Asynchronous Generators and Pipelines in JavaScript

Nested Software Apr 23 *Updated on May 09, 2018*

#javascript **#async** **#generator** **#es2018**

Introducing Asynchronous Generators

Both this article and the last one, The Iterators Are Coming, which deals with asynchronous iterators, were motivated by a question that occurred to me as I was programming with some `async` functions: *Would it be possible to `yield` in an `async` function?* In other words, can we combine an `async` function with a generator function?

To explore this question, let's start with a normal synchronous generator function, `numberGenerator`:

```
const random = (min, max) => Math.floor(Math.random() * (max - min + 1)) + m

const getValue = () => {
    return random(1,10)
}

const numberGenerator = function* () {
    for (let i=0; i<5; i++) {
        const value = getValue()
        yield value**2
    }
}

const main = () => {
    const numbers = numberGenerator()
    for (const v of numbers) {
        console.log('number = ' + v)
    }
}

main()
```

This code produces the expected squares of 5 random numbers:

```
C:dev>node gen.js
number = 1
number = 64
number = 36
number = 25
number = 49
```

My idea was to change `getValue` to return a promise and to

search

value. I tried something like the following:

```
const random = (min, max) => Math.floor(Math.random() * (max - min + 1)) + m

const getValue = () => {
    //return promise instead of value
    return new Promise(resolve=>{
        setTimeout(()=>resolve(random(1,10)), 1000)
    })
}

const numberGenerator = function* () {
    for (let i=0; i<5; i++) {
        const value = await getValue() //await promise
        yield value**2
    }
}

const main = () => {
    const numbers = numberGenerator()
    for (const v of numbers) {
        console.log('number = ' + v)
    }
}

main()
```

Let's see what happens:

```
C:\dev\gen.js:12
                const value = await getValue() //await promise
                              ^^^^^

SyntaxError: await is only valid in async function
    at new Script (vm.js:51:7)
```

Okay, that makes sense: We need to make our `numberGenerator` function `async`. Let's try that!

```
const numberGenerator = async function* () { //added async
```

Does it work?

```
C:\dev\gen.js:10
const numberGenerator = async function* () { //added async
                                        ^

SyntaxError: Unexpected token *
    at new Script (vm.js:51:7)
```

Ouch, it didn't work. This is what led me to do some online searching on the topic. It turns out this kind of functionality is going to be released in ES2018, and we can use it already in a recent version of node with the `--harmony-async-iteration` flag.

Let's see this in action:

```
const timer = () => setInterval(()=>console.log('tick'), 1000)

const random = (min, max) => Math.floor(Math.random() * (max - min + 1)) + m

const getValue = () => {
    //return promise instead of value
    return new Promise(resolve=>{
        setTimeout(()=>resolve(random(1,10)), 1000)
    })
}

const numberGenerator = async function* () { //added async
    for (let i=0; i<5; i++) {
        const value = await getValue() //await promise
}

//main is 'async'
const main = async () => {
    const t = timer()
    const numbers = numberGenerator()

    //use 'for await...of' instead of 'for...of'
    for await (const v of numbers) {
        console.log('number = ' + v)
    }

    clearInterval(t)
}

main()
```

There are a few small changes from the previous version of the code:

- The `main` function's `for...of` loop becomes a `for await...of` loop.
- Since we are using `await`, `main` has to be marked as `async`

> A timer was also added so we can confirm that the generator is indeed asynchronous.

Let's take a look at the results:

```
C:\dev>node --harmony-async-iteration gen.js
tick
number = 16
```

```
tick
number = 100
tick
number = 100
tick
number = 49
```

It worked!

> The `yield` in an `async` generator function is similar to the `yield` in a normal (synchronous) generator function. The difference is that in the regular version, `yield` produces a `{value, done}` tuple, whereas the asynchronous version produces a promise that *resolves* to a `{value, done}` tuple.

Pipelining Asynchronous Generators Together

Let's look at a neat little application of this technology: We will create an asynchronous generator function that drives another one to produce statistics on an asynchronous stream of numbers.

This kind of pipeline can be used to perform arbitrary transformations on asynchronous data streams.

