ntander Coders 2024.1 | Front-End | #1178

usca simples em vetores e matrizes

Isaias Soares
Id: 1178008

Busca simples em vetores e matrizes

find

A função `find()` permite procurar a existência de um item em um array. Em tópicos anteriores usamos `indexOf()` para esse propósito, no entanto, `indexOf()` só funciona para arrays com tipos primitivos, como números ou strings. Se tivermos um array de objetos e desejarmos procurar um objeto que tem determinado ID, por exemplo, o `indexOf()` já não é capaz de nos ajudar.

Para fazer essas buscas mais complexas passamos ao `find()` uma função de busca, se ela retornar `true` isso quer dizer que o `find()` deve retornar aquele objeto pois é o que procuramos, se retornar `false` quer dizer que o objeto não é o que procuramos e o `find()` passará para o próximo.

Exemplo: Array de contatos (busca por nome)

```
const contatos = [
    { nome : 'contato1', telefone : '000000000'},
    { nome : 'contato2', telefone : '111111111'},
    { nome : 'contato3', telefone : '222222222'},
    { nome : 'contato4', telefone : '333333333'},
```

```
    { nome : 'contato5', telefone : '444444444'},
];

const contato = contatos.find(c => c.nome === "contato3");
console.log(contato); // -> Object { nome: "contato3", telefone: "222222222" }
```

A função passada ao `find()` recebe um parâmetro, que é o item atual que o `find()` está testando. A partir de um teste usando esse item devemos retornar `true` ou `false`. É aconselhável que o teste seja capaz de ser `true` apenas para um item, mas isso não é obrigatório. O `find()` retornará o primeiro item que satisfizer a função de busca. Caso nenhum item satisfaça o teste, teremos `undefined` como retorno.

Podemos sem problemas criar a função antes e utilizá-la no `find()` também, para o caso de fazermos buscas mais complexas e reutilizáveis.

findIndex

`findIndex()` funciona exatamente da mesma forma que o `find()` com uma única diferença, ao invés de retornar o objeto encontrado ele retorna a posição (index) desse objeto no array. Se você substituir nos exemplos acima verá o resultado.

Vale dizer que essas funções são mais úteis quando os itens são objetos, no entanto, nada impede seu uso para arrays de tipos como number e string.

Exemplo: busca em um array numérico

```
const nums = [10,20,30];
const indexOf20 = nums.findIndex(n => n === 20);
console.log(indexOf20); // -> 1
```

A única diferença é que usamos o próprio valor na comparação.

every

`every()` à primeira vista não parece uma função interessante, mas é. O que ele faz é aplicar uma função a todos os itens do array que retorne um booleano (parecido com `find()`), mas ele reduz os resultados usando o operador `&&`.

Imagine que tenho a função `x => x === 10` que verifica se um valor é 10. Se aplicada a um array `[10, 20, 30]` essa função produziria `[true, false, false]`, no entanto, se ela for aplicada ao array `[10, 10]` ela produziria `[true, true]`. Se aplicarmos o operador `&&` nos resultados, teremos `true && false && false` que produz `false` e `true && true` que produz `true`.

É assim que o `every()` funciona. Se todos os valores estiverem de acordo com o critério da função passada por parâmetro, seu retorno será `true`. Por outro lado, se ao menos um dos valores não estiver de acordo com esse critério, ele produzirá `false`.

Exemplo: Todos os números do array são pares?

```
const nums1 = [10, 20, 30, 40, 50];
const nums2 = [11, 20, 30, 40, 50];

const isEven = num => num % 2 === 0;

const nums1IsEven = nums1.every(isEven);
const nums2IsEven = nums2.every(isEven);

console.log(nums1IsEven); // -> true
console.log(nums2IsEven); // -> false
```

filter

Lembra do `find()` permite que encontrássemos um item dentro de um array, para isso você tinha que escrever uma função que retorna um booleano e essa função podia ter até 3 parâmetros valor, index e array. `filter()` é muito parecido, mas com ele você

não busca necessariamente 1 item, mas todos aqueles que retornarem `true`. O retorno de `filter()`, portanto, é sempre um array.

Para formalizar, dado um array, filter copiará para um novo array apenas aqueles elementos onde a função de filtragem retornar `true` (ou truthy).

Exemplo: Filtrando os pares

```
const nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const evens = nums.filter(x => x % 2 === 0);
console.log(evens); // -> [ 2, 4, 6, 8, 10 ]
```

Uma técnica interessante de uso de filter é filtrar pelos elementos avaliados como truthy. Isso remove qualquer valor falsy. Só tenha cuidado porque remove também 0 e strings vazias, então se precisar desses valores faça uma função que leve isso em conta.

Exemplo: Filtrando usando truthy

```
const arr = [null, undefined, 0, false, 10, 20, 30, 40];
const result = arr.filter(x => x);
console.log(result); // -> [ 10, 20, 30, 40 ]
```

Esse exemplo é bem útil para remover itens `undefined` de uma lista.

Observe que a função é muito simples: `x => x`. Pode parecer contraintuitivo, mas quando x é convertido para booleano ele se torna o valor truthy/falsy equivalente.

Imagine que filter seja algo assim:

```
const filter = (fn, arr) => {
    let result = [];
```

```
    for(let index in arr){
        const ok = fn(arr[index], index, arr);

        if(ok)
            result.push(arr[index]);
    }

    return result;
}

//uso

const nums = [10, 20, 30, undefined, 50];
const result = filter(x => x);
console.log(result); // -> [ 10, 20, 30, 50 ]
```

O legal dessa versão é que você pode colocar alguns `console.log` dentro para ver como ela funciona. O `filter()` real tem mais rigor, validações, e em tese, deveria usar recursão. Então depois de fazer suas experiências com esse, volte a usar o "oficial".

Vou fazer em um array de objetos também, para ficar de exemplo.

Exemplo: filtrando objetos (arrow function)

```
const alunos = [
    {nome : "aluno1", nota : "8.0"},
    {nome : "aluno2", nota : "5.0"},
    {nome : "aluno3", nota : "4.0"},
    {nome : "aluno4", nota : "9.0"},
    {nome : "aluno5", nota : "7.5"},
    {nome : "aluno6", nota : "2.5"},
```

```
];

const notaCorte = 6.0;

const aprovados = alunos.filter(x => x.nota >= notaCorte);

console.log(aprovados);

/*Saída
[
    Object { nome: "aluno1", nota: "8.0" }
    Object { nome: "aluno4", nota: "9.0" }
    Object { nome: "aluno5", nota: "7.5" }
]
*/
```

E se no exemplo acima eu quisesse apenas a lista dos nomes dos alunos que reprovaram, não os objetos completos? A gente já sabe que `filter()` apenas filtra, e precisamos transformar os valores. Esse é um trabalho para o `map()`.

Exemplo: filtrando objetos (arrow function)

```
const alunos = [
    {nome : "aluno1", nota : "8.0"},
    {nome : "aluno2", nota : "5.0"},
    {nome : "aluno3", nota : "4.0"},
    {nome : "aluno4", nota : "9.0"},
    {nome : "aluno5", nota : "7.5"},
    {nome : "aluno6", nota : "2.5"},
];
```

```
const notaCorte = 6.0;

const reprovados = alunos.filter(x => x.nota < notaCorte);
const nomes = reprovados.map(x => x.nome);

console.log(nomes); // -> [ "aluno2", "aluno3", "aluno6" ]
```

Referências e Materiais Complementares

- Array

Próximo Tópico