Clojure

Clojure é uma linguagem de programação **dinâmica** que roda da **JVM**, **CLR** e **JavaScript**. Ela foi projetada para ser **simples**, **interativa, flexível** e ter facilidades para **concorrência**.

O que é clojure?

```
(+ 1 2) ;; 3
```

Sintaxe

Código em Clojure é escrito com suas próprias **estruturas de dados**.

Homoiconicity

```
def
if
do
let
quote
var
fn
```

```
throw
try
monitor-enter
monitor-exit
loop
recur
```

Formas especiais em Clojure

```
BEGIN   next    super   defined?
END     nil     then    do
alias   not     true    else
and     or      undef   elsif
begin   redo    unless  end
break   rescue  until   ensure
\case   retry   when    false
class   return  while   for
def     self    yield   if
                                ...
```

Formas especiais em Ruby

```
def
if
do
let
quote
var
fn
```

```
throw
try
monitor-enter
monitor-exit
loop
recur
```

Formas especiais em Clojure

DEMO

Sintaxe

Literais

Numbers - `123`
Doubles - `98.9`
BigInt - `822322N`
BigDecimal - `87549.23M`
Ratios - `22/7`
String - `“Uma string”`
Character - `\c, \a`
Symbol - `double, even?`
Keyword - `:azul?, :callback-fn`
Booleans - `true, false`
Null - `nil`
Regex - `#”[a-z]+”`

