Introdução à programação funcional com Clojure

Gregório Melo

Chapter 1

Programação funcional, o começo

A gente acabou de se familiarizar com Clojure e agora está na hora de começar a conversa sobre os princípios da programação funcional. A ideia é abordar o assunto de forma um tanto pragmática. Ao fim desta parte, espero que você tenha uma boa compreensão do essencial da programação funcional, já levando algumas ideias para o seu dia a dia. E, claro, aproveitar este conhecimento para escrever programas melhores. Daí, caso você queira se aprofundar em teorias e conceitos matemáticos, há referências esperando por você no fim do livro.

Programação funcional não é novidade alguma. De fato, é baseada no cálculo lambda, desenvolvido em meados dos anos 1930. Mas, por muito tempo, linguagens de programação funcional eram tidas como lentas e que exigiam maior consumo de memória se comparadas às linguagens de programação imperativas. Hoje, estes argumentos são infundados, já que compiladores de linguagens funcionais, há bastante tempo, trazem recursos como *tail call optimization* (tema para o capítulo 7) que melhoram, e muito, o uso de recursos computacionais.

Por um lado, a gente vai poder tirar proveito de novidades interessantes. Como dito na introdução do livro, o aumento do número de núcleos computacionais pede por programas que lidem bem com concorrência. Dada a natureza de linguagens funcionais, que prezam por dados imutáveis, elas se tornam ideias para cenários assim, já que é minimizado o risco de alterações indesejadas no estado de um sistema.

Por um outro lado, precisaremos sair da nossa zona de conforto. Abriremos mão de alguns conceitos do mundo de linguagens imperativas que não estarão mais presentes, como *loops* com `for` ou até mesmo variáveis. E assim precisaremos pensar de uma forma diferente para resolver alguns problemas. Um exemplo disto são os loops, como vimos no capítulo 3, no exemplo do uso da função `map`.

E aqui fica o convite para a introdução do primeiro conceito: a importância de funções.

Funções: primeira classe e grandeza superior

Por mais óbvio que soe, preciso dizer que funções são muito relevantes em programação funcional, não é à toa o nome! Funções são tão importantes que são consideradas de primeira classe. Isto quer dizer que funções são tratadas como valores: assim como você passa um número ou uma string para uma função, você também pode passar uma função como argumento. Em JavaScript, por exemplo, temos o seguinte:

```
// código JavaScript
var autor = "George Orwell";
var nomeDoLivro = "1984";
var anoDePublicacao = 1949;
// variáveis 'normais'

var descricaoCompleta = function (autor, nomeDoLivro, anoDePublicacao) {
  return nomeDoLivro + ", " + autor + ", " + anoDePublicacao
};
// variável que é uma função
```

Em JavaScript, podemos declarar variáveis cujo tipo são funções, e isto faz com que JavaScript seja uma linguagem que possui funções como cidadãs de primeira classe. Java é um exemplo de linguagem onde funções (métodos) não são consideradas cidadãs de primeira classe, apesar de ser possível a passagem de funções como argumento para um método (a partir do Java 8).

Daí, com toda esta importância, ganhamos um recurso a mais para lidar com funções, que são as funções de grandeza superior. Funções de grandeza superior são funções que recebem funções como argumento, ou que retornam uma função como resultado. Como, por exemplo, este trecho de código para lidar com o registro de senhas, usando a técnica (simplificada) de *salting*, que é uma técnica que anexa uma string aleatória a uma senha antes que ela seja criptografada e salva:

```
// código JavaScript
var salgarSimples = function (valor) {
  return valor + "sal";
};
// função que aplica um sal para um texto

salgarSimples("senhainsegura");
// "senhainsegurasal"
// perceba que "sal" foi anexado ao final da senha, como esperado
```

```
var gravarCliente = function (nome, senha, salgar) {
  console.log("Salvando cliente de nome '" + nome + "'");
  console.log("Senha salva: " + salgar(senha));
};
// função que 'salva' cliente no console

gravarCliente("Gregório", "senhaseguradema is", salgarSimples);
// Salvando cliente de nome 'Gregório'
// Senha salva: senhaseguradema issal
```

Repare que o argumento `salgar` é uma função e é utilizada como argumento para `gravarCliente`. Isto permite que `gravarCliente` não se importe com a estratégia que será utilizada para salgar a senha.

