Iterables

Iterable objects is a generalization of arrays. That's a concept that allows to make any object useable in a `for..of` loop.

Of course, Arrays are iterable. But there are many other built-in objects, that are iterable as well. For instance, Strings are iterable also. As we'll see, many built-in operators and methods rely on them.

If an object represents a collection (list, set) of something, then `for..of` is a great syntax to loop over it, so let's see how to make it work.

Symbol.iterator

We can easily grasp the concept of iterables by making one of our own.

For instance, we have an object, that is not an array, but looks suitable for `for..of`.

Like a `range` object that represents an interval of numbers:

```
let range = {
  from: 1,
  to: 5
};

// We want the for..of to work:
// for(let num of range) ... num=1,2,3,4,5
```

To make the `range` iterable (and thus let `for..of` work) we need to add a method to the object named `Symbol.iterator` (a special built-in symbol just for that).

1. When `for..of` starts, it calls that method once (or errors if not found). The method must return an *iterator* – an object with the method `next`.
2. Onward, `for..of` works *only with that returned object*.
3. When `for..of` wants the next value, it calls `next()` on that object.
4. The result of `next()` must have the form `{done: Boolean, value: any}`, where `done=true` means that the iteration is finished, otherwise `value` must be the new value.

Here's the full implementation for `range`:

```
let range = {
  from: 1,
  to: 5
};

// 1. call to for..of initially calls this
range[Symbol.iterator] = function() {

  // ...it returns the iterator object:
  // 2. Onward, for..of works only with this iterator, asking it for next values
  return {
    current: this.from,
    last: this.to,

    // 3. next() is called on each iteration by the for..of loop
    next() {
      // 4. it should return the value as an object {done:.., value :...}
      if (this.current <= this.last) {
        return { done: false, value: this.current++ };
      } else {
        return { done: true };
      }
    }
  };
};

// now it works!
for (let num of range) {
  alert(num); // 1, then 2, 3, 4, 5
}
```

Please note the core feature of iterables: an important separation of concerns:

- The `range` itself does not have the `next()` method.
- Instead, another object, a so-called "iterator" is created by the call to `range[Symbol.iterator]()`, and it handles the whole iteration.

So, the iterator object is separate from the object it iterates over.

Technically, we may merge them and use `range` itself as the iterator to make the code simpler.

Like this:

```
let range = {
  from: 1,
  to: 5,

  [Symbol.iterator]() {
    this.current = this.from;
    return this;
  },

  next() {
    if (this.current <= this.to) {
      return { done: false, value: this.current++ };
    } else {
      return { done: true };
    }
  }
};

for (let num of range) {
  alert(num); // 1, then 2, 3, 4, 5
}
```

Now `range[Symbol.iterator]()` returns the `range` object itself: it has the necessary `next()` method and remembers the current iteration progress in `this.current`. Shorter? Yes. And sometimes that's fine too.

The downside is that now it's impossible to have two `for..of` loops running over the object simultaneously: they'll share the iteration state, because there's only one iterator – the object itself. But two parallel for-ofs is a rare thing, doable with some async scenarios.

Infinite iterators

Infinite iterators are also possible. For instance, the `range` becomes infinite for `range.to = Infinity`. Or we can make an iterable object that generates an infinite sequence of pseudorandom numbers. Also can be useful.

There are no limitations on `next`, it can return more and more values, that's normal.

Of course, the `for..of` loop over such an iterable would be endless. But we can always stop it using `break`.

String is iterable

Arrays and strings are most widely used built-in iterables.

For a string, `for..of` loops over its characters:

```
for (let char of "test") {
  // triggers 4 times: once for each character
  alert( char ); // t, then e, then s, then t
}
```

And it works correctly with surrogate pairs!

```
let str = '𝒳😂';
for (let char of str) {
    alert( char ); // 𝒳, and then 😂
}
```

Calling an iterator explicitly

Normally, internals of iterables are hidden from the external code. There's a `for..of` loop, that works, that's all it needs to know.

But to understand things a little bit deeper let's see how to create an iterator explicitly.

We'll iterate over a string the same way as `for..of`, but with direct calls. This code gets a string iterator and calls it "manually":

```
let str = "Hello";

// does the same as
// for (let char of str) alert(char);

let iterator = str[Symbol.iterator]();

while (true) {
  let result = iterator.next();
  if (result.done) break;
  alert(result.value); // outputs characters one by one
}
```

That is rarely needed, but gives us more control over the process than `for..of`. For instance, we can split the iteration process: iterate a bit, then stop, do something else, and then resume later.