First we'll write an asynchronous generator that produces an endless stream of values. Every second it generates a random value between 0 and 100:

```
const random = (min, max) => Math.floor(Math.random() * (max - min + 1)) + m

const asyncNumberGenerator = async function* () {
    while (true) {
        const randomValue = random(0,100)

        const p = new Promise(resolve=>{
            setTimeout(()=>resolve(randomValue), 1000)
        })

        yield p
    }
}
```

Now we'll write a function, `createStatsReducer`. This function returns a callback function, `exponentialStatsReducer`, that will be used to iteratively calculate the exponential moving average on this stream of data:

```
const createStatsReducer = alpha => {
    const beta = 1 - alpha

    const exponentialStatsReducer = (newValue, accumulator) => {
        const redistributedMean = beta * accumulator.mean

        const meanIncrement = alpha * newValue

        const newMean = redistributedMean + meanIncrement

        const varianceIncrement = alpha * (newValue - accumulator.mean)**2

        const newVariance = beta * (accumulator.variance + varianceIncrement

        return {
            lastValue: newValue,
            mean: newMean,
            variance: newVariance
        }
    return exponentialStatsReducer
}
```

Next up we have a second asynchronous generator function, `asyncReduce`. This one applies a reducer to an asynchronous iterable. It works like JavaScript's built-in `Array.prototype.reduce`. However, the standard version goes through an entire array to produce a final value, whereas our version applies the reduction lazily. This allows us to use an infinite sequence of values (our asynchronous number generator above) as the data source:

```
const asyncReduce = async function* (iterable, reducer, accumulator) {
    for await (const item of iterable) {
        const reductionResult = reducer(item, accumulator)

        accumulator = reductionResult

        yield reductionResult
    }
}
```

Let's tie this all together. The code below will pipe an endless sequence of asynchronously-generated numbers into our asynchronous reduce. We will loop through the resulting values (forever), obtaining the updated mean, variance, and standard deviation as new values arrive:

```
const timer = () => setInterval(()=>console.log('tick'), 1000)
```

```
    const numbers = asyncNumberGenerator()

    const firstValue = await numbers.next()

    //initialize the mean to the first value
    const initialValue = { mean: firstValue.value, variance: 0 }

    console.log('first value = ' + firstValue.value)

    const statsReducer = createStatsReducer(0.1)

    const reducedValues = asyncReduce(numbers, statsReducer, initialValue)

    for await (const v of reducedValues) {
        const lastValue = v.lastValue
        const mean = v.mean.toFixed(2)
        const variance = v.variance.toFixed(2)
        const stdev = Math.sqrt(v.variance).toFixed(2)

        console.log(`last value = ${lastValue}, stats = { mean: ${mean}`
            + `, variance: ${variance}, stdev: ${stdev} }`)
    }

    clearInterval(t)
}

main()
```

Let's take a look at some sample output:

```
C:\dev>node --harmony-async-iteration async_stats.js
tick
first value = 51
tick
last value = 97, stats = { mean: 55.60, variance: 190.44, stdev: 13.80 }
tick
last value = 73, stats = { mean: 57.34, variance: 198.64, stdev: 14.09 }
```

```
tick
last value = 42, stats = { mean: 51.64, variance: 345.16, stdev: 18.58 }
tick
last value = 42, stats = { mean: 50.67, variance: 319.00, stdev: 17.86 }
tick
last value = 60, stats = { mean: 51.60, variance: 294.93, stdev: 17.17 }
^C
```

We now get continually updating statistics on our asynchronous stream of values. Neat!

I think that asynchronous generator functions will be especially useful to do processing on sources of asynchronous data along these lines.

Let me know what you think, or if you have ideas for other ways asynchronous generators and iterators can be used!

References:

- ES2018: asynchronous iteration
- Array.prototype.reduce

Related:

- The Iterators Are Coming
- Careful Examination of JavaScript Await
- Exponential Moving Average on Streaming Data
- How to Serialize Concurrent Operations in Javascript: Callbacks, Promises, and Async/Await

Filter, and Reduce

❤️ 9 🦄 6 🔖 20

dev.to is where software developers stay in the loop and avoid career stagnation.

Signing up (for free!) is the first step.

Nested Software + FOLLOW
nestedsoftware

Add to the discussion

PREVIEW SUBMIT

code of conduct - report abuse

Classic DEV Post from May 4

What programming sub-disciplines seem to be trending up in terms of career options?

Ben Halpern

Which types of roles are gaining momentum in our field? What are the jobs that ...

❤️ 89 💬 28

READ POST SAVE FOR LATER

I was not ready to become the maintainer of Babel

Henry Zhu

I had never published my own npm package before or explored much of the codebase, but slowly (sometimes really slowly) I got used to it.

249 16

READ POST SAVE FOR LATER

Another Post You Might Like

But really, what is a JavaScript mock?

Kent C. Dodds

Photo by Dmitri Popov on Unsplash Let's take a step back and understand what m...

57

READ POST SAVE FOR LATER

Play With the React 🐘Router
Sai gowtham - Jun 1

Under the Hood of the Most Powerful Video JavaScript API
The JW Player Team - Jun 1

An alternative to handle state in React: the URL !

GaelS - Jun 1