E agora veremos como utilizaremos este recurso no nosso domínio.

Funções de grandeza superior e nossas finanças

A gente já viu um exemplo de uso de função de grandeza superior no capítulo 3, utilizando a função `map`. `map` é o recurso que Clojure provê (função) para que a gente iteraja sobre uma coleção, e, a cada iteração, aplica-se uma função. Por fim, retorna uma outra coleção como resultado. Vamos ver um exemplo?

Iterando sobre transações

Imagina que a gente tem uma coleção de transações e, para a aplicação cliente, nosso serviço deve retornar um conjunto reduzido de dados: o valor, o tipo da transação (receita ou despesa) e a data. Nosso programa, então, deve varrer cada mapa e pegar só chaves e valores que nos interessam:

```
(defn resumo [transacao]
  (select-keys transacao [:valor :tipo :data]))
;; função que cria um resumo de uma transação

(def transacoes
  [{:valor 33.0 :tipo "despesa" :comentario "Almoço" :data "19/11/2016"}
   {:valor 2700.0 :tipo "receita" :comentario "Bico" :data "01/12/2016"}
   {:valor 29.0 :tipo "despesa" :comentario "Livro de Clojure" :data "03/12/2016"}])
;; as transações

(map resumo transacoes)
;; ({:valor 33.0,   :tipo "despesa", :data "19/11/2016"}
;;  {:valor 2700.0, :tipo "receita", :data "01/12/2016"}
;;  {:valor 29.0,   :tipo "despesa", :data "03/12/2016"})
```

Perceba que o resultado foram transações sem o campo comentário. A função nativa `select-keys` permite que a gente pegue só alguns valores de um mapa. Daí, com `map`, pegamos o `resumo` de cada item de `transacoes`. E se a gente quiser procurar algum padrão nas despesas, e filtrando as transações que são do tipo `"despesa"`?

Filtrando transações

A função `filter` pode nos ajudar aqui. Assim:

```
(defn despesa? [transacao]
  (= (:tipo transacao) "despesa"))
;; função que verifica se uma transação é uma despesa,
;; verificando o valor para a chave :tipo

(filter despesa? transacoes)
;; ({:valor 33.0, :tipo "despesa",
;;   :comentario "Almoço", :data "19/11/2016"}
;;  {:valor 29.0, :tipo "despesa",
;;   :comentario "Livro de Clojure", :data "03/12/2016"})
```

Das 3 transações que temos, apenas uma é uma receita. E `filter` separou pra gente só as que a gente quer, deixando a receita de fora.

Agora que as despesas estão separadas, que tal somar o valor delas?

Condensando resultado de iterações

Depois de aplicarmos `filter`, conseguimos uma outra coleção com os valores que a gente queria. E agora a gente precisa somar todos eles. No mundo não-funcional do Java anterior à versão 8, teríamos que fazer algo assim:

```
public Integer somaDosValores(List<Integer> valores) {
    Integer soma = 0;

    for (Integer valor : valores) {
        soma += valor;
    }

    return soma;
}
```

Linguagens de programação com recursos funcionais facilitam esta operação através de uma função conhecida por diversos nomes, de acordo com a linguagem. Em Clojure, ela se chama `reduce`, mas pode ser encontrada com os seguintes

nomes em outras linguagens: fold, aggregate, accumulate, inject, dentre outros nomes.

