Generatori Asincroni in JavaScript



INTRODUZIONE

È bene ricordare che i generatori ci permettono, tra le altre cose, di creare iteratori. Ecco che i generatori asincroni ci aiutano a rispettare l'interfaccia dell'iteratore asincrono in modo da poter implementare più agilmente un iterabile asincrono.

Se queste poche righe vi suonano strane significa che avete saltato i tre appuntamenti precedenti su iteratori, generatori e iteratori asincroni, e vi conviene mettervi in pari per poter seguire al meglio quest'ultimo articolo.

i GENERATORI ASINcroni

Vediamone subito uno nel suo ambiente naturale:

```
const asyncSource = { async *[Symbol.asyncIterator]() { //
```

```
metti lo yield togli l'await // metti lo yield togli l'await }
}
```

Carino, vero? C’è la keyword **async**, c’è la stellina, tipico segno che contraddistingue una funzione normale da un generatore. Ma il bello viene quando si scopre quello che c’è all’interno.

Possiamo infatti attendere il completamento di una Promise, per poi *yieldare* o *restituire* il valore contenuto in essa:

```
const asyncSource = { async *[Symbol.asyncIterator]() { const v = await new Promise(ok => setTimeout(() => ok(1), 1000)); yield v; }
}
```

Se vogliamo fare gli esibizionisti, possiamo ridurre tutto in una singola linea:

```
const asyncSource = { async *[Symbol.asyncIterator]() { yield await new Promise(ok => setTimeout(() => ok(1), 1000)); }
}
```

In questo caso possiamo anche evitare di utilizzare la keyword **await**. Come avevamo accennato nel precedente articolo, il **value** di ogni iterazione asincrona non dovrebbe mai essere una Promise, perciò se ne “*yieldiamo*” una essa verrà attesa implicitamente, come se avessimo anteposto la keyword await:

```
const asyncSource = { async *[Symbol.asyncIterator]() { yield new Promise(ok => setTimeout(() => ok(1), 1000)); }
}
```

Ovviamente quando il valore da restituire è legato a più Promise, rientra in gioco l’*await*:

```
const asyncSource = { async *[Symbol.asyncIterator]() { const one = await new Promise(ok => setTimeout(() => ok(1), 1000)); const two = await new Promise(ok => setTimeout(() => ok(2), 2000)); yield one + two; }
}
```

Riattiviamo la modalità esibizionista:

```
const asyncSource = { async *[Symbol.asyncIterator]() { yield ( await new Promise(ok => setTimeout(() => ok(1), 1000)) + await new Promise(ok => setTimeout(() => ok(2), 2000)) ); }
}
```

Una situazione degna di nota è la seguente:

```
const asyncSource = { async *[Symbol.asyncIterator]() { await new Promise(ok => setTimeout(() => ok(1), 1000)); await new Promise(ok => setTimeout(() => ok(2), 2000)); yield 42; }
}
```

Sebbene il valore *42* potrebbe venire immediatamente restituito, verrà comunque atteso il completamento delle due Promise anche se il loro valore non viene nemmeno utilizzato. Questo perchè, come nelle normali funzioni **async**, la keyword **await** interrompe momentaneamente l'esecuzione.

esempio pratico

Come esempio verrà ripresa la lettura di un file in node; questa volta faremo pieno uso delle possibilità offerte dagli iterabili asincroni. Inoltre vedremo un pattern veramente interessante che riguarda i generatori asincroni: l'invocazione di un generatore asincrono produce un iteratore asincrono che è anche un iterabile asincrono, perciò essi possono essere utilizzati come sorgenti asincrone di dati per altri generatori asincroni collegati in serie, i quali consumano questi dati con il ciclo **for-await-of**. La dualità iteratore-iterabile era stata ampliamente spiegata nella prima puntata di questa serie, perciò vi consiglio di ridarci un'occhiata perché sarà molto utile avere confidenza con questo concetto.

In altre parole possiamo definire questo pattern un *piping*, simile a quello Unix, ma asincrono. È uno strumento molto potente, come avremo modo di osservare.

Ovviamente è possibile ottenere il piping anche con i generatori sincroni e il ciclo *for-of*, esso sarà però un piping sincrono.

Per ricadere meno nel banale ho creato una classe, *AsyncFileReader*, le quali istanze hanno un metodo, *read()*, per la lettura asincrona dei file, linea dopo linea. Il metodo *read()* non è altro che un generatore asincrono che provvede un iterabile, asincrono, il quale ad ogni iterazione fornisce una linea del file.

```
// yaffee
(async function IIAFE() { const fileStream = new AsyncFileReader('test.txt') for await (const line of fileStream.read()) { console.log('> ' + line); }
})();
```

