EBook Gratuito

APPRENDIMENTO clojure

Free unaffiliated eBook created from
Stack Overflow contributors.

#clojure

Sommario

Di

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: clojure

It is an unofficial and free clojure ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official clojure.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con Clojure

Osservazioni

Clojure **è un linguaggio di programmazione generico con tipizzazione dinamica con sintassi Lisp.**

Le sue funzioni supportano lo stile funzionale della programmazione con funzioni di prima classe e valori immutabili per impostazione predefinita. L'uso di variabili riassegnabili non è così facile in Clojure come in molti linguaggi tradizionali, dal momento che le variabili devono essere create e aggiornate come oggetti contenitore. Questo incoraggia l'uso di valori puri che rimarranno come erano nel momento in cui sono stati visti l'ultima volta. Questo in genere rende il codice molto più prevedibile, verificabile e compatibile con la concorrenza. Questo funziona anche per le raccolte, dal momento che le strutture di dati incorporate di Clojure sono persistenti.

Per quanto riguarda le prestazioni, Clojure supporta il suggerimento del tipo per eliminare il riflesso non necessario laddove possibile. Inoltre, i gruppi di modifiche alle raccolte persistenti possono essere eseguiti su versioni *transitorie* , riducendo la quantità di oggetti coinvolti. Ciò non è necessario la maggior parte del tempo, poiché le collezioni persistenti sono veloci da copiare poiché condividono la maggior parte dei loro dati. Le loro garanzie prestazionali non sono lontane dalle loro controparti mutevoli.

Tra le altre caratteristiche, Clojure ha anche:

- memoria transazionale software (STM)
- diverse primitive di concorrenza che non prevedono il blocco manuale (atomo, agente)
- trasformatori di sequenza componibili (trasduttori),
- strutture funzionali per la manipolazione degli alberi (cerniere)

Grazie alla sua sintassi semplice ed elevata estensibilità (tramite macro, implementazione di interfacce e riflessioni core), è possibile aggiungere alcune funzionalità di linguaggio comunemente viste a Clojure con le librerie. Ad esempio, `core.typed` porta un controllo di tipo statico, `core.async` porta meccanismi di concorrenza semplici basati su canali, `core.logic` porta la programmazione logica.

Progettato come linguaggio ospitato, può interagire con la piattaforma su cui gira. Mentre l'obiettivo principale è JVM e l'intero ecosistema dietro Java, le implementazioni alternative possono essere eseguite anche in altri ambienti, come ClojureCLR in esecuzione su Common Language Runtime o ClojureScript in esecuzione su runtime JavaScript (inclusi i browser Web). Mentre le implementazioni alternative possono mancare di alcune funzionalità dalla versione di

JVM, sono ancora considerate una famiglia di lingue.

Versioni

Versione	Registro delle modifiche	Data di rilascio
1.8	Ultimo registro delle modifiche	2016/01/19
1.7	Registro delle modifiche 1.7	2015/06/30
1.6	Registro modifiche 1.6	2014/03/25
1.5.1	Registro delle modifiche 1.5.1	2013/03/10
1.4	Registro delle modifiche 1.4	2012-04-15
1.3	Registro delle modifiche 1.3	2011-09-23
1.2.1		2011-03-25
1.2		2010-08-19
1.1		2010-01-04
1.0		2009-05-04

Examples

Installazione e configurazione

Opzione 1: Leiningen

Richiede JDK 6 o versioni successive.

Il modo più semplice per iniziare con Clojure è scaricare e installare Leiningen, lo strumento standard di fatto per gestire i progetti Clojure, quindi eseguire `lein repl` per aprire un REPL .

Linux

```
curl https://raw.githubusercontent.com/technomancy/leiningen/stable/bin/lein > ~/bin/lein
export PATH=$PATH:~/bin
chmod 755 ~/bin/lein
```

OS X

Segui i passaggi di Linux sopra o installa con i gestori di pacchetti macOS.

Installa con Homebrew

```
brew install leiningen
```

Installa con MacPorts

Prima installa Clojure

```
sudo port -R install clojure
```

Installa `Leiningen` , uno strumento di costruzione per Clojure

```
sudo port -R install leiningen
```

```
lein self-install
```

finestre

Vedi la documentazione ufficiale .

Opzione 2: distribuzione ufficiale

Richiede JRE 6 o versioni successive.

Le versioni di Clojure sono pubblicate come semplici file JAR da eseguire su JVM. Questo è ciò che accade normalmente all'interno degli strumenti di costruzione Clojure.

1. Vai su http://clojure.org e scarica l'ultimo archivio Clojure
2. Estrai il file ZIP scaricato in una directory a tua scelta
3. Esegui `java -cp clojure-1.8.0.jar clojure.main` in quella directory

   Potrebbe essere necessario sostituire `clojure-1.8.0.jar` in quel comando per il nome del file JAR effettivamente scaricato.

   Per una migliore esperienza REPL da riga di comando (ad es. `rlwrap rlwrap java -cp clojure-1.8.0.jar clojure.main` comandi precedenti), è possibile installare `rlwrap` : `rlwrap java -cp clojure-1.8.0.jar clojure.main`

Opzione 3: avvio

Richiede JDK 7 o versioni successive.

