JavaScript Asynchronous Iterators

Summary: in this tutorial, you will learn about the JavaScript asynchronous iterators that allow you to access asynchronous data sequentially.

Introduction to JavaScript Asynchronous Iterators

ES6 introduced the iterator interface that allows you to access data sequentially. The iterator is well-suited for accessing the synchronous data sources like arrays, sets, and maps.

The main method of an iterator interface is the `next()` that returns the `{value, done}` object, where `done` is a boolean indicating whether the end of the sequence is reached and `value` is the yielded value in the sequence.

The synchronous data means that the next `value` in the sequence and the `done` state is known at the time the `next()` method returns.

Besides the synchronous data sources, JavaScript often has to access asynchronous data sources like I/O access. For the asynchronous data sources, the `value` and `done` state of the iterator is often unknown at the time the `next()` method returns.

To deal with the asynchronous data sources, ES2018 introduced the asynchronous iterator (or async iterator) interface.

An async iterator is like an iterator except that its `next()` method returns a promise that resolves to the `{value, done}` object.

The following illustrates the `Sequence` class that implements the iterator interface. (Check it out the iterator tutorial for more information on how to implement `Sequence` class.)

```
class Sequence {
    constructor(start = 0, end = Infinity, interval = 1) {
            this.start = start;
```

```
            this.end = end;
            this.interval = interval;
        }
        [Symbol.iterator]() {
            let counter = 0;
            let nextIndex = this.start;
            return {
                next: () => {
                    if (nextIndex <= this.end) {
                        let result = {
                            value: nextIndex,
                            done: false
                        }
                        nextIndex += this.interval;
                        counter++;
                        return result;
                    }
                    return {
                        value: counter,
                        done: true
                    };
                }
            }
        }
}
```

To make this `Sequence` class asynchronously, you need to modify it as follows:

- Use the `Symbol.asyncIterator` instead of the `Symbol.iterator`
- Return a Promise from the `next()` method.

The following code transforms the `Sequence` class to the `AsyncSequence` class:

```
class AsyncSequence {
    constructor(start = 0, end = Infinity, interval = 1) {
            this.start = start;
            this.end = end;
            this.interval = interval;
        }
```

```
        [Symbol.asyncIterator]() {
            let counter = 0;
            let nextIndex = this.start;
            return {
                next: async () => {
                    if (nextIndex <= this.end) {
                        let result = {
                            value: nextIndex,
                            done: false
                        }
                        nextIndex += this.interval;
                        counter++;

                        return new Promise((resolve, reject) => {
                            setTimeout(() => {
                                resolve(result);
                            }, 1000);
                        });
                    }
                    return new Promise((resolve, reject) => {
                        setTimeout(() => {
                            resolve({
                                value: counter,
                                done: true
                            });
                        }, 1000);

                    });
                }
            }
        }
}
```

The `AsyncSequence` returns the next number in the sequence after every 1 second.

The for await...of statement

To iterate over an asynchronous iterable object, ES2018 introduced the `for await...of` statement:

```
for await (variable of iterable) {
    // statement
}
```

Since we can use the `await` keyword in an `async` function only, we can create an async IIFE as that uses the `AsyncSequence` class as follows:

```
(async () => {

    let seq = new AsyncSequence(1, 10, 1);

    for await (let value of seq) {
        console.log(value);
    }

})();
```

Output (each number is returned after every second)

```
1
2
3
4
5
6
7
8
9
10
```

The following table illustrates the differences between the iterators and async iterators:

#	Iterators	Async iterators
Well-known Symbol	`Symbol.iterator`	`Symbol.asyncIterator`

#	Iterators	Async iterators
`next()` return value is	`{value, done }`	`Promise` that resolves to `{value, done}`
Loop statement	`for...of`	`for await...of`