Making use of built-in higher order array functions in JavaScript

Some examples to help you master reduce, map, filter, some, find, and every.

bytefish

Photo by Bharat Patil on Unsplash

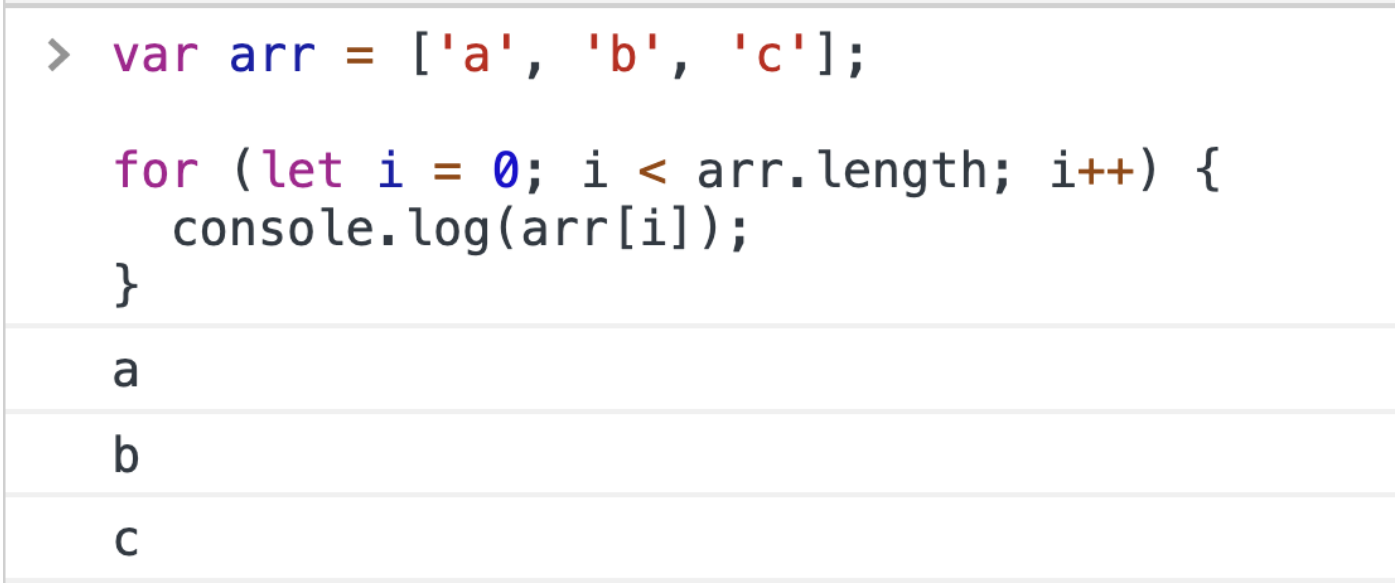
`Array` is the most common data structure in programming. And traversal is one of our most common operations on arrays. So, do you know how many

ways to loop an array?

for-index

First of all, an array is an indexed data structure. We can traverse an array through its index.

```
var arr = ['a', 'b', 'c'];for (let i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}
```



from Chrome console

for in

At the same time, in JavaScript, we can traverse the enumerable properties of an object through the for-in grammar. All arrays are objects, so we can also apply this to arrays.

```
var arr = ['a', 'b', 'c'];for(let key in arr) {
    console.log(arr[key]);
```

```
}
```

for of

ES2015 adds *iterators* to JavaScript. The easiest way to use iterators is the new `for-of` statement. It looks like this:

```
var arr = ['a', 'b', 'c'];for ( let element of arr) {
    console.log(element);
}
```

The above several for-loops are classic methods for traversing arrays. There is no doubt that there is no error in using the above methods in the code. But our JavaScript is a functional programming language, which provides us with many higher-order functions to loop arrays. If we can use them properly, we can make the code more concise and elegant, and inspire us to think from a different perspective.

