() {
19   console.log('async');
20 })();
21
22 setImmediate(function() {
23   console.log('setImmediate');
24 });
25
26 process.nextTick(function() {
27   console.log('nextTick2');
28 });
29
30 console.log('end');
```

我的output窗

```
start
promise
async
end
nextTick1
nextTick2
```

那8個queue的狀態



好了，nextTick Queue算是清空了。接著換優先層級第二高的microTask Queue。還記得在最一開始在創立promise物件時已經有踩過他的 `resolve()` ，因此被堆在nextTick Queue裡面的callback已經不是pending而是resolve的狀態了，可以執行。

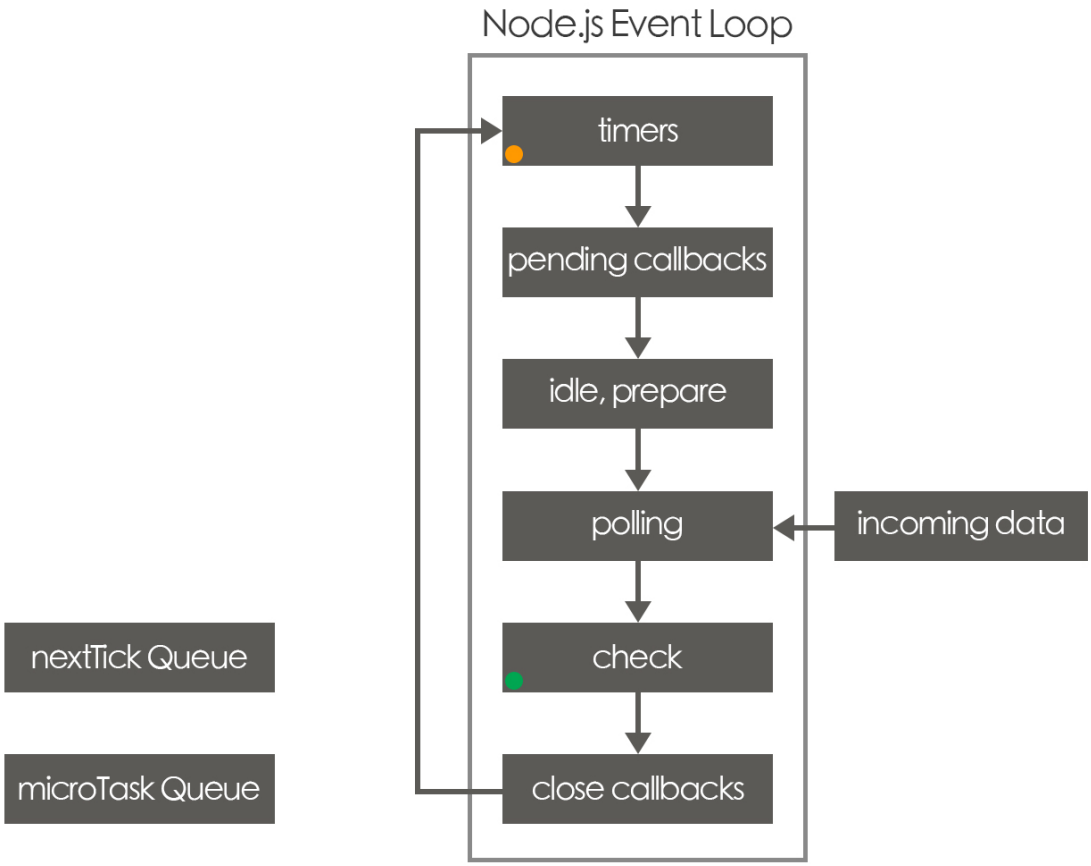
```
console.log('start');

process.nextTick(function() {
  console.log('nextTick1');
});

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

new Promise(function(resolve, reject) {
  console.log('promise');
  resolve('resolve');
}).then(function(result) {
  console.log('promise then');
});

(async function() {
  console.log('async');
})();
```

```
21
22  setImmediate(function() {
23    console.log('setImmediate');
24  });
25
26  process.nextTick(function() {
27    console.log('nextTick2');
28  });
29
30  console.log('end');
```

我的output窗

