Artigo

Programação funcional com JavaScript

Esse artigo introduz os conceitos de programação funcional no universo JavaScript, assunto muito abordado em outras plataformas e em alta no momento. Traz desde os conceitos básicos até customização de funções no JavaScript.

Introdução a programação em Jav

O JavaScript foi criado como uma linguagem de programação de scripts baseada essencialmente em funções. Até mesmo os conceitos núcleo da orientação a objetos (classes, objetos, herança etc.), perfeitamente possíveis de serem aplicados na linguagem, são implementados via **funções no JavaScript.** Mas talvez a principal característica que a fez se tornar funcional tenha sido sua natureza *runtime*, isto é, seus códigos são executados em tempo de execução e não de compilação como temos em outras linguagens famosas, como o Java. Não compilamos **código em JavaScript** e, como já discutido por inúmeros outros artigos, isso pode ser um ponto negativo da linguagem em alguns casos, já que camufla determinados erros que só acontecem em situações especiais e quando o usuário já está usando a aplicação. Por essa razão, o uso de frameworks de testes (unitários, integrados e automatizados) se faz tão necessário. Mas isso é assunto para outro artigo.

Relacionado: JavaScript Replace

A essência da programação funcional, um conceito genérico associado e implementado por muitas outras linguagens (vide a recente adição no Java, via lambas, por exemplo), é extremamente importante na programação dos dias de hoje, porque adiciona um grande poder à estrutura e arquitetura de nossos projetos. E o JavaScript não poderia estar atrás, já que galga cada vez mais

portáveis, além de principal escolha para a maioria dos *frameworks híbridos* do mercado. Neste artigo, trataremos de expor as principais características da programação funcional no universo JavaScript, tomando como biblioteca auxiliar o Underscore.js (vide seção **Links** para site de download), que dispõe de uma série de funções utilitárias que facilitam esse tipo de implementação.

Se você tiver o mínimo de experiência em JavaScript, provavelmente já terá visto algo como o código a seguir:

```
[1, 2].forEach(alert);
// exibe um alerta com o valor 1
// exibe um alerta com o valor 2
```

Aprenda mais sobre JavaScript

- Curso JavaScript
- Desafio: PHP + Ajax + JavaScript
- Mensagens de erro com abstração em JavaScript
- Guia de referência JavaScript

O método `Array#forEach`, adicionado na quinta edição padrão da linguagem ECMA-262, recebe uma função (no caso, `alert`) e passa cada elemento do vetor para a função, um após o outro.

O seu modelo de execução é tão flexível a ponto de fornecer o famoso método apply a todas as funções no JavaScript. Esse método, por sua vez, permite que [illegible] em um array desde que os elementos desse array sejam os argumentos da função em si, isto é, por meio dessa função auxiliar podemos enviar um vetor e transformar cada um dos elementos dele em parâmetros para a função que estiver imediatamente interna à função original. Vejamos um exemplo na **Listagem 1**. Nela, temos uma função principal de nome multiplicar(), que recebe a função a ser considerada em tempo de execução (veja que ela é genérica o suficiente para receber qualquer tipo de função que receba um **array** como parâmetro, dessa forma, não definimos o "que" vai acontecer dentro dela, apenas recebemos o comportamento respectivo) e retorna uma outra função que usa o array a ser passado no momento de sua chamada para aplicar a função original `apply`. Note que o valor null que passamos como primeiro argumento dirá ao JavaScript que todo valor nulo deverá ser substituído pelo objeto global (`this`) da função.

Listagem 1. Exemplo de função usando a utilitária apply().

```
function multiplicar(funcao) {
       return function(array) {
             return funcao.apply(null, array);
       };
}


var multiplicarElementosArray = multiplicar(function(x, y) { return x * y });
```

```
// Saída: 4
```


Esse, portanto, é nosso primeiro contato com a programação funcional: uma função que retorna outra. O ponto é que o JavaScript fornece inúmeras outras funções utilitárias que podemos usar para flexibilizar nossa programação. Vejamos um segundo exemplo demonstrado na **Listagem 2.** Nele, temos exatamente o comportamento inverso da listagem anterior, isto é, a função `desmanchar()` também recebe uma função e devolve outra, porém dessa vez essa recebe qualquer número de argumentos de quaisquer tipos distintos e chama a função original com um **array**. Veja que dessa vez estamos fazendo uso da função `call` do JavaScript, que efetua praticamente o mesmo papel de `apply`, a única diferença é que ela aceita quaisquer quantidade/tipo de argumentos, enquanto `apply` aceita apenas um array como parâmetro.

Listagem 2. Exemplo de função usando a utilitária call().

```
function desmanchar(funcao) {
	return function() {
		return funcao.call(null, _.toArray(arguments));
	};
}

var juntarElementos = desmanchar(function(array) { return array.join(' ') });

juntarElementos(2, 2);
// Saída: "2 2"
```

Funções como unidades de abstração

Métodos de abstração são funções que ocultam os detalhes de implementação das mesmas. Por exemplo, quando precisamos implementar algum tipo de mecanismo de gerenciamento e exibição de erros de report e/ou avisos sistêmicos, poderíamos escrever algo como o que temos na **Listagem 3.**

Listagem 3. Exemplo de função para tratamento de erros/avisos.

```
function converterIdade(idade) {
 if (!_.isString(idade)) throw new Error("Uma string era esperada");
 var a;
 console.log("Tentativa de converter uma idade ");
 a = parseInt(idade, 10);
 if (_.isNaN(a)) {
   console.log(["Não pode converter a idade:", idade].join(' '));
   a = 0;
 }
 return a;
}
```

Essa função, embora não seja abrangente o suficiente para converter strings de idade, é bem ilustrativa. O uso da função `converterIdade` pode se dar da seguinte maneira:

```
converterIdade("abc");
```


O primeiro exemplo recebe um valor em string e efetua a conversão com sucesso, exibindo o resultado *42* no console. O segundo, por sua vez, lança um *Error: Uma string era esperada* em vista do tipo de dado do parâmetro não ter sido enviado corretamente. Por fim, o último exemplo lança uma mensagem "Não pode converter a idade: abc" justamente porque o dado passado sequer representa um número, seja em string ou tipo numérico.

