JavaScript

Event loop - Promises - Async/Await

Giovanni Iovino: giovanni.iovino@intecs.it

Contenuto:

Come viene eseguito il codice javascript, modello di parallelismo e come scrivere codice non bloccante.

Linguaggio interpretato

runtime environment

- Normalmente utilizzato come linguaggio di scripting all'interno di altri applicativi
  - Il programma ospite (es. browser) offre il runtime environment in cui è eseguito il codice JS offrendo i metodi per interagire con l'environment (es. accesso alle risorse di sistema)
- Per essere eseguito il codice js richiede un interprete o engine (es. V8 sviluppato da Google e utilizzato all'interno di Chrome)
- Node.js utilizza lo stesso motore V8 di chrome utilizzandolo fuori dal browser
- Altri esempi di motori js sono:
  - Spidermonkey -Firefox
  - JavaScriptCore - Safari

Single threaded

at least the Main Thread

- Ogni applicazione js viene eseguita in un processo separato (es. nei browser uno per ogni pagina)
- Il codice che scriviamo viene eseguito in un singolo thread (Main thread)

"Significa che tutto il codice eseguito è single-threaded?
Non necessariamente. Ad esempio alcuni moduli nativi di Node.js possono essere eseguiti automaticamente in thread separati ..."

(per saperne di più https://github.com/libuv/libuv)

Single threaded

at least the Main Thread

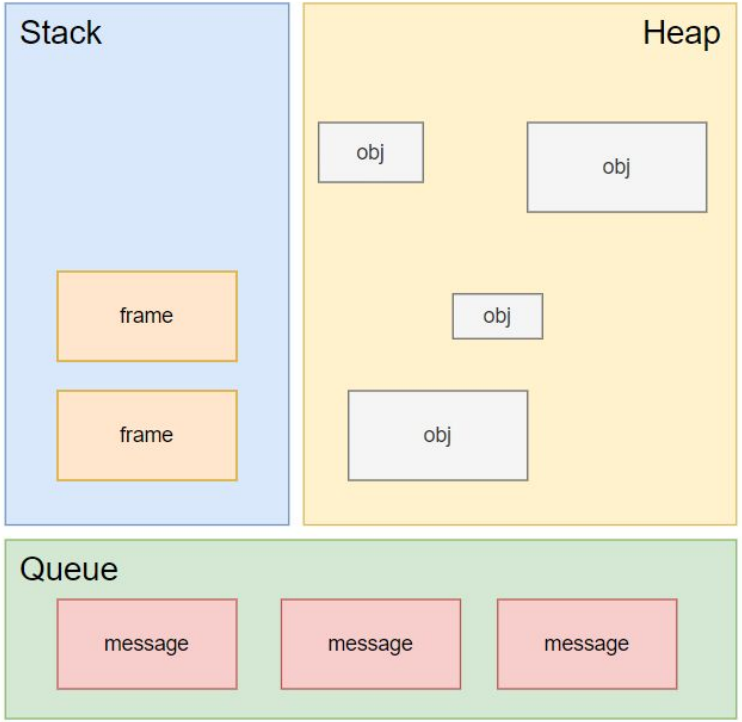
- Nonostante il codice che scriviamo venga eseguito in un unico thread, L'interprete Javascript implementa meccanismi che permettono di eseguire operazioni di I/O non bloccanti.

Normalmente in Javascript, la scrittura di codice bloccante è l'eccezione e non il contrario.

Modello di esecuzione del codice JS

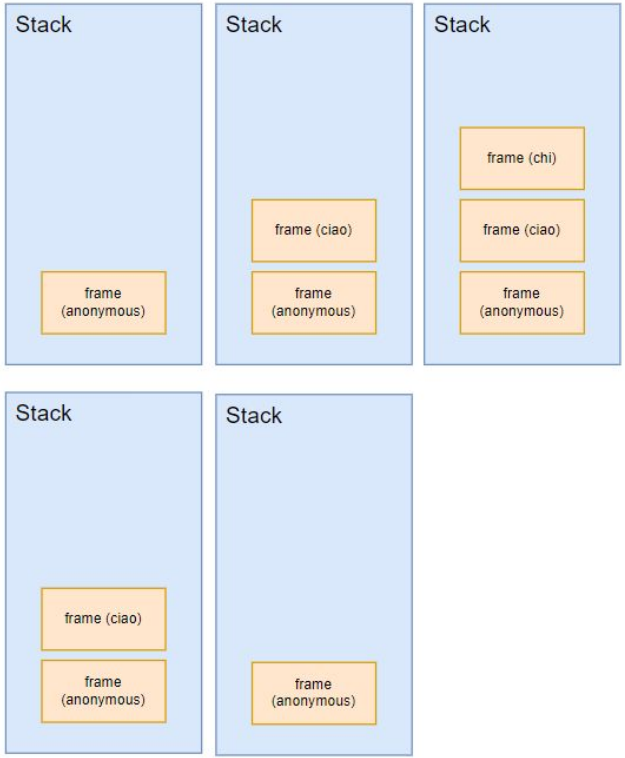
Il modello teorico di esecuzione, implementato e ottimizzato dai diversi motori Javascript, è composto da:

