Entendendo Programação Funcional em JavaScript de uma vez

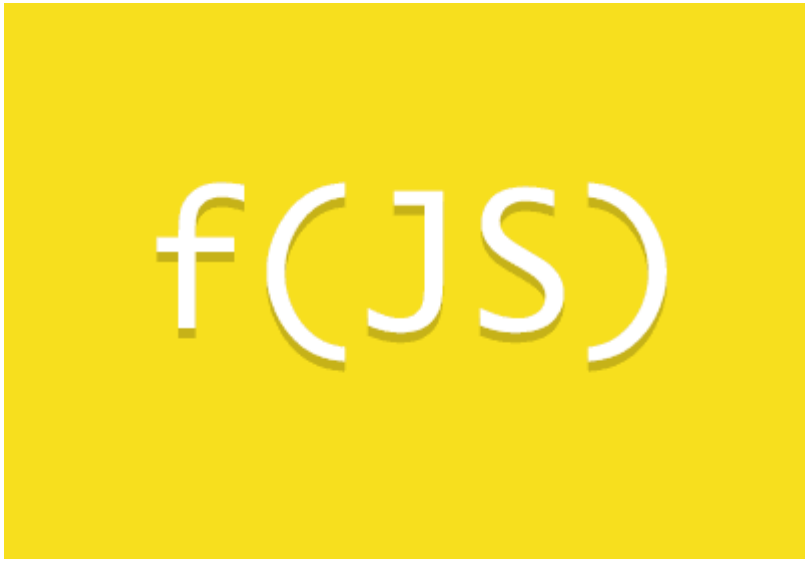
Matheus Lima Follow
Mar 3, 2016 · 6 min read

Você já percebeu que cada vez mais o termo **Programação Funcional** vem sendo usado pela comunidade?

No meu último *post,* por exemplo: O que TODO desenvolvedor JavaScript precisa saber, um dos pontos que gerou mais dúvida foi justamente o da Programação Funcional.

Continue lendo esse artigo para aprender:
1. Quais as vantagens de usar a Programação Funcional
2. Como usá-la tanto em ES5 quanto em ES6

3. O que são *Pure Functions* e *Higher-Order Functions*
4. Qual a diferença entre *Map*, *Filter* e *Reduce*
5. O que é *Currying*
6. Como compor funções de maneira eficaz



Para entender as verdadeiras motivações, temos obrigatoriamente que voltar aos conceitos básicos.

A função abaixo possui *inputs* e *outputs* bem definidos:

```
1  function square(x) {
2      return x * x;
```

```
}

square(2); // 4
```

square.js hosted with ♡ by GitHub view raw

Ela recebe como parâmetro uma variável **x** e retorna um *int* que é a multiplicação de **x** com ele mesmo.

A função abaixo porém, não possui *inputs* e *outputs* tão bem definidos:

```
function generateDate() {
    var date = new Date();
    generate(date);
}

generateDate(); // ???
```

date.js hosted with ♡ by GitHub view raw

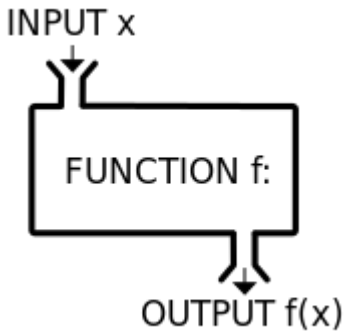
Ela não recebe nada como parâmetro e retorna o que parece ser uma data processada mas não temos como ter certeza.

Um ponto que vale ser reforçado: só porque não declaramos explicitamente os *inputs* e *outputs* dessa função não quer dizer que a mesma não os tenha.

Eles apenas estão ocultos. E isso pode gerar um dos piores problemas nas aplicações modernas: os Efeitos Colaterais (side-effects).

Funções como as de cima que possuem *inputs* e *outputs* ocultos e podem gerar *side-effects* são chamadas de Funções Impuras (impure functions). Outra característica importante delas é que se invocarmos uma função impura diversas vezes, o retorno dela nem sempre será o mesmo. O que dificulta a manutenção e os testes na sua aplicação.

Funções Puras (pure functions) por outro lado, como o primeiro exemplo desse *post*, tem *inputs* e *outputs* declarados e não geram *side-effects*. Além disso, o retorno de uma função pura dado um parâmetro será sempre o mesmo. Obviamente os seus testes serão mais fáceis de desenvolver, assim como a manutenção da sua aplicação.



Pure Functions

E por que estamos falando sobre isso?