A função `converterIdade` funciona como está escrito, mas se você deseja modificar a maneira em que os erros, informações e avisos são apresentados em seguida, as mudanças precisam ser feitas nas linhas apropriadas, e em qualquer outro lugar onde padrões semelhantes são usados. Uma abordagem melhor é "abstrair" a noção de erros, informações e avisos em funções, padronizando assim o código, tal como temos na **Listagem 4**.

Listagem 4. Exemplo de funções para tratar erros e mensagens.

```
function fail(msg) {
 throw new Error(msg);
}
function warn(msg) {
 console.log(["AVISO:", msg].join(' '));
}
function note(msg) {
```




Veja que apenas encapsulamos o comportamento que a[illegible]sim reusar tais funções em qualquer lugar ao longo da nossa implementação. Usando essas funções, a função `converterIdade` pode ser reescrita da mesma forma que vemos na **Listagem 5.**

Listagem 5. Nova função converterIdade com uso das funções de erro/aviso.

```
function converterIdade(idade) {
 if (!_.isString(idade)) fail("Uma string era esperada");
 var a;
 note("Tentativa de converter uma idade ");
 a = parseInt(idade, 10);
 if (_.isNaN(a)) {
   warn(["Não pode converter a idade:", idade].join(' '));
   a = 0;
 }
 return a;
}
```

Portanto, ao tentar executar o novo código, seu comportamento se dará de forma semelhante, porém com logs mais apropriados:

```
converterIdade("frob");
// (console) AVISO: Não pode converter a idade: abc
// Saída: 0
```

Não é muito diferente do antigo comportamento, exceto que agora a ideia de erros de report informações e advertências foram abstraídas. O report [illegible] pode, assim, ser modificado completamente a bel prazer, tal como vemos na **Listagem 6**.

Listagem 6. Modificando comportamento das mensagens.

```
function note() {}
function warn(msg) {
 alert("Isso não parece uma idade válida");
}
converterIdade("abc");
// (alert box) Isso não parece uma idade válida
// Saída: 0
```

Portanto, já que o comportamento está contido dentro de uma única função, podemos substituí-la por novas funções que fornecem um comportamento semelhante ou diferente quando usadas em conjunto.

Encapsulando e escondendo

Ao longo dos anos, fomos ensinados que um dos pilares da programação orientada a objetos é o encapsulamento. O termo encapsulamento em referência à programação orientada a objetos refere-

se a uma forma de embalar certas partes dos dados com as mesmas operações que os manipulam, como pode ser visto na **Figura 1.**

Figura 1. Exemplo de entidade encapsulada

Como vimos, a maioria das linguagens orientadas a objeto usam o próprio objeto para embalar os elementos de dados com as operações que trabalham neles; uma classe *Pilha*, portanto, empacota um vetor de elementos com operações `push`, `pop` e `peek` usadas para manipulá-los. O JavaScript fornece um sistema de objetos que, de fato, permite encapsular dados com seus manipuladores. No entanto, por vezes, o encapsulamento é usado para restringir a visibilidade de certos elementos, e esse ato é conhecido como "ocultação de dados". O sistema de objetos do JavaScript fornece uma maneira de esconder dados diretamente, assim os dados são escondidos usando uma estrutura chamada de `closure` (funções criadas dentro de outras funções que podem ser retornadas e alocadas em variáveis para uso posterior no JavaScript), como mostrado na **Figura 2.**

Figura 2. Closure para encapsular os dados e ocultar detalhes da view de um cliente.

Ao usar técnicas funcionais que envolvam *closures*, você pode conseguir esconder dados tão eficazmente quanto a mesma capacidade oferecida pela maioria das linguagens orientadas a objetos. Todavia, enquanto eles são diferentes na prática, ambos fornecem maneiras semelhantes de construção de certos tipos de abstração. De fato, este artigo não intenciona encorajá-lo a jogar fora tudo o que já aprendeu a favor da programação funcional; em vez disso, é destinado a explicar alguns itens da programação funcional em seus próprios termos para que você mesmo decida se é a melhor opção para suas necessidades.

Funções como unidade de comportamento

Esconder dados e comportamentos (que tem o efeito colateral de fornecer uma mudança mais ágil de experiência) é apenas uma das maneiras em que funções podem ser unidades de abstração. Outra é o fornecimento de uma maneira fácil de armazenar unidades discretas de comportamento básico.

Tomemos, por exemplo, a sintaxe JavaScript para denotar a procurar um valor em uma matriz pelo índice:

```
var letras = ['a', 'b', 'c'];
letras[1];
// Saída: 'b'
```

Já que a indexação de matriz é um comportamento núcleo do JavaScript, não há nenhuma maneira de agarrar o comportamento e usá-lo em uma função. Portanto, um simples exemplo de uma função que abstrai o comportamento de indexação da matriz poderia ser chamado de `nth`. A implementação de `nth` é a seguinte:

```
function nth(a, index) {
 return a[index];
}
```

Como você pode observar, a `nth` opera ao longo do caminho mais adequado:

```
nth(letras, 1);
// Saída: "b"
```

No entanto, a função irá falhar se for passado algo inesperado:

```
nth({}, 1);
//Saída: undefined
```


Portanto, se pensarmos sobre a abstração em torno de uma função `nth`, poderíamos conceber a seguinte declaração: `nth` retorna o elemento localizado em um índice válido dentro de um tipo de dados que permite o acesso indexado. Uma parte fundamental dessa declaração é a ideia de um tipo de dados *indexado*. Para determinar se algo é um tipo de dados indexado, podemos criar uma função `isIndexed`, implementada como segue:

```
function isIndexed(dado) {
 return _.isArray(dado) || _.isString(dado);
}
```

A função também fornece uma abstração sobre como verificar se o dado é uma *string* ou um *array*. Construir abstração sobre abstração nos leva à implementação completa de `nth`, tal como vemos na **Listagem 7**.

Listagem 7. Função nth completa.

```
function nth(a, index) {
 if (!_.isNumber(index)) fail("Um número era esperado como index");
 if (!isIndexed(a)) fail("Não suporta em tipos não-indexados");
 if ((index < 0) || (index > a.length - 1)) fail("O valor do index está fora dos limites ");
```




Portanto, a implementação completa do `nth` funciona c ...ger...

Listagem 8. Testando função nth completa.

```
nth(letras, 1);
// Saída: 'b'
nth("abc", 0);
// Saída: "a"
nth({}, 2);
// Erro: Não suporta em tipos não-indexados
nth(letras, 4000);
// Erro: O valor do index está fora dos limites
nth(letras, 'aaaaa');
// Erro: Um número era esperado como index
```

Da mesma forma que construímos uma abstração `nth` de uma abstração indexada, podemos também construir uma segunda abstração:

```
function segunda(a) {
 return nth(a, 1);
}
```

A segunda função permite apropriar-se do comportamento correto de `nth` e transcrevê-lo para um diferente, mas relacionado ao caso de uso (**Listagem 9**).

Listagem 9. Testando função segunda().


```
segunda(['a','b']);
// Saída: "b"
segunda("fogo");
// Saída: "o"
segunda({});
// Erro: Não suporta em tipos não-indexados
```

Outra unidade de comportamento básica em JavaScript é a ideia de um `comparator` (comparador). Um comparador é uma função que leva dois valores e retorna <1 se o primeiro for menor do que o segundo, >1 se for maior e 0 se eles forem iguais. De fato, o próprio JavaScript pode usar a natureza dos números para fornecer um método de classificação padrão:

```
[3, 2, -7, 0, -108, 42].sort();
// Saída: [-108, -7, 0, 2, 3, 42]
```

Mas surge um problema quando você tem um mix de diferentes números:

```
[1, 3, -1, -6, 0, -100, 22, 20].sort();
// Saída: [-1, -100, -6, 0, 20, 1, 3, 22]
```

O problema é que quando nenhum argumento é passado, o método **Array#sort** faz uma comparação

vez disso, escreve um código igual ao que vemos na **Listagem 10**.

Listagem 10. Exemplo de ordenação dos valores passando uma função.

```
[1, 3, -1, -6, 0, -100, 22, 20].sort(function(x,y) {
 if (x < y) return -1;
 if (y < x) return 1;
 return 0;
});
// Saída: [-100, -6, -1, 0, 1, 3, 20, 22]
```

Essa implementação parece melhor, mas há uma maneira de torná-la mais genérica. Afinal de contas, pode ser necessário classificar novamente os dados em outra parte do código, então talvez seja melhor extrair a função anônima e dar-lhe um nome (**Listagem 11**).

Listagem 11. Função para comparar valores.

```
function compararMenorQueOuIgual(x, y) {
 if (x < y) return -1;
 if (y < x) return 1;
 return 0;
}
[2, 3, -1, -6, 0, -108, 42, 10].sort(compararMenorQueOuIgual);
// Saída: [-108, -6, -1, 0, 2, 3, 10, 42]
```

O problema com essa função é que ela está acoplada à ideia de "comparatorness" (conceito comum no JavaScript para descrever comportamentos que devem ser comparados uns aos outros, e não pode facilmente ser usada em prol de uma operação de comparação genérica:

```
if (compararMenorQueOuIgual(1,1))
 console.log("less or equal");
// nothing prints
```

Para conseguir o efeito desejado, seria necessário saber sobre o `compararMenorQueOuIgual` de natureza comparador:

```
if (_.contains([0, -1], compararMenorQueOuIgual(1,1)))
 console.log("menor or igual");
// menor ou igual
```

Veja que chamamos a função contains() do objeto global _ para checar se o vetor passado como parâmetro está contido no retorno da função `compararMenorQueOuIgual`. Mas isso é menos do que satisfatório, especialmente quando há uma possibilidade de algum desenvolvedor vir futuramente e mudar o valor de retorno da função `compararMenorQueOuIgual`. A melhor maneira de escrever `compararMenorQueOuIgual` pode ser a seguinte:

```
}
```


Funções que sempre retornam um valor booleano (isto é, verdadeiro ou falso, apenas), são chamadas predicados. Assim, em vez de uma elaborada construção comparadora, a função `menorOuIgual` é simplesmente uma "*skin*" sobre o operador <=:

```
[2, 3, -1, -6, 0, -108, 42, 10].sort(menorOuIgual);
// Saída: [42, 10, 3, 2, 0, -1, -6, -108]
```

Se a função `sort` espera um comparador, e a função `menorOuIgual` só retorna verdadeiro ou falso, então você precisa de alguma forma obter tal comportamento sem duplicar um monte de clichês *if/then/else*. A solução está na criação de uma função `comparator` , que leva um predicado e converte o seu resultado para o -1/0/1 esperado das funções `comparator` (**Listagem 12**). A função `truthy` usada na listagem será detalhada mais adiante.

Listagem 12. Função comparator com predicado.

```
function comparator(pred) {
 return function(x, y) {
   if (truthy(pred(x, y)))
     return -1;
   else if (truthy(pred(y, x)))
     return 1;
```




```
  };
};
```


Agora, a função de comparação pode ser utilizada para devolver uma função nova que mapeia os resultados do predicado `menorOuIgual` (ou seja, verdadeiro ou falso) sobre os resultados esperados dos comparadores (isto é, -1, 0 ou 1).

Na programação funcional, você verá quase sempre funções interagindo de uma forma que permite um tipo de dados ser trazido para o mundo de outro tipo de dados. Observe o `comparator` em ação:

```
[120, 3, 0, 20, -1, -4, -1].sort(comparator(menorOuIgual));
  // Saída: [-4, -1, -1, 0, 3, 20, 120]
```

Vale a pena notar que agora o `comparator` é uma função de ordem superior (isso porque é preciso uma nova função assim como o retorno dessa nova função). Tenha em mente que nem todos os predicados fazem sentido para o uso com a função de `comparator`, no entanto. Por exemplo, o que significa usar a função `_.isEqual` como base para um `comparator`? Tente ver o que acontece.

Dados como abstração

O modelo de objetos de protótipo JavaScript é um esquema de dados rico e fundacional, além de

flexibilidade não encontrado em muitas outras linguagens de programação tradicionais. No entanto, muitos programadores de JavaScript, como é de costume, [illegible] sistema de objetos baseados em classes usando o protótipo ou funcionalidades *closure* (ou ambos). Embora um sistema de classes tenha seus pontos fortes, muitas vezes o "modelo de dados" de uma **aplicação JavaScript** se faz mais simples do que ele.

Em vez disso, usando dados JavaScript primitivos, objetos e matrizes, grande parte dos dados e tarefas de modelagem que atualmente são servidos por classes são subsumidos. Historicamente, a programação funcional tem se centrado em torno de funções de construção que trabalham para alcançar comportamentos de nível superior e trabalhar em construções de dados muito simples. A flexibilidade nesses dois tipos de dados simples é surpreendente, é uma pena que eles são muitas vezes negligenciados em favor de mais um sistema baseado em classes. Imagine que você está encarregado de escrever um aplicativo JavaScript que lida com valores separados por vírgulas (CSV), que são uma forma padrão de representar tabelas de dados. Por exemplo, suponha que você tenha um arquivo CSV que é o seguinte:

```
nome, idade, cabelo
Marcio, 35, preto
João, 64, branco
```

Deve ficar claro que esse dado representa uma tabela com três colunas (nome, idade e cabelo) e três

analisar essa limitada representação CSV armazenada em uma string é implementada na **Listagem 13.**

Listagem 13. Função para mapear os valores do CSV.

```
function lidarComCSV(str) {
   return _.reduce(str.split("\n"), function(table, row) {
     table.push(_.map(row.split(","), function(c) { return c.trim()}));
     return table;
   }, []);
  };
```

Você irá notar que a função `lidarComCSV` processa as linhas, uma por uma, dividindo-as pela expressão \n (nova linha) e, em seguida, tirando o espaço em branco de cada célula no seu interior (via função `trim`). A tabela de dados inteira é uma matriz de submatrizes, cada uma contendo strings. Do ponto de vista conceitual mostrado na **Figura 3**, matrizes aninhadas podem ser vistas como uma tabela.

Figura 3. Matrizes simples aninhadas são uma forma de abstrair uma tabela de dados.

Para analisar os dados armazenados em uma string via função `lidarComCSV` podemos implementar o seguinte código:

```
var tabelaPessoas = lidarComCSV("nome,idade,cabelo\nMarcio,35,preto\nJoão,64,branco");
```

O resultado, consequentemente, será:

```
// [["nome", "idade", "cabelo"],
// ["Marcio", "35", "preto"],
// ["João", "64", "branco"]]
```

Quando usamos o espaçamento seletivo destacamos a natureza da tabela da matriz retornada. Na programação funcional, funções como `lidarComCSV` e a previamente definida (`comparator`) são fundamentais na tradução de um tipo de dados em outro. A **Figura 4** ilustra como as transformações de dados em geral podem ser vistas como a obtenção de um “mundo” em outro.

Figura 4. As funções podem preencher a lacuna entre dois "mundos"

Há melhores maneiras de representar uma tabela de tais dados, mas essa matriz aninhada nos serve bem por enquanto. Na verdade, há pouca motivação para construir uma hierarquia de classes complexa que representa a própria tabela, as linhas, as pessoas ou o que for. Em vez disso, manter os dados de representação mínima nos permite usar campos de matriz existentes, bem como métodos externos. Por exemplo, observe o exemplo a seguir considerando que a função `rest()` retorna uma lista filtrada de elementos da tabela de pessoas:

```
_.rest(tabelaPessoas).sort();
  // [["Marcio", "64", "preto"],
  // ["João", "35", "branco"]]
```

Da mesma forma, desde que saibamos a forma dos dados originais, podemos criar apropriadamente funções de seletor para acessar os dados de uma forma mais descritiva, tal como vemos na **Listagem 14.** Nela, é possível ver uma analogia aos métodos de consulta que faríamos a uma base de dados (`select`), os quais recebem o objeto de tabela, filtram os objetos que serão processados (`rest`) e chamam a função `map()` do JavaScript que, por sua vez, se encarrega de chamar a função de `callback` passada como segundo argumento em todos os objetos do vetor passado como primeiro argumento. A função `zip` é uma utilitária conhecida dos desenvolvedores JavaScript que se encarrega de empacotar todos os vetores passados à mesma em um só. Veja que a associamos à uma

Listagem 14. Abstração das funções de consulta dos valores nas tabelas


```
function selectNomes(tabela) {
 return _.rest(_.map(tabela, _.first));
}
function selectIdades(tabela) {
 return _.rest(_.map(tabela, segunda));
}
function selectCorCabelo(tabela) {
 return _.rest(_.map(tabela, function(row) {
   return nth(row, 2);
 }));
}
var mergeResults = _.zip;
```

As funções selecionadas definidas aqui utilizam funções de processamento de matriz existentes para fornecer acesso fluente a todos tipos de dados simples. Vejamos na **Listagem 15** o resultado de alguns testes usando as mesmas.

Listagem 15. Testando as funções criadas de seleção.

```
selectNomes(tabelaPessoas);
  // Saída: ["Marcio", "João"]
  selectIdades(tabelaPessoas);
  // Saída: ["35", "64"]
  selectCorCabelo(tabelaPessoas);
  // Saída: ["preto", "branco"]
```

A simplicidade de implementação e utilização é um arg[...] de dados do núcleo do JavaScript para fins de modelagem de dados. Isso não quer dizer que não há lugar para uma abordagem orientada a objetos ou baseada em classe. Uma abordagem funcional centrada em torno de funções de processamento de coleção genéricas é ideal para a manipulação de dados sobre as pessoas e uma abordagem orientada a objetos funciona melhor para simular pessoas, por exemplo. A tabela de dados também pode ser alterada para um modelo baseado em classes personalizado.

JavaScript Funcional: existy e truthy

A função `existy` pretende definir a existência de algo. O JavaScript tem dois valores – *null* e *undefined* – que denotam inexistência. Assim, `existy` verifica se o seu argumento é algum desses valores, e é implementada da seguinte maneira:

```
function existy(x) { return x != null };
```

Usando o operador de desigualdade (!=) é possível fazer a distinção entre *null, undefined* e tudo mais. Vejamos na **Listagem 16** um exemplo de como a função `existy` (que faz uso do mesmo) pode ser

Listagem 16. Testando a função existy().

```
existy(null);
// Saída: false
existy(undefined);
// Saída: false
existy({}.notHere);
// Saída: false
existy((function(){})());
// Saída: false
existy(0);
// Saída: true
existy(false);
// Saída: true
```

O uso de `existy` simplifica o que significa "algo existir" no JavaScript. Minimamente, ele coloca a verificação de existência em uma função de fácil uso. A segunda função mencionada, `truthy`, é definida como segue:

```
function truthy(x) { return (x !== false) && existy(x) };
```

A função `truthy` é usada para determinar se algo deve ser considerado um sinônimo de *true* (verdadeiro) e é usada como mostrado na **Listagem 17.**

Listagem 17. Testando a função truthy()

```
truthy(false);
// Saída: false
truthy(undefined);
// Saída: false
truthy(0);
// Saída: true
truthy('');
// Saída: true
```


Em JavaScript, às vezes é útil executar alguma ação somente se uma condição for verdadeira e retornar algo como *undefined* ou *null* caso contrário. O padrão geral é como este:

```
if(condicao) return _.isFunction(facaAlgo) ? facaAlgo () : facaAlgo;
  else return undefined;
```

Usando a função `truthy`, podemos encapsular essa lógica da seguinte forma:

```
function facaQuando(cond, action) {
   if(truthy(cond)) return action();
   else return undefined;
  }
```

Agora, sempre que precisarmos de uma execução mais arrojada com gerenciamento de log e precisarmos receber os parâmetros de condição e ação de forma genérica, podemos criar uma função tal como temos na **Listagem 18**. A mesma recebe dois parâmetros: o primeiro se refere ao

interna facaQuando, por sua vez, checa se a função de fato existe e imprime no console o resultado da execução.

Listagem 18. Função para executar somente se tiver algum campo interno.

```
function executaSeTemCampo(target, name) {
   return facaQuando(existy(target[name]), function() {
     var result = _.result(target, name);
     console.log(['O resultado é', result].join(' '));
     return result;
   });
  }
```

Vejamos na **Listagem 19** alguns casos de teste para a execução da função em questão para os casos de sucesso e de erro:

Listagem 19. Testes envolvendo a função executaSeTemCampo().

```
executaSeTemCampo([1,2,3], 'reverse');
// (console) O resultado é 3, 2, 1
// Saída: [3, 2, 1]
executaSeTemCampo({foo: 42}, 'foo');
// (console) O resultado é 42
// Saída: 42
executaSeTemCampo([1,2,3], 'notHere');
// Saída: undefined
```

Isto é programação funcional:

- A definição de uma abstração para a "existência" é o disfarce de uma função;
- A definição de uma abstração para "*truthiness*" construída a partir de funções existentes;
- A utilização das referidas funções por outras funções através da passagem do parâmetro para alcançar algum comportamento.

Sobre a velocidade

Você deve estar pensando em como o material de programação funcional é lento. Não há como negar que o uso do formato *array* índice de *array[0]* irá executar mais rápido do que qualquer um *nth(array, 0)* ou *_.first(array).* Da mesma forma, um loop imperativo como o demonstrado a seguir será muito rápido:

```
for (var i=0, len=array.length; i < len; i++) {
   facaAlgo(array[i]);
  }
```

Se fizer uso de algum **framework JavaScript** componentizado, como o Underscore, por exemplo, a mesma função pode existir de forma análoga:

```
    facaAlgo(array[i]);
});
```


No entanto, é muito provável que todos os fatores não serão iguais. Felizmente, os dias da ponderosa lentidão JavaScript estão chegando ao fim e em alguns casos já são uma coisa do passado. Por exemplo, o lançamento do motor V8 do Google inaugurou uma era de otimizações de tempo de execução, que tem trabalhado para motivar os ganhos de desempenho em todos os motores de fornecedores JavaScript. Mesmo que outros fornecedores não estejam seguindo o exemplo do Google, a prevalência do motor V8 está crescendo e na verdade impulsiona o **navegador Chrome** e o próprio **:Node.js**. No entanto, outros fornecedores estão seguindo a liderança V8 e introduzindo melhorias de velocidade de execução, *just in time*, coleta de lixo mais rápida, local de armazenamento em cache, no qual alinham seus próprios motores JavaScript.

A necessidade de apoiar o envelhecimento de navegadores como o Internet Explorer 6 é um requisito muito real para alguns programadores de JavaScript. Há dois fatores a considerar quando confrontados com plataformas legadas: (1) o uso do IE6 está diminuindo (veja na seção **Links** um site interessante que rastreia o uso do IE6 no mundo hoje), e (2) há outras maneiras de ganhar velocidade antes, no código que nunca atinge o browser. Por exemplo, o objeto de revestimento (*in-lining*) pode ser realizado estaticamente, ou antes do código, onde é sempre executado. O código *in-lining* é o ato de tomar um pedaço de código contido em, digamos, uma função e colar no lugar de

algum lugar nas profundezas de implementação do Underscore, a função _.each atua como um circuito muito parecido com o loop mostrado anteriormente:

```
_.each = function(obj, iterator, context) {
   for (i = 0, j = obj.length; i < j; i++) { }
  }
```

Imagine que você tem um código que se parece com este:

```
function efetuarTarefa(array) {
   _.each(array, function(elem) {
     facaAlgo(array[i]);
   });
  }
```

Em algum lugar do código de teste, você efetua a seguinte checagem:

```
// ... algum tempo depois
efetuarTarefa([1,2,3,4,5]);
```

Um otimizador de estática pode transformar o corpo da função *efetuarTarefa* para o seguinte:

```
function efetuarTarefa(array) {
```

```
  }
}
```


E uma ferramenta de otimização sofisticada poderia otimizar isso ainda mais, eliminando a chamada de função por completo:

```
var array = [1,2,4,6];
for (var i = 0, j = array.length; i < j; i++) {
 facaAlgo(array[i]);
```

Finalmente, um analisador estático realmente surpreendente poderia otimizá-lo ainda mais em cinco chamadas separadas:

```
efetuarTarefa(array[1]);
efetuarTarefa(array[2]);
efetuarTarefa(array[3]); // ...
```

E, para terminar este conjunto incrível de transformações otimizadas, você pode imaginar que se essas chamadas não têm efeitos ou nunca são chamadas, em seguida, a transformação ideal é:

```
// ... algum tempo depois
```




Isto é, se uma parte do código pode ser determinada como "morta" (isto é, não chamada), então ela pode seguramente ser eliminada através de um processo de *dead code elimination*. Já existem programas otimizadores disponíveis para JavaScript que realizam esses tipos de otimizações – como o compilador primário de *closures* do Google. O compilador de *closures* é uma incrível peça de engenharia que compila JavaScript em JavaScript altamente otimizado.

Há maneiras diferentes para acelerar as bases de código mesmo altamente funcionais usando uma combinação de melhores práticas e ferramentas de otimização. No entanto, muitas vezes não paramos para examinar as questões de velocidade de computação bruta antes de escrevermos um código correto. O Underscore é uma biblioteca de programação funcional muito popular para JavaScript, e um grande número de aplicações fazem um bom uso dela. O mesmo pode ser dito para o campeão dos pesos pesados das bibliotecas JavaScript, o jQuery, que promove muitas expressões funcionais.

Certamente, existem domínios legítimos para a velocidade crua (por exemplo, programação de jogos e sistemas de baixa latência). No entanto, mesmo em face de demandas de execução de tais sistemas, técnicas funcionais não são garantidas para retardar as coisas.

O caso do Underscore

O *Underscore* é uma agradável biblioteca que oferece uma API pragmática e bem funcional em grande estilo. Em segundo lugar, há uma chance maior que isso que a execução dos trechos de códigos anteriores usando *Array#map* não funcione. A razão provável é que em qualquer ambiente que você escolher executá-los pode ser que não tenhamos o método `map`. O que temos que evitar, a qualquer custo, é ficar atolado em problemas de compatibilidade do *cross-browser*. Esse ruído, embora extremamente importante, é uma distração para o propósito maior de introdução de programação funcional. O uso do *Underscore* elimina isso quase completamente.

Finalmente, o JavaScript por sua própria natureza permite aos programadores reinventar a roda com bastante frequência. O JavaScript tem a combinação perfeita de poderosas construções de baixo nível, juntamente com a ausência de recursos de linguagens de médio e de alto nível. É essa condição estranha que quase atreve as pessoas a criarem recursos de linguagens a partir das partes de nível inferior. A evolução da linguagem evitará a necessidade de reinventar algumas rodas existentes (por exemplo, sistemas de módulos), mas é improvável ver uma eliminação completa do desejo ou necessidade de construção de características de linguagem. No entanto, quando estiverem disponíveis, bibliotecas existentes de alta qualidade devem ser reutilizadas.

Funções de primeira classe

A programação funcional facilita a utilização e criação de funções de primeira classe. O termo "primeira classe" significa que algo é apenas um valor. A função de primeira classe é uma que pode ir a qualquer lugar que outro valor possa ir – há poucas ou nenhumas restrições. Um número no JavaScript é certamente algo de primeira classe, e, portanto, uma função de primeira classe tem natureza semelhante:

- Um número pode ser armazenado em uma variável, da mesma forma que uma função pode:

```
var quarentaedois = function() { return 42 };
```

- Um número pode ser armazenado em um slot de um vetor, da mesma forma que uma função pode:

```
var quarentaedois = [42, function() { return 42 }];
```

- Um número pode ser armazenado em um campo de objeto, da mesma forma que uma função pode:

```
var quarentaedois = {number: 42, fun: function() { return 42 }};
```

- Um número pode ser criado conforme necessário, da mesma forma que uma função pode:

```
42 + (function() { return 42 })();
// Saída: 84
```




```
function add(n, f) { return n + f() }
add(42, function() { return 42 });
// Saída: 84
```


- Um número pode ser retornado de uma função, da mesma forma que uma função pode:

```
return 42;
return function() { return 42 };
```

Os dois últimos pontos definem, por exemplo, o que poderíamos chamar de uma função “de ordem superior”; colocada diretamente, uma função de ordem superior pode fazer um ou ambos dos seguintes procedimentos:

- Tomar uma função como argumento;
- Retornar uma função como resultado.

Anteriormente o `comparator` foi usado como um exemplo de uma função de ordem superior, mas aqui temos outro exemplo:

```
_.each(['suco', 'tango', 'melão'], function(palavra) {
   console.log(palavra.charAt(0).toUpperCase() + palavra.substr(1));
  });
```

O resultado, consequentemente, será:




```
// (console) Suco
// (console) Tango
// (console) Melão
```


A função do `Underscore _.each` leva uma coleção (objeto ou *array*) e itera ao longo de seus elementos, chamando a função *given* como o segundo argumento para cada elemento.

Múltiplos paradigmas JavaScript

É claro que o **JavaScript não é estritamente uma linguagem de programação funcional**, mas em vez disso facilita o uso de outros paradigmas de igual forma:

- **Programação imperativa**: baseada na descrição de ações em detalhes;
- **Programação orientada a objetos baseada em *protoypes:*** baseada em objetos prototípicos bem como nas instâncias deles;
- **Metaprogramação:** programação que manipula a base do modelo de execução do JavaScript, focando principalmente nos dados e na forma como eles serão gerenciados ao longo do código.

Incluindo apenas o paradigma imperativo, orientado a objetos e metaprogramação nos restringimos a esses paradigmas suportados diretamente pelas construções de linguagens embutidas. Você poderia suportar ainda outros paradigmas, como orientação de classe e programação eventual, usando a própria linguagem como um meio de execução, mas não iremos nos aprofundar sobre

Programação imperativa

Um estilo de programação imperativa é classificado por sua requintada atenção aos detalhes de implementação do algoritmo. Além disso, os programas são imperativos e muitas vezes construídos em torno da manipulação direta e inspeção de estado do programa. Por exemplo, imagine que você gostaria de escrever um programa para construir uma folha com a letra para a canção “99 barris of beer” (99 barris de cerveja). A maneira mais direta de descrever os requisitos desse programa seria:

- Comece com 99;
- Cante o seguinte para cada número até 1:

— X barris de cerveja na parede

— X barris de cerveja

— Tome um primeiro, passe para frente

— X-1 barris de cerveja na parede

- Subtraia um do último X e comece novamente com o novo valor;
- Quando você finalmente chega ao número 1, cante o seguinte na última linha:

Como se vê, essa especificação tem uma implementação imperativa bastante simples no JavaScript mostrado aqui (**Listagem 20**).

Listagem 20. Iterando sobre a lista de letras.

```
var letras = [];
for (var barris = 99; barris > 0; barris--) {
 letras.push(barris + " barris de cerveja na parede");
 letras.push(barris + " barris de cerveja");
 letras.push("Tome uma, passe pra frente");
 if (barris > 1) {
 letras.push((barris - 1) + " barris de cerveja na parede.");
 }
 else {
 letras.push("Sem mais barris de cerveja na parede!");
 }
}
```

Essa versão imperativa, apesar de um pouco artificial, é emblemática de um estilo imperativo de programação. Ou seja, a implementação descreve um programa “99 barris of beer” e exatamente um programa “99 barris of beer”. Como o código imperativo opera em um nível tão preciso de detalhes, são muitas vezes implementações difíceis de reutilizar. Além disso, linguagens imperativas são muitas vezes restritas a um nível de detalhe que é bom para os seus compiladores, mas não para seus programadores. Em comparação, uma abordagem mais funcional para esse mesmo problema pode

```
function segmentosLetra(n) {
   return _.chain([])
     .push(n + " barris de cerveja na parede")
     .push(n + " barris de cerveja")
     .push("Tome uma, passe pra frente")
     .tap(function(letras) {
       if (n > 1)
         letras.push((n - 1) + " barris de cerveja na parede.");
       else
         letras.push("Sem mais barris de cerveja na parede!");
       })
     .value();
  }
```


A função `segmentosLetra` faz muito pouco por conta própria – na verdade, ela só gera letras para um único verso da canção de um determinado número:

```
  segmentosLetra(9);
  // Saída: ["9 barris de cerveja na parede", "9 barris de cerveja", "Tome uma,
passe para frente", "8 barris de cerveja na parede."]
```

Programação orientada a objetos baseada em prototypes

O JavaScript é muito semelhante ao **Java** ou **C#**, em que as funções construtoras são classes, mas o método de utilização é um nível inferior, onde todas as instâncias em um programa Java são geradas




para servir como protótipos para instâncias especializadas. A especialização do objeto, juntamente com uma lógica *built-in* de despacho (*dispatching*) que [illegible] de uma cadeia de protótipos, é muito mais baixo nível do que a programação orientada a classes, mas é extremamente flexível e poderosa.

Por agora, a forma como isso se relaciona com a programação funcional se dá através das funções que podem também existir como valores dos campos do objeto, e o *Underscor*e em si é a ilustração perfeita disso:

```
_.each;
  // Saída: function (array, n, guard) {
  // }
```

Isso é ótimo, certo? Bem, não exatamente, pois o JavaScript é orientado em torno de objetos, então ele deve ter uma semântica para auto referências. Como se constata, a sua semântica de auto referência conflita com a noção de programação funcional. Observe o seguinte:

```
var a = {nome: "a", fun: function () { return this; }};
  a.fun();
  // Saída: {nome: "a", fun: ...};
```

Você irá notar que a auto referência **this**, no retorno da função `fun` embutida, retorna o objeto *a* (ou

```
var bFunc = function () { return this };
var b = {nome: "b", fun: bFunc};
b.fun();
```


O resultado da execução desse exemplo pode ser verificado na **Figura 5**. Quando uma função é criada fora do contexto de uma instância do objeto, sua referência **this** aponta para o objeto global.

Figura 5. Resultado da execução da função fun()

Metaprogramação

Muitas linguagens de programação apoiam a metaprogramação, mas raramente fornecem o mesmo nível de potência oferecida pelo JavaScript. Na metaprogramação, a programação ocorre quando você escreve o código para fazer algo, e a metaprogramação ocorre quando você escreve o código

que muda a maneira como algo é interpretado. Vamos dar uma olhada em um exemplo de metaprogramação para que você possa entender melhor.

No caso do JavaScript, a natureza dinâmica dessa referência pode ser explorada para executar um pouco de metaprogramação. Por exemplo, observe a seguinte construção de função:

```
function Point2D(x, y) {
   this._x = x;
   this._y = y;
  }
```

Quando usada com um *new*, a função *Point2D* dá uma nova instância do objeto com os campos adequados definidos:

```
new Point2D(0, 1);
// Saída: {_x: 0, _y: 1}
```

No entanto, o método `Function.call` pode ser usado para metaprogramar uma derivação do *Point2D* com o comportamento do construtor para um novo tipo *Point3D*, por exemplo:

```
function Point3D(x, y, z) {
   Point2D.call(this, x, y);
   this._z = z;
```




O *this* representa o objeto global onde o elemento está inserido (ou seja, o *window*). E a criação de uma nova instância funciona exatamente como esperávamos.

```
new Point3D(10, -1, 100);
// Saída: {_x: 10, _y: -1, _z: 100}
```

Em nenhum lugar o *Point3D* define explicitamente os valores para *this._x* e *this._y*, mas por ligação dinâmica a essa referência em uma chamada para o *Point2D*, tornou-se possível alterar o alvo do seu código de criação das referidas propriedades.

A programação funcional é um universo muito grande e por muitas vezes é relativa ao contexto do que se está programando naquele exato momento. Essa relatividade, portanto, se traduz na criatividade e experiência dos profissionais alocados num dado projeto. A melhor forma de entender sua real aplicabilidade é comparando seus conceitos com os de outras linguagens essencialmente funcionais, como o Java (a partir da versão 8) e o C#. Assim, você pode analisar como transcrever suas regras para o modelo de protótipos, OO ou até mesmo baseado em funções do JavaScript.

- **Introdução ao Javascript**:

  Veja neste artigo uma breve introdução, de forma simples e prática, à ling isolic das bases do desenvolvimento web.

- **Introdução ao JavaScript**:

  Nesse guia conheceremos a linguagem de programação JavaScript, baseada em scripts client-side de páginas web e orientada a objetos. Veremos a sintaxe básica e bibliotecas que vão nos ajudar a criar páginas ricas em recursos.

Link

Página do framework Underscore.js

www.underscorejs.org/

Tecnologias:

JavaScript POO

Suporte ao aluno - Deixe a sua dúvida.

8