Non lasciatevi spaventare dalla sua implementazione, che esamineremo punto per punto:

```
const fs = require('fs'); const AsyncFileReader = function () { // richiede in ingresso un iterabile asincrono che restituisce i chunk // ritorna (yield) ogni linea costruita con uno o più chunk async function* chunksToLines(chunksSource) { let previous = ''; // itera l'iterabile ricevuto in ingresso che provvede // i chunk in modo asincrono for await (const chunk of chunksSource) { previous += chunk; let eolIndex; while ((eolIndex = previous.indexOf('\n')) >= 0) { const line = previous.slice(0, eolIndex + 1); // feed dell'iteratore prodotto dal generatore asincrono chunksToLines yield line; previous = previous.slice(eolIndex + 1); } } if (previous.length > 0) { // feed dell'iteratore prodotto dal generatore asincrono chunksToLines yield previous; } } return class AsyncFileReader { constructor(file) { this.file = file; } async *read() { const readStream = fs.createReadStream(this.file, { encoding: 'utf8', // codifica highWaterMark: 1024 // grandezza massima del buffer (usato internamente) }); // piping for await (const line of chunksToLines(readStream)) { yield line; } } }
}();
```

Analizziamo la situazione:

```
async *read() { const readStream = fs.createReadStream(this.file, { encoding: 'utf8', // codifica highWaterMark: 1024 // grandezza massima del buffer (utilizzato internamente) }); // piping for await (const line of chunksToLines(readStream)) { yield line; }
}
```

Il metodo *read()*, generatore asincrono, si occupa di gestire la sorgente dei dati, ovvero il file, con un **readStream**. La chiamata al metodo *createReadStream()* dell'oggetto *fs* imposta la codifica e la grandezza massima del buffer, utilizzato internamente per gestire il problema della backpressure, del quale accennammo nell'articolo precedente.
Lo stream *readStream* quando viene iterato, ad ogni iterazione, restituisce il contenuto del buffer, buffer che viene popolato ovviamente in modo asincrono.

Il seguente ciclo *for-await-of*, usato come esempio, stamperebbe a video il contenuto del buffer mano mano che esso si riempie, e questo fino alla fine del file:

```
for await (const buffer of readStream) { console.log(buffer);
}
```

Abbiamo però parlato di **piping**, pattern che ci permette di comporre più generatori in serie. Ecco che entra in gioco la funzione *chunksToLines()*, memorizzata tramite closure perché non deve essere disponibile all'esterno. Anche questa funzione è un generatore asincrono:

```
// traforma uno o più chunk in una linea
async function* chunksToLines(chunksSource) { let previous = ''; // itera l'iterabile ricevuto in ingresso che provvede // i chunk in modo asincrono for await (const chunk of chunksSource) { previous += chunk; let eolIndex; while ((eolIndex = previous.indexOf('\n')) >= 0) { const line = previous.slice(0, eolIndex + 1); // feed dell'iteratore prodotto dal generatore asincrono chunksToLines yield line; previous = previous.slice(eolIndex + 1); } } if
```

```
(previous.length > 0) { // feed dell'iteratore prodotto dal generatore asincrono chunksToLines yield previous; }
}
```

Essa riceve in ingresso un iterabile asincrono, lo stream, e lo consuma grazie al ciclo *for-await-of*. Il ciclo richiede un iteratore dall'iterabile invocato il metodo **Symbol.asyncIterator** e lo utilizza. Il generatore *chunksToLines* inoltre restituisce tramite yield un valore solo quando lo ritiene opportuno. Altrimenti prosegue iterando la sorgente dei dati. Dove vanno a finire i valori che restituisce? Dove viene eseguito questo acclamato *piping*?

Essendo *chunksToLines* un generatore asincrono, produrrà un iteratore asincrono. Un iteratore asincrono è anche un iterabile asincrono, perciò possiamo darlo in pasto al *for-await-of* dentro al metodo *read()*:

```
for await (const line of chunksToLines(readStream)) { yield line;
}
```

Il *piping* è proprio racchiuso in questo ciclo, ed implica quello che abbiamo esaminato fin'ora: lo stream *readStream* non viene iterato direttamente dal metodo *read()*, ma viene passato prima al generatore asincrono *chunksToLines*.
In altre parole:

- *chunksToLines* itera lo stream *readStream,* occupandosi di gestire i chunk
- *read()* itera l'iteratore-iterabile asincrono prodotto da *chunksToLines*, ottenendo una linea del file ad ogni iterazione

Da questo è facile capire che il numero di iterazioni può solo aumentare mano mano che si scende all'interno della catena di piping, fino ad arrivare alla sorgente che sarà iterata il maggior numero di volte. Ovvero, per comporre una singola linea *chunksToLines* potrebbe aver bisogno di iterare più volte lo *readStream*. Ma solo quando ne ha ottenuta una risolverà la

Promise che il *for-await-of* dentro a *read()* sta attendendo, provocando l'iterazione dentro ad esso.

