Iteratori Asincroni in JavaScript



INTRODUZIONE

Novità fresca fresca (si parla di ECMASCRIPT 2018), gli iteratori asincroni, ed i corrispettivi generatori asincroni, entrano brutalmente in campo per risolvere un problema tanto sottile quanto importante.
Abbiamo visto che ogni iterazione effettuata con un iteratore sincrono restituisce un oggetto { done, value } dove il **done** è un flag che indica se la fine dell'iterazione è stata raggiunta.
Sono perfetti quindi per le sorgenti di dati sincrone. Cosa intendiamo per sorgente sincrona? Intendiamo una sorgente di dati che, nel momento in cui riceve la richiesta di un dato, è subito in grado di determinare se questo dato sarà l'ultimo disponibile oppure no.

Attenzione a questo particolare: è la sorgente ad essere sincrona, non il dato, che può essere sia sincrono che asincrono. Veloce dimostrazione con un array di Promise:

```
const array = [ new Promise(ok => setTimeout(() => ok(1), 1000)), new Promise(ok => setTimeout(() => ok(2), 2000)), new Promise(ok => setTimeout(() => ok(3), 3000)), new Promise(ok => setTimeout(() => ok(4), 4000)), ] for(const v of array) { console.log(await v); // 1 2 3 4
}
```

L'array fornisce un iteratore sincrono, e anche se il valore ricavato di volta in volta fosse una Promise non è necessario scomodare gli iteratori asincroni, perché è possibile determinare in modo sincrono la fine della collezione di dati. Ogni volta che l'array viene iterato, ogni volta che il *for-of* implicitamente chiama il metodo *next()* dell'iteratore, la sorgente è in grado di determinare sincronicamente se quel dato è l'ultimo oppure no, e quindi può subito impostare il flag *done*.

Notiamo che questo è possibile farlo se la collezione è completamente presente in memoria. Ma saprete meglio di me che è facile doversi interfacciare con sorgenti di dati completamente esterne alla nostra applicazione. Ogni sorgente di dati che, ad esempio, richiede I/O, viene di solito rappresentata nel codice tramite un entità che espone una API asincrona basata sul concetto di evento o, con qualche astrazione in più, con il concetto di stream. Purtroppo gli iteratori sincroni non possono essere utilizzati per rappresentarle ne per interfacciarsi con esse, perché essi costringono a determinare in modo sincrono la fine dell'iterazione. Quelle entità non contengono i dati: forse ne contengono una piccola parte ma non tutti, perché spesso non è fisicamente possibile.

Possiamo vederle, per semplificare un po' il discorso, come un tramite tra sorgenti esterne ed il codice che viene eseguito. Quando richiediamo a queste entità un dato tramite un iteratore, dovendo esse interagire con l'esterno, ed essendo questa interazione spesso e volentieri asincrona, non sono in grado di determinare immediatamente, cioè nel momento in cui

iteriamo/richiediamo il dato successivo, se esso sarà l'ultimo e quindi se l'iterazione è terminata. Ed ecco che entrano in gioco gli iteratori asincroni, nei quali la risoluzione del flag *done* è asincrona.

GLI ITERATORI ASINCRONI

Facciamo un esempio che può aiutare ad afferrare il punto.
Avete su un server un file da 400Gb da inviare ai client che si connetteranno. Ovviamente è fuori discussione inglobarlo interamente nel programma per poterlo inviare, a meno che non abbiate minimo 400Gb di Ram.
Quello che potete fare, su node, è utilizzare un **readStream**. Un readStream è quell'entità che si occuperà di interagire con la sorgente esterna, ovvero con l'enorme file.
All'interno del codice voi potere richiedere al readStream, di volta in volta, qualche byte del file. Il readStream però non solo non è in grado di darvi i byte richiesti, ma deve interagire con il file anche per capire se c'è altro da leggere o meno. Appena ottiene dal file tutte le informazioni necessarie, e ricordiamo che questa interazione è asincrona, può sia restituirci i byte richiesti che dirci se il file è appena terminato o meno, così che a nostra volta possiamo inoltrarli al client e in caso chiudere la comunicazione.

Sorge una domanda: ogni volta che è presente una sorgente asincrona si può implementare un iteratore asincrono per la richiesta dei dati? Non esattamente.

Famose librerie come RxJS utilizzano design patterns come l'Observable per implementare quello che viene definito un approccio **push based**. Con esso, l'entità che nel codice è adibita al rilascio dei dati nel corso del tempo, l'Osservabile, mano mano che la vera sorgente dei dati li rende disponibili, li "pusha" ai subscriber in ascolto che devono essere pronti a reagire al cambiamento.
In questo caso non ha senso implementare l'iteratore asincrono

perché nessuna parte nel nostro codice utilizzatrice di quei dati, avendo come scopo la gestione della UI, può arrogarsi il diritto di controllarne il flusso.

L'approccio degli iteratori asincroni viene invece definito **pull based**. In questo caso la sorgente di dati non li inoltra finché essi non sono richiesti. Perché viene utilizzato questo approccio in Node? Semplificando molto, spesso l'entità che utilizza il dato è più lenta del producer del dato. Ed entra in gioco il problema della backpressure. Se si utilizzano gli stream direttamente, ad esempio tramite il metodo *pipe()*, tutto ciò passa abbastanza inosservato, specialmente perché entrano in gioco tutta una serie di meccanismi implementati dietro le quinte per sopperire a questo problema.
Ad ogni modo il principio che permette di richiedere un dato ad una sorgente asincrona quando lo si ritiene più opportuno, che ripeto già di base è previsto in Node, ha aperto la strada agli iteratori asincroni in questo ambiente.
Infatti tutti gli stream che permettono operazioni di lettura implementano l'interfaccia dell'iteratore asincrono.