Boot è uno strumento di creazione Clojure multiuso. Comprenderlo richiede una certa conoscenza di Clojure, quindi potrebbe non essere l'opzione migliore per i principianti. Vedere il sito Web (fare clic su *Inizia* qui) per le istruzioni di installazione.

Una volta installato e nel tuo `PATH` , puoi avviare `boot repl` ovunque per avviare un REPL Clojure.

"Ciao mondo!" nella REPL

La comunità di Clojure pone una grande enfasi sullo sviluppo interattivo, quindi un sacco di interazione con Clojure avviene all'interno di un REPL (read-eval-print-loop) . Quando si inserisce un'espressione in esso, Clojure lo **legge** , lo **valuta** e **stampa** il risultato della valutazione, il tutto in un **ciclo** .

Dovresti essere in grado di lanciare un REPL Clojure adesso. Se non sai come, segui la sezione **Installazione e configurazione** in questo argomento. Una volta eseguito, digita quanto segue:

```
(println "Hello, world!")
```

Quindi premi `Invio` . Questo dovrebbe stampare `Hello, world!` seguito dal valore restituito da questa espressione, `nil` .

Se vuoi eseguire qualche clojure all'istante, prova REPL online. Ad esempio http://www.tryclj.com/ .

Crea una nuova applicazione

Dopo aver seguito le istruzioni sopra e aver installato Leiningen, avviare un nuovo progetto eseguendo:

```
lein new <project-name>
```

Questo imposterà un progetto Clojure con il modello Leiningen predefinito all'interno della cartella `<project-name>` . Esistono diversi modelli per Leiningen, che influiscono sulla struttura del progetto. Più comunemente viene utilizzata la "app" di modello, che aggiunge una funzione principale e prepara il progetto da impacchettare in un file jar (che è la funzione principale il punto di accesso dell'applicazione). Questo può essere ottenuto con questo eseguendo:

```
lein new app <project-name>
```

Supponendo che tu abbia usato il modello di app per creare una nuova applicazione, puoi verificare che tutto sia stato impostato correttamente, inserendo la directory creata ed eseguendo l'applicazione usando:

```
lein run
```

Se vedi `Hello, World!` sulla tua console, sei pronto e pronto per iniziare a costruire la tua applicazione.

Puoi impacchettare questa semplice applicazione in due file jar con il seguente comando:

```
lein uberjar
```

"Ciao mondo!" usando Boot

Nota: è necessario installare Boot prima di provare questo esempio. Vedi la sezione **Installazione e configurazione** se non l'hai ancora installato.

Boot consente di creare file Clojure eseguibili usando la linea **shebang** (#!). Inserisci il seguente testo in un file a tua scelta (questo esempio presuppone che sia nella "directory di lavoro corrente" e si chiami `hello.clj`).

```
#!/usr/bin/env boot

(defn -main [& args]
  (println "Hello, world!"))
```

Quindi contrassegnalo come eseguibile (se applicabile, in genere eseguendo `chmod +x hello.clj`).

... ed `./hello.clj` (`./hello.clj`).

Il programma dovrebbe produrre "Hello, world!" E finito.

Crea una nuova applicazione (con avvio)

```
boot -d seancorfield/boot-new new -t app -n <appname>
```

Questo comando dirà all'avvio di acquisire l'attività `boot-new` da https://github.com/seancorfield/boot-new e di eseguire l'attività con il modello `app` (vedere il collegamento per altri modelli). L'attività creerà una nuova directory denominata `<appname>` con una tipica struttura dell'applicazione Clojure. Vedi il README generato per maggiori informazioni.

Per eseguire l'applicazione: `boot run` . Altri comandi sono specificati in `build.boot` e descritti nel README.

Leggi Iniziare con Clojure online: https://riptutorial.com/it/clojure/topic/827/iniziare-con-clojure

Capitolo 2: Atomo

introduzione

Un atomo in Clojure è una variabile che può essere cambiata nel tuo programma (spazio dei nomi). Poiché la maggior parte dei tipi di dati in Clojure sono immutabili (o immutabili) - non è possibile modificare il valore di un numero senza ridefinirlo - gli atomi sono essenziali nella programmazione Clojure.

Examples

Definisci un atomo

Per definire un atomo, usa un `def` ordinario, ma aggiungi una funzione `atom` prima di esso, in questo modo:

```
(def counter (atom 0))
```

Questo crea un `atom` di valore `0` . Gli atomi possono essere di qualsiasi tipo:

```
(def foo (atom "Hello"))
(def bar (atom ["W" "o" "r" "l" "d"]))
```

Leggi il valore di un atomo

Per leggere il valore di un atomo, basta mettere il nome dell'atomo, con un `@` prima di esso:

```
@counter ; => 0
```

Un esempio più grande:

```
(def number (atom 3))
(println (inc @number))
;; This should output 4
```

Aggiorna il valore di un atomo

Ci sono due comandi per cambiare un atomo, `swap!` e `reset!` . `swap!` viene dato comandi e cambia l'atomo in base al suo stato corrente. `reset!` cambia completamente il valore dell'atomo, indipendentemente da quale fosse il valore dell'atomo originale:

```
(swap! counter inc) ; => 1
(reset! counter 0) ; => 0
```

Questo esempio emette le prime 10 potenze di `2` usando gli atomi:

```
(def count (atom 0))

(while (< @atom 10)
  (swap! atom inc)
  (println (Math/pow 2 @atom)))
```

Leggi Atomo online: https://riptutorial.com/it/clojure/topic/7519/atomo

Capitolo 3: CLJ-time

introduzione

Questo documento riguarda come manipolare la data e l'ora in clojure.