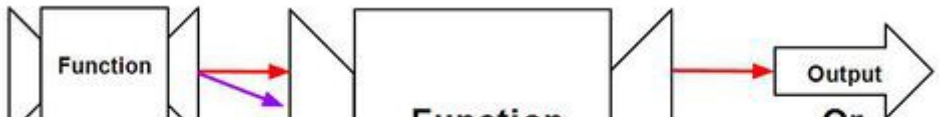
Porque escrever funções puras e remover *side-effects* é a base da Programação Funcional.

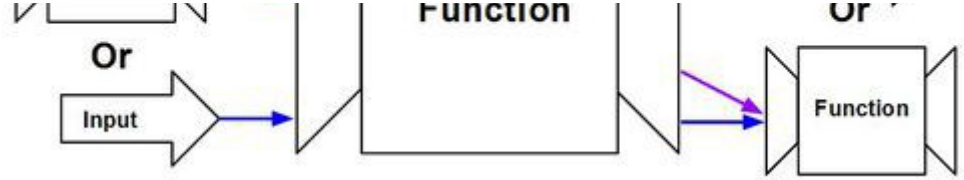
. . .

Agora que temos uma ideia melhor da motivação de se usar Programação Funcional, podemos começar a ver os casos reais e aprender na prática como usá-la.

1) Higher-Order Functions

Matematicamente falando, funções que operam sobre outras funções ou as recebendo como parâmetro ou as retornando são chamadas de Higher-Order Functions.





Higher-Order Functions

Essas funções nos permitem fazer abstrações não apenas de valores mas também de ações, como no exemplo abaixo:

```
var calculate = function(fn, x, y) {
    return fn(x, y);
};
```

calculate.js hosted with ♡ by GitHub view raw

A função **calculate** recebe três parâmetros. O primeiro é uma função qualquer que será invocada passando como parâmetro **x** e **y**.

Pensando em um cenário em que precisamos tanto de uma soma quanto de uma multiplicação, podemos pensar na solução dessa forma em ES5:

```
var sum = function(x, y) {
    return x + y;
};
```

```
4
5  var mult = function(x, y) {
6      return x * y;
7  };
8
9  calculate(sum, 2, 5); // 7
10 calculate(mult, 2, 5); // 10
```

calculateES5.js hosted with ♡ by GitHub view raw

Ou dessa, bem mais curta, em ES6:

```
1  const sum = (x, y) => x + y;
2  const mult = (x, y) => x * y;
3
4  calculate(sum, 2, 5); // 7
5  calculate(mult, 2, 5); // 10
```

calculateES6.js hosted with ♡ by GitHub view raw

Higher-order functions estão em todos os lugares no ecossistema do JavaScript. Se você já usou testes unitários com Jasmine ou Mocha, então o trecho abaixo deve ser familiar:

```
1  describe("A suite", function() {
2      it("contains spec with an expectation", function() {
3          expect(true).toBe(true);
```

```
    });
});
```

jasmine.js hosted with ♡ by GitHub view raw

Percebemos que o segundo parâmetro tanto de **describe** quanto de **it** são funções. Por definição ambas são *higher-order functions*.

Outro exemplo também é o bom e velho jQuery. Podemos perceber que praticamente todo o código gerado por ele era composto de *higher-order functions*, como o exemplo abaixo:

```
$(btn1).click(function() {
    doSomething();
});

$(btn2).bind("click", function() {
    doSomethingElse();
});

$(document).ajaxStart(function() {
    $(log).text("Triggered ajaxStart handler.");
});
```

jquery.js hosted with ♡ by GitHub view raw

Em aplicações desenvolvidas com AngularJS também não é diferente, observe atentamente a definição de um *controller*:

```
var app = angular.module('app');
app.controller('MyController', function() {
    /* */
});
```

angular.js hosted with ♡ by GitHub view raw

Agora que já temos conhecimento dos fundamentos: *pure functions* e *higher-order functions* podemos nos aprofundar um pouco mais…

2) Map

A função **map** invoca um *callback* e retorna um novo *array* com o resultado desse *callback* aplicado em cada item do *array* inicial.

Imaginando um cenário em que temos um *array* de inteiros e precisamos do quadrado de cada valor desse *array*, podemos fazer dessa forma bem simples usando a função **map** em ES5:

```
var numbers = [1, 2, 3];
```

```
2
3  var square = function(x) {
4      return x * x;
5  };
6
7  var squaredNumbers = numbers.map(square); // [1, 4, 9]
```

mapES5.js hosted with ♡ by GitHub view raw

Ou assim em ES6:

```
1  const numbers = [1, 2, 3];
2  const square = x => x * x;
3  const squaredNumbers = numbers.map(square); // [1, 4, 9]
```

mapES6.js hosted with ♡ by GitHub view raw

Nesse outro cenário abaixo, percebemos o reaproveitamento de código que podemos conseguir ao usar o **map**.

