JavaScript TUTORIAL

The Essential Guide to JavaScript Iterators

Summary: in this tutorial, you will learn about JavaScript iterators and how to use iterators to process a sequence of data more efficiently.

The for loop issues

When you have an array of data, you typically use a `for` loop to iterate over its elements. For example:

```
let ranks = ['A', 'B', 'C'];

for (let i = 0; i < ranks.length; i++) {
    console.log(ranks[i]);
}
```

The `for` loop uses the variable `i` to track the index of the `ranks` array. The value of `i` increments each time the loop executes as long as the value of `i` is less than the number of elements in the `ranks` array.

This code is straightforward. However, its complexity grows when you nest a loop inside another loop. In addition, keeping track of multiple variables inside the loops is error-prone.

ES6 introduced a new loop construct called `for...of` to eliminate the standard loop's complexity and avoid the errors caused by keeping track of loop indexes.

To iterate over the elements of the `ranks` array, you use the following `for...of` construct:

```
for(let rank of ranks) {
    console.log(rank);
}
```

The `for...of` is far more elegant than the `for` loop because it shows the true intent of the code – iterate over an array to access each element in the sequence.

On top of this, the `for...of` loop has the ability to create a loop over any **iterable** object, not just an array.

To understand the iterable object, you need to understand the iteration protocols first.

Iteration protocols

There are two iteration protocols: **iterable protocol** and **iterator protocol**.

Iterator protocol

An object is an iterator when it implements an interface (or API) that answers two questions:

- Is there any element left?
- If there is, what is the element?

Technically speaking, an object is qualified as an iterator when it has a `next()` method that returns an object with two properties:

- `done`: a boolean value indicating whether or not there are any more elements that could be iterated upon.
- `value`: the current element.

Each time you call the `next()`, it returns the next value in the collection:

```
{ value: 'next value', done: false }
```

If you call the `next()` method after the last value has been returned, the `next()` returns the result object as follows:

```
{done: true: value: undefined}
```

The value of the `done` property indicates that there is no more value to return and the `value` of the property is set to `undefined`.

Iterable protocol

An object is iterable when it contains a method called `[Symbol.iterator]` that takes no argument and returns an object that conforms to the iterator protocol.

The `[Symbol.iterator]` is one of the built-in well-known symbols in ES6.

Iterators

Since ES6 provides built-in iterators for the collection types `Array`, `Set`, and `Map`, you don't have to create iterators for these objects.

If you have a custom type and want to make it iterable so that you can use the `for...of` loop construct, you need to implement the iteration protocols.

The following code creates a `Sequence` object that returns a list of numbers in the range of (`start`, `end`) with an `interval` between subsequent numbers.

```
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 };
            }
        }
    }
};
```

The following code uses the `Sequence` iterator in a `for...of` loop:

```
let evenNumbers = new Sequence(2, 10, 2);

for (const num of evenNumbers) {
    console.log(num);
}
```

Output:

```
2
4
6
8
10
```

You can explicitly access the `[Symbol.iterator]()` method as shown in the following script:

```
let evenNumbers = new Sequence(2, 10, 2);
let iterator = evenNumbers[Symbol.iterator]();

let result = iterator.next();

while( !result.done ) {
    console.log(result.value);
    result = iterator.next();
}
```

Cleaning up

In addition to the `next()` method, the `[Symbol.iterator]()` may optionally return a method called `return()`.

The `return()` method is invoked automatically when the iteration is stopped prematurely. It is where you can place the code to clean up the resources.

The following example implements the `return()` method for the `Sequence` object:

```
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 };
            },
            return: () => {
                console.log('cleaning up...');
                return { value: undefined, done: true };
            }
        }
    }
}
```

The following snippet uses the `Sequence` object to generate a sequence of odd numbers from 1 to 10. However, it prematurely stops the iteration. As a result, the `return()` method is automatically invoked.

```
let oddNumbers = new Sequence(1, 10, 2);

for (const num of oddNumbers) {
    if( num > 7 ) {
        break;
    }
    console.log(num);
}
```

Output:

```
1
3
5
7
cleaning up...
```

In this tutorial, you have learned about the JavaScript iterator and how to use the iteration protocols to implement customized iteration logic.