

Sviluppo



Di Michele Costabile



Un linguaggio creato per la virtual machine di Java, con radici profonde nella tradizione del Lisp.

È arrivata l'ora di Clojure

Rich Hickey è un uomo che sa quello che vuole ed è un hacker con competenze sui linguaggi di sviluppo con pochi confronti. Hickey, infatti, di fronte a una lunga lista di critiche agli ambienti di produzione esistenti, ha scelto di girarla in positivo e adottarla come specifica per un nuovo linguaggio, realizzato da zero. La sua scelta ha dato i natali a Clojure. Il nome viene da chiusura (closure), un termine familiare a chi ci ha seguito, su queste pagine, nei meandri di JavaScript e nelle notizie sull'evoluzione dei linguaggi più diffusi, da C++ a C#. La storpiatura dell'ortografia, dà una poco rassicurante assonanza con *conjurare* (congiura), ma è servita per avere un nome con zero occorrenze su Google all'inizio del progetto. Con il senno di poi, alcuni pensano che la *j* venga da Java, mentre qualcuno si spinge anche a pensare alle prime lettere come un acronimo di Common Lisp on Java. Clojure, infatti, è proprio questo: un Lisp che gira sulla Java Virtual Machine. In realtà, non è l'unico Lisp che può girare sulla Jvm (e sulla macchina virtuale CLR della piattaforma Microsoft) e

non basterebbe solo questo a renderlo interessante. Il punto centrale è che Clojure è un linguaggio pensato per rendere più produttivo il lavoro di chi crea applicazioni web, senza perdere una virgola della qualità e della robustezza dell'ambiente della Jvm. Un punto di partenza del progetto, infatti, è che Clojure è il linguaggio e la Jvm, l'ambiente di programmazione. Come di consueto con le piattaforme Lisp, lo sviluppo con Clojure è dinamico, ovvero non c'è necessità di invocare il compilatore mentre si scrive il codice e si definiscono funzioni, un vantaggio non da poco, dato che i cicli di compilazione e test, semplicemente non esistono più.

Ma la dinamicità non è l'unico dono di Clojure, ci sono anche due aspetti importanti del linguaggio a venire in aiuto della produttività. Il primo è che Clojure supporta un paradigma tipico dei linguaggi funzionali: strutture di dati immutabili. Il secondo

è il supporto nativo per la concorrenza. Dobbiamo anche aggiungere una buona dose di scetticismo per la programmazione a oggetti e il supporto per il polimorfismo e abbiamo le caratteristiche principali del cocktail.

PERCHÉ UNA NUOVA ARCHITETTURA

Le motivazioni per la creazione di Clojure sono espresse con molta chiarezza alla pagina clojure.org/rationale. Leggendole, colpisce quanto sia difficile non condividere tutti i punti esposti. In primis, quello che Hickey si era proposto di realizzare era un Lisp, con estensioni per la programmazione funzionale, con un alto grado di integrazione con una piattaforma software affermata.

Le macchine virtuali sono piattaforme più affidabili dei sistemi operativi, secondo Hickey, perché definiscono un sistema di tipi e un insieme di librerie che astraggono il sistema operativo. Per questo sono più interessanti degli OS stessi per chi investe in un ambiente di sviluppo. Come conferma di questo punto di vista abbiamo la macchina virtuale Java e il modo in cui ha finito per sostituirsi a Unix e Windows presso i grandi clienti.

La scelta del Lisp come linguaggio si basa su ottimi motivi, che vanno da quanto già accennato (la rapidità di sviluppo in un ambiente così dinamico da non richiedere mai la compilazione quando si modificano i sorgenti) fino all'efficienza dell'architettura nonostante la sua età: realizzare un

Tra Lisp e Java

La doti di Clojure sono la sua dinamicità e la modesta richiesta di risorse

ambiente Lisp richiede poche risorse e il core del linguaggio è semplicissimo. Clojure non mira alla compatibilità con un ambiente Lisp in particolare, anzi cestina allegramente questi requisiti in favore della compatibilità con la Java Virtual Machine. Una scelta condivisibile, dato che Common Lisp e Scheme sono già disponibili (per esempio, si può provare Kawa, uno Scheme open source). Piuttosto che imbarcarsi in un percorso certamente costellato di problemi di compatibilità, Clojure sceglie l'integrazione con la Jvm. Questo è ancora più corretto considerando che, tradizionalmente, i Lisp fanno piattaforma a sé e portare un ambiente con tutta la sua piattaforma conduce a una duplicazione delle librerie inevitabile. Nel momento in cui si sceglie di usare il Lisp come linguaggio e la Jvm come ambiente operativo, la scelta è logica.