Possuímos dois *arrays* de objetos diferentes, porém ambos tem o campo *name*, e precisamos de uma função que retorne um novo *array* apenas com os *names* dos objetos:

```
1  var students = [
```

```
    { name: 'Anna', grade: 6 },
    { name: 'John', grade: 4 },
    { name: 'Maria', grade: 9 }
];

var teachers = [
    { name: 'Mark', salary: 2500 },
    { name: 'Todd', salary: 3700 },
    { name: 'Angela', salary: 1900 }
];

var byName = function(object) {
    return object.name;
};

var byNames = function(list) {
    return list.map(byName);
};

byNames(students); // ["Anna", "John", "Maria"]
byNames(teachers); // ["Mark", "Todd", "Angela"]
```

map2ES5.js hosted with ♡ by GitHub view raw

Em ES6:

```
const students = [
    { name: 'Anna', grade: 6 },
    { name: 'John', grade: 4 },
    { name: 'Maria', grade: 9 }
];
```

```
6
7  const teachers = [
8      { name: 'Mark', salary: 2500 },
9      { name: 'Todd', salary: 3700 },
10     { name: 'Angela', salary: 1900 }
11 ];
12
13 const byName = object => object.name;
14 const byNames = list => list.map(byName);
15
16 byNames(students); // ["Anna", "John", "Maria"]
17 byNames(teachers); // ["Mark", "Todd", "Angela"]
```

map2ES6.js hosted with ♡ by GitHub view raw

Podemos melhorar ainda mais esse trecho de código alterando as funções *byName* e *byNames* para que o atributo *name* não esteja mais tão acoplado. Podemos simplesmente receber como parâmetro qualquer atributo e aplicá-lo a função. Fica como exercício :)

3) Filter

A função **filter** é bem semelhante ao *map*: ela também recebe um *callback* como parâmetro e também retorna um novo *array*, a única diferença é que **filter**, como o próprio nome diz, retorna um filtro dos elementos do *array* inicial baseado na função de *callback*.

Imaginando que temos um *array* de inteiros e desejamos retornar apenas aqueles que são maiores do que 4. Podemos resolver assim usando o **filter** com ES5:

```
var numbers = [1, 4, 7, 10];

var isBiggerThanFour = function(value) {
    return value > 4;
};

var numbersBiggerThanFour = numbers.filter(isBiggerThanFour); // [7, 10]
```

filterES5.js hosted with ♡ by GitHub view raw

Ou com ES6:

```
const numbers = [1, 4, 7, 10];
const isBiggerThanFour = value => value > 4;

const numbersBiggerThanFour = numbers.filter(isBiggerThanFour); // [7, 10]
```

filterES6.js hosted with ♡ by GitHub view raw

Outro exercício é uma melhoria na função **isBiggerThanFour**, deveríamos alterá-la para receber como parâmetro qualquer inteiro que desejamos

fazer a comparação.

4) Reduce

Uma das funções que mais gera dúvidas é o **reduce**. Ele recebe como parâmetro um *callback* e um valor inicial, com o objetivo de reduzir o array a um único valor. O cenário mais comum para explicar o **reduce** é uma soma:

```
var numbers = [1, 2, 3];

var sum = function(x, y) {
    return x + y;
};

var numbersSum = numbers.reduce(sum, 0); // 6
```

reduceES5.js hosted with ♡ by GitHub view raw

Com ES6 seria assim:

```
const numbers = [1, 2, 3];
const sum = (x, y) => x + y;
const numbersSum = numbers.reduce(sum, 0); // 6
```

reduceES6.js hosted with ♡ by GitHub view raw

O primeiro parâmetro é a função que será aplicada, no caso uma soma. E o segundo parâmetro é o valor inicial. Se por algum motivo precisássemos começar a soma com 10, faríamos dessa forma:

```
const numbers = [1, 2, 3];
const sum = (x, y) => x + y;
const numbersSum = numbers.reduce(sum, 10); // 16
```

reduce2ES6.js hosted with ♡ by GitHub view raw

Mas o **reduce** não serve apenas para somas, podemos também trabalhar com *strings*. Imaginando que nós temos um *array* de meses e precisamos retornar o meses dessa forma: **JAN/FEV/MAR ... / DEZ**.
Podemos fazer assim:

```
var months = ['JAN', 'FEV', 'MAR', /*...*/ , 'DEZ'];

var monthsShortener = function(previous, current) {
    return previous + '/' + current;
};

var shortenedMonths = months.reduce(monthsShortener, '');
// /JAN/FEV/MAR ... /DEZ
```

reduceMonths.js hosted with ♡ by GitHub view raw

Não era bem o que a gente queria inicialmente.
Nós queríamos isso: **JAN/FEV/MAR … / DEZ**
Mas obtivemos isso: **/JAN/FEV/MAR … / DEZ**