- **Stack**: di tipo LIFO, ogni volta che si chiama una funzione viene creato un **frame** che contiene il valore delle variabili che utilizza la funzione (context), gli argomenti passati alla funzione e altre informazioni come puntatori (es. next frame) e il valore da restituire
- **Heap**: Memoria senza una struttura precisa dove vengono allocati gli oggetti. Javascript utilizza un **Garbage Collector** per la gestione della memoria.
- **Message Queue**: Lista di messaggi da processare utilizzata dall'**Event Loop**. Ad ogni messaggio è associata una funzione che sarà invocata quando il messaggio viene processato. Utilizzato per implementare il paradigma di programmazione orientato agli eventi.



Modello di esecuzione del codice JS: Stack

```
var b = 10;
function ciao() {
  console.log(1)
  console.trace();
  let nome = chi();
  console.log(3);
  console.trace();
  console.log("ciao " + nome);
}

function chi() {
  console.log(2);
  console.trace();
  return "mondo";
}

console.trace();
ciao();
console.trace():
```



Modello di esecuzione del codice JS: Stack

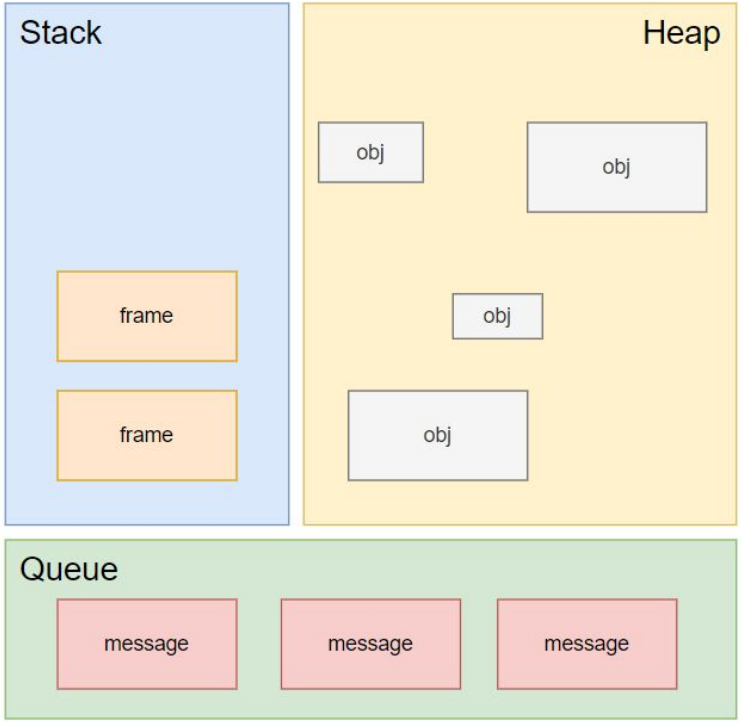
- Quando un funzione viene eseguita non è rilasciabile (non-preemptive) fino alla fine della sua esecuzione.
  Ovvero il modello di esecuzione dei task è “**run-to-completition**”. Il task in esecuzione è sempre portato a termine prima che venga eseguito il task successivo.
- Una funzione che impiega troppo tempo ad essere eseguita blocca l’esecuzione di tutto il codice Javascript. Ad esempio in una pagina web non sarà possibile interagire con gli elementi della pagina o scrollare la pagina

```
function infinite(){
 for(let i = 0; i>0; i++){
    //blocco del codice Javascript
 }
}

infinite();

//questo codice non verrà mai eseguito
let a = 10;
a++;
```

EventLoop

I messaggi vengono aggiunti alla coda ogni volta che avviene un evento e una callback è associata all'evento. Esempi di eventi sono:

- il click su un elemento di una pagina
- lo scadere di un timer
- la risposta di un'operazione di I/O (es. operazione http)



EventLoop: setTimeout/setInterval

- setTimeout: esegue una funzione allo scadere di un timer
  sintassi: **var timeoutID = setTimeout(function[, delay, arg1, arg2, ...]);**

  timoutID può essere utilizzato per cancellare il timer prima che scada utilizzando la funzione **clearTimeout(id)**