La programmazione funzionale è in voga di questi tempi. Abbiamo visto affermarsi Scala, grazie anche all'endorsement di Twitter, fino a diventare la soluzione più votata per la categoria "i linguaggi che vorrei imparare" nel sondaggio dei Rebel Labs (zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/). Questo, grazie anche al fatto che conoscere Scala è uno skill monetizzabile, e, almeno negli Stati Uniti, può fruttare un colloquio di lavoro.

Uno dei figli di ML, F#, è saldamente fuori dai laboratori e fa parte a tutti gli effetti dell'offerta di linguaggi Microsoft. Abbiamo parlato di lambda e type inference in C++11 e vediamo elementi del paradigma funzionale anche nell'ultimo nato di casa Apple, il linguaggio Swift, da poco presentato alla WWDC 2014.

Quello che rende interessante il modello di programmazione funzionale, oltre all'ovvio focus sulle funzioni, piuttosto che sulle strutture dati, è l'accesso alla memoria.

Alcuni linguaggi, come Clojure, si fondano sull'immutabilità delle variabili, che possono essere assegnate una sola volta. Anche una lista può essere estesa solo creando un'altra lista. Ci sono modi di realizzare questo modello senza penalizzare le performance in modo avvertibile, riducendo al minimo le copie. Il beneficio di questo approccio è che è possibile ottenere che l'elaborazione di una funzione non abbia effetti secondari sull'ambiente

circostante e agisca solo sul suo valore di ritorno. Questo approccio rende il codice sicuro e prevedibile. In alcuni linguaggi, come ML, possiamo assumere che il codice compilato non possa generare errori di run time.

In Clojure, come in Scala, l'approccio funzionale è morbido, nel senso che il linguaggio non impedisce a una funzione di creare effetti secondari, cioè di alterare i parametri o variabili globali, ma comunque le funzioni di libreria sono prive di effetti secondari. Sta al programmatore scegliere quando mantenere la disciplina di un approccio puramente funzionale e quando, invece, optare per un approccio più pragmatico, cosa che non ci sentiamo di deprecare, perché avere delle scelte è una buona cosa, così come è una buona cosa avere un linguaggio sicuro. Parlando di funzioni, osserviamo che

gli argomenti sono tipizzati e Clojure supporta il polimorfismo attraverso una parola chiave specifica *defmulti*, invece di *def*. Si veda il riquadro per un esempio di funzione polimorfa in pratica. In generale, le funzioni accettano argomenti non tipizzati, ma la valutazione degli argomenti può portare a errori di runtime, come in questo esempio.

```
user=> (+ "ciao" "michele")
```

```
ClassCastException java.lang.String
cannot be cast to java.lang.Number
clojure.lang.Numbers.add (Numbers.
java:126)
```

SPERIMENTARE CON CLOJURE

Muovere i primi passi è semplice. Non è nemmeno indispensabile scaricare

LISTATO 1

Questo è il main del server di Try Clojure (triclj.com), nulla di particolare e nulla di diverso, come è ovvio, da quello che potremmo scrivere in Node.js o con altre piattaforme server. Si veda come viene usato il server Jetty, una popolare alternativa light a Tomcat.

```
(ns tryclojure.server
  (:use compojure.core)
  (:require [compojure.route :as route]
            [noir.util.middleware :as nm]
            [ring.adapter.jetty :as jetty]
            [tryclojure.views.home :as home]
            [tryclojure.views.tutorial :as tutorial]
            [tryclojure.views.eval :as eval]))

(def app-routes
  [(GET "/" [] (home/root-html))
   (GET "/about" [] (home/about-html))
   (GET "/links" [] (home/links-html))
   (POST "/tutorial" [:as {args :params}] (tutorial/tutorial-html
                                           (args :page)))
   (POST "/eval.json" [:as {args :params}] (eval/eval-json (args
                                                             :expr) (args :jsonp)))
   (GET "/eval.json" [:as {args :params}] (eval/eval-json (args :expr)
                                                           (args :jsonp)))
   (route/resources "/")
   (route/not-found "Not Found")])

(def app (nm/app-handler app-routes))

(defn -main [port]
  (jetty/run-jetty app {:port (Long. port) :join? false}))
```



