

JAVASCRIPT

Riccardo Cattaneo

Lezione 8



Gestione processi asincroni

Per capire come vengono gestite le chiamate asincrone e che cosa sono, facciamo un esempio partendo dalle nozioni base. Andiamo a creare 4 funzioni : a, b, c, d. Queste funzioni vengono eseguite all'interno di un'unica funzione calcolo() in questo modo :

```
1  function a(){
2      |   console.log("funzione A");
3  }
4
5  function b(){
6      |   console.log("funzione B");
7  }
8
9  function c(){
10     |   console.log("funzione C");
11 }
12
13 function d(){
14     |   console.log("funzione D");
15 }
16
17 function calcolo(){
18     |   a();
19     |   b();
20     |   c();
21     |   d();
22 }
23
24 calcolo();
```

Se proviamo ad eseguire la funzione calcolo, notiamo che le funzioni vengono eseguite in modo «sincrono», in quanto **javascript esegue le funzioni in modo «sincrono»**.

```
D:\Google Drive\Corsi\Javascript\workspace>node promise.js  
funzione A  
funzione B  
funzione C  
funzione D
```

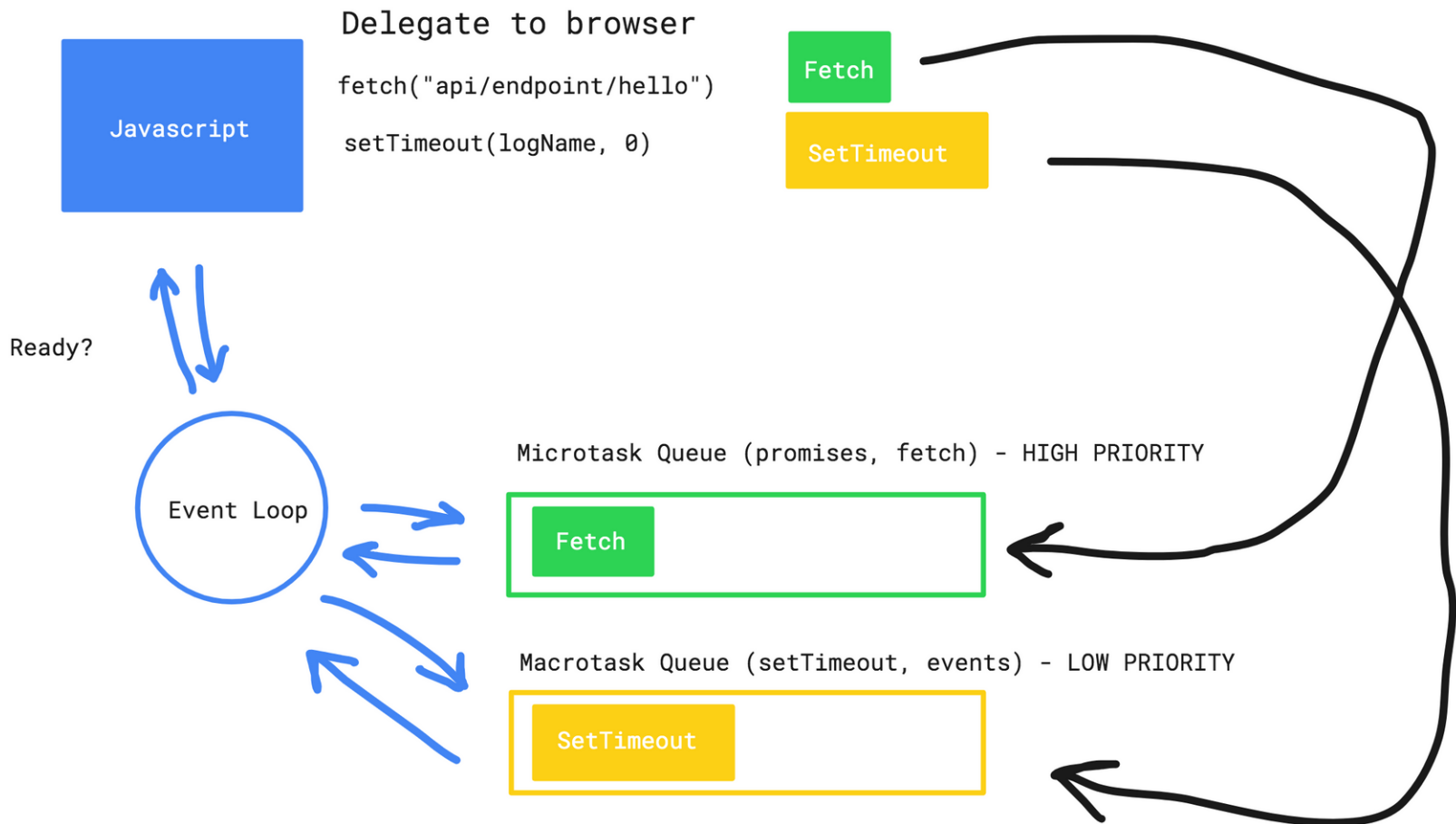
Il problema nasce quando provo ad eseguire funzioni o API specifiche **dell'ambiente** (e non di javascript) **in cui si trova** (browser o Node.js).