- setInterval: esegue una funzione allo scadere di un timer periodicamente.
  sintassi: **var intervalID = setInterval(function[, delay, arg1, arg2, ...]);**

  intervalID può essere utilizzato per cancellare il timer utilizzando la funzione **clearInterval(id)**

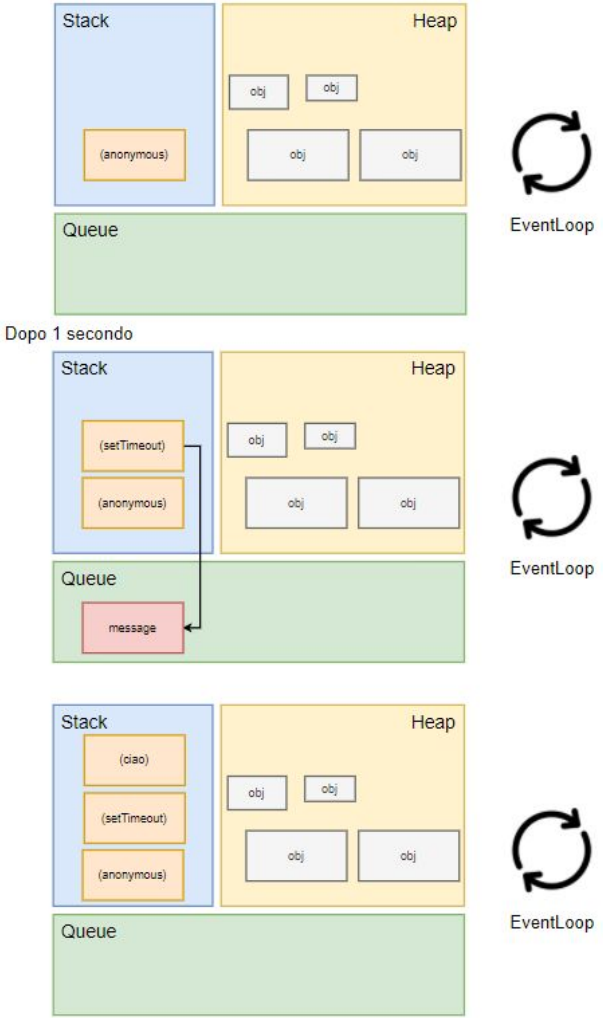
```
function ciao(){
  console.log("ciao");
}

var id = setTimeout(ciao, 1000);

let n = 0;
function stampaN(){
  console.log("n: "+n);
  if(n===3){
    clearInterval(id2);
  }
  else{
    n++;
  }
}

var id2 = setInterval(stampaN, 1000);
```

EventLoop: setTimeout example

```
function ciao(){
  console.log("ciao");
}

var id = setTimeout(ciao, 1000);
```

Nell'esempio: dopo un secondo, quando scade il timer, un messaggio associato alla funzione **ciao()** viene aggiunto alla coda.

L'event loop controlla continuamente la presenza di messaggi nella coda, se un messaggio è processabile ed ha una o più callback associate, per ognuna viene creato uno stack-frame e inserito nello stack per essere eseguito.



EventLoop:

I messaggi nella coda vengono processati in ordine.

Attenzione: Dato che le funzioni non sono prerilasciabili non c'è garanzia su quando verrà eseguita la funzione associata ad un messaggio

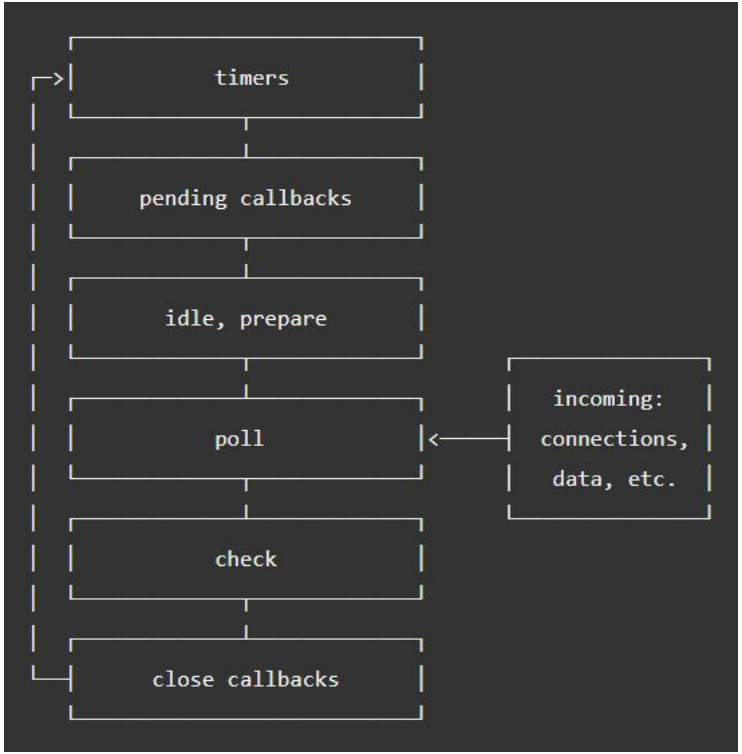
```
function f(){
  console.log("ciao");
}
var id = setTimeout(()=>{
  console.log("eseguita dopo 500ms?");
}, 500);

const start = new Date().getTime();
while(true){
  const now = new Date().getTime();
  if(now-start>2000){
    console.log("trascorsi: "+(now-start)+"ms");
    break;
  }
}
```