These higher-order functions are:

- Array.protorype.map
- Array.protorype.filter
- Array.protorype.reduce
- Array.protorype.some
- Array.protorype.every

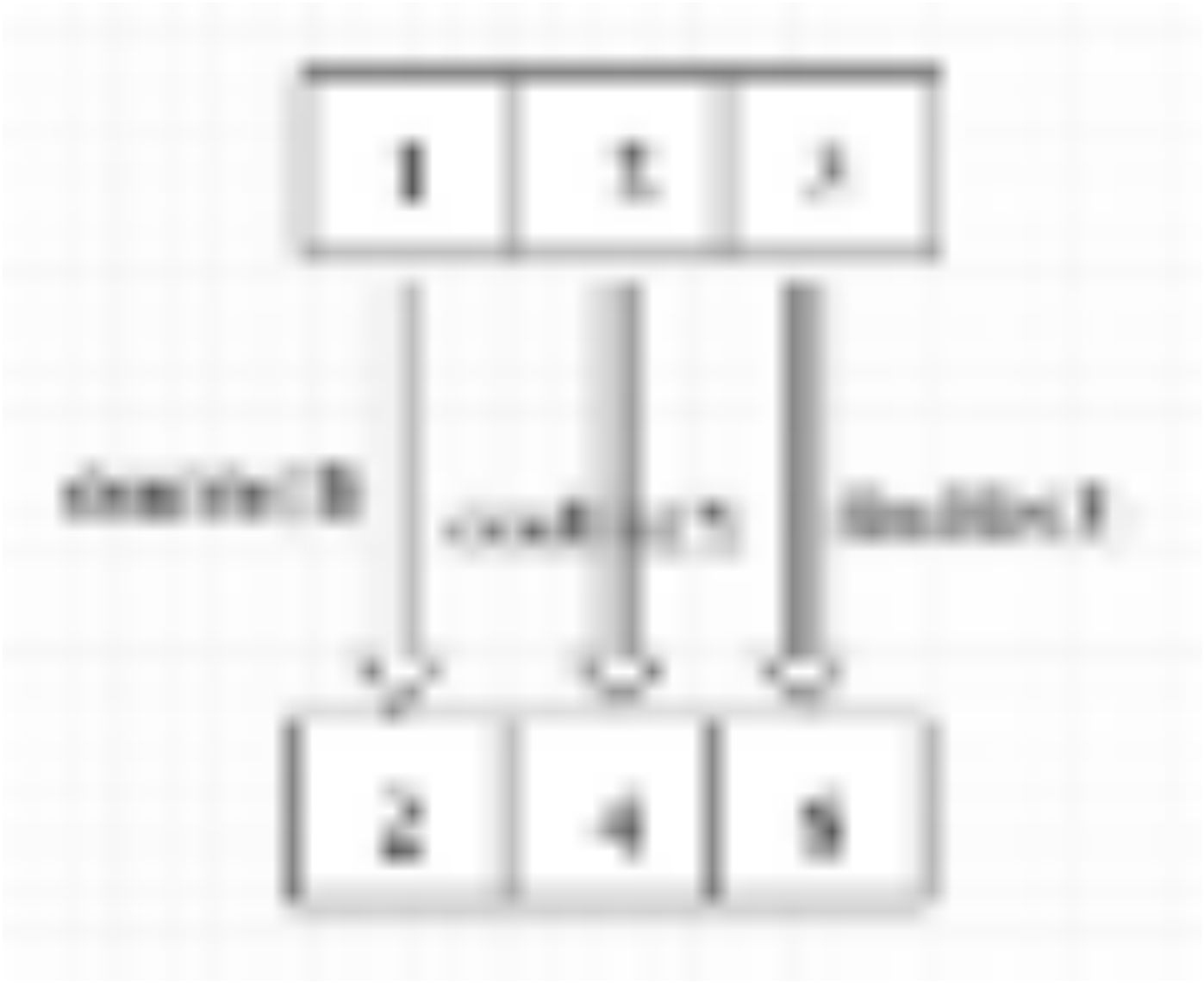
Map

The `map()` method **creates a new array** populated with the results of calling a provided function on every element in the calling array.