Então, vamos utilizar o `reduce` para fazer a some de todos os valores da coleção de valores das despesas? Assim, ó:

```
(defn so-valor [transacao]
  (:valor transacao))
;; função que pega só o valor de uma transação

(map so-valor (filter despesa? transacoes))
;; (33.0 29.0)
;; pegamos só os valores das despesas, que serão condensados com reduce

(reduce + (map so-valor (filter despesa? transacoes)))
;; 62.0
;; a soma dos valores das despesas
```

`reduce` é uma função que combina todos os itens de uma coleção num valor só. `reduce` aplica uma função que vai resultando num valor acumulado. Aqui, a função `reduce` começa com um valor nulo e aplica a função passada (`+`, neste caso) ao primeiro item da lista, e coloca o resultado numa variável interna que costumamos chamar de acumulador. Daí parte para os elementos seguintes, aplicando `+` entre o valor acumulado e o próximo item da coleção.

Se você conhece Ruby, já deve ter visto uma função similar a `reduce` chamada `inject`. Ela funciona mais ou menos assim:

```
# código Ruby
valores = [33.0, 29.0]

valores.inject(:+)
# 62.0
```

Esta combinação `filter`, `map`, e `reduce` é muito popular e poderosíssima! Você certaente encontrará muitas destas funções por aí, e por isso fiz questão de trazê-las o mais cedo possível no livro!

Funções anônimas

Até aqui a gente usou duas coisas bem comuns para associar valores a nomes: `def` para dar nome a algum dado, e `defn` para dar nome a um função que nós criamos. Acontece que nem sempre a gente precisa dar um nome a alguma função, já que às vezes elas são utilizadas apenas uma vez. Nestes casos, talvez prefiramos que tais funções sejam anônimas. Por exemplo, no código JavaScript que salga uma senha, a função de salgar, quando anônima, ficaria assim:

```
gravarCliente("Gregório", "senhaseguradamais", function(senha) {
  return "salgrosso" + senha;
});
```

Vamos ver como ficaria em Clojure? Que tal a gente pegar o exemplo para ver somente valores acima de 100? Começando com a função de filtragem declarada:

```
(defn valor-grande? [transacao]
  (> (:valor transacao) 100))
;; verifica se o valor de uma transação é maior que 100

(filter valor-grande? transacoes)
;; ({:valor 2700.0, :tipo "receita", :comentario "Bico", :data "01/12/2016"})
```

Ora, o trecho de código na função que utilizamos para filtrar valores acima de 100 é bem pequeno e talvez seja utilizado num só lugar. Parece ter um bom potencial para virar uma função anônima! Então acho que chegou a hora da gente entender melhor o que o `defn` faz. Clojure é sintaticamente muito simples, com poucas palavras-chaves e notações nativas, sendo `def` um destes recursos nativos (*special form*, como consta na documentação da linguagem). Só que Clojure provê um recurso chamado *macros* (não são como as macros do Excel, tá?), que permite que a linguagem seja extendida. Às vezes o mais simples é entender que algumas macros proveem atalhos. Que é o caso de `defn`: uma macro para definição de funções.

Lembra que funções são cidadãs de primeira classe em Clojure? Isto significa que funções podem ser associadas a nomes, assim como uma coleção, por exemplo. `def` é o que utilizamos para dar nome a algo, como nestes exemplos:

```
(def impares '(1 3 5 7 9))
(def ano-do-pentacampeonato-do-brasil 2002)
(def pi 3.14)
(def um-terco 1/3) ;; acredite, uma razão é um tipo válido em Clojure
```

Existe outra forma especial (como `def`) para criar uma função: `fn`. Só que é uma função sem nome, que ninguém consegue chamar:

```
(fn [])
;; uma função que não recebe nenhum argumento e não faz nada

(fn [nome]
  (str "Olá, " nome "!"))
;; uma função que recebe um argumento, nome, e retorna uma mensagem carinhosa.
```

Acontece que estas funções acabam ficando no limbo, porque não há como nós a referenciarmos. A não ser que a gente já as execute no momento da sua criação:

```
((fn [nome]
   (str "Olá, " nome "!"))
```

```
 "mundo novo")
;; "Olá, mundo novo!"
```