setTimeout , fetch e DOM sono tutti esempi di API Web. Sono strumenti **integrati nel browser** e resi disponibili quando il nostro codice viene eseguito. E poiché eseguiamo sempre JavaScript in un ambiente, sembra che questi facciano parte del linguaggio. **Ma non lo sono.**

L'ambiente si fa carico del lavoro

E' quindi l'ambiente che si fa carico del lavoro, e il modo per far sì che l'ambiente faccia quel lavoro, è usare la funzionalità che appartiene all'ambiente. Ad esempio `fetch` o `setTimeout` nell'ambiente del browser.

Ci sono regole rigide relative al momento in cui JavaScript può ricevere il risultato del lavoro delegato. Tali regole sono dettate dal ciclo degli eventi in questo modo :



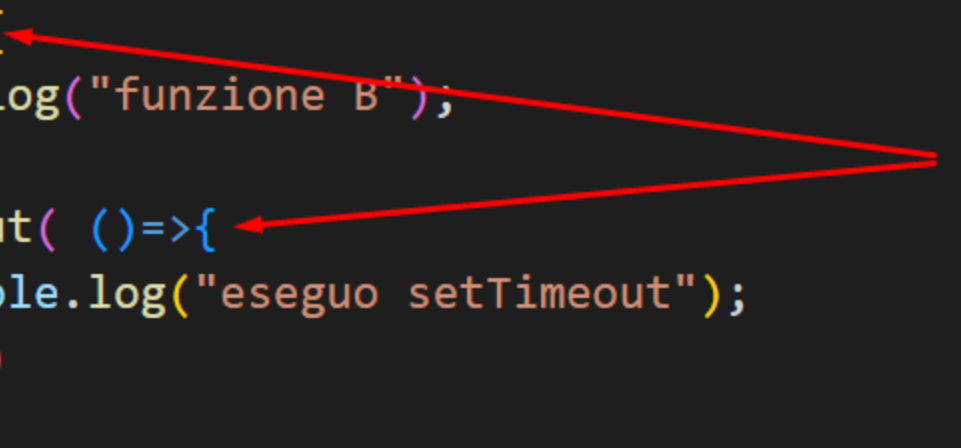
Quando deleghiamo codice asincrono al browser, il browser prende ed esegue il codice e si assume quel carico di lavoro. Ma potrebbero esserci più attività che vengono assegnate al browser, quindi dobbiamo assicurarci di poter dare priorità a queste attività.

È qui che entrano in gioco la coda del **microtask** (priorità più alta) e la coda del **macrotask** (priorità più bassa). Il browser prenderà il lavoro, lo farà, quindi collocherà il risultato in una delle due code in base al tipo di lavoro che riceve.

Ora, una volta che il lavoro è terminato e viene posizionato in una delle due code, il ciclo di eventi verrà eseguito avanti e indietro e verificherà se JavaScript è pronto o meno per ricevere i risultati.

Solo quando JavaScript ha finito di eseguire tutto il suo codice sincro ed è pronto e funzionante, il ciclo di eventi inizierà a prelevare dalle code e restituire le funzioni a JavaScript per l'esecuzione.

```
5  function b(){  
6      console.log("funzione B");  
7  
8      setTimeout( ()=>{  
9          console.log("eseguo setTimeout");  
10     } , 3000)  
11 }
```



Se proviamo ora ad eseguire la nostra funzione calcolo() possiamo notare come javascript esegue sempre in modo sincrono le funzioni a,b,c e d, ma quando esegue la funzione b() fa partire anche la chiamata alla funzione setTimeout() che però è asincrona (del browser), questo vuol dire che javascript non si ferma ad aspettare la sua esecuzione ma prosegue :

```
D:\Google Drive\Corsi\Javascript\workspace>node promise.js
funzione A
funzione B
funzione C
funzione D
eseguo setTimeout
```

Il problema nella maggior parte delle applicazioni è che il codice «deve» aspettare l'esecuzione della funzione «asincrona» perché il dato tornato dalla funzione è necessaria per eseguire poi la funzione c() e d().