This is a basic usage of `reduce` :

```
var arr = [1, 2, 3];function double(ele){
  return ele * 2
}arr.map(double)
```

The `map` method takes a function as an argument. This function is then called for each element in the array, and the result of the function call is returned to form a new array



Now let's take a look at how the `reduce` method replaces for-loop with a few examples.

1. Copy an array

Suppose now we want to copy an array (regardless of deep cloning), what would you do? By for-loop, we may write like this:

```
function copy(arr){
  let newArray = [];
  for(let ele of arr){
```

```
    newArray.push(ele)
  }
  return newArray;
}
```

But if you're used to using `.map()`, it only takes one line of code.

2. Walkthrough all the elements and perform the same modification

Let's say we have an array of characters, and we want all the elements in that array to be uppercase strings, so what do we do?

With for-loop:

```
var names = ["Jon", "Sonw", "bitfish", "Tom"];
var upperCaseNames = [];
for(let i = 0; i < names.length ; i= i +1) {
  upperCaseNames[i] = names[i].toUpperCase();
}
console.log(upperCaseNames)
```

But with `map`, the task is very simple.

```
var names = ["Jon", "Sonw", "bitfish", "Tom"];var upperCaseNames
```

Filter

It is also very common to filter some unqualified elements in an array, so how do we write the code?

So let's say we have an array of integers, and we want to remove all of the odd integers and keep only the even integers.

With for-loop:

```
function filterOdd(arr){
  function isEven(n) {
    return n % 2 === 0;
  }
  var evens = [];  for(let number of numbers) {
    if(isEven(number)) {
       evens.push(number);
    }
  }
  return evens;
}
```

Using `filter`:

The `filter()` method **creates a new array** with all elements that pass the test implemented by the provided function.

Reduce

`Array.prototype.reduce()` is one of the most powerful methods in an array, and it is also an attractive feature in JavaScript functional programming. But unfortunately, I found that many friends are not used to using it. Let me introduce this method in detail and hope it can help you.

This is a basic usage of `reduce` :

```
var arr = [1, 2, 3];function reducer(parmar1, parmar2){
}arr.reduce(reducer)
```

`reduce` is a method on the array prototype object, which can help us operate the array. It will take another function as its parameter, which can be called a `reducer`.

`reducer` takes two parameters. The first parameter, param1, is the result of the last reducer run. If this is the first time `reducer` is run, the default value of param1 is the value of the first element of the array.

The `reduce` method loops through each element in the array much like it

would in a for-loop. And the current value in the loop was taken as param2.

After traversing the array, `reduce` will return the result of the last reducer calculation.

Let's take a look at a detailed example.

Next, let's explore how the above code is executed.

In this code, `reducer` is `add` .

First, because we execute `add` for the first time, the first element `'a'` in the array will be treated as the first parameter of `add`, and then the loop will start from the second element `'b'` of the array. This time, `'b'` is the second parameter of `add`.

After the first calculation, we get the result `'ab'`. This result will be cached and used as param1 in the next `add` calculation. Meanwhile, the third parameter `'c'` in the array will be used as param2 of `add`.

Similarly, `reduce` will continue to traverse the elements in the array, running `'abc'` and `'d'` as parameters of `add`.

Finally, after traversing the last element in the array, the calculation result will be returned.

Now we have the result: `'abcde'`.

So, we can see that `reduce` is also a way to loop arrays! It takes the value of each element in the array in turn and executes the `reducer` function.

But we can see that the above cycle does not have that kind of harmonious aesthetic feeling. Because we take the first element in the array, that is, `'a'`, as the initial param1, and then param2 is obtained by looping from the

second element in the array.

In fact, we can specify the second parameter in `reduce` as the initial value of the `reducer` function's param1, so param2 will be obtained by looping from the first element in the array.

The code is as follows:

This time, we take `'s'` as param1 when we call `reducer` for the first time, and then start to traverse the array from the first element in turn.

So we can use this syntax to rewrite our first code piece.

```
var arr = ['a', 'b', 'c', 'd', 'e'];function add(x, y) {
    return x + y;
}arr.reduce(add, '')
```

Well, the above is the basic introduction of the `reduce` method. Let's go back to the example at the beginning.

Now let's take a look at how the `reduce` method replaces for-loop with a few examples.

1. Accumulation and cumulative multiplication

What would you do if we wanted to get the sum of all the elements in the numerical array?

In general, you might write like this:

You may use`for-in` or `for-of` instead, but as long as you use the for loop, the code will appear redundant. In this case, `Array.prototype.reduce` can help us write code in a better way.

Then let's see what the accumulation function above does:

- Set an initial `sum` to zero
- Take out the first element in the array and sum it
- Cache the results of the previous step in the `sum`
- Take out the other elements in the array in turn and perform the above operation
- Return the final result

We can see that when we describe the above steps in plain English, it is obvious that it conforms to the usage of `reduce`. So we can use `reduce` to rewrite the above code:

If you are used to using arrow functions, the above code will look more

concise:

All the code in one line!

Of course, cumulative multiplication and accumulation are exactly the same:

A lot of times, we need to add a weight when we are summing up, which can better reflect the elegance of `reduce`.

```
const scores = [
  { score: 90, subject: "HTML", weight: 0.2 },
  { score: 95, subject: "CSS", weight: 0.3 },
  { score: 85, subject: "JavaScript", weight: 0.5 }
];const result = scores.reduce((x, y) => x + y.score * y.weight,
```

2. Get the maximum and minimum values of an array

If you want to get the maximum and minimum values of an array, you might write like this:

It's the same as before. If we use `reduce`, we can do it in one line of code.

```
let arr = [3.24, 2.78, 999];arr.reduce((x, y) => Math.max(x, y))
```

3. Count the frequency of elements in an array

We often need to count the number of times each element in the array appears. By the grammar of the for-loop, we can write like this

The `reduce` method can also help us to achieve this in a different way.

Note that we use a map object instead of an object to store the frequency after statistics, because the elements in the array may be of an object type, and the key of an object can only be of a string or symbol type.

Here are two examples:

Similarly, if you want to count the frequency of each character in the string, you can first convert the string to a character array, and then follow the above method.

Because character types can be used as keys for objects, we don’t use Map here.

Some

We may often check whether an array contains a value that meets a condition. For example, check whether any element in a numeric array is a multiple of 7.

With for-loop:

```
var numbers = [233, 324, 5666, 324, 456];function hasMutipleOfSe
  for (let number of numbers) {
    if(number % 7 === 0) {
      return true
    }
  }
  return false
}
```

The `some()` method tests whether at least one element in the array passes the test implemented by the provided function. It returns a Boolean value.

```
var numbers = [233, 324, 5666, 324, 456];numbers.some(ele => ele
```

Every

Sometimes we need to check whether every element in an array meets a condition.

For example, we want to check that the first letter of all the elements in a character array is uppercase.

With for-loop:

```
var strArray = ["Tom", "bitfish", "Jerry"]function checkFistLett
  for(let str of arr){
    let firstLetter = str.charAt(0);
    if(firstLetter < "a" || firstLetter > "z"){
      return false
    }
  }
  return true
}
```

The `every()` method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value.

```
var strArray = ["Tom", "bitfish", "Jerry"]strArray.every(str =>
  let firstLetter = str.charAt(0);
  if(firstLetter < "a" || firstLetter > "z"){
    return false
  }
})
```

Using these higher-order functions to traverse array makes our code more semantic and readable. At the same time, do not forget that JavaScript is a functional programming language, the use of these higher-order functions can let us get used to using functional programming thinking to think.

Finally, thank you for reading this, and I hope this article will help you.

A note from the Plain English team

Did you know that we have four publications? Show some love by giving them a follow: **JavaScript in Plain English**, **AI in Plain English**, **UX in Plain English**, **Python in Plain English** — thank you and keep learning!

Also, we're always interested in helping to promote good content. If you have an article that you would like to submit to any of our publications, send an email to **submissions@plainenglish.io** with your Medium username and what you are interested in writing about and we will get back to you!