**99% of Beginners Do Not Understand the Difference Between ‘for’ Loops & ‘forEach’ -** [Gejiufelix](https://medium.com/@cowdog?source=post_page-----6cb0bc375595--------------------------------) Aug 4, 2022

**Let’s first look at the essential difference:**

The ‘for’ loop has been in existence since the beginning of JavaScript. ‘forEach’, introduced in ES5, is mounted on the prototype of iterable objects, such as Array Set Map.

**‘forEach’** is an iterator responsible for traversing an iterable object. So what are traversal, iteration, and iterable objects? This is a bit more detailed, and I leave it to you to review for yourself.

Knowing that ‘forEach’ is actually an iterator, the essential difference between it and the ‘for’ loop is that ‘forEach’ is responsible for traversing (Array Set Map) iterable objects, while ‘for’ loop is a looping mechanism, but it can traverse an array through it.

**Explanation:** What is an iterator? Here, the Generator generator is used as an example. When it is called, it will generate an iterator object (Iterator Object). It has a .next() method, and each call returns an object {value:value,done:Boolean }, value returns the return value after yield, when yield ends, done becomes true, and the internal value is accessed by calling and iterating in turn.

An iterator is a special kind of object. In the ES6 specification, its sign is the next() method of the returned object, and the iterative behavior is judged as done. Iterators implement traversal without exposing the internal representation.

See code:

let arr = [4, 5, 6, 7] let iterator = arr[Symbol.iterator]() console.log(iterator.next()); console.log(iterator.next()); console.log(iterator.next()); console.log(iterator.next()); console.log(iterator.next());

As we can see from the above, as long as it is an iterable object, calling the internal Symbol.iterator will provide an iterator, and access the internal according to the next method returned by the iterator, which is also the implementation principle of for…of.

let arr = [4, 5, 6, 7]  
for (const item of arr) {  
 console.log(item);  
}

Call the next method to return the value of the object and save it in the item until done is true to jump out of the loop, and all iterable objects can be consumed by for…of. Let’s look at other iterable objects:

function num(params) {  
 console.log(arguments);  
 let iterator = arguments[Symbol.iterator]()  
 console.log(iterator.next());  
 console.log(iterator.next());  
 console.log(iterator.next());  
 console.log(iterator.next());  
 console.log(iterator.next());  
}  
num(4, 5, 6, 7)let set = new Set(‘4567’)  
set.forEach(item => {  
 console.log(item);  
})  
let iterator = set[Symbol.iterator]()  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());  
console.log(iterator.next());

Then we can also intuitively see that the Symbol.iterator property in the iterable object can generate an iterator when it is called, and forEach also generates an iterator, passing the value of each element in the internal callback function.

**Description: I think it is the iterator that is called here, calling next continuously, and passing parameters to the callback function. You can leave a message in the comment area and talk about your own views.**

Syntax difference between ‘for’ loop and ‘forEach’:

1. Parameters of forEach.  
2. Interruption of forEach.  
3. forEach deletes its own element, and the index cannot be reset.  
4. The for loop can control the starting point of the loop.

**Parameters for forEach**  
Let’s understand the complete parameter content of forEach:

arr.forEach((self,index,arr) =>{},this)

**self:** The element currently traversed by the array. By default, the array elements are obtained in order from left to right.

**index:** The index of the current element of the array, the index of the first element is 0, and so on.

**arr:** The currently traversed array.

**this:** This refers to the callback function.

let arr = [4, 5, 6, 7];  
let person = {  
 name: ‘BOb’  
};  
arr.forEach(function (self, index, arr) {console.log(`${self} ${index}, ${arr}`);  
 console.log(this.name+=’high’);  
}, person)

We can use arr to achieve array deduplication:

let arr1 = [4, 5, 4, 6, 4];  
let arr2 = [];  
arr1.forEach(function (self, index, arr) {  
 arr1.indexOf(self) === index ? arr2.push(self) : null;  
});  
console.log(arr2);

**forEach interrupt**

In JavaScript, there is break-return-continue to interrupt the function or jump out of the loop. We will use some interruption behaviors in the for loop, which is very good for optimizing array traversal and search, but because forEach belongs to iterators, it can only be ordered in order The sequential traversal is completed, so the above interrupting behavior is not supported.

*let arr = [4,5,6,7],  
i = 0,  
length = arr.length;  
for (; i < length; i++) {  
console.log(arr[i]);  
if (arr[i] === 5) {  
break;  
};  
};*

*arr.forEach((self,index) => {   
 console.log(self);  
 if (self === 5) {  
 break;  
 };  
});*

*arr.forEach((self,index) => {  
 console.log(self);  
 if (self === 5) {  
 continue;  
 };  
});*

What if I have to break out of the loop in forEach? In fact, there is a way, with the help of try/catch:

*try {  
 var arr = [4, 5, 6, 7];  
 arr.forEach(function (item, index) {  
 if (item ===6 ) {  
 throw new Error(“LoopTerminates”);  
 }  
   
 console.log(item);  
 });  
} catch (e) {  
 if (e.message !== “LoopTerminates”) throw e;  
};*

If return is encountered, no error will be reported, but it will not take effect:

*let arr = [4, 5, 6, 7];function find(array, num) {  
 array.forEach((self, index) => {  
 if (self === num) {  
 return index;  
 };  
 });  
};  
let index = find(arr, 5);*

**forEach deletes its own element, index cannot be reset**  
In forEach, we can’t control the value of index, it will just mindlessly increment until it is greater than the length of the array and jump out of the loop. Therefore, it is not possible to delete itself to reset the index.

Let’s look at a simple example:

*let arr = [4,5,6,7]  
arr.forEach((item, index) => {  
 console.log(item);  
 index++;  
});*

index does not change as it is added or subtracted from within the function body. In actual development, it is very common to traverse an array and delete an item at the same time. Pay attention when using forEach to delete.

**for loop can control the starting point of the loop**As mentioned above, the starting point of the forEach loop can only be 0 without human intervention, but the for loop is different.

The previous operation of traversing the array and deleting the breeding can be written as:

*let data = [];let data2 = [1,2,3];data2.map(item=>data.push(item));*

**Performance difference between for loop and forEach**  
In terms of performance comparison, we add a map iterator, which, like filter, generates a new array. We compare the performance of for forEach map in a browser environment:

**Performance comparison: for > forEach > map**  
In chrome 62 and Node.js v9.1.0 environment: for loop is 1x faster than forEach, and forEach is about 20% faster than map.

**Cause Analysis**

**for:** The for loop has no additional function call stack and context, so its implementation is the simplest.

**forEach:** For forEach, its function signature contains parameters and context, so the performance will be lower than for loop.

**map:**The reason why map is the slowest is because map returns a new array, and the creation and assignment of arrays will result in allocation of memory space, so it will bring a large performance overhead. If the map is nested in a loop, it will bring more unnecessary memory consumption.

When you use an iterator to traverse an array, it is against the design intent to use map without returning a new array.