Iterables and array-likes

There are two official terms that look similar, but are very different. Please make sure you understand them well to avoid the confusion.

- *Iterables* are objects that implement the `Symbol.iterator` method, as described above.
- *Array-likes* are objects that have indexes and `length`, so they look like arrays.

Naturally, these properties can combine. For instance, strings are both iterable (`for..of` works on them) and array-like (they have numeric indexes and `length`).

But an iterable may be not array-like. And vice versa an array-like may be not iterable.

For example, the `range` in the example above is iterable, but not array-like, because it does not have indexed properties and `length`.

And here's the object that is array-like, but not iterable:

```
let arrayLike = { // has indexes and length => array-like
  0: "Hello",
  1: "World",
  length: 2
};

// Error (no Symbol.iterator)
for (let item of arrayLike) {}
```

What do they have in common? Both iterables and array-likes are usually *not arrays*, they don't have `push`, `pop` etc. That's rather inconvenient if we have such an object and want to work with it as with an array.

Array.from

There's a universal method Array.from that brings them together. It takes an iterable or array-like value and makes a "real" `Array` from it. Then we can call array methods on it.

For instance:

```
let arrayLike = {
  0: "Hello",
  1: "World",
  length: 2
};

let arr = Array.from(arrayLike); // (*)
alert(arr.pop()); // World (method works)
```

`Array.from` at the line `(*)` takes the object, examines it for being an iterable or array-like, then makes a new array and copies there all items.

The same happens for an iterable:

```
// assuming that range is taken from the example above
let arr = Array.from(range);
```

```
alert(arr); // 1,2,3,4,5 (array toString conversion works)
```

The full syntax for `Array.from` allows to provide an optional "mapping" function:

```
Array.from(obj[, mapFn, thisArg])
```

The second argument `mapFn` should be the function to apply to each element before adding to the array, and `thisArg` allows to set `this` for it.

For instance:

```
// assuming that range is taken from the example above

// square each number
let arr = Array.from(range, num => num * num);

alert(arr); // 1,4,9,16,25
```

Here we use `Array.from` to turn a string into an array of characters:

```
let str = '𝒳😂';

// splits str into array of characters
let chars = Array.from(str);

alert(chars[0]); // 𝒳
alert(chars[1]); // 😂
alert(chars.length); // 2
```

Unlike `str.split`, it relies on the iterable nature of the string and so, just like `for..of`, correctly works with surrogate pairs.

Technically here it does the same as:

```
let str = '𝒳😂';

let chars = []; // Array.from internally does the same loop
for (let char of str) {
  chars.push(char);
}

alert(chars);
```

...But is shorter.

We can even build surrogate-aware `slice` on it:

```
function slice(str, start, end) {
  return Array.from(str).slice(start, end).join('');
}

let str = '𝒳😂𩷶';

alert( slice(str, 1, 3) ); // 😂𩷶

// native method does not support surrogate pairs
alert( str.slice(1, 3) ); // garbage (two pieces from different surrogate pairs)
```

Summary

Objects that can be used in `for..of` are called *iterable*.

- Technically, iterables must implement the method named `Symbol.iterator`.
  - The result of `obj[Symbol.iterator]` is called an *iterator*. It handles the further iteration process.
  - An iterator must have the method named `next()` that returns an object `{done: Boolean, value: any}`, here `done:true` denotes the iteration end, otherwise the `value` is the next value.
- The `Symbol.iterator` method is called automatically by `for..of`, but we also can do it directly.
- Built-in iterables like strings or arrays, also implement `Symbol.iterator`.
- String iterator knows about surrogate pairs.

Objects that have indexed properties and `length` are called *array-like*. Such objects may also have other properties and methods, but lack the built-in methods of arrays.

If we look inside the specification – we'll see that most built-in methods assume that they work with iterables or array-likes instead of "real" arrays, because that's more abstract.

`Array.from(obj[, mapFn, thisArg])` makes a real `Array` of an iterable or array-like `obj`, and we can then use array methods on it. The optional arguments `mapFn` and `thisArg` allow us to apply a function to each item.

Previous lesson Next lesson

Share

Tutorial map

Comments

- You're welcome to post additions, questions to the articles and answers to them.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen...)
- If you can't understand something in the article – please elaborate.