EventLoop: fasi

Il meccanismo con cui vengono controllati i messaggi è leggermente più complesso. Vengono utilizzate delle fasi, ognuna della quali ha la propria coda, che viene processata fin quando non si esaurisce.

Per maggiori dettagli:

https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/



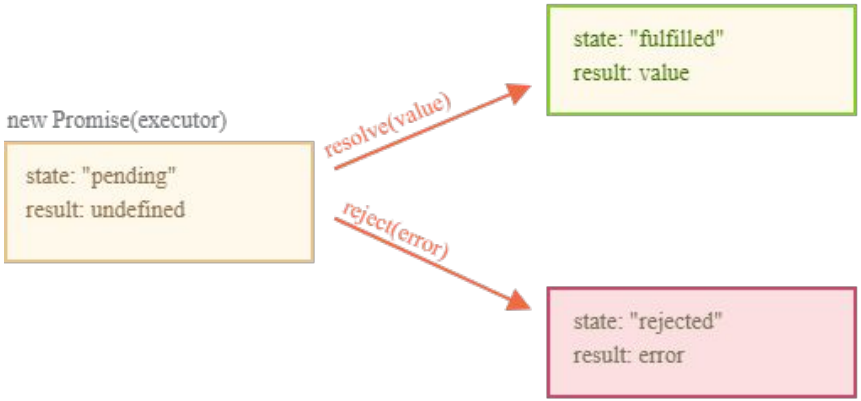
Promise - Definizione e utilizzo

Introdotte in ES6 le Promise sono oggetti che rappresentano il risultato di un'operazione differita o asincrona, ovvero un'operazione che non è ancora completata ma che lo sarà in futuro.

Le librerie moderne utilizzano le promise invece degli eventi per gestire i risultati di operazioni asincrone.

Le Promise possono essere in uno dei seguenti stati:

- **pending**: stato iniziale.
- **fulfilled**: operazione completata con successo.
- **rejected**: operazione terminata a causa di un errore.



JS

Promise - Utilizzo

Per utilizzare le Promise possiamo associare delle callback alla loro risoluzione e/o al loro rigetto utilizzando i metodi del Promise.prototype **then** e **catch**

- La callback passata al metodo then sarà invocata se la promise si risolve senza errori
- Viceversa in caso di errore sarà invocato la callback associata al metodo catch

```
const p = longAsyncFunctionOfLib();

p.then((res)=>{
    console.log(res); //Codice eseguito se
l'operazione termina con successo
    console.log(p); //Resolved
}).catch((error)=>{
    console.log(error); //Codice eseguito se
l'operazione termina con errore
    console.log(p); //Rejected
});

console.log(p); //Pending

//Codice eseguito prima che la promise sia
risolta
let a = 10;
a++;
console.log(a);
```

JS

Promise - Creazione

Possiamo creare delle promise che eseguano un codice scritto da noi, sia esso sincrono o asincrono.

Spesso questo approccio viene utilizzato per *inglobare* vecchie librerie che usano ancora un meccanismo basato su eventi e callback. (Evitare il callback-hell)

```
const p = new Promise((resolve, reject) => {
    const r = Math.random()*100;
    if(r>=50){
        resolve(r);
    }
    reject("Numero troppo piccolo");
});
p.then((res)=>{
    console.log(res);
}).catch((error)=>{
    console.log(error);
});

console.log(p);

//O ancora meglio se:

const wait = (t)=>{
  return new Promise((resolve, reject) => {
    setTimeout(()=>{
      resolve(t);
    },t);
})
};
wait(2500).then(res=>{
  console.log(res);
})
```

JS

Promise - Chaining

Le promise possono essere concatenate:

Ogni chiamata ai metodi then/catch di una Promise restituisce a sua volta una Promise

Per maggiori informazioni:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then

Possiamo utilizzare un solo catch alla fine della chain - Verrà eseguito non appena avviene un errore nella catena

```
const p = new Promise((resolve, reject)=>{
    resolve(0);
  }
).then(res =>{
  return res+1; //Restituisce una Promise
})
.then(res =>{
  return new Promise((resolve, reject)=> {
    resolve(res+1);
  })
}).then(res=>{
  console.log(res);
}).catch(error=>{
  console.log(error);
});
```

Promise - Chaining

In alternativa possiamo usare il catch dopo qualsiasi then, per valutare un errore in un punto specifico.