```
start
promise
async
end
nextTick1
nextTick2
promise then
```

有優先權的queue都空了，換Event Loop內部。



還記得我們曾經設定了倒數計時0毫秒的計時器吧，他已經倒數完callback可以被執行了。

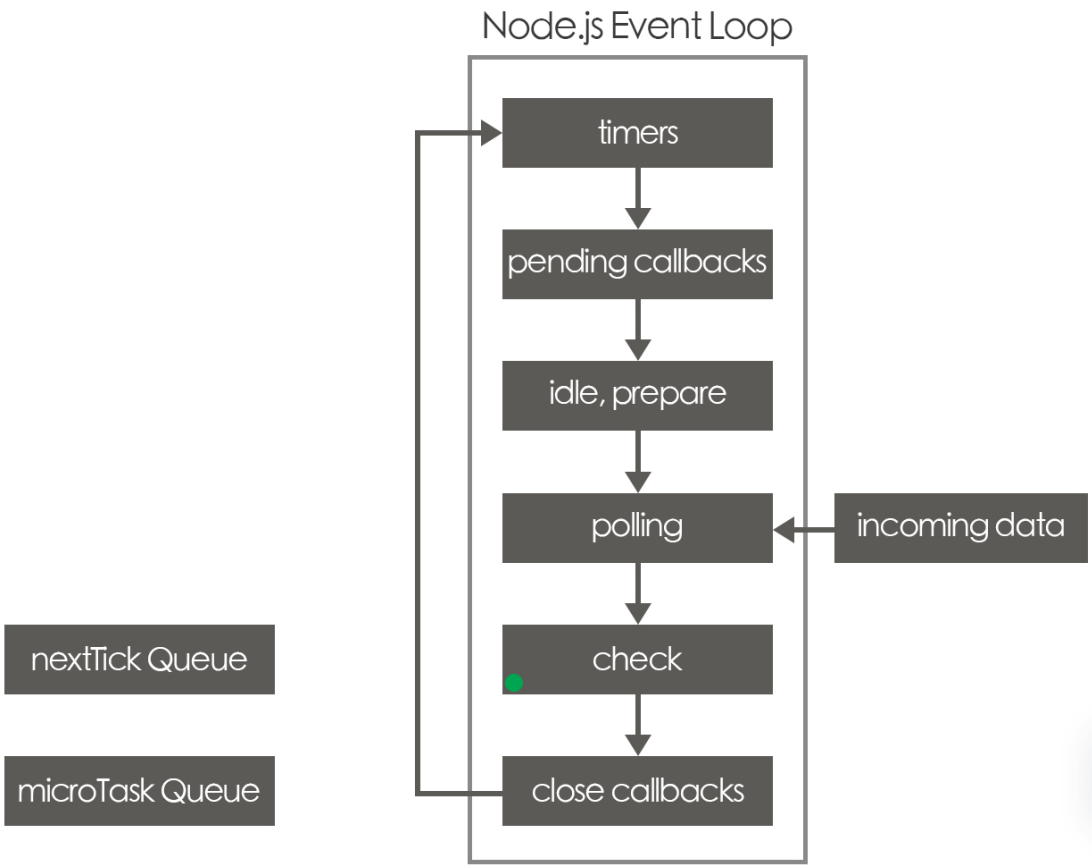
```
console.log('start');

process.nextTick(function() {
  console.log('nextTick1');
});

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

new Promise(function(resolve, reject) {
  console.log('promise');
  resolve('resolve');
}).then(function(result) {
  console.log('promise then');
});

(async function() {
  console.log('async');
})();

setImmediate(function() {
  console.log('setImmediate');
```

```
24  });
25
26  process.nextTick(function() {
27    console.log('nextTick2');
28  });
29
30  console.log('end');
```

我的output窗

