Variabili mutabili e strutture dati persistenti

Concorrenza in Clojure

Sommario

- Clojure
- Strutture dati persistenti
- Variabili mutabili
  - Atom
  - Agents
  - Ref
- Esempio filosofi a cena con STM
- Conclusioni

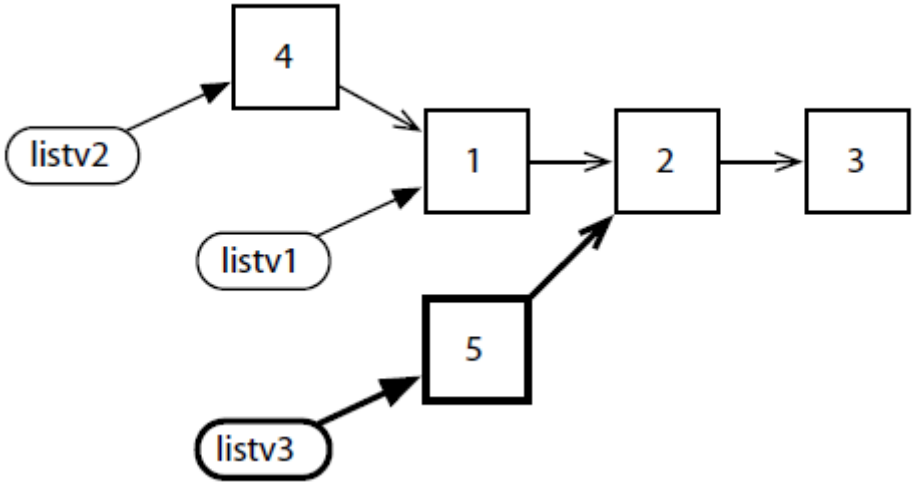
Clojure

- Clojure è un linguaggio funzionale impuro;
- Per la concorrenza non utilizza lock né synchronzed
- Strutture dati persistenti;
- Variabili mutabili:
  - Atoms
  - Agents
  - Refs

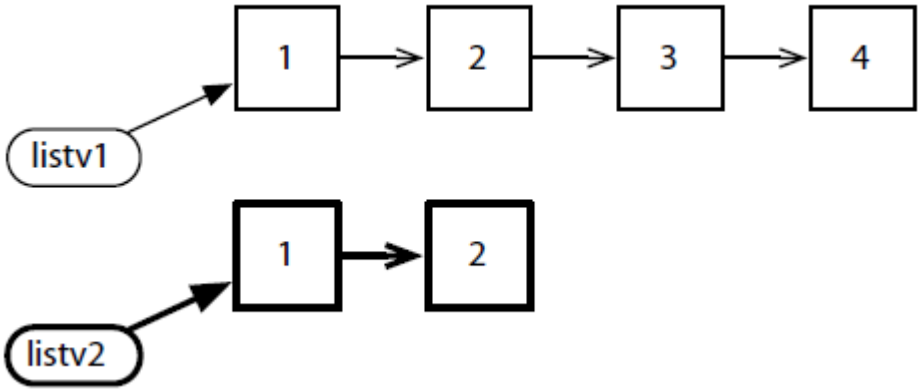
Strutture dati persistenti

- Le strutture dati persistenti preservano le proprie versioni precedenti alle modifiche
- Non sempre sono necessarie copie
- Tutte le collezioni in Clojure sono persistenti
- Distinzione tra stato e identità

```
user=> (def listv1 (list 1 2 3))
#'user/listv1
user=> listv1
(1 2 3)
user=> (def listv2 (cons 4 listv1))
#'user/listv2
user=> listv2
(4 1 2 3)
user=> (def listv3 (cons 5 (rest listv1)))
#'user/listv3
user=> listv3
(5 2 3)
```




```
user=> (def listv1 (list 1 2 3 4))
#'user/listv1
user=> (def listv2 (take 2 listv1))
#'user/listv2
user=> listv2
(1 2)
```



Atoms

- Variabili mutabili
- Permettono modifiche indipendenti e sincrone su singoli valori senza uso di lock
- Costruiti a partire da java.util.concurrent.atomic
- Il metodo di modifica swap!, in caso di modifiche da parte di altri altri thread, si ripete

```
Line 1 (def players (atom ()))
  -
  - (defn list-players []
  -   (response (json/encode @players)))
  5
  - (defn create-player [player-name]
  -   (swap! players conj player-name)
  -   (status (response "") 201))
  -
 10 (defroutes app-routes
  -   (GET "/players" [] (list-players))
  -   (PUT "/players/:player-name" [player-name] (create-player player-name)))
  - (defn -main [& args]
  -   (run-jetty (site app-routes) {:port 3000}))
```

Agents

- Permettono cambi indipendenti e asincroni su singoli valori
- Metodo di modifica send, se in serie vengono serializzati
- Fuzioni await e await-for per gestire azioni asincrone

```
user=> (def my-agent (agent 0))
#'user/my-agent
user=> @my-agent
0
user=> (send my-agent inc)
#<Agent@2cadd45e: 1>
user=> @my-agent
1
user=> (send my-agent + 2)
#<Agent@2cadd45e: 1>
user=> @my-agent
?
```

Refs

- Permettono cambiamenti coordinati e sincroni su più valori
- Possono essere modificati solo all'interno di transazioni, ovvero operazioni atomiche, consistenti e isolate
  - Dosync avvia una transazione
- Meccanismo di Software Transactional Memory
- In caso di conflitti, le transazioni si ripetono

```
(defn transfer [from to amount]
  (dosync
    (alter from - amount)
    (alter to + amount)))

user=> (def checking (ref 1000))
#'user/checking
user=> (def savings (ref 2000))
#'user/savings
user=> (transfer savings checking 100)
1100
user=> @checking
1100
user=> @savings
1900
```

Filosofi a cena con STM - 1

```
Line 1  (def philosophers (into [] (repeatedly 5 #(ref :thinking))))
    -
    -   (defn think []
    -     (Thread/sleep (rand 1000)))
    5
    -   (defn eat []
    -     (Thread/sleep (rand 1000)))
    -
    -   (defn philosopher-thread [n]
   10     (Thread.
    -       #(let [philosopher (philosophers n)
    -              left (philosophers (mod (- n 1) 5))
    -              right (philosophers (mod (+ n 1) 5))]
    -          (while true
   15            (think)
    -            (when (claim-chopsticks philosopher left right)
    -              (eat)
    -              (release-chopsticks philosopher))))))
    -
   20   (defn -main [& args]
    -     (let [threads (map philosopher-thread (range 5))]
    -       (doseq [thread threads] (.start thread))
    -       (doseq [thread threads] (.join thread))))
```

Filosofi a cena con STM - 2

```
(defn release-chopsticks [philosopher]
  (dosync (ref-set philosopher :thinking)))
(defn claim-chopsticks [philosopher left right]
  (dosync
    (when (and (= (ensure left) :thinking) (= (ensure right) :thinking))
      (ref-set philosopher :eating))))
```

Conclusioni

- Pro
  - ✓ Pensato per raccogliere il meglio della programmazione funzionale e imperativa
  - ✓ Riduce lo sforzo richiesto al programmatore
- Contro
  - × Nessun supporto per la programmazione distribuita
  - × Minore efficienza rispetto ai linguaggi imperativi.