Attenzione: in questo caso il catch deve restituire un valore, per fare in modo che si possa continuare ad eseguire la catena di promise

Se non si ritorna un valore esplicitamente il then successivo avrà come argomento **undefined.**

```
const p = new Promise((resolve, reject)=>{
    resolve(0);
  }
).then(res =>{
  throw("Errore1")
}).catch(error=>{
  console.log(error)
  return 100;
})
.then(res =>{
  return new Promise((resolve, reject)=> {
    resolve(res+1);
  })
}).then(res=>{
  console.log(res);
});
```

JS

Promise - All/Any

Il metodo Promise.all(iterable) permette di creare una singola promise che viene risolta quando tutte le promise dell'argomento iterable sono risolte.

Il metodo Promise.any(iterable) permette di creare una singola promise che viene risolta quando almeno una delle promise dell'argomento iterable è risolta.

```
const wait = (t)=>{
    return new Promise((resolve, reject) => {
        setTimeout(()=>{
        resolve(t);
        },t);
    })
};
const p1 = wait(1000);
const p2 = wait(500);
const p3 = wait(1000);

const start = new Date().getTime();
Promise.all([p1,p2,p3]).then((value)=>{
  console.log(value);
  console.log((new Date().getTime()-start)+"ms"); //???
})

const start2 = new Date().getTime();
Promise.any([p1,p2,p3]).then((value)=>{
  console.log(value);
  console.log((new Date().getTime()-start2)+"ms"); //???
})
```

JS

Promise - Vantaggi

Le Promise permettono di scrivere un codice più pulito e offrono un metodo che permette di evitare il callback-hell, ovvero un codice che utilizza una serie di callback annidate per gestire degli eventi in sequenza.

```
setTimeout(()=>{
    console.log(1);
    setTimeout(()=>{
        console.log(2);
        setTimeout(()=>{
            console.log(3);
        },250);
    },500);
}, 1000);

const wait = (t)=>{
    return new Promise((resolve, reject) => {
        setTimeout(()=>{
        resolve(t);
        },t);
    })
};
wait(1000).then(()=>{
    console.log(1);
    return wait(500);
}).then(()=>{
    console.log(2);
    return wait(200);
}).then(()=>{
    console.log(3);
});
```

Promise e EventLoop

Le promise non usa la stessa coda messaggi utilizzata dall'event loop. Utilizzano una coda dedicata chiamata **Job Queue** che ha una priorità più alta rispetto a quella dell'event loop!

```
//Crea una Promise che viene risolta immediatamente
const makeZeroPromise = ()=>{
  return new Promise(function(resolve, reject) {
   resolve();
});
}

console.log('1');
//setTimeout con timer = 0!
setTimeout(function() {
   console.log('2');
}, 0);

var promise = makeZeroPromise();
promise.then(function(resolve) {
   console.log('3');
})
.then(function(resolve) {
   console.log('4');
});
console.log('5');
```

JS

Async/Await

Le keyword **async** e **await** sono state introdotte nella versione Javascript ECMAScript2017, e permettono di scrivere del codice basato sulle promise in modo ancora più pulito, evitando di dover utilizzare le chain di promise.

- **async** viene utilizzata per dichiarare una funzione come asincrona. Una funzione asincrona restituisce sempre una promise
- **await** può essere utilizzata solo all'interno di funzioni asincrone e serve ad attendere il risultato di una promise, come si farebbe normalmente con del codice sincrono.

```
const wait = (t)=>{
    return new Promise((resolve, reject) => {
        setTimeout(()=>{
        resolve(t);
        },t);
    })
};
//Promise chain
wait(1000).then(()=>{
    console.log(1);
    return wait(500);
}).then(()=>{
    console.log(2);
    return wait(200);
}).then(()=>{
    console.log(3);
});

//async await
async function waitAll(){
    await wait(1000);
    console.log(1);
    await wait(500);
    console.log(2);
    await wait(500);
    console.log(3);
}

const p = waitAll();
console.log(p);
```