LISTATO 2

Questo è un esempio del polimorfismo di Clojure, estratto dalla documentazione del wiki ufficiale. Si definisce un metodo encounter (incontro), che ha un comportamento diverso a seconda della specie degli oggetti che si incontrano, in questo caso leoni e conigli. Si noti che defmulti definisce una funzione polimorfa, mentre defmethod definisce un metodo per la funzione (che probabilmente si traduce in una classe Java). Vediamo che defmethod contiene una lista di tipi, una lista di parametri e il corpo della funzione.

```
(defmulti encounter (fn [x y]
  [(:Species x) (:Species y)]))
```

```
(defmethod encounter [[:Bunny :Lion]
  [b1] :run-away)
(defmethod encounter [[:Lion :Bunny]
  [l1 b] :eat)
(defmethod encounter [[:Lion :Lion]
  [l1 l2] :fight)
(defmethod encounter [[:Bunny
  :Bunny] [b1 b2] :mate)
```

```
(def b1 {:Species :Bunny :other
  :stuff})
(def b2 {:Species :Bunny :other
  :stuff})
(def l1 {:Species :Lion :other
  :stuff})
(def l2 {:Species :Lion :other
  :stuff})
```

```
(encounter b1 b2)
-> :mate
(encounter b1 l1)
-> :run-away
(encounter l1 b1)
-> :eat
(encounter l1 l2)
-> :fight
```

l'ambiente clojure, anche se nella versione slim è più piccolo di un mp3. Si può cominciare con un ambiente di esecuzione Clojure, un Repl (read eval print loop) in gergo Lisp, online all'indirizzo <http://tryclj.com>. L'ambiente di esecuzione guida l'utente nei suoi primi passi con il linguaggio in modo soffice, secondo uno schema reso famoso dal programmatore passato alla storia come Why the lucky stiff con Ruby.

Scaricare l'ambiente Clojure non è difficile. Le istruzioni sono alla pagina clojure.org/downloads. Chi non è interessato ai sorgenti può scaricare l'archivio jar con la distribuzione all'indirizzo build.clojure.org. La versione "magra" del software è meno di un MByte, mentre l'archivio completo ne richiede tre e mezzo. Per eseguire il Repl si può usare la riga di comando:

```
java -cp clojure-1.6.0.jar
clojure.main
```

nella cartella in cui si è scaricato l'archivio jar. La somma di due numeri è in perfetto stile Lisp

```
> (+ 3 3)
6
```

Provando una divisione scopriamo il supporto di Clojure per i numeri razionali

```
> (/ 10 3)
10/3
> (/ 10 3.0)
3.3333333333333335
```

Solo se forziamo un parametro in virgola mobile avremo un risultato in virgola mobile.

Le funzioni possono anche avere liste di parametri, ecco qualche esempio

```
> (+ 1 2 3 4 56 )
21
> (/ 10 2 5)
1
> (/ 12 4 2)
3/2
```

Le funzioni si definiscono con defn

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

```
#'sandbox17930/square
> (square 10)
100
```

Questo, invece è l'esempio di una funzione anonima invocata dove è definita

```
> ((fn [x] (* x x)) 10)
100
```

In effetti, defn è un'abbreviazione di def. Ecco un esempio in cui def definisce il nome e fn definisce la funzione

```
> (def square (fn [x] (* x x)))
#'sandbox17930/square
```

Ecco un esempio di funzione applicata a una lista di argomenti, in questo caso la funzione inc, che incrementa una lista di numeri.

```
> (map inc [1 2 3 4])
(2 3 4 5)
```

CONCLUSIONI

Clojure, alla fine dei test, si è guadagnato la nostra simpatia. È un ambiente pragmatico che offre il beneficio di un ciclo di produzione leggero, nel modo tipico degli ambienti interpretati, ma con le performance e l'affidabilità della macchina virtuale Java. Ricordiamo che la Jvm è un ambiente ottimizzato per più di un decennio.

In Clojure il Lisp è il linguaggio di programmazione, cosa che ci piace, ma l'ambiente di esecuzione è quello abituale della macchina virtuale Java, cosa che ci piace ancora di più. Non ci sono nuovi framework da imparare, solo un linguaggio piccolo piccolo.

Le performance dell'ambiente di esecuzione sono eccellenti, come si può verificare sul sito Debian dedicato al confronto fra i vari linguaggi (bit.ly/clojureperf). Si vede che la differenza di prestazioni fra Java e Clojure è microscopica e siamo nella stessa categoria di C#, Go, Scala, Haskell e OCaml, poco dietro il C++.

In sostanza, se ci va di sperimentare un ambiente di esecuzione più dinamico e un linguaggio più leggero e flessibile di Java, possiamo farlo, continuando a lavorare con i soliti strumenti e possiamo anche non dire al nostro cliente che il Jar, che installiamo nel consueto ambiente enterprise (si veda bit.ly/cljdeploy), è stato creato con un linguaggio diverso da Java. •