```
start
promise
async
end
nextTick1
nextTick2
promise then
setTimeout
```

Event Loop只剩下最後一個callback了。



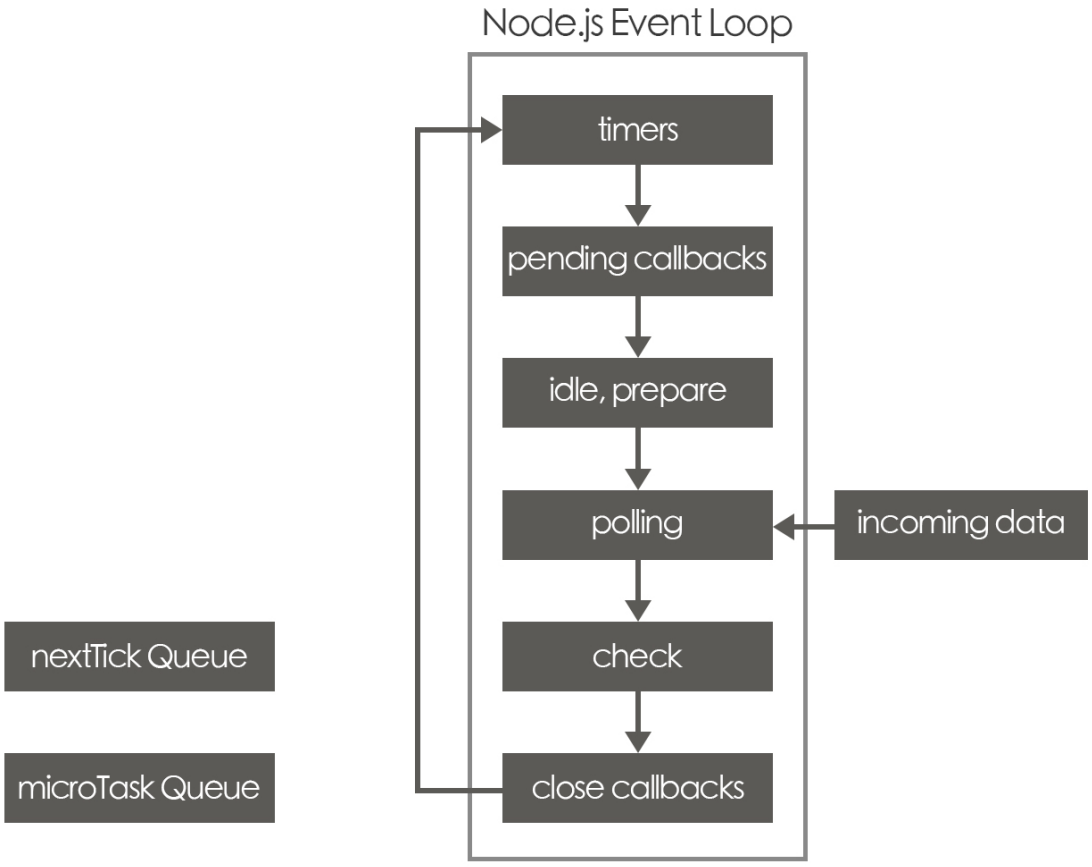
剩下 `setImmediate()` 所註冊的callback，拔出來執行掉。

```
1  console.log('start');
2
3  process.nextTick(function() {
4    console.log('nextTick1');
5  });
6
7  setTimeout(function() {
8    console.log('setTimeout');
9  }, 0);
10
11 new Promise(function(resolve, reject) {
12   console.log('promise');
13   resolve('resolve');
14 }).then(function(result) {
15   console.log('promise then');
16 });
17
18 (async function() {
19   console.log('async');
20 })();
21
22 setImmediate(function() {
23   console.log('setImmediate');
24 });
25
26 process.nextTick(function() {
27   console.log('nextTick2');
28 });
29
30 console.log('end');
```

我的output窗

```
start
promise
async
end
nextTick1
nextTick2
promise then
setTimeout
setImmediate
```

以上輸出就是本篇最一開始那個問題的答案。現在Event Loop也都清空了，程式正常終止。



現在是不是對Node.js異步函式的運作步驟有更清楚的認識了呢。

只要清楚掌握了那2+6個queue，以後再看到問執行順序的問題就再也不用擔心了：)

← 白話文講解支持向量機(二) 非線性SVM

[Git] commit至未來或過去的特定時間 →

12 Comments

ZHIH
2021-04-0122:05:16

簡單明瞭的方式解釋event loop，受益良多，感恩