Tipos de dados

Listas

```
(1 2 3) (clojure symbols) (list 1 2 5)
```

Vetores

```
[1 2 3 4 5] [:keyword symbol 1 2/3]
```

Maps

```
{:chave "valor" :chave2 "valor2"} {1 :val}
```

Sets

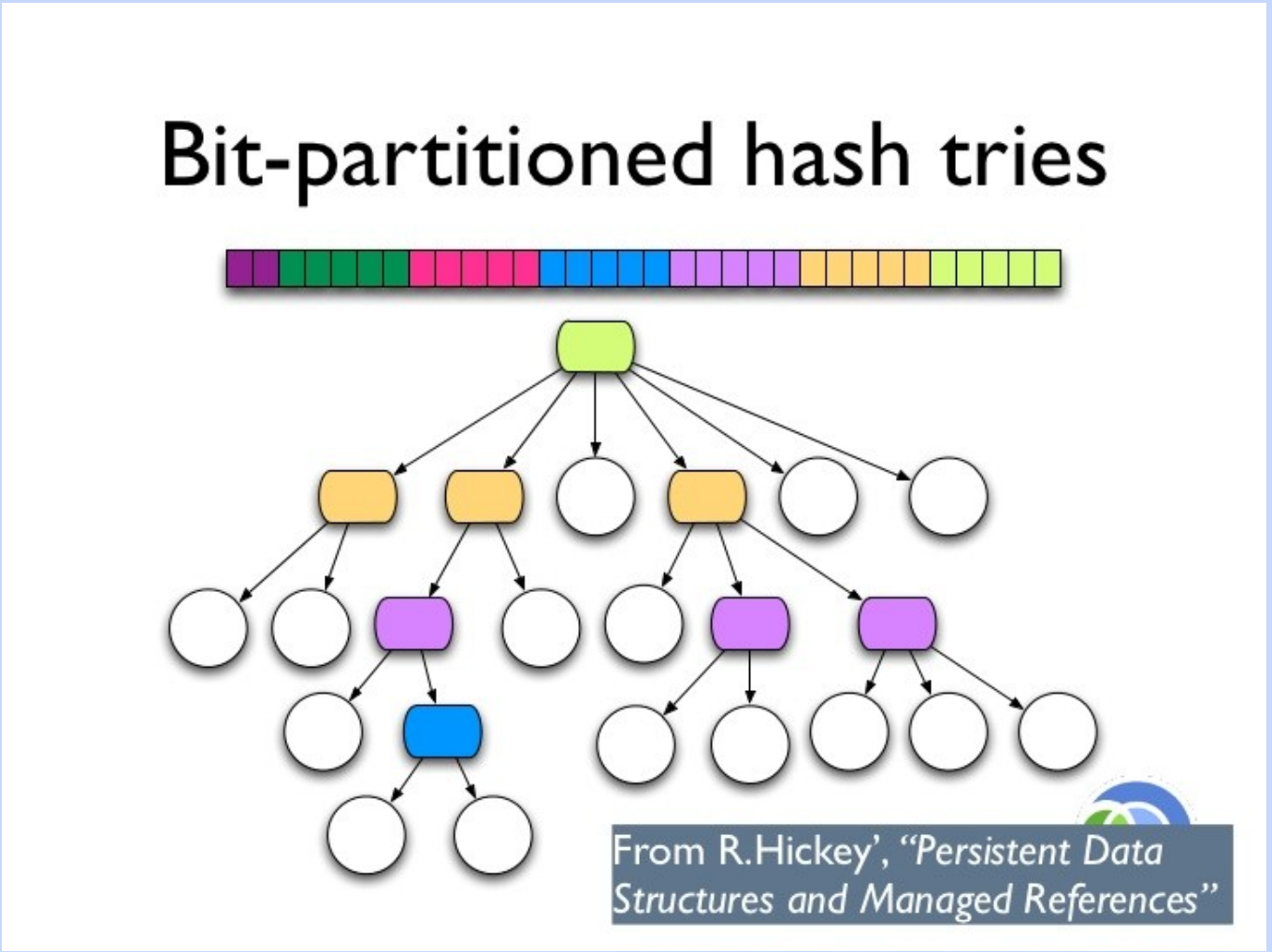
```
#{1 2 3 4} (set [1 1 3 4 5])
```

As estruturas de dados são **imutáveis** e **persistentes**.

```
(def my-map {:name "Heisenberg"})
;;=> my-map

(assoc my-map :job "Drug dealer")
;;=> {:name "Heisenberg", :job "Drug dealer"}

my-map
;;=> {:name "Heisenberg"}
```



Persistent Data Structures

DEMO

Tipos de dados

Meios de criação de funções

- fn
- defn
- #()

```
(fn [x] (* x x))

(defn square [x]
  (* x x))

#(* % %)
```

Funções de alta ordem

```
(map square [1 2 3 4]) ;;=> (1 4 9 16)

(reduce + [1 2 3 4]) ;;=> 10

(filter #(zero? (rem % 2)) [1 2 3 4]) ;;=> (2 4)
```

Functions

```clojure
(def double (partial * 2))
(double 2) ;;=> 4

(def upper-cased-chars
  (comp seq upper-case))
(upper-cased-chars "abc") ;;=> (\A \B \C)
```

Partial application e Function Composition

DEMO

Functions

Clojure implementa abstrações de sequencia para

- Listas
- Vetores
- Maps
- Sets
- Strings
- Diretórios
- Arquivos
- Flexível para implementar em outras estruturas de dados ou até mesmo plataformas

Sequences

```clojure
(ns sequence-demo
  (:require [clojure.java.io :as io]
            [clojure.string :as s]))
;; Sequencia de arquivos e diretorios
(def arvore-de-diretorio (file-seq (io/file "some/path")))
;;=> (<#clojure.java.io.File#347fd723 /some/path/folder>
;;    <#clojure.java.io.File#584ffd22 /some/path/folder/file.txt>)
;; Filtrar somente arquivos
(filter (fn [file-object] (.isFile file-object)) arvore-de-diretorio)
;;=> (<#clojure.java.io.File#584ffd22 /some/path/folder/file.txt>
;; Sequencia de linhas em um arquivo
(def lines (line-seq (io/reader (io/file "some/file.txt"))))
;;=> ("first line"
;;    "second line")
;; Deixa todas as linhas em caixa alta
(map s/upper-case lines)
;; => ("FIRST LINE"
;;     "SECOND LINE")
;; Filtra todos os numeros pares num Vetor
(filter even? [1 2 3 4 5])
;;=> (2 4)
```

Sequences

Resultados computados em demanda

Laziness

```
;; Sequencia infinita
(def infinite-range (range))

;; Sequencia infinita de numeros impares
(def infinite-odd (filter odd? (range)))

;; Computando somente 3 numeros impares
(take 3 infinite-odd)
;;=> (1 3 5)

;; Sequencia de Fibonacci
(defn fib
  ([] (fib 0 1))
  ([a b]
   (lazy-seq
    (cons a
          (fib b (+ a b))))))

(take 5 (fib))
;;=> (0 1 1 2 3)
```

Java Interop

```clojure
(ns java-interop-demo
  (:import java.util.Date
           java.util.ArrayList))
(def now (Date.))
;; Acessando metodos do objeto
(. now getSeconds)
;; 26
(.getSeconds now)
;; 26
;; Metodos estaticos
(Math/pow 2 2)
;; 4.0
;; Acessando propriedades e metodos em corrente
(.. System getProperties (get "java.version"))
;; "1.8.0_45"
;; Com macros
(def arr-list (ArrayList.))
(doto arr-list (.add 0 1) (.add 1 2))
(str arr-list)
;;=> [1, 2]
```

Código gerado por código

- Em tempo de compilação
- Funcionalidades de linguagem podem ser macros

```
(defmacro infix-notation [form]
  `(~(second form) ~(first form) ~(last form)))

(infix-notation (1 + 2)) ;;=> 3
```

DEMO

Macros

Atom

```clojure
(def medias (atom []))

(swap! medias conj 1)
;;=> #<Atom@asd3j2 [1]

@medias
;;=> [1]

(reset! medias [])
(deref medias)
;;=> []
```

Concorrência

Agent

```
(ns logger
  (:import java.io BufferedWriter FileWriter))

(def wtr (agent (BufferedWriter. (FileWriter. "log.txt"))))

(defn log [message]
  (send wtr
        (fn [out message]
              (.write out message))
        message))

(log "logging")
(log "warning")
```

Concorrência

Ref

```
(def conta-do-paulo (ref 10))
(def conta-do-wilson (ref 100))

(defn transferir [acc-from acc-to amount]
  (dosync
   (alter acc-from - amount)
   (alter acc-to + amount)))

(transferir conta-do-wilson conta-do-paulo 10)
@conta-do-paulo ;; => 20
@conta-do-wilson ;; => 90
```

Concorrência

Future

```
(def f (future (expensive-computation)))

;; Retorna o resultado da computacao,
;; vai bloquear a thread ate que a funcao seja executada
@f
```

Promise

```
(def p (promise))

(realized? p) ;; => false

(deliver p 50)

(realized? p) ;; => true

@p ;; => 50
```

Concorrência

core.async

```
(ns sandbox.core-async-events
  (:require-macros [cljs.core.async.macros :refer [go]])
  (:require [cljs.core.async :as async :refer [put! >! <! chan]]
            [domina.css :as css]
            [domina.event :as event]))

(defn events [element event-type]
  (let [c (chan)]
    (event/listen! element event-type (fn [e] (put! c e)))
    c))

(let [click-events (events (css/sel "#some-button") :click)]
  (go (while true
        (do-something-with (<! click-events))))
  click-events)
```

Concorrência

Protocols

```
(defprotocol IArea
  (area [this]))

(defrecord Square [x]
  IArea
  (area [this]
    (let [x (:x this)]
      (* x x))))

(defrecord Circle [r]
  IArea
  (area [this]
    (let [radius (:r this)]
      (* Math/PI
         (Math/pow radius 2)))))
```

Polimorfismo

```
(def circle (->Circle 3))
(def sq (->Square 4))

(area circle) ;;=> 28.27433...
(area sq) ;;=> 16
```

Multimethods

```clojure
(defmulti area (fn [[type]] type))

(defmethod area :square
  [[_ x]]
  (* x x))

(defmethod area :circle
  [[_ radius]]
  (* Math/PI
     (Math/pow radius 2)))

(defmethod area :default
  [_]
  "Argument not supported")

(def circle [:circle 3])
(def sq [:square 4])

(area circle) ;; => 28.27433...
(area sq) ;; => 16
```

Polimorfismo

```
(def you! (all-of devmt-community))

(thank you!)
```

Como me achar:

eduardomrb@gmail.com
eduardomrb (GitHub, devmt.slack.com)

Fim