Anche il metodo *read()* è stato impostato come generatore asincrono e quindi produrrà a sua volta un iteratore asincrono, il quale riceverà i valori che il metodo *read()* restituisce tramite **yield**. Chi utilizza quest'altro iteratore? È la top-level IIAFE, per gli amici "yaffee", con il suo *for-await-of*. Anche in questo caso è presente un'operazione di pipe, quasi implicita, che non ha alcun effetto sul flusso di dati:

```
// yaffee
(async function IIAFE() { const fileStream = new AsyncFileReader('test.txt') for await (const line of fileStream.read()) { console.log('> ' + line); }
})();
```

Il piping a questo livello può essere considerato implicito ma è presente, perché nella IIAFE non utilizziamo direttamente l'iteratore prodotto dall'invocazione chunksToLines(readStream) ma quello fornitoci dal metodo *read()*. Proprio perché questo generatore asincrono non da alcun contributo effettivo, possiamo eliminare lo step nel seguente modo:

```
read() { const readStream = fs.createReadStream(this.file, { encoding: 'utf8', highWaterMark: 1024 }); return chunksToLines(readStream);
}
```

Adesso *read()* è stato declassato a normale funzione, ma essa restituisce comunque un iteratore-iterabile asincrono, perciò il codice presente nella top-level IIAFE non necessita di essere modificato. Abbiamo semplicemente eliminato quell'operazione di piping aggiuntiva.
Ed abbiamo potuto farlo solo perché essa non contribuiva in alcun modo al valore corrente di ogni dato richiesto dalla IIAFE.
In caso contrario, anche se non fosse un richiesto uno

sfasamento tra il numero di iterazioni come avviene con *chunksToLines*, saremmo costretti a ripristinare l'operazione di piping. Ad esempio, nel seguente caso, vogliamo capitalizzare la prima lettera di ogni linea:

```
async *read() { const readStream = fs.createReadStream(this.file, { encoding: 'utf8', // codifica highWaterMark: 1024 // grandezza massima del buffer }); for (const line of chunksToLines(readStream)) { yield str[0].toUpperCase() + str.slice(1); }
}
```

Dato che il file viene splittato in chunk di 1024 bytes, questo codice può gestire tranquillamente anche file molto grandi, perché ne tiene in memoria solo una piccola parte.
Il punto critico è il generatore asincrono *chunksToLines*, il quale richiede i chunk finché non trova il carattere di newline, per poi restituire tutta la linea. Quindi analizzare file JavaScript minificati potrebbe non essere una buona idea.

Sicuramente avrete notato il fatto che la classe *AsyncFileReader* non implementa l'interfaccia dell'iterabile asincrono, prediligendo anzi l'uso di un metodo apposito: *read()*. Questa implementazione è lasciata al lettore volenteroso; lo scopo finale è quello di poter modificare la top-level IIAFE nel seguente modo:

```
// yaffee
(async function IIAFE() { const fileStream = new AsyncFileReader('test.txt') for await (const line of fileStream) { console.log('> ' + line); }
})();
```

CONCLUSIONE

Non è immediato comprendere il pattern del *piping* asincrono, non tanto per il flusso seguito dai dati ma per quello che accade dietro le quinte. Personalmente mi aiuta immaginarlo come l'incremento delle cifre in un numero. Mentre le cifre

delle unità scorrono (iterano), quelle delle decine, delle centinaia, delle migliaia, ecc. non si muovono. Ad un certo punto la cifra delle unità passa da 9 a 0, “yeldando” una decina, e così la cifra delle decine aumenta (itera) di uno, per poi far riprendere l’incremento (l’iterazione) delle unità.
A volte può accadere che vi siano più 9 in fila in attesa e che quindi il singolo incremento delle unità porti a più incrementi in serie.

Questo esempio può rappresentare abbastanza bene un generatore, sincrono o asincrono, che con un *for-of* o un *for-await-of*, itera un iteratore-iterabile provveduto da un altro generatore e “yielda” alcuni valori, ma solo quando lo ritiene opportuno.

L’esempio non è completamente adeguato alla reale situazione perché, per essere precisi, dobbiamo immaginare che le cifre delle unità scorrano per colpa delle decine, che le decine scorrano per via delle centinaia e così via. Questo perché l’iterazione dei generatori “più interni”, più vicini alla sorgente dei dati, è controllata da quelli “più esterni”, più lontani da tale sorgente, nella catena di piping.
Nell’esempio della lettura di un file è infatti la top-level IIAFE ad aprire le danze con:

```
for await (const line of fileStream.read()) { console.log('> ' + line);
}
```

Se essa non iterasse l’iteratore-iterabile prodotto dal metodo *read()*, la funzione *chunksToLines* non richiederebbe i chunk allo stream *readStream* e nemmeno un chunk del file verrebbe letto.
Questo non toglie nulla al fatto che finché *chunksToLines* non completa la sua iterazione, i generatori asincroni che dipendono da essa non possono proseguire con l’iterazione successiva. Sono le cifre delle unità ad influenzare le

decine, le decine ad influenzare le centinaia e così via. Perciò *chunksToLines* agisce in base a quello che *readStream* restituisce e la IIAFE agisce in base a quello che *chunksToLines* restituisce.

Probabilmente adesso siete più confusi di prima. Un abbraccio.

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