Programmation Fonctionnelle

Manuel Clergue

Programmer doit être amusant !

Programmer doit être beau !

Chapitre 4 : Déstructuration et références

1 - Déstructuration des données

2 - Références

1 - Déstructuration des données

a - Principes

L’accès aux données d’une structure (list, vector, map) peut être fastidieux si on utilise les manipulations de base (first, second, get …).

Exemple :

```
user=> (def segment [[0 0] [10 0]])
#'user/segment
user=> (let [p1 (first segment)
      p2 (second segment)
      x1 (first p1)
      y1 (second p1)
      x2 (first p2)
      y2 (second p2)]
  (Math/sqrt (+ (Math/pow (- x1 x2) 2) (Math/pow (- y1 y2) 2))))
```

C’est long à écrire et difficilement lisible.

a - Principes

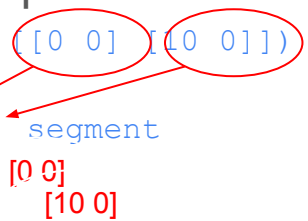
La déstructuration permet un accès aux données plus concis :

```
user=> (def segment [[0 0] [10 0]])
#'user/segment
user=> (let [[p1 p2] segment
             [x1 y1] p1
             [x2 y2] p2]
  (Math/sqrt (+ (Math/pow (- x1 x2) 2) (Math/pow (- y1 y2) 2))))

10.0
```

a - Principes

La déstructuration permet un accès au données plus concis :

```
user=> (def segment [[0 0] [10 0]])
#'user/segment
user=> (let [[p1 p2] segment
      [x1 y1] p1
      [x2 y2] p2]
  (Math/sqrt (+ (Math/pow (- x1 x2) 2) (Math/pow (- y1 y2) 2))))

10.0
```



Il existe deux déstructurations :

- la déstructuration de séquence
- la déstructuration associative

b - Déstructuration de séquences

La déstructuration de séquence est adaptée à toutes les structures de données pouvant être traversées (les vectors, les lists, les chaînes de caractères):

```
user=> (let [[x y z] [1 2 3]] (println x y z))
1 2 3

user=> (let [[x y z]  '(1 2 3)] (println x y z))
1 2 3

user=> (let [[x y z] "abc"] (println x y z))
a b c

user=> (let [[x y z] {"a" 1 "b" 2 "c" 3 }] (println x y z))
Execution error (UnsupportedOperationException) at user/eval1148 (REPL:1).
nth not supported on this type: PersistentArrayMap
```

b - Déstructuration de séquences

La déstructuration est souple à utiliser, il n'est pas obligé d'avoir le même nombre d'éléments des deux côtés :

```
user=> (let [[x y z]  '(1 2 3 4 5)] (println x y z))
1 2 3

user=> (let [[x y z]  '(1 2)] (println x y z))
1 2 nil
```

On peut choisir les éléments à extraire :

```
user=> (let [[x _ z]  '(1 2 3)] (println x  z))
1 3
```

Et récupérer le reste des éléments dans une liste :

```
user=> (let [[x & r]  '(1 2 3)] (print x) (println r))
1(2 3)
```

b - Déstructuration de séquences

La déstructuration est imbriquable :

```
user=> (let [[[x1 y1] [x2 y2]] segment]
  (Math/sqrt (+ (Math/pow (- x1 x2) 2) (Math/pow (- y1 y2) 2))))
10.0
```

Le mot-clé :as permet d'aliaser les objets déstructurés:

```
user=> (let [[[x1 y1 :as p1] [x2 y2 :as p2]] segment]
  (println "la longueur du segment entre les points " p1 " et " p2 " : "
  (Math/sqrt (+ (Math/pow (- x1 x2) 2) (Math/pow (- y1 y2) 2)))))
la longueur du segment entre les points  [0 0]  et  [10 0]  :  10.0
```

b - Déstructuration de séquences

La déstructuration est utilisable pour tous les types de bindings, explicites ou implicites :

```
user=> (defn longueur [[[x1 y1] [x2 y2]]]
  (Math/sqrt (+ (Math/pow (- x1 x2) 2) (Math/pow (- y1 y2) 2))))
#'user/longueur
user=> (longueur segment)
10.0
```

Avec les mêmes contraintes :

```
user=> (longueur  {"a" 1 "b" 2 "c" 3 })
Execution error (UnsupportedOperationException) at user/longueur (REPL:1).
nth not supported on this type: PersistentArrayMap
```

c - Déstructuration associative

On peut également déstructurer les structures associatives:

```
user=> (def personne {:nom "Dujardin" :prenom "Jean" :age "50"})
#'user/personne
user=> (let [{n :nom p :prenom a :age} personne] (println n p a))
Dujardin Jean 50
```

Eventuellement en omettant des clefs :

```
(let [{n :nom p :prenom} personne] (println n p))
```

Ou en en ajoutant :

```
user=> (let [{n :nom p :prenom t :tel} personne] (println n p t))
Dujardin Jean nil
```

Avec éventuellement une valeur par défaut :

```
user=> (let [{n :nom p :prenom t :tel :or {t "Info inconnue"} } personne] (println n p t))
Dujardin Jean Info inconnue
```

c - Déstructuration associative

Cela peut être encore plus concis :

```
user=> (let [{:keys [nom prenom age]} personne] (println nom prenom age))
Dujardin Jean 50
```

Avec les mêmes possibilités :

```
user=> (let [{:keys [nom prenom]} personne] (println nom prenom))
Dujardin Jean

user=> (let [{:keys [nom prenom tel]} personne] (println nom prenom tel))
Dujardin Jean nil

user=> (let [{:keys [nom prenom tel]  :or {tel "Info inconnue"}} personne]
         (println nom prenom tel))
Dujardin Jean Info inconnue
```

c - Déstructuration associative

Si les clefs ne sont pas des mots-clés, cela fonctionne aussi

```
user=> (let [{:strs [nom prenom]} {"nom" "Dujardin" "prenom" "Jean"}]
         (println nom prenom))
Dujardin Jean

user=> (let [{:syms [nom prenom]} {'nom "Dujardin" 'prenom "Jean"}]
         (println nom prenom))
Dujardin Jean
```

c - Déstructuration associative

Cela fonctionne de la même manière pour les records

```
user=> (defrecord Acteur [nom prenom age])
user.Acteur
user=> (def dujardin (->Acteur "Dujardin" "Jean" 50))
#'user/dujardin
user=> (defn printActeur [a]
  (let [{:keys [nom prenom age]} a]
    (println "Nom :" nom)
    (println "Prenom :" prenom)
    (println "Age :" age)))
#'user/printActeur
user=> (printActeur dujardin)
Nom : Dujardin
Prenom : Jean
Age : 50
nil
```

c - Déstructuration associative

Et tout cela peut être imbriqué :

```
user=> (def etudiant {:nom "alice" :notes [10 20 15]})
#'user/etudiant

user=> (let [{nom :nom [n1 n2 n3] :notes} etudiant] (println nom n1 n2 n3))
alice 10 20 15
```

2 - Références

a - Motivation

Il est parfois nécessaire d’accéder à des informations dynamiques. Clojure fournit un moyen de pouvoir le faire de façon sécurisée. Pour cela, il existe quatre mécanismes implémentés dans clojure : les vars, les refs, les agents et les atoms. Le premier est celui que l’on connaît déjà (définition de bindings par def). Les deux suivants sont plus adaptés au multithreading. Le dernier est plus général.

b - Les atoms

Les atoms fournissent un moyen de manipuler des états partagés, synchrones et indépendants.
L'usage des atoms est de contenir une structure immutable de Clojure et de permettre d'en changer la valeur en appliquant une fonction sur l'ancienne valeur.
Le changement est atomique (tout est fait ou rien), thread-safe (si un autre processus a changé la valeur pendant le changement, le changement est réappliqué sur la nouvelle valeur).
Parce que la fonction de changement peut être appliquée plusieurs fois, il est nécessaire qu'elle soit sans effet de bord.
C'est un moyen efficace de représenter des états indépendants avec des changements synchrones.

c - Premier exemple : la mémoïsation

Retour sur Fibonacci :

```
(defn fib [n]
  (if (<= n 1)
    n
    (+ (fib (dec n)) (fib (- n 2)))))
```

On a déjà vu que cette fonction refait plusieurs fois (et même un nombre exponentiel de fois) le même calcul. Ce qui se voit lorsqu'on calcule le temps d'exécution :

```
user=> (time (fib 35))
"Elapsed time: 1892.890188 msecs"
9227465
```

On souhaiterait pouvoir "sauvegarder" les calculs intermédiaires pour pouvoir les retrouver quand on en a besoin ...

c - Premier exemple : la mémoïsation

On peut le faire en créant un atom portant une table associative :

```
(def m (atom {}))
```

Au fur et à mesure du calcul, chaque valeur calculée va être ajoutée à la table. Lorsqu'on appelle la fonction avec une valeur, on commence par vérifier qu'elle n'a pas déjà été calculée. Si c'est le cas, la valeur est disponible dans la table, et on la renvoie. Sinon, on fait le calcul et on intègre la valeur dans la table.

```
(defn fibo [n]
  (if-let [e (find @m n)]
    (val e)
    (let [ret (if (<= n 1)
                n
                (+ (fibo (dec n)) (fibo (- n 2))))]
      (swap! m assoc n ret)
      ret)))
```

c - Premier exemple : la mémoïsation

On peut le faire en créant un atom portant une table associative :

```
(def m (atom {}))
```

Au fur et à mesure du calcul, chaque valeur calculée va être ajoutée à la table. Lorsqu'on appelle la fonction avec une valeur, on commence par vérifier qu'elle n'a pas déjà été calculée. Si c'est le cas, la valeur est disponible dans la table, et on la renvoie. Sinon, on fait le calcul et on intègre la valeur dans la table.

```
(defn fibo [n]
  (if-let [e (find @m n)]
    (val e)
    (let [ret (if (<= n 1)
                n
                (+ (fibo (dec n)) (fibo (- n 2))))]
      (swap! m assoc n ret)
      ret)))
```

déréférencement

modification

c - Premier exemple : la mémoïsation

Le temps de calcul est réduit drastiquement :

```
user=> (time (fibo 35))
"Elapsed time: 0.986724 msecs"
9227465
```

Il y a évidemment un prix à payer : on échange du temps contre de l'espace. Mais dans le cas de fibonacci, c'est très rentable : on passe d'une complexité exponentielle à linéaire en temps, tout en passant d'une complexité constante à linéaire en espace (et encore, c'est sans compter la taille de la pile d'exécution).

```
user=> @m
{0 0, 7 13, 20 6765, 27 196418, 1 1, 24 46368, 4 3, 15 610, 21 10946, 31 1346269, 32
2178309, 33 3524578, 13 233, 22 17711, 29 514229, 6 8, 28 317811, 25 75025, 34 5702887, 17
1597, 3 2, 12 144, 2 1, 23 28657, 35 9227465, 19 4181, 11 89, 9 34, 5 5, 14 377, 26 121393,
16 987, 30 832040, 10 55, 18 2584, 8 21}
user=> (count @m)
36
```

c - Premier exemple : la mémoïsation

On peut intégrer l'atom à la définition de la fonction :

```
(defn fib-mem [n]
  (let [memo (atom {0 0, 1 1})]
    (letfn  [(fib-aux  [n]
                       (if-let [e (find @memo n)]
                         (val e)
                         (let [ret (+ (fib-aux (dec n)) (fib-aux (- n 2)))]
                           (swap! memo assoc n ret)
                           ret)))]
      (fib-aux n))))

user=> (time (fib-mem 35))
"Elapsed time: 1.059933 msecs"
9227465
```

Mais l'atom ne persiste plus à l'extérieur de la fonction …

c - Premier exemple : la mémoïsation

Il est même possible de créer une fonction de niveau supérieur (ako *décorateur*) pour la mémoïsation :

```
(defn memoize [f]
  (let [mem (atom {})]
    (fn [& args]
      (if-let [e (find @mem args)]
        (val e)
        (let [ret (apply f args)]
          (swap! mem assoc args ret)
          ret)))))

(def fib (memoize fib))

user=> (time (fib 35))
"Elapsed time: 0.223625 msecs"
9227465
```

La fonction memoize existe par défaut dans clojure ...

c - Premier exemple : la mémoïsation

La mémoïsation fonctionne également quand il y a plusieurs arguments :

```
(defn binom [n p]
  (cond (zero? p) 1
        (= n p) 1
        (> p n) 0
        :else (+'  (binom (dec n) p) (binom (dec n) (dec p)) )))

user=> (time (binom 32 16))
"Elapsed time: 29530.316404 msecs"
601080390

(def binom (memoize binom))

user=>  (time (binom 32 16))
"Elapsed time: 11.175657 msecs"
601080390
```

d - Deuxième exemple : la construction de résultats

Le problème des n-reines est un problème classique en informatique. Il s'agit de poser n reines (d'échec) sur un échiquier nxn de manière à ce qu'aucune reine ne soit en situation de prise avec une autre. Le problème se brute-force, en étant un peu malin :

```
(def plateau 8)

(defn reine-test [rcol rvect]
  (letfn [(f [x] (let [[ligne colonne] x]
                (or
                  (= rcol colonne)
                  (= rcol (- colonne (+ ligne  1)))
                  (= rcol (+ colonne (+ ligne  1))))))]
  (empty? (filter f (map-indexed vector (rseq rvect))))))

(defn nreine [sol]
  (doseq [p (range plateau)]
    (if (reine-test p sol)
      (let [nsol  (conj sol p)]
        (if (= (count nsol) plateau) (println nsol) (nreine nsol))))))
```

d - Deuxième exemple : la construction de résultats

En rassemblant le tout dans une fonction :

```
(defn nreine [taille]
  (letfn [(reine-test [rcol rvect]
            (letfn [(f [x] (let [[ligne colonne] x]
                             (or
                              (= rcol colonne)
                              (= rcol (- colonne (+ ligne  1)))
                              (= rcol (+ colonne (+ ligne  1))))))]
              (empty? (filter f (map-indexed vector (rseq rvect))))))
          (nreine-aux [sol]
            (doseq [p (range taille)]
              (if (reine-test p sol)
                (let [nsol  (conj sol p)]
                  (if (= (count nsol) taille) (println nsol) (nreine-aux nsol)))))))]
    (nreine-aux [])))
```

d - Deuxième exemple : la construction de résultats

Le problème de cette fonction, c'est qu'elle fournit un affichage. Il serait intéressant de pouvoir avoir une liste (ou un vecteur) des différentes solutions pour pouvoir éventuellement le traiter.
Pour cela, on peut triturer la fonction de manière à obtenir le résultat souhaité, mais c'est difficile et risque d'amener à un algorithme beaucoup moins clair.
Avec un atom permettant de manipuler un état, on obtient avec très peu de modifications, ce que l'on souhaite !

d - Deuxième exemple : la construction de résultats

En rassemblant le tout dans une fonction :

```
(defn nreine [taille]
  (let [solutions (atom [])]
    (letfn [(reine-test [rcol rvect]
              (letfn [(f [x] (let [[ligne colonne] x]
                               (or
                                (= rcol colonne)
                                (= rcol (- colonne (+ ligne  1)))
                                (= rcol (+ colonne (+ ligne  1))))))]
                (empty? (filter f (map-indexed vector (rseq rvect))))))
            (nreine-aux [sol]
              (doseq [p (range taille)]
                (if (reine-test p sol)
                  (let [nsol  (conj sol p)]
                    (if (= (count nsol) taille)
                      (swap! solutions conj nsol)
                      (nreine-aux nsol)))))))]
      (nreine-aux [])
      @solutions)))
```

e - programmation concurrente

C’est l’une des motivations majeures de la création du langage Clojure !
Fournir une interface propre et sécurisée aux threads de la JVM.

Les principes de la programmation concurrente sont hors champ de ce cours …
(vous pouvez consulter : https://www.braveclojure.com/concurrency/ pour une intro dans le cadre de Clojure)

Il y a 3 mécanismes :

- future
- delay
- promise