Clojure

Isabella Mendonça
Italo Moraes
Lucas Romão

Breve histórico

- Criada por Rich Hickey inspirada em LISP
- Roda na JVM e na CLR
- Criada para ser uma versão moderna do Lisp e própria para concorrência
- Utilizada por empresas como Walmart, Nubank, Heroku, Netflix e Spotify

Utilização

Conceitos fundamentais:

- Valores são imutáveis;
- Funções são cidadãos de primeira classe, isto é, são valores como qualquer outro;
- Funções podem receber outras funções como parâmetro e também podem retornar uma função (*High order function)*

Valores imutáveis

```
(let [my-vector [1 2 3 4]
      my-map {:fred "ethel"}
      my-list (list 4 3 2 1)]
  (list
    (conj my-vector 5)
    (assoc my-map :ricky "lucy")
    (conj my-list 5)
    ;the originals are intact
    my-vector
    my-map
    my-list))
-> ([1 2 3 4 5] {:ricky "lucy", :fred "ethel"} (5 4 3 2 1) [1 2 3 4] {:fred "ethel"}
(4 3 2 1))
```

Valor vs Identidade

- Como visto no slide anterior, valores em clojure são imutáveis e qualquer “alteração” cria um novo valor, porém o anterior permanece inalterado;
- Em Clojure existe o conceito de identidade que são entidades que em um dado momento fazem referência a um valor;
- Existem três tipos de identidades: *atoms*, *agents* e *refs*

Funções como valor e *High Order Function*

```clojure
(defn inc-maker
  "Create a custom incrementor"
  [inc-by]
  #(+ % inc-by))

(def inc3 (inc-maker 3))

(inc3 7)
; => 10
```

```clojure
(map #(str "Hello " % "!" ) ["Ford"
"Arthur" "Tricia"])
;;=> ("Hello Ford!" "Hello Arthur!" "Hello
Tricia!")
```

Recursão ao invés de loop

- Assim como nas outras linguagens funcionais, *loops* e iterações são feitas usando recursão;
- Diferente das outras linguagens, em Clojure é necessário utilizar o operador *recur* para garantir otimização das chamadas recursivas.

```
(defn sum [a-seq]
  (if (empty? a-seq)
    0
    (+ (first a-seq) (sum (rest a-seq)))))
```

```
(defn sum [a-seq]
  (let [sum-helper (fn [acc a-seq]
                     if (empty? a-seq)
                       acc
                       recur (+ acc (first a-seq))
                             rest a-seq))))]
    (sum-helper 0 a-seq)))
```

Paralelismo

Estruturas Fundamentais da Linguagem:

- Como já dito antes os valores são imutáveis;
- Suporte nativo a *thread pool* por padrão, também é possível definir *thread pool* personalizada;
- Estruturas de dados com suporte a alterações atômica e com escopo de transação;
- Variáveis podem ter escopo dinâmico.

Estruturas de Dados

Atom

```
(def currently-connected (atom []))

@currently-connected
;; ⇒ []
(deref currently-connected)
;; ⇒ []
currently-connected
;; ⇒ #<Atom@614b6b5d: []>
(swap! currently-connected conj "chatty-joe")
;; ⇒ ["chatty-joe"]
currently-connected
;; ⇒ #<Atom@614b6b5d: ["chatty-joe"]>
@currently-connected
;; ⇒ ["chatty-joe"]
```

Agents

```
(def errors-counter (agent 0))
;; ⇒ 0
(send errors-counter inc)
;; ⇒ #<Agent@6a6287b2: 0>
@errors-counter
;; ⇒ 1

;; 10 is an additional parameter. The + function will be invoked as
`(+ @errors-counter 10)`.
(send errors-counter + 10)
;; ⇒ #<Agent@6a6287b2: 1>
@errors-counter
;; ⇒ 11
```

Delay

```
(def d (delay (System/currentTimeMillis)))
;; ⇒ #'user/d
d
;; ⇒ #<Delay@21ed22af: :pending>
;; dereferencing causes the value to be realized, it happens only
once
(realized? d)
;; ⇒ false
@d
;; ⇒ 1350997967984
(realized? d)
;; ⇒ true
```

Ref

```
(def account-a (ref 1000))
;; ⇒ #'user/account-a
(def account-b (ref 1000))
;; ⇒ #'user/account-b

(dosync
  ;; will be executed as (+ @account-a 100)
  (alter account-a + 100)
  ;; will be executed as (- @account-b 100)
  (alter account-b - 100))
;; ⇒ 900
@account-a
;; ⇒ 1100
@account-b
;; ⇒ 900
```

Routines

```
;; promises have no code body (no code to evaluate)
(def p (promise))
;; ⇒ #'user/p
p
;; ⇒ #<core$promise$reify__6153@306a0a21: :pending>
(realized? p)
;; ⇒ false

;; delivering a promise makes it realized
(deliver p {:result 42})
;; ⇒ #<core$promise$reify__6153@306a0a21: {:result
42}>
(realized? p)
;; ⇒ true
@p
;; ⇒ {:result 42}
```

```
(def ft (future (+ 1 2 3 4 5 6)))
;; ⇒ #'user/ft
ft
;; ⇒ #<core$future_call$reify__6110@effa25e:
21>
(realized? d)
;; ⇒ Will be true if future ended processing
@ft
;; ⇒ 21
```