Perceba que há dois parênteses antes de `fn`: Um significa que vamos chamar alguma função. O outro é o começo da construção da função sem nome. Daí, logo após o parêntese que fecha a definição da função, a gente passa o único argumento que a função sem nome espera. Se a gente tivesse usado `defn`, ficaria assim:

```
(defn ola [nome]
  (str "Olá, " nome "!"))

(ola "mundo novo")
```

O truque com `defn` é que ele nada mais é que uma macro que nos proporciona um atalho que, no fundo, nos serve como isso aqui:

```
(def ola (fn [nome]
           (str "Olá, " nome "!")))

(ola "mundo novo")
```

O que acontece aqui é que a gente usa `fn` pra criar uma função, `def` para dar nome às coisas, e `defn` pra facilitar nossa vida quando precisamos dar nomes a funções.

Agora que sabemos que como criar nossas funções anônimas, como faríamos para filtrar nossa coleção de transações sem aquela `valor-grande?`? Assim:

```
(filter (fn [transacao]
          (> (:valor transacao) 100))
        transacoes)
;; ({:valor 2700.0, :tipo "receita", :comentario "Bico", :data "01/12/2016"})
```

O primeiro argumento para `filter` é uma função anônima, e o segundo é a coleção, `transações`.

Como funções anônimas são tão comuns nessa vida, existe uma forma curta de descrevê-las. Lembra que existem funções para criar listas e mapas, mas que existem formas curtas para criar estas estruturas de dados? O mesmo recurso é disponibilizado para funções anônimas. Assim:

```
(filter #(> (:valor %) 100)
        transacoes)
```

Aqui utilizamos a forma `#(...)` para encurtar a declaração da função anônima. Perceba que agora a gente não tem mais um argumento com o nome; antes tínhamos um argumento chamado `transacao`, e agora ele não é mais tão explícito. Mesmo assim, o argumento ainda existe. Lembre-se, `filter` vai passar por cada elemento da coleção e passá-lo pra função que citamos como primeiro argumento. Daí cada elemento da coleção vira argumento da função anônima.

Só que a gente não precisa dar um nome a este argumento. Tanto faz. Por isso podemos referenciá-lo com `%`, como em `(:valor %)`.

> **E se minha função anônima receber mais de um argumento?**
>
> Daí você poderá utilizar %1, %2, %3 e por aí em diante, para referenciar os argumentos na ordem em que eles são passados.

Tendo visto como lidar com funções anônimas, podemos utilizá-las para o caso da gente condensar todos os valores das despesas num só valor. Só pra lembrar, o código era assim:

```
(defn despesa? [transacao]
  (= (:tipo transacao) "despesa"))

(defn so-valor [transacao]
  (:valor transacao))

(reduce + (map so-valor (filter despesa? transacoes)))
;; 62.0
```

E pode ficar assim:

```
(reduce + (map #(:valor %) (filter #(= (:tipo %) "despesa") transacoes)))
;; 62.0
```

Mas, e agora? Qual forma devo utilizar? Veja, a gente atinge o mesmo resultado de ambas as formas. Fica a seu critério qual forma utilizar quando ambas estiverem disponíveis.

Lendo código Clojure de uma outra forma

O código do exemplo anterior nos ajuda muito a contar como que código em Clojure é lido diferente de outras linguagens. A gente primeiro procura saber o que acontece com `filter`, depois com `map` e, por fim, `reduce`. Mesmo que nossos olhos enxerguem primeiro `reduce`, depois `map` e `filter`. É bem diferente de muitas outras linguagens, né?