Non ci occuperemo di analizzare gli iteratori asincroni nel dettaglio come abbiamo fatto per quelli sincroni, perché l'implementazione delle interfacce che li riguardano può diventare complicata in brevissimo tempo. Ci limiteremo a prendere confidenza con le loro caratteristiche principali. Nel prossimo articolo vedremo come i generatori asincroni ci aiutano ad implementarli. Il metodo da aggiungere ai nostri oggetti non sarà più **Symbol.iterator**, ma sarà **Symbol.asyncIterator**.

Dato che vengono utilizzati nelle situazioni in cui non è possibile determinare in modo sincrono la fine dell'iterazione, gli iteratori asincroni restituiscono **SEMPRE** una Promise, che si completerà nel buon vecchio oggetto { done, value }.
Quando si completa? Si completa appena sarà possibile determinare il *done* per quella specifica iterazione.

Il modo migliore per consumare questo tipo di iteratori è il ciclo **for-await-of**, anch'esso una novità del linguaggio.

il ciclo FOR-await-of

```
for await (const v of asyncSource) { console.log(v);
}
```

Questo particolare tipo di ciclo, utilizzabile solo nelle funzioni **async**, può essere sfruttato per iterare sia sorgenti di dati sincrone, esattamente come il suo cugino *for-of*, sia sorgenti asincrone.
L'oggetto **asyncSource** potrebbe essere anche un iteratore asincrono, perché gli iteratori asincroni rispettano quella convenzione per la quale se usati come *Iterable*, o per meglio dire *AsyncIterable*, restituiscono se stessi.

Vediamo qualche esempio per capire meglio come si comporta il *for-await-of*:

```
// sorgente sincrona, value sincrono
// ogni itera<ione restituisce { value:number, done:boolean }
for await (const v of [1, 2, 3, 4]) { console.log(v); // 1 2 3
4
} // sorgente sincrona, value asincrono
// ogni itera<ione restituisce { value:Promise<number>,
done:boolean }
const array = [ new Promise(ok => setTimeout(() => ok(1),
1000)), new Promise(ok => setTimeout(() => ok(2), 2000)), new
Promise(ok => setTimeout(() => ok(3), 3000)), new Promise(ok
=> setTimeout(() => ok(4), 4000)), ]
for await (const v of array) { console.log(v); // 1 2 3 4
} // sorgente asincrona:
// ogni iterazione restituisce Promise<{ value:number,
done:boolean }>
for await (const v of asyncSource) { console.log(v); // serie
di numeri
} // sorgente asincrona:
// ogni iterazione restituisce Promise<{
value:Promise<number>, done:boolean }>
```

```
for await (const v of asyncSource) { console.log(v); // serie di Promise<number>
}
```

Probabilmente uno o più risultati potrebbero suonarvi strani, perciò proviamo a fare chiarezza.

Se viene rilevata una sorgente asincrona (metodo **Symbol.asyncIterator** implementato), il ciclo si limita ad eseguire l'operazione di **await** sulla Promise restituita dall'iterazione. Quando la Promise viene risolta, se il flag **done** è **false**, il ciclo rende disponibile il **value**, qualunque esso sia, per poi procedere con l'iterazione successiva. Nessun'altra operazione verrà eseguita sul **value** ottenuto, ecco spiegato perché nel terzo esempio vediamo una serie di numeri e nel quarto una serie di Promise. Invece, se il flag **done** è **true**, il ciclo termina.

Se invece l'oggetto iterato non è una sorgente asincrona, il ciclo prova a vedere se esso è una sorgente sincrona, ripiegando sul metodo **Symbol.iterator**. In tal caso ogni iterazione produce direttamente un oggetto { done, value }. Si potrebbe pensare che questo oggetto venga banalmente inserito in una Promise immediatamente risolta dall'**await**, ma le cose non stanno realmente così. Anche perché altrimenti non si spiegherebbe il risultato del secondo esempio, che dovrebbe essere simile al quarto a rigor di logica.
Quello che accade veramente è una promozione a Promise di qualunque sia il valore contenuto in **value** tramite il metodo Promise.resolve(). Ecco che, da questo punto in poi, non esiste più alcuna differenza tra il primo e il secondo caso. Di questa Promise verrà atteso il valore di completamento, che verrà poi utilizzato come **value** nell'oggetto inglobato nella Promise che l'**await** del ciclo sta aspettando.
Possiamo tradurre in codice quest'ultimo periodo nel seguente modo:

```
nextIterationPromise = Promise.resolve(valueObtainedFromSyncIteration).then(value =>
```

({value, done:false}));

Da questa spiegazione comprendiamo che il **value** restituito da una sorgente **asincrona** non dovrebbe mai essere una Promise a sua volta, perché il *for-await-of* diventerebbe inutile. Inoltre, a pensarci bene, questo si avvicinerebbe un po' troppo ad una Promise di Promise, concetto dal quale il JavaScript si è sempre tenuto molto lontano.

I generatori asincroni, dei quali parleremo nel prossimo articolo, si attengono strettamente a questa logica. Vedremo infatti che quando una Promise viene restituita direttamente, tramite **yield** o **return**, senza quindi utilizzare la keyword **await** per attenderne il completamento, quella specifica iterazione non verrà comunque risolta finché la Promise restituita non si completerà. In questo modo il **value** non sarà la Promise restituita, ma diventerà direttamente il suo valore di completamento.

https://italiancoders.it/iteratori-asincroni-in-javascript/