Devemos alterar nossa função *monthsShortener* para adicionar uma condição que faça a prevenção desse erro:

```
var months = ['JAN', 'FEV', 'MAR', /*...*/ , 'DEZ'];

var monthsShortener = function(previous, current) {
    if (previous === '') {
        return current;
    } else {
        return previous + '/' + current;
   }
};

var shortenedMonths = months.reduce(monthsShortener, '');
// JAN/FEV/MAR ... /DEZ
```

reduceMonths2.js hosted with ♡ by GitHub view raw

Feito! E também na versão em ES6:

```
const months = ['JAN', 'FEV', 'MAR', /*...*/ , 'DEZ'];

const monthsShortener = (previous, current) => {
    if (previous === '') {
        return current;
    } else {
        return previous + '/' + current;
   }
};

const shortenedMonths = months.reduce(monthsShortener, '');
// JAN/FEV/MAR ... /DEZ
```

reduceMonthsES6.js hosted with ♡ by GitHub view raw

[UPDATE — 03/04/2016]

Verificar o comentário do Marcus Tenório para um algoritmo melhor

5) Currying

A técnica de transformar uma função com múltiplos parâmetros em uma sequência de funções que aceitam apenas um parâmetro é chamada de Currying.

Se na teoria ficou confuso, na prática seria transformar isso:

```
var add = function(x, y) {
   return x + y;
};

add(1, 2) // 3
```

noCurrying.js hosted with ♡ by GitHub view raw

Nisso:

```
var add = function(x) {
    return function(y) {
        return x + y;
    };
};

add(1)(2); // 3
```

currying.js hosted with ♡ by GitHub view raw

A princípio parece que estamos apenas adicionando mais dificuldade sem nenhum ganho. Porém temos uma grande vantagem: transformar 0 código em pequenos pedaços mais expressivos e com maior reuso.

Pensando em uma aplicação que possui diversos trechos do código uma soma com 5 e outra com 10, podemos usar a segunda versão da função *add*

dessa forma:

```
var addFive = add(5);
var addTen = add(10);

addFive(3); // 8
addFive(1); // 6

addTen(1); // 11
addTen(10); //20
```

curryingES5.js hosted with ♡ by GitHub view raw

Mais um exemplo seria um *Hello World* simples com uma *curried function*. Podemos implementá-lo desse jeito com ES5:

```
var greeting = function(greet) {
    return function(name) {
        return greet + ' ' + name;
    };
};

var hello = greeting('Hello');
hello('World'); // Hello World
hello('Matheus'); // Hello Matheus
```

currying2ES5.js hosted with ♡ by GitHub view raw

Ou com ES6:

```
1  const greeting = greet => name => greet + ' ' + name;
2  const hello = greeting('Hello');
3
4  hello('World'); // Hello World
5  hello('Matheus'); // Hello Matheus
```

currying2ES6.js hosted with ♡ by GitHub view raw

6) Compose

Podemos compor funções pequenas para gerar outras mais complexas de forma bem fácil em JavaScript. A vantagem é o poder de usar essas funções mais complexas, de forma simples, em toda aplicação. Ou seja, aumentamos o reuso.

Por exemplo, em uma aplicação em que necessitamos de uma função para transformar uma *string* passada pelo usuário em um grito: mudar para caracteres maiúsculos e adicionar uma exclamação no final. Podemos fazer assim em ES5:

```
1  var compose = function(f, g) {
```

```
    return function(x) {
        return f(g(x));
    };
};

var toUpperCase = function(x) {
    return x.toUpperCase();
};

var exclaim = function(x) {
    return x + '!';
};

var angry = compose(toUpperCase, exclaim);

angry('ahhh'); // AHHH!
```

composeES5.js hosted with ♡ by GitHub view raw

Ou em ES6:

```
const compose = (f, g) => x => f(g(x));

const toUpperCase = x => x.toUpperCase();
const exclaim = x => x + '!';

const angry = compose(toUpperCase, exclaim);

angry('ahhh'); // AHHH!
```

composeES6.js hosted with ♡ by GitHub view raw

. . .

Se você gostou do *post* não se esqueça de dar um ❤ aqui embaixo!
E se quiser receber de antemão mais *posts* como esse, assine nossa newsletter.

Seja um apoiador, doe BitCoins: 1BGVKwjwQxkr3w1Md2X8WHAsyRjDjyJiPZ

JSCasts

É difícil encontrar conteúdo bom e atualizado em português. Com isso em mente criamos o JSCasts, onde você vai se manter em dia com o JavaScript e todo o seu ecossistema de forma fácil e interativa.

Cursos:

- Começando com React.js
- React.js com ES6

Obrigado por ler! ♥

ES6 JavaScript Functional Programming