Per utilizzarlo nella tua applicazione, vai al tuo file project.clj e includi [clj-time "<version_number>"] nella tua sezione: dependencies.

Examples

Creare un tempo Joda

```
(clj-time/date-time 2017 1 20)
```

Ti dà un tempo Joda del 20 gennaio 2017 alle 00:00:00.

Ore, minuti e secondi possono anche essere specificati come

```
(clj-time/date-time year month date hour minute second millisecond)
```

Ottenere giorno mese anno ora minuto secondo dall'ora della data

```
(require '[clj-time.core :as t])

(def example-time (t/date-time 2016 12 5 4 3 27 456))

(t/year example-time) ;; 2016
(t/month example-time) ;; 12
(t/day example-time) ;; 5
(t/hour example-time) ;; 4
(t/minute example-time) ;; 3
(t/second example-time) ;; 27
```

Confronto tra due date

```
(require '[clj-time.core :as t])

(def date1 (t/date-time 2016 12 5))
(def date2 (t/date-time 2016 12 6))

(t/equal? date1 date2) ;; false
(t/equal? date1 date1) ;; true

(t/before? date1 date2) ;; true
(t/before? date2 date1) ;; false

(t/after? date1 date2) ;; false
(t/after? date2 date1) ;; true
```

Controllare se un tempo è all'interno di un intervallo di tempo

Questa funzione indica se un dato tempo si trova all'interno di un dato intervallo di tempo.

```
(require '[clj-time.core :as t])

(def date1 (t/date-time 2016 11 5))
(def date2 (t/date-time 2016 12 5))

(def test-date1 (t/date-time 2016 12 20))
(def test-date2 (t/date-time 2016 11 15))

(t/within? (t/interval date1 date2) test-date1) ;; false
(t/within? (t/interval date1 date2) test-date2) ;; true
```

La funzione intervallo è utilizzata è **esclusiva** , il che significa che non include il secondo argomento della funzione nell'intervallo. Come esempio:

```
(t/within? (t/interval date1 date2) date2) ;; false
(t/within? (t/interval date1 date2) date1) ;; true
```

Aggiunta di data e ora joda da altri tipi di orario

La libreria clj-time.coerce può aiutare a convertire altri formati di data e ora in formato joda time (clj-time.core / date-time). Gli altri formati includono **Java long** format, **String** , **Date** , **SQL Date** .

Per convertire il tempo da altri formati temporali, includere la libreria e utilizzare la funzione da, ad es

```
(require '[clj-time.coerce :as c])

(def string-time "1990-01-29")
(def epoch-time 633571200)
(def long-time 633551400)

(c/from-string string-time) ;; #<DateTime 1990-01-29T00:00:00.000Z>
(c/from-epoch epoch-time) ;; #<DateTime 1990-01-29T00:00:00.000Z>
(c/from-long 633551400) ;; #<DateTime 1990-01-29T00:00:00.000Z>
```

Aggiunta di data e ora ad altre date

cljs-time ci dà l'opzione di aggiungere / sottrarre date-time ad altre date-time. I tempi di data aggiunti sottratti dovrebbero essere in forma di giorni, mesi, anni, ore ecc.

```
(require '[clj-time.core :as t])

(def example-date (t/date-time 2016 1 1)) ;; #<DateTime 2016-01-01T00:00:00.000Z>

;; Addition
(t/plus example-date (t/months 1))        ;; #<DateTime 2016-02-01T00:00:00.000Z>
(t/plus example-date (t/years 1))         ;; #<DateTime 2017-01-01T00:00:00.000Z>
```

```
;; Subtraction
(t/minus example-date (t/days 1))          ;; #<DateTime 2015-12-31T00:00:00.000Z>
(t/minus example-date (t/hours 12))        ;; #<DateTime 2015-12-31T12:00:00.000Z>
```

Leggi CLJ-time online: https://riptutorial.com/it/clojure/topic/9127/clj-time

Capitolo 4: Clojure destrutturante

Examples

Distruzione di un vettore

Ecco come puoi distruggere un vettore:

```
(def my-vec [1 2 3])
```

Quindi, ad esempio all'interno di un blocco `let` , è possibile estrarre i valori dal vettore in modo molto succinto come segue:

```
(let [[x y] my-vec]
 (println "first element:" x ", second element: " y))
;; first element: 1 , second element: 2
```

Distruzione di una mappa

Ecco come puoi distruggere una mappa:

```
(def my-map {:a 1 :b 2 :c 3})
```

Quindi, ad esempio, all'interno di un blocco let è possibile estrarre i valori dalla mappa in modo molto succinto come segue:

```
(let [{x :a y :c} my-map]
  (println ":a val:" x ", :c val: " y))
;; :a val: 1 , :c val: 3
```

Si noti che i valori estratti in ogni mappatura sono sulla sinistra e le chiavi a cui sono associati sono sulla destra.