Para ajudar um pouco com a legibilidade, às vezes até vemos códigos organizados da seguinte forma, e daí a gente tenta ler de dentro pra fora:

```
(reduce +
        (map so-valor
             (filter despesa?
                     transacoes)))
```

O que ainda pode não ser tão legível em casos complexos, ou nem ser a preferência de algumas pessoas. Eis que surge uma macro interessante: `->`. Ela ajuda a gente a escrever código deixando clara a ordem de execução. Imagine um

caso simples onde a gente quer pegar só o valor da primeira transação de uma coleção:

```
(so-valor (first transacoes))
;; 33.0
```

A gente lê de dentro pra fora: primeiro pega o primeiro elemento com `first` e depois pega só o valor através de `so-valor`. Com a macro `->`, este código ficaria da seguinte forma:

```
(-> (first transacoes)
    (so-valor))
;; 33.0
```

Fica bem mais claro qual a ordem de execução, né? Esta macro se chama *thread-first*. Apesar de ter *thread* no nome, não há nada de concorrência aqui. Mas vale pensar em *thread* como um fluxo, e o *first* (de *thread first*) é importante porque significa que o resultado de uma linha é usado como o primeiro argumento da função seguinte. É por isso que `so-valor` aparenta não receber nenhum argumento, quando, de fato, recebe o resultado da função anterior como seu primeiro argumento.

Existe uma outra macro parecida: `->>`, chamada de *thread-last*. Ela é útil nos casos onde a gente precisa passar o resultado da aplicação de uma função como o último argumento da função seguinte. Por exemplo:

```
(->> (filter despesa? transacoes)
     (map so-valor)
     (reduce +))
;; 62.0
```

Tanto `map` quanto `reduce` recebem uma função como primeiro argumento, e por isso a macro `->`, *thread-first*, não nos serve. Daí, o resultado de `(filter despesa? transacoes)` é passado como último argumento para `(map so-valor)`, e seu resultado é passado como último argumento para `(reduce +)`.

É bem comum ver ambas macros utilizadas por aí! Então, sinta-se à vontade para utilizá-las. Meu critério para usar uma destas macros é quando há muitas operações em série em cima de um mesmo argumento inicial. Um caso muito comum é na escrita de *middlewares* para o Ring (https://github.com/ring-clojure/ring), uma biblioteca para aplicações web. Um exemplo do uso do Ring é o seguinte:

```
(def app
  (-> (wrap-json-body app-routes {:keywords? true :bigdecimals? true})
      (wrap-json-response)
      (wrap-request-logger)
      (wrap-exception-handler)
      (wrap-response-logger)))
```

Exercício 5.1

Lembra da solução para o FizzBuzz? Como você implementaria uma solução que nos trouxesse todos os números que não fossem nem `"fizz"`, nem `"buzz"`?

Exercício 5.2

E como ficaria a solução acima utilizando a macro *thread-last*?

Respostas aos exercícios

5.1

Uma alternativa seria:

```
(defn nao-eh-fizzbuzz? [resultado]
  (not (or (= "fizz" resultado)
           (= "buzz" resultado)
           (= "fizzbuzz" resultado))))

(filter nao-eh-fizzbuzz?
        (map fizzbuzz (range 1 16)))
;; (1 2 4 7 8 11 13 14)
```

Primeiro criei uma função pra dizer se é uma das palavras que a gente não quer, e depois usei como função para filtragem de valores.

5.2

E aqui pode ser assim:

```
(defn nao-eh-fizzbuzz? [resultado]
  (not (or (= "fizz" resultado)
           (= "buzz" resultado)
           (= "fizzbuzz" resultado))))

(->> (range 1 16)
     (map fizzbuzz)
     (filter nao-eh-fizzbuzz?))
;; (1 2 4 7 8 11 13 14)
```

Conclusão

Um vasto capítulo, hein? Aqui a gente viu algo essencial para a prática da programação funcional: funções de grandeza superior e funções como cidadãs de primeira classe. Junto com alguns exemplos de uso de funções célebres (como `filter`, `map` e `reduce`), vimos como Clojure acaba se diferenciando no processo de aplicar funções a itens de uma coleção. De quebra, ainda teve um pouco de detalhes sobre Clojure com as macros `->` e `->>`. No próximo capítulo vamos um pouco além nesse mundo de funções, abordando os temas de composição e *currying*.