External Libraries

```
;; ⇒ Simple example of Core.async begin used
(let [c1 (chan)])
      c2 (chan)]
  (go (while true
        (let [[v ch] (alts! [c1 c2])]
          (println "Read"
 v "from" ch))))
  (go (>! c1 "hi"))
  (go (>! c2 "there")))
```

- **Core.async:** Essa biblioteca não realmente externa, porém vem sendo implementada depois de Clojure já estar madura. Essa biblioteca tem principal foco um interação mais responsiva entre “threads”, utilizando uma sintaxe muito focada em CSP(Communicating Sequential Processes). Essa biblioteca não foca necessariamente em execuções de threads reais paralelas, mas apenas na execução concorrente de código mesmo em ambientes que não possuem mais de uma thread real.

Map Reduce

```
;; A function that simulates a long-running process by
calling Thread/sleep:
(defn long-running-job [n]
    (Thread/sleep 3000) ; wait for 3 seconds
    (+ n 10))

;; Use `doall` to eagerly evaluate `map`, which
evaluates lazily by default.

;; With `map`, the total elapsed time is just under 4 *
3 seconds:
user=> (time (doall (map long-running-job (range 4))))
"Elapsed time: 11999.235098 msecs"
(10 11 12 13)

;; With `pmap`, the total elapsed time is just over 3
seconds:
user=> (time (doall (pmap long-running-job (range 4))))
"Elapsed time: 3200.001117 msecs"
(10 11 12 13)
```

```
;; This example ilustrates how would be the usage of the
new reducers class.

(require '[closure.core.reducers :as r])

(r/fold + (r/filter even? (r/map inc [1 1 1 2])))

;; 6
```

Exemplo

Usando *delay* e *future* para evitar problemas de corrida:

```
(def gimli-headshots ["serious.jpg" "fun.jpg" "playful.jpg"])
(defn email-user
  [email-address]
  (println "Sending headshot notification to" email-address))
(defn upload-document
  "Needs to be implemented"
  [headshot]
  true)
(let [notify (delay ❶(email-user "and-my-axe@gmail.com"))]
  (doseq [headshot gimli-headshots]
    (future (upload-document headshot)
            ❷(force notify))))
```

Exemplo 2 - usando *future* e *promisse*

Suponha que você deseja encontrar uma manteiga que tenha uma determinada suavidade e que o preço esteja dentro do seu orçamento.

Exemplo 2- Definições básicas

```
(def yak-butter-international
  {:store "Yak Butter International"
    :price 90
    :smoothness 90})
(def butter-than-nothing
  {:store "Butter Than Nothing"
   :price 150
   :smoothness 83})
;; This is the butter that meets our requirements
(def baby-got-yak
  {:store "Baby Got Yak"
   :price 94
   :smoothness 99})

(defn mock-api-call
  [result]
  (Thread/sleep 1000)
  result)

(defn satisfactory?
  "If the butter meets our criteria, return the butter, else return false"
  [butter]
  (and (<= (:price butter) 100)
       (>= (:smoothness butter) 97)
       butter))
```

Versão sequencial x Versão paralela

```
(time (some (comp satisfactory? mock-api-call)
            [yak-butter-international butter-than-nothing baby-got-yak]))
; => "Elapsed time: 3002.132 msecs"
; => {:store "Baby Got Yak", :smoothness 99, :price 94}
```

```
(time
 (let [butter-promise (promise)]
   (doseq [butter [yak-butter-international butter-than-nothing baby-got-yak]]
     (future (if-let [satisfactory-butter (satisfactory? (mock-api-call butter))]
               (deliver butter-promise satisfactory-butter))))
   (println "And the winner is:" @butter-promise)))
; => "Elapsed time: 1002.652 msecs"
; => And the winner is: {:store Baby Got Yak, :smoothness 99, :price 94}
```

Comparativo

- Elaboração de um *websocket* com duas mensagens: echo e *broadcast* codificadas em JSON
  - Echo é retornado ao cliente que enviou a solicitação
  - *Broadcast* é enviado para todos clientes conectados
- Desenvolvido em Clojure, C++, Go, NodeJS e Ruby

Implementação em Clojure

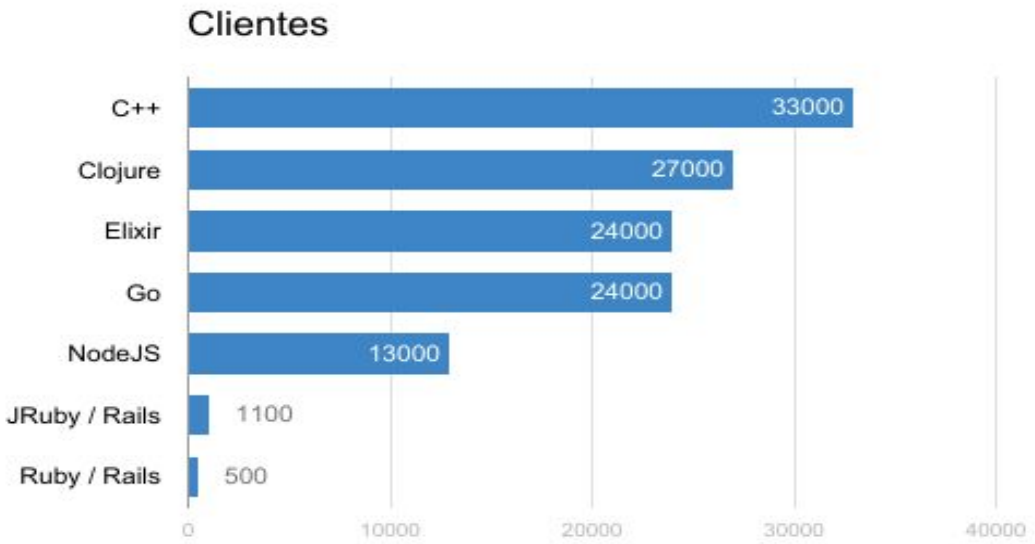
- Utilização do *webserver* HTTP Kit
- Conexão com o cliente foi armazenada em atoms, garantindo concorrência

```clojure
(defonce channels (atom #{}))

(defn connect! [channel]
  (log/info "channel open")
  (swap! channels conj channel))

(defn disconnect! [channel status]
  (log/info "channel closed:" status)
  (swap! channels disj channel))
```

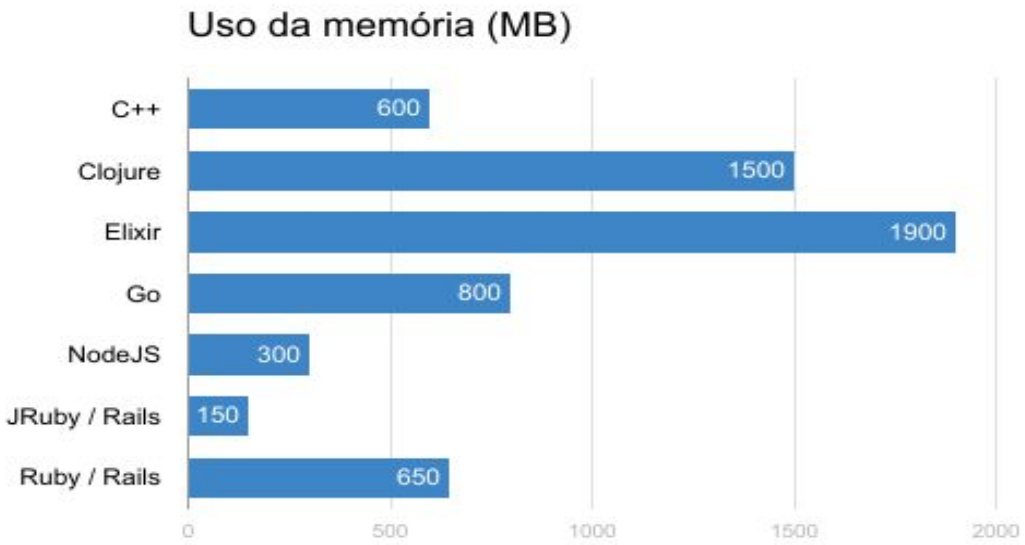
Benchmarks

- Realizada sobrecarga do servidor para checar o número de conexões não-ociosas possíveis mantendo certa performance
- Servidores localizados em uma máquina e o benchmark em outra de mesma configuração conectadas via ethernet

Benchmarks


Clientes
C++
Clojure
Elixir
Go
NodeJS
JRuby / Rails
Ruby / Rails
33000
27000
24000
24000
13000
1100
500
0
10000
20000
30000
40000

Memória


Uso da memória (MB)
C++
600
Clojure
1500
Elixir
1900
Go
800
NodeJS
300
JRuby / Rails
150
Ruby / Rails
650
0
500
1000
1500
2000

C++

- Melhor performance
- Possui o maior e mais complexo desenvolvimento entre as linguagens testadas
- Inclui trechos de código de baixo nível
- Desenvolvimento de makefile
- Tempo de compilação elevado

Clojure

- Segunda melhor performance
- Por rodar na JVM, beneficia-se das bibliotecas Java
- Fácil desenvolvimento
- Código conciso
- Contido em um único arquivo jar

Conclusão

- Clojure é uma linguagem funcional que conseguiu se disseminar de forma muito rápida e foi bem aceita pela comunidade;
- Por ser executada na JVM é capaz de rodar em diversas plataformas e ainda sim é capaz de obter excelente performance e simplicidade de código;
- É capaz de acessar todas as estruturas de dados de Java, e implementa uma camada robusta para implementação de código concorrente de maneira simples.