Programmation Fonctionnelle

Manuel Clergue

Programmer doit être amusant !

Programmer doit être beau !

Chapitre 3 : Structures avancées, boucles et séquences

1 - Données hashées

2 - Boucles et séquences

1 - Données hashées

a - Principes

Clojure offre deux structures de données hashées pour lesquelles la recherche d'éléments est efficace :

- les ensembles (Sets)
- les tables associatives (Maps).

Les opérations (recherche, ajout, suppression) sur ces structures sont efficaces (en temps constant, ou presque).

b - Ensembles

Les ensembles en clojure sont des structures de données fonctionnant sur le même principe que les ensembles mathématiques : une collection de données non ordonnée et sans doublons.

Définition d'un ensemble :

```
user=> (def legumes #{"tomate" "artichaut" "concombre"})
#'user/legumes
user=> legumes
#{"concombre" "tomate" "artichaut"}
```

Les doublons sont vérifiés :

```
user=> (def legumes #{"tomate" "artichaut" "concombre" "tomate"})
Syntax error reading source at (REPL:592:58).
Duplicate key: tomate
```

b - Ensembles

Ajout d'éléments :

```
user=> (conj legumes "courgette")
#{"concombre" "tomate" "artichaut" "courgette"}
user=> (conj legumes "tomate")
#{"concombre" "tomate" "artichaut"}
user=> (conj legumes "courgette" "aubergine")
#{"aubergine" "concombre" "tomate" "artichaut" "courgette"}
```

Suppression d'éléments :

```
user=> (disj legumes "tomate")
#{"concombre" "artichaut"}
user=> (disj legumes "tomate" "concombre")
#{"artichaut"}
user=> (disj legumes "courgette")
#{"concombre" "tomate" "artichaut"}
```

b - Ensembles

Recherche d'élément :

```
user=> (contains? legumes "tomate")
true
user=> (contains? legumes "courgette")
false
```

Fusion de collections :

```
user=> (def fruits ["cerise" "fraise"])
#'user/fruits
user=> (into legumes fruits)
#{"fraise" "concombre" "tomate" "cerise" "artichaut"}
user=> (into fruits legumes)
["cerise" "fraise" "concombre" "tomate" "artichaut"]
```

b - Ensembles triés

Il est parfois nécessaire de maintenir un ordre dans des ensembles.

Pour cela clojure offre une structure adaptée : les **`sorted-set`**

Constructeur :

```
user=> (sorted-set)
#{}
user=> (conj (sorted-set) "tomate" "artichaut" "concombre")
#{"artichaut" "concombre" "tomate"}
```

Si on veut un ordre différent de l'ordre *naturel* (qui est donné par la fonction `compare`) :

```
user=> (defn mycomp [a b] (<= (count a) (count b)))
#'user/mycomp
user=> (conj (sorted-set-by mycomp) "tomate" "artichaut" "concombre")
#{"tomate" "concombre" "artichaut"}
```

c - Tables associatives

Les tables associatives (ou tables de hachage, ou dictionnaires) permettent d'enregistrer des associations clé-valeur.

Constructeur :

```
user=> (def notes {"alain" 10, "robert" 15, "alice" 13})
#'user/notes
user=> notes
{"alain" 10, "robert" 15, "alice" 13}
```

Ajout d'une valeur :

```
user=> (assoc notes "hector" 7)
{"alain" 10, "robert" 15, "alice" 13, "hector" 7}
```

Modification d'une valeur :

```
user=> (assoc notes "alice" 17)
{"alain" 10, "robert" 15, "alice" 17}
```

c - Tables associatives

Recherche d'éléments :

```
user=> (get notes "alice")
13
user=> (get notes "hector")
nil
user=> (get notes "hector" :abs)
:abs
user=> (get notes "alice" :abs)
13
```

On peut aussi invoquer directement la table :

```
user=> (notes "alice")
13
user=> (notes "hector")
nil
user=> (notes "hector" :abs)
:abs
```

c - Tables associatives

Recherche d'éléments (si on veut savoir si l'élément est présent) :

```
user=> (contains? notes "alice")
true
user=> (contains? notes "hector")
false
```

Recherche d'éléments (si on veut obtenir l'association complète) :

```
user=> (find notes "alice")
["alice" 13]
user=> (find notes "hector")
nil
```

c - Tables associatives

On peut obtenir la liste des clés :

```
user=> (keys notes)
("alain" "robert" "alice")
```

Ou la liste des valeurs :

```
user=> (vals notes)
(10 15 13)
```

Et on peut associer deux listes pour en faire une table :

```
user=> (zipmap (keys notes) (vals notes))
{"alain" 10, "robert" 15, "alice" 13}

user=> (zipmap '("alain" "robert" "alice") (repeat 0))
{"alain" 0, "robert" 0, "alice" 0}
user=> (zipmap '("alain" "robert" "alice" "alice") (repeat 0))
{"alain" 0, "robert" 0, "alice" 0}
```

c - Tables associatives

Fusion de tables :

```
user=> (merge notes {"hector" 12 "germain" 15})
{"alain" 10, "robert" 15, "alice" 13, "hector" 12, "germain" 15}
```

En cas de doublons, c'est la valeur de la table la plus à droite qui est prise en compte :

```
user=> (merge notes {"hector" 12 "alice" 2})
{"alain" 10, "robert" 15, "alice" 2, "hector" 12}
```

On peut utiliser une fonction pour régler les conflits :

```
user=> (merge-with max notes {"hector" 12 "alice" 2})
{"alain" 10, "robert" 15, "alice" 13, "hector" 12}
```

d - Tables associatives triées

Comme pour les ensembles, il est possible de définir des tables dont l'ordre des clés est maintenu :

```
user=> (def notes-triees (sorted-map "alain" 10, "robert" 15, "alice" 13))
#'user/notes-triees
user=> (keys  notes-triees)
("alain" "alice" "robert")
```

Et même avec une fonction de comparaison :

```
user=> (sorted-map-by (fn [a b] (<= (count a) (count b)))
                      "alain" 10, "robert" 15, "alice" 13)
{"alice" 13, "alain" 10, "robert" 15}
```

e - Tables associatives utilisées comme enregistrement

Il est possible d'utiliser les tables associatives pour définir des enregistrements :

```
user=> (def acteur {:nom "Dujardin", :prenom "Jean", :age 50})
#'user/acteur
user=> acteur
{:nom "Dujardin", :prenom "Jean", :age 50}
```

Cela reste une table :

```
user=> (get acteur :nom)
"Dujardin"
user=> (acteur :nom)
"Dujardin"
```

Mais on peut invoquer la clé sur la table :

```
user=> (:nom acteur)
"Dujardin"
user=> (:films acteur)
nil
```

f - Enregistrements

Il est possible de définir des enregistrements (Records) :

```
user=> (defrecord Acteur [nom prenom age])
user.Acteur
```

Pour instancier un enregistrement, on peut utiliser les arguments positionnels :

```
(def dujardin (->Acteur "Dujardin" "Jean" 50))
#'user/dujardin
user=> dujardin
#user.Acteur{:nom "Dujardin", :prenom "Jean", :age 50}
```

Ou les arguments nommés :

```
user=> (def depardieu (map->Acteur {:prenom "Gérard" :age 70 :nom "Depardieu"}))
#'user/depardieu
user=> depardieu
#user.Acteur{:nom "Depardieu", :prenom "Gérard", :age 70}
```

f - Enregistrements

La différence avec les tables est sur la possibilité d'invoquer l'instance :

```
user=> (:nom depardieu)
"Depardieu"
user=> (depardieu :nom)
Execution error (ClassCastException) at user/eval589 (REPL:1).
user.Acteur incompatible with clojure.lang.IFn
```

2 - Boucles et séquences

a - `doseq`, `dotimes`

Ce sont des boucles utilisables pour des effets de bord (la valeur de retour de l'expression est `nil`) :

```
user=> (dotimes [i 3] (println i))
0
1
2
nil

user=> (doseq [n (range 3)] (println n))
0
1
2
nil
```

a - doseq, dotimes

Elles peuvent être utilisées pour faire des combinaisons :

```
user=> (doseq [nom '("depardieu" "dujardin")
               prenom '("jean" "gérard")]
  (prn [prenom nom]))
["jean" "depardieu"]
["gérard" "depardieu"]
["jean" "dujardin"]
["gérard" "dujardin"]
nil
```

b - Listes en compréhension

La fonction for est un générateur de liste :

```
user=> (for [n (range 3)] (* n 3))
(0 3 6)
```

Qui peut aussi être utilisé de façon imbriquée :

```
user=> (for [nom '("depardieu" "dujardin")
      prenom '("jean" "gérard")]
  [prenom nom])
(["jean" "depardieu"] ["gérard" "depardieu"] ["jean" "dujardin"] ["gérard" "dujardin"])
```

c - application d’une fonction sur les séquences

Clojure permet la manipulation des collections (`vector`, `list`, `set` et `map`) à travers une abstraction commune : la *`sequence`* ou *`seq`*
Il faut voir seq comme une *interface* de certaines structures de données
Pour qu’une structure de données *implémente* seq, il est nécessaire qu’elle réponde à la fonction `seq`

c - application d’une fonction sur les séquences

Les 4 collections vues précédemment implémentent l’interface seq

```
user=> (seq '(1 2 3))
(1 2 3)
user=> (seq  [1 2 3])
(1 2 3)
user=> (seq  #{1 2 3})
(1 3 2)
user=> (seq  {:a 1 :b 2 :c 3})
([:a 1] [:b 2] [:c 3])
```

Notes :

- même si cela ressemble à une liste,le résultat de seq n’est pas (tout à fait) une liste
- pour les collections non ordonnées, l’ordre de la séquence n’est pas significatif
- pour les maps, seq renvoie une séquence de vectors [clef valeur]

c - application d’une fonction sur les séquences

La transformation en sequence est réversible par la fonction `into` :

```
user=> (into '() (seq '(1 2 3)))
(3 2 1)
ser=> (into [] (seq  [1 2 3]))
[1 2 3]
user=> (into #{} (seq  #{1 2 3}))
#{1 3 2}
user=> (into {} (seq  {:a 1 :b 2 :c 3}))
{:a 1, :b 2, :c 3}
```

Vous noterez que la transformation into + seq a inversé la liste. Cela est dû à la façon dont cette fonction est implémentée.

c - application d’une fonction sur les séquences

Une seq répond à ces 4 fonctions de base :

- `empty?` : renvoie vrai si la seq est vide
- `first` : renvoie le “premier” élément de la seq
- `rest` : renvoie la seq sans le “premier” élément
- `cons` : prend une seq et un élément et renvoie la seq avec l’élément ajouté en tête.

Ces 4 fonctions permettent de définir les autres fonctions opérants sur les seq

c - application d’une fonction sur les séquences

Exemple de la fonction into

```
user=> (defn $into [st s]
  (if (empty? s) st ($into (conj st (first s)) (rest s))))
#'user/$into
user=> ($into {} (seq  {:a 1 :b 2 :c 3}))
{:a 1, :b 2, :c 3}
```

Note : conj est une fonction polymorphique !

c - application d’une fonction sur les séquences

On peut appliquer une fonction sur une séquence:

```
user=> (map inc [1 2 3 4 5])
(2 3 4 5 6)
user=> (map #(* % 2) '(1 2 3 4 5))
(2 4 6 8 10)
user=> (map + [1 2 3] (iterate inc 1))
(2 4 6)
user=> (map #(% 2) [inc dec])
(3 1)
```

c - application d’une fonction sur les séquences

On peut *aplatir* une séquence :

```
user=> (reduce + [1 2 3 4 5])
15
```

C’est différent de `apply` :

```
user=> (reduce (fn [a b] (+ a b)) [1 2 3 4 5])
15
user=> (apply + [1 2 3 4 5])
15
user=> (apply (fn [a b] (+ a b)) [1 2 3 4 5])
Execution error (ArityException) at user/eval730 (REPL:1).
Wrong number of args (5) passed to: user/eval730/fn--731
```

c - application d'une fonction sur les séquences

Les fonctions `take`, `drop`, `take-while` et `drop-while` produisent des seq à partir d'une seq :

```
user=> (take 3 (range 1 10))
(1 2 3)
user=>  (drop 3 (range 1 10))
(4 5 6 7 8 9)
user=> (take-while #(< % 4) (range 1 10))
(1 2 3)
user=> (drop-while #(< % 4) (range 1 10))
(4 5 6 7 8 9)
```

Et encore d'autres fonctions : `filter`, `some`, `sort`, `sort-by`, ...

d - seq infinie et paresseuse

Il existe des sequences qui sont infinies !

Par exemple, `(repeat 1)` va générer une sequence infinie de 1 ! (à tester à vos risques et périls …)

Certaines fonctions opérant sur des seq ne vont pas évaluer toute la seq, mais seulement les éléments nécessaires :

```
user=> (take 3 (repeat 1))
(1 1 1)
user=>  (take 10 (iterate inc 2))
(2 3 4 5 6 7 8 9 10 11)
```

d - seq infinie et paresseuse

Un exemple des seq paresseuses en action :

```
user=> (defn crible [L]
  (lazy-seq
  (cons (first L) (crible  (filter  #(not= 0 (mod % (first L)))  (next
L))))))
#'user/crible
user=>  (take 100 (crible (iterate inc 2)))
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101
103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197
199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311
313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431
433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541)
```