Se vuoi distruggere i valori con i bind con lo stesso nome dei tasti puoi usare questa scorciatoia:

```
(let [{:keys [a c]} my-map]
  (println ":a val:" a ", :c val: " c))
;; :a val: 1 , :c val: 3
```

Se le tue chiavi sono stringhe puoi usare quasi la stessa struttura:

```
(let [{:strs [foo bar]} {"foo" 1 "bar" 2}]
  (println "FOO:" foo  "BAR: " bar ))
;; FOO: 1 BAR: 2
```

E allo stesso modo per i simboli:

```
(let [{:syms [foo bar]} {'foo 1 'bar 2}]
  (println "FOO:" foo "BAR:" bar))
;; FOO: 1 BAR: 2
```

Se si desidera distruggere una mappa nidificata, è possibile nidificare i moduli di associazione illustrati sopra:

```
(def data
  {:foo {:a 1
         :b 2}
   :bar {:a 10
         :b 20}})

(let [{{:keys [a b]} :foo
       {a2 :a b2 :b} :bar} data]
  [a b a2 b2])
;; => [1 2 10 20]
```

Distruzione di elementi rimanenti in una sequenza

Diciamo che hai un vettore come questo:

```
(def my-vec [1 2 3 4 5 6])
```

E vuoi estrarre i primi 3 elementi e ottenere gli elementi rimanenti come sequenza. Questo può essere fatto come segue:

```
(let [[x y z & remaining] my-vec]
 (println "first:" x ", second:" y "third:" z "rest:" remaining))
;= first: 1 , second: 2 third: 3 rest: (4 5 6)
```

Distruzione di vettori annidati

Puoi distruggere i vettori annidati:

```
(def my-vec [[1 2] [3 4]])

(let [[[a b][c d]] my-vec]
  (println a b c d))
;; 1 2 3 4
```

Distruzione di una mappa con valori predefiniti

A volte si desidera distruggere la chiave sotto una mappa che potrebbe non essere presente nella mappa, ma si desidera un valore predefinito per il valore destrutturato. Puoi farlo in questo modo:

```
(def my-map {:a 3 :b 4})
(let [{a :a
       b :b
       :keys [c d]
       :or {a 1
```

```
           c 2}} my-map]
  (println a b c d))
  ;= 3 4 2 nil
```

Param di destrutturazione di un fn

La destrutturazione funziona in molti posti, così come nella lista param di un fn:

```
(defn my-func [[_ a b]]
  (+ a b))

(my-func [1 2 3]) ;= 5
(my-func (range 5)) ;= 3
```

La destrutturazione funziona anche per il costrutto `& rest` nella lista param:

```
(defn my-func2 [& [_ a b]]
  (+ a b))

(my-func2 1 2 3) ;= 5
(apply my-func2 (range 5)) ;= 3
```

Convertire il resto di una sequenza in una mappa

La destrutturazione ti dà anche la possibilità di interpretare una sequenza come una mappa:

```
(def my-vec [:a 1 :b 2])
(def my-lst '("smthg else" :c 3 :d 4))

(let [[& {:keys [a b]}] my-vec
      [s & {:keys [c d]} my-lst]
  (+ a b c d)) ;= 10
```

È utile per definire funzioni con **parametri denominati** :

```
(defn my-func [a b & {:keys [c d] :or {c 3 d 4}}]
  (println a b c d))

(my-func 1 2) ;= 1 2 3 4
(my-func 3 4 :c 5 :d 6) ;= 3 4 5 6
```

Panoramica

La destrutturazione consente di estrarre dati da vari oggetti in variabili distinte. In ogni esempio di seguito, ogni variabile è assegnata alla propria stringa (`a` = `"a"` , b = `"b"` , & c.)

genere	Esempio	Valore di `data` / *commenti*
vec	`(let [[abc] data ...)`	`["a" "b" "c"]`
vec annidato	`(let [[[ab] [cd]] data ...)`	`[["a" "b"] ["c" "d"]]`

genere	Esempio	Valore di `data` / *commenti*
`map`	`(let [{a :ab :bc :c} data ...)`	`{:a "a" :b "b" :c "c"}`
- alternativa:	`(let [{:keys [abc]} data ...)`	*Quando le variabili prendono il nome dalle chiavi.*

Suggerimenti:

- I valori predefiniti possono essere forniti usando `:or` , altrimenti il valore predefinito è `nil`
- Usa `& rest` per memorizzare un `seq` di eventuali valori extra in `rest` , altrimenti i valori extra vengono ignorati
- Un uso comune e utile della destrutturazione è per i parametri di funzione
- Puoi assegnare parti indesiderate a una variabile throw-away (convenzionalmente: _)

Distruzione e associazione al nome delle chiavi

A volte, quando si distruggono le mappe, si desidera associare i valori destrutturati al rispettivo nome della chiave. A seconda della granularità della struttura dei dati, l'utilizzo dello schema di destrutturazione *standard* può essere un po ' *prolisso* .

Diciamo, abbiamo un record basato sulla mappa in questo modo:

```
(def john {:lastname "McCarthy" :firstname "John" :country "USA"})
```

normalmente lo distruggeremo in questo modo:

```
(let [{lastname :lastname firstname :firstname country :country} john]
    (str firstname " " lastname ", " country))
;;"John McCarthy, USA"
```

qui, la struttura dei dati è piuttosto semplice con solo 3 slot (*nome, cognome, nazione*) ma immagina quanto sarebbe ingombrante se dovessimo ripetere due volte tutti i nomi dei tasti per una struttura dati più granulare (avendo più slot di appena 3) .

Invece, un modo migliore per gestirlo è usando `:keys` (dato che le nostre chiavi sono *parole chiave* qui) e selezionando il nome della chiave che vorremmo associare a like like:

```
(let [{:keys [firstname lastname country]} john]
    (str firstname " " lastname ", " country))
;;"John McCarthy, USA"
```

La stessa *logica intuitiva* vale per altri tipi di chiavi come i *simboli* (usando `:syms`) e le semplici *stringhe* vecchie (usando `:strs`)

```
;; using strings as keys
(def john {"lastname" "McCarthy" "firstname" "John" "country" "USA"})
;;#'user/john
```

```
;; destructuring string-keyed map
(let [{:strs [lastname firstname country]} john]
    (str firstname " " lastname ", " country))
;;"John McCarthy, USA"

;; using symbols as keys
(def john {'lastname "McCarthy" 'firstname "John" 'country "USA"})

;; destructuring symbol-keyed map
(let [{:syms [lastname firstname country]} john]
    (str firstname " " lastname ", " country))
;;"John McCarthy, USA"
```

Distruzione e dare un nome al valore dell'argomento originale

```
(defn print-some-items
   [[a b :as xs]]
  (println a)
  (println b)
  (println xs))

(print-some-items [2 3])
```

Questo esempio stampa l'output

```
2
3
[2 3]
```

L'argomento è destrutturato e gli elementi `2` e `3` sono assegnati ai simboli `a` e `b` . L'argomento originale, l'intero vettore `[2 3]` , viene anche assegnato al simbolo `xs` .

Leggi Clojure destrutturante online: https://riptutorial.com/it/clojure/topic/1786/clojure-destrutturante

Capitolo 5: clojure.core

introduzione

Questo documento offre varie funzionalità di base offerte da Clojure. Non è necessaria alcuna dipendenza esplicita per questo e viene fornito come parte di org.clojure.

Examples

Definire funzioni in clojure

```
(defn x [a b]
        (* a b)) ;; public function

=> (x 3 2) ;; 6
=> (x 0 9) ;; 0

(defn- y [a b]
        (+ a b)) ;; private function

=> (x (y 1 2) (y 2 3)) ;; 15
```

Assoc - aggiornamento dei valori di mappa / vettore in clojure

Quando applicato su una mappa, restituisce una nuova mappa con val coppie chiave nuove o aggiornate.

Può essere usato per aggiungere nuove informazioni nella mappa esistente.

```
(def userData {:name "Bob" :userID 2 :country "US"})
(assoc userData :age 27)            ;; { :name "Bob" :userID 2 :country "US" :age 27}
```

Sostituisce il vecchio valore di informazione se viene fornita la chiave esistente.

```
(assoc userData :name "Fred")       ;; { :name "Fred" :userID 2 :country "US" }
(assoc userData :userID 3 :age 27)  ;; {:name "Bob" :userID 3 :country "US" :age 27}
```

Può anche essere utilizzato su un vettore per sostituire il valore nell'indice specificato.

```
(assoc [3 5 6 7] 2 10)              ;; [3 5 10 7]
(assoc [1 2 3 4] 6 6)               ;; java.lang.IndexOutOfBoundsException
```

Operatori di comparazione in Clojure

I confronti sono funzioni in clojure. Cosa significa in `(2>1)` è `(> 2 1)` in clojure. Ecco tutti gli operatori di confronto in clojure.

1. Più grande di

```
(> 2 1) ;; true
(> 1 2) ;; false
```

2. Meno di

```
(< 2 1) ;; false
```

3. Maggiore o uguale a

```
(>= 2 1) ;; true
(>= 2 2) ;; true
(>= 1 2) ;; false
```

4. Minore o uguale a

```
(<= 2 1) ;; false
(<= 2 2) ;; true
(<= 1 2) ;; true
```

5. Uguale a

```
(= 2 2) ;; true
(= 2 10) ;; false
```

6. Non uguale a

```
(not= 2 2) ;; false
(not= 2 10) ;; true
```

Dissoc - dissociare una chiave da una mappa del clojure

Ciò restituisce una mappa senza le coppie chiave-valore per le chiavi menzionate nell'argomento della funzione. Può essere usato per rimuovere informazioni dalla mappa esistente.

```
(dissoc {:a 1 :b 2} :a) ;; {:b 2}
```

Può anche essere usato per dissociare più chiavi come:

```
(dissoc {:a 1 :b 2 :c 3} :a :b) ;; {:c 3}
```

Leggi clojure.core online: https://riptutorial.com/it/clojure/topic/9585/clojure-core

Capitolo 6: clojure.spec

Sintassi

- :: è una abbreviazione di una parola chiave qualificata per lo spazio dei nomi. Ad esempio se siamo nello spazio dei nomi utente: :: foo è una scorciatoia per: user / foo
- #: o # - sintassi letterale per le chiavi qualificanti in una mappa da uno spazio dei nomi

Osservazioni

Clojure spec è una nuova libreria di specifiche / contratti per clojure disponibile dalla versione 1.9.

Le specifiche vengono sfruttate in vari modi, tra cui l'inclusione nella documentazione, la convalida dei dati, la generazione di dati per il test e altro ancora.

Examples

Utilizzo di un predicato come specifica

Qualsiasi funzione di predicato può essere utilizzata come specifica. Ecco un semplice esempio:

```
(clojure.spec/valid? odd? 1)
;;=> true

(clojure.spec/valid? odd? 2)
;;=> false
```

il `valid?` la funzione prenderà una specifica e un valore e restituirà true se il valore è conforme alle specifiche e false altrimenti.

Un altro predicato interessante è l'appartenenza al set:

```
(s/valid? #{:red :green :blue} :red)
;;=> true
```

fdef: scrivere una specifica per una funzione

Diciamo che abbiamo la seguente funzione:

```
(defn nat-num-count [nums] (count (remove neg? nums)))
```

Possiamo scrivere una specifica per questa funzione definendo una specifica della funzione con lo stesso nome:

```
(clojure.spec/fdef nat-num-count
```

```
        :args (s/cat :nums (s/coll-of number?))
        :ret integer?
        :fn #(<= (:ret %) (-> % :args :nums count)))
```

`:args` prende una specifica regex che descrive la sequenza di argomenti da un'etichetta di parola chiave corrispondente al nome dell'argomento e una specifica corrispondente. Il motivo per cui le specifiche richieste da `:args` è una specifica regex è il supporto di più origini per una funzione. `:ret` specifica una specifica per il valore di ritorno della funzione.

`:fn` è una specifica che vincola la relazione tra `:args` e `:ret` . Viene utilizzato come proprietà quando viene eseguito attraverso test.check. Viene chiamato con un singolo argomento: una mappa con due chiavi:: `:args` (gli argomenti conformi alla funzione) e `:ret` (il valore di ritorno conforme della funzione).

Registrazione di una specifica

Oltre ai predicati che funzionano come specifiche, è possibile registrare una specifica globalmente utilizzando `clojure.spec/def` . `def` richiede che una specifica che viene registrata sia nominata da una parola chiave qualificata nello spazio dei nomi:

```
(clojure.spec/def ::odd-nums odd?)
;;=> :user/odd-nums

(clojure.spec/valid? ::odd-nums 1)
;;=> true
(clojure.spec/valid? ::odd-nums 2)
;;=> false
```

Una volta registrato, una specifica può essere referenziata globalmente ovunque in un programma Clojure.

La sintassi `::odd-nums` è una scorciatoia per `:user/odd-nums` , assumendo che ci troviamo nello spazio dei nomi `user` . `::` qualificherà il simbolo che precede con la namesapce corrente.

Piuttosto che passare nel predicato, possiamo passare il nome della specifica a `valid?` e funzionerà allo stesso modo.

clojure.spec / and & clojure.spec / o

`clojure.spec/and` & `clojure.spec/or` può essere utilizzato per creare specifiche più complesse, utilizzando più specifiche o predicati:

```
(clojure.spec/def ::pos-odd (clojure.spec/and odd? pos?))

(clojure.spec/valid? ::pos-odd 1)
;;=> true

(clojure.spec/valid? ::pos-odd -3)
;;=> false
```

`or` funziona allo stesso modo, con una differenza significativa. Quando si definisce una `or` una

specifica, è necessario contrassegnare ogni ramo possibile con una parola chiave. Questo è usato per fornire rami specifici che falliscono nei messaggi di errore:

```
(clojure.spec/def ::big-or-small (clojure.spec/or :small #(< % 10) :big #(> % 100)))

(clojure.spec/valid? ::big-or-small 1)
;;=> true

(clojure.spec/valid? ::big-or-small 150)
;;=> true

(clojure.spec/valid? ::big-or-small 20)
;;=> false
```

Quando si conforma una specifica usando `or` , verrà restituita la specifica applicabile che ha reso il valore conforme:

```
(clojure.spec/conform ::big-or-small 5)
;; => [:small 5]
```

Registra le specifiche

È possibile specificare un record come segue:

```
(clojure.spec/def ::name string?)
(clojure.spec/def ::age pos-int?)
(clojure.spec/def ::occupation string?)

(defrecord Person [name age occupation])

(clojure.spec/def ::person (clojure.spec/keys :req-un [::name ::age ::occupation]))

(clojure.spec/valid? ::person (->Person "john doe" 25 "programmer"))
;;=> true

(clojure.spec/valid? ::person (->Person "john doe" "25" "programmer"))
;;=> false
```

A un certo punto, in futuro, potrebbe essere introdotta una sintassi del lettore o un supporto integrato per le chiavi di registrazione idonee dallo spazio dei nomi dei record. Questo supporto esiste già per le mappe.

Specifiche della mappa

Puoi specificare una mappa specificando quali chiavi dovrebbero essere presenti nella mappa:

```
(clojure.spec/def ::name string?)
(clojure.spec/def ::age pos-int?)
(clojure.spec/def ::occupation string?)

(clojure.spec/def ::person (clojure.spec/keys :req [::name ::age ::occupation]))

(clojure.spec/valid? ::person {::name "john" ::age 25 ::occupation "programmer"})
```

```
;; => true
```

`:req` è un vettore di chiavi che devono essere presenti nella mappa. È possibile specificare ulteriori opzioni come `:opt` , un vettore di chiavi che sono opzionali.

Gli esempi finora richiedono che le chiavi nel nome siano qualificate per lo spazio dei nomi. Ma è comune che le chiavi della mappa non siano qualificate. Per questo caso, `clojure.spec` fornisce: req e: opt equivalents per le chiavi non qualificate `:req-un` e `:opt-un` . Ecco lo stesso esempio, con chiavi non qualificate:

```
(clojure.spec/def ::name string?)
(clojure.spec/def ::age pos-int?)
(clojure.spec/def ::occupation string?)

(clojure.spec/def ::person (clojure.spec/keys :req-un [::name ::age ::occupation]))

(clojure.spec/valid? ::person {:name "john" :age 25 :occupation "programmer"})
;; => true
```

Si noti come le specifiche fornite nel `:req-un` vettore come ancora qualificato. clojure.spec, confermerà automaticamente le versioni non qualificate nella mappa quando si conformano i valori.

la sintassi letterale della mappa dello spazio dei nomi consente di qualificare in modo succinto tutte le chiavi di una mappa da un singolo spazio dei nomi. Per esempio:

```
(clojure.spec/def ::name string?)
(clojure.spec/def ::age pos-int?)
(clojure.spec/def ::occupation string?)

(clojure.spec/def ::person (clojure.spec/keys :req [::name ::age ::occupation]))

(clojure.spec/valid? ::person #:user{:name "john" :age 25 :occupation "programmer"})
;;=> true
```

Notare il numero speciale `#:` sintassi del lettore. Lo seguiamo con lo spazio dei nomi per cui desideriamo qualificare tutte le chiavi della mappa. Questi verranno quindi confrontati con le specifiche corrispondenti allo spazio dei nomi fornito.

collezioni

Puoi specificare le raccolte in diversi modi. coll-of consente di spec collezioni e fornire alcuni vincoli aggiuntivi. Ecco un semplice esempio:

```
(clojure.spec/valid? (clojure.spec/coll-of int?) [1 2 3])
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int?) '(1 2 3))
;; => true
```

Le opzioni di vincolo seguono le specifiche / il predicato principali per la raccolta. Puoi vincolare il

tipo di raccolta con `:kind` come questo:

```
(clojure.spec/valid? (clojure.spec/coll-of int? :kind vector?) [1 2 3])
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int? :kind vector?) '(1 2 3))
;; => false
```

Quanto sopra è falso perché la collezione passata non è un vettore.

```
(clojure.spec/valid? (clojure.spec/coll-of int? :kind list?) '(1 2 3))
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int? :kind set?) #{1 2 3})
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int? :kind set?) #{1 "2" 3})
;; => false
```

Quanto sopra è falso perché non tutti gli elementi nel set sono interi.

Puoi anche limitare la dimensione della raccolta in pochi modi:

```
(clojure.spec/valid? (clojure.spec/coll-of int? :kind vector? :count 3) [1 2 3])
;; => true

    (clojure.spec/valid? (clojure.spec/coll-of int? :kind vector? :count 3) [1 2])
;; => false

(clojure.spec/valid? (clojure.spec/coll-of int? :min-count 3 :max-count 5) [1 2 3])
;; => true

    (clojure.spec/valid? (clojure.spec/coll-of int? :min-count 3 :max-count 5) [1 2])
;; => false
```

Puoi anche far rispettare l'unicità degli elementi nella raccolta con `:distinct` :

```
(clojure.spec/valid? (clojure.spec/coll-of int? :distinct true) [1 2])
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int? :distinct true) [2 2])
;; => false
```

`coll-of` assicura che tutti gli elementi di una sequenza siano controllati. Per le collezioni di grandi dimensioni, questo può essere molto inefficiente. `every` comporta come `coll-of` , eccetto che campiona solo un numero relativamente piccolo di elementi delle sequenze per la conformità. Questo funziona bene per le grandi collezioni. Ecco un esempio:

```
(clojure.spec/valid? (clojure.spec/every int? :distinct true) [1 2 3 4 5])
;; => true
```

`map-of` è simile a `coll-of` , ma per le mappe. Poiché le mappe hanno sia chiavi che valori, devi fornire sia una specifica per la chiave che una specifica per il valore:

```
(clojure.spec/valid? (clojure.spec/map-of keyword? string?) {:red "red" :green "green"})
;; => true
```

Come `coll-of` , `map-of` conformità ai controlli di tutte le chiavi / valori della mappa. Per le mappe di grandi dimensioni questo sarà inefficiente. Come `coll-of` , `map-of` supplies `every-kv` per campionare in modo efficiente un numero relativamente piccolo di valori da una grande mappa:

```
(clojure.spec/valid? (clojure.spec/every-kv keyword? string?) {:red "red" :green "green"})
;; => true
```

sequenze

le specifiche possono descrivere ed essere usate con sequenze arbitrarie. Supporta questo tramite una serie di operazioni spec regex.

```
(clojure.spec/valid? (clojure.spec/cat :text string? :int int?) ["test" 1])
;;=> true
```

`cat` richiede etichette per ogni specifica utilizzata per descrivere la sequenza. cat descrive una sequenza di elementi e una specifica per ognuno.

`alt` è usato per scegliere tra un numero di possibili specifiche per un dato elemento in una sequenza. Per esempio:

```
(clojure.spec/valid? (clojure.spec/cat :text-or-int (clojure.spec/alt :text string? :int
int?)) ["test"])
;;=> true
```

`alt` richiede anche che ogni specifica sia etichettata da una parola chiave.

Le sequenze di Regex possono essere composte in alcuni modi molto interessanti e potenti per creare specifiche di descrizione della sequenza arbitrariamente complesse. Ecco un esempio un po 'più complesso:

```
(clojure.spec/def ::complex-seq (clojure.spec/+ (clojure.spec/cat :num int? :foo-map
(clojure.spec/map-of keyword? int?))))
(clojure.spec/valid? ::complex-seq [0 {:foo 3 :baz 1} 4 {:foo 4}])
;;=> true
```

Qui `::complex-seq` convaliderà una sequenza di una o più coppie di elementi, la prima essendo una int e la seconda una mappa di parola chiave in int.

Leggi clojure.spec online: https://riptutorial.com/it/clojure/topic/2325/clojure-spec

Capitolo 7: clojure.test

Examples

è

Il `is` macro è il nucleo della `clojure.test` libreria. Restituisce il valore della sua espressione corporea, stampando un messaggio di errore se l'espressione restituisce un valore falso.

```
(defn square [x]
  (+ x x))

(require '[clojure.test :as t])

(t/is (= 0 (square 0)))
;;=> true

(t/is (= 1 (square 1)))
;;
;; FAIL in () (foo.clj:1)
;; expected: (= 1 (square 1))
;;   actual: (not (= 1 2))
;;=> false
```

Raggruppare i test relativi con la macro di test

È possibile raggruppare le asserzioni correlate nei test unitari più `deftest` all'interno di un contesto utilizzando la macro di `testing` :

```
(deftest add-nums
  (testing "Positive cases"
    (is (= 2 (+ 1 1)))
    (is (= 4 (+ 2 2))))
  (testing "Negative cases"
    (is (= -1 (+ 2 -3)))
    (is (= -4 (+ 8 -12)))))
```

Ciò contribuirà a chiarire l'output del test durante l'esecuzione. Si noti che il `testing` deve avvenire all'interno di un punto di `deftest` .

Definire un test con il risultato migliore

`deftest` è una macro per la definizione di un test unitario, simile ai test unitari in altre lingue.

È possibile creare un test come segue:

```
(deftest add-nums
  (is (= 2 (+ 1 1)))
  (is (= 3 (+ 1 2))))
```

Qui stiamo definendo un test chiamato `add-nums` , che verifica la funzione `+` . Il test ha due asserzioni.

È quindi possibile eseguire il test in questo modo nello spazio dei nomi corrente:

```
(run-tests)
```

Oppure puoi semplicemente eseguire i test per lo spazio dei nomi in cui si trova il test:

```
(run-tests 'your-ns)
```

siamo

L' `are` macro è anche parte del `clojure.test` biblioteca. Ti permette di fare più asserzioni su un modello.

Per esempio:

```
(are [x y] (= x y)
   4 (+ 2 2)
   8 (* 2 4))
=> true
```

Qui, `(= xy)` agisce come un modello che prende ogni argomento e crea una `is` un'affermazione fuori di esso.

Questo espande multiplo `is` affermazioni:

```
(do
  (is (= 4 (+ 2 2)))
  (is (= 8 (* 2 4))))
```

Avvolgi ogni test o tutti i test con i dispositivi d'uso

`use-fixtures` consente di racchiudere ogni `deftest` nel namespace con il codice che viene eseguito prima e dopo il test. Può essere usato per infissi o stub.

Le fixture sono solo funzioni che prendono la funzione di test e la eseguono con altri passi necessari (prima / dopo, wrap).

```
(ns myapp.test
  (require [clojure.test :refer :all])

(defn stub-current-thing [body]
  ;; with-redefs stubs things/current-thing function to return fixed
  ;; value for duration of each test
  (with-redefs [things/current-thing (fn [] {:foo :bar})]
    ;; run test body
    (body)))

(use-fixtures :each stub-current-thing)
```

Se utilizzato con `:once` , esegue l'intera serie di test nello spazio dei nomi corrente con la funzione

```
(defn database-for-tests [all-tests]
  (setup-database)
  (all-tests)
  (drop-database))

(use-fixtures :once database-for-tests)
```

Esecuzione di test con Leiningen

Se si sta utilizzando Leiningen e i test si trovano nella directory di test nella root del progetto, è possibile eseguire i test utilizzando il test `lein test`

Leggi clojure.test online: https://riptutorial.com/it/clojure/topic/1901/clojure-test

Capitolo 8: Collezioni e sequenze

Sintassi

- `'()` → `()`
- `'(1 2 3 4 5)` → `(1 2 3 4 5)`
- `'(1 foo 2 bar 3)` → `(1 'foo 2 'bar 3)`
- `(list 1 2 3 4 5)` → `(1 2 3 4 5)`
- `(list* [1 2 3 4 5])` → `(1 2 3 4 5)`
- `[]` → `[]`
- `[1 2 3 4 5]` → `[1 2 3 4 5]`
- `(vector 1 2 3 4 5)` → `[1 2 3 4 5]`
- `(vec '(1 2 3 4 5))` → `[1 2 3 4 5]`
- `{}` => `{}`
- `{:keyA 1 :keyB 2}` → `{:keyA 1 :keyB 2}`
- `{:keyA 1, :keyB 2}` → `{:keyA 1 :keyB 2}`
- `(hash-map :keyA 1 :keyB 2)` → `{:keