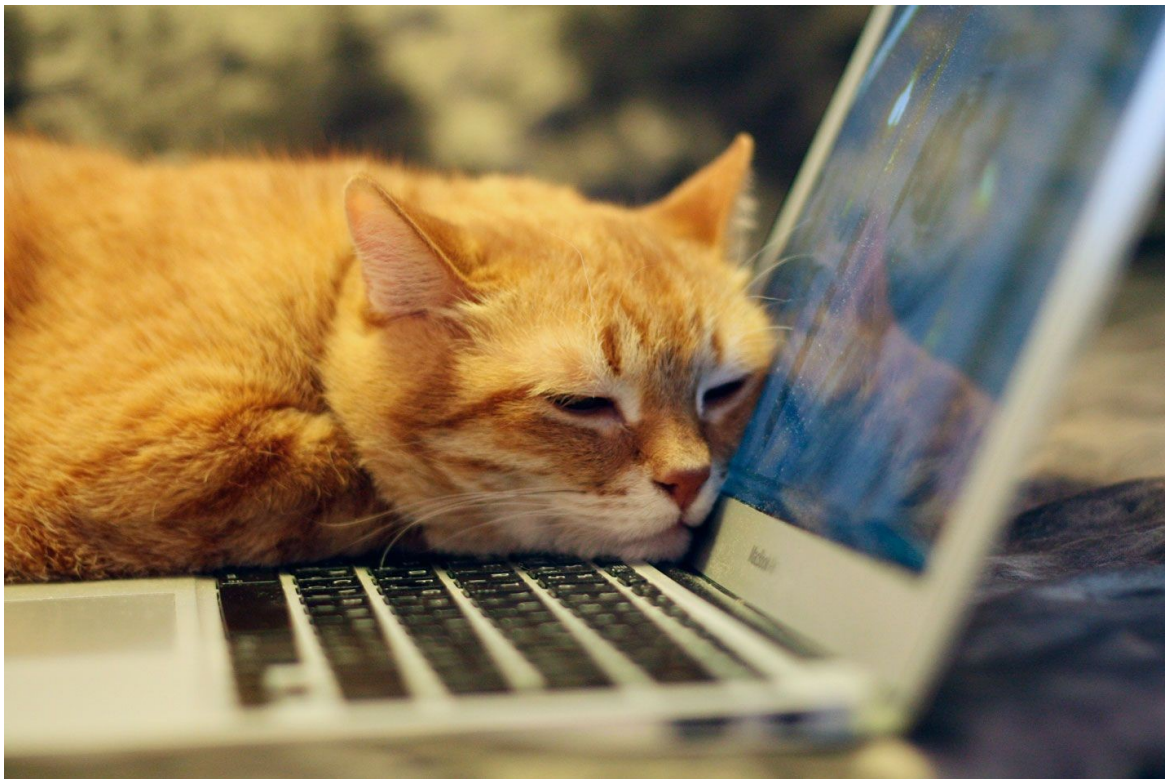
itdo.com

JavaScript. Métodos de iteración: Map, Filter, Reduce, forEach

Sergio Vergara

5-6 minutos



La popularidad de JavaScript te ha puesto a programar ahora mismo en este lenguaje tan particular, o con alguno de sus *frameworks*. Es probable que hayas tenido dudas a la hora de iterar valores de un vector (*array, o matriz unidimensional)* y te haya entrado la gran tentación de utilizar el padre de todos los bucles: *for*.

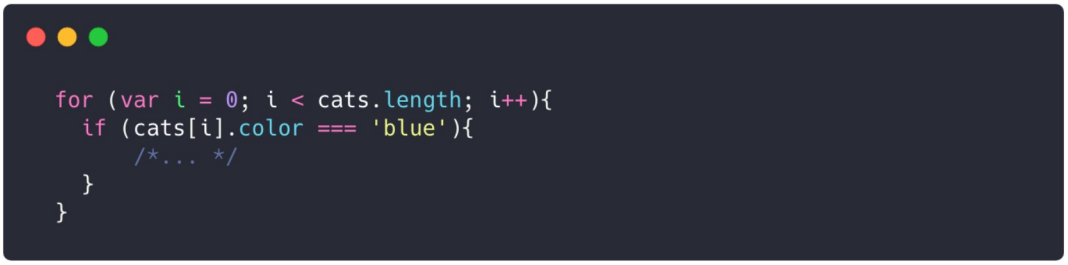
Lejos de querer juzgarte por hacerlo, hoy quiero compartir contigo algunas de las opciones modernas más utilizadas por los programadores y por qué deberías usar cada una de ellas para

mejorar la legibilidad de tu código y, naturalmente el rendimiento de tu aplicación.

Antes de seguir, te recomiendo que leas nuestros artículos sobre JavaScript. Si tienes alguna pregunta, ¡no dudes en dejar un comentario!

Me va bien iterar con *For*

Una función de JavaScript desarrollada usando *for* es legítima y funciona correctamente, ya que te permite obtener el resultado que necesitas. Lo que pasa es que por sus características, a veces necesitas hacer demasiadas cosas en una misma función. Esto dificulta la lectura y normalmente aquél código solo sirve una vez. No es reusable. Es aquí que ¡*Map*, *Filter*, *Reduce*, *forEach*, *Every* y *Some* llegan al rescate¡ Son métodos soportados por todos los navegadores modernos. Pero, ¿qué son estos métodos? Algunos se parecen un poco, ¿para qué sirven?



for loop

Filter

filter() te permite crear un **nuevo vector** con elementos del vector inicial que pasen un determinado test. Por ejemplo, imagínate que de un listado de *gatos* quieres filtrar todos los *azules* creando un otro vector solamente con estos gatos azules. Lo único que tendrías que hacer sería:

```
var cats = [
  {
    color: 'blue',
    name: 'gru'
  },
  {
    color: 'red',
    name: 'mpy'
  },
   {
    color: 'blue',
    name: 'chiy'
  },
];

var blues = cats.filter(myFunction);
//console.log(blues)

function myFunction(cat) {
  return cat.color === 'blue';
}
```

método filter()

Como puedes ver, declaramos un vector de objetos, utilizamos una función pura de llamada para hacer el control y ejecutamos *filter().* El resultado en este caso sería un nuevo vector de objetos solamente con los gatos azules.

Map

El método *map()* crea un nuevo vector (array, o matriz unidimensional) ejecutando una función en cada uno de los elementos del vector. Al mismo tiempo no ejecuta la función en elementos sin valor y no cambia el vector original.

Ejemplo

Imagínate, por ejemplo que lo que quieres es multiplicar por 2 la edad de cada uno de los gatos y crear un nuevo vector con estas edades. Con map() harías lo siguiente:

```
```

```
var cats = [
  {
    color: 'blue',
    name: 'gru',
    age: 2
  },
  {
    color: 'red',
    name: 'mpy',
    age: 5
  },
   {
    color: 'blue',
    name: 'chiy',
    age: 37
  },
];

var doubleAges = cats.map(doubleAge);
//console.log(doubleAges)

function doubleAge(cat) {
  return cat.age * 2;
}
```

método map()

Además, también podrías utilizar el método anterior para duplicar solamente la edad de tus gatos azules encadenando los dos métodos. Por ejemplo:

```
var cats = [
  {
    color: 'blue',
    name: 'gru',
    age: 2
  },
  {
    color: 'red',
    name: 'mpy',
    age: 5
  },
   {
    color: 'blue',
    name: 'chiy',
    age: 37
  },
];

var doubleAges = cats.filter(getBlue).map(doubleAge);
//console.log(doubleAges);

function getBlue(cat) {
  return cat.color === 'blue';
}

function doubleAge(cat) {
  return cat.age * 2;
}
```

método filter() encadenado por map()

Reduce

El método *reduce()* ejecuta una función en cada uno de los elementos del vector para “reducirlo” a un valor único. Es importante tener en cuenta que *reduce()* funciona de izquierda a derecha y no “reduce” el vector original.

Imagina por ejemplo que quieres sumar el total del doble de las edades de los gatos azules. Solamente tendrías que hacer:

```
var cats = [
  {
    color: 'blue',
    name: 'gru',
    age: 2
  },
  {
    color: 'red',
    name: 'mpy',
    age: 5
  },
   {
    color: 'blue',
    name: 'chiy',
    age: 37
  },
];

var sum = cats.filter(getBlue).map(doubleAge).reduce(sumAge);
console.log(sum);


function getBlue(cat) {
  return cat.color === 'blue';
}

function doubleAge(cat) {
  return cat.age * 2;
}

function sumAge(total, cat) {
  return total + cat;
}
```

método filter(), encadenado por map() y reduce()

Y obtendrías el resultado de la suma. En este caso 78.

forEach

forEach() llama y ejecuta una función una vez por cada elemento del vector. Comparando con *map()* o *reduce(), forEach()* es ideal cuando lo que quieres no es cambiar los datos de tu vector, si no guardarlos en alguna base de datos. *forEach()* siempre devuelve el valor *undefined* y no es encadenable. Lo podrías ejecutar de la siguiente forma:

```
var txt = "";
var ages = [2, 5, 37];
ages.forEach(myFunction);

function myFunction(value) {
  txt = txt + value + "<br>";
}
```

método forEach()

every

El método *every()* verifica si **todos** los valores del vector pasan en un determinado test. Por ejemplo, quieres verificar que todos los gatos tienen más de 18 años, solo tendrías que hacer:

```
var cats = [
  {
    color: 'blue',
    name: 'gru',
    age: 2
  },
  {
    color: 'red',
    name: 'mpy',
    age: 5
  },
   {
    color: 'blue',
    name: 'chiy',
    age: 37
  },
];
```

```
var allOver18 = cats.every(isOver18);
//console.log(allOver18)

function isOver18(cat) {
  return cat.age > 18;
}
```

método every()

Y obtendrías un *true* o *false*.

some

Al contrario de *every()*, el método *some()* verifica si **alguno** de los valores del vector pasa un determinado test. Si por ejemplo lo que quieres es saber si hay por lo menos un gato mayor de 18 harías lo siguiente:

```
var cats = [
  {
    color: 'blue',
    name: 'gru',
    age: 2
  },
  {
    color: 'red',
    name: 'mpy',
    age: 5
  },
   {
    color: 'blue',
    name: 'chiy',
    age: 37
  },
];


var someOver18 = cats.some(isOver18);
//console.log(someOver18)

function isOver18(cat) {
  return cat.age > 18;
}
```

método some()

¿Conclusión?

¿Cuál es el mejor método? Pues, como siempre depende de lo que quieres hacer, de tu objetivo y la necesidad de la aplicación. Si quieres comparar el rendimiento de cada uno de los métodos, te recomiendo utilizar herramientas como *jsPerf* para hacerlo.

Creo que es importante dominar estos métodos de forma a simplificar tu código y tu vida como programador de JavaScript. Evidentemente, hay otros métodos relacionados con la iteración de vectores que también te pueden ayudar en tu jornada. ¡Te animo a estudiarlos en W3schools o en Mozilla!

¿De qué forma utilizas los seis métodos de iteración?

Fuentes:

- Recursos para desarrolladores, creados por desarrolladores de Mozilla
- JavaScript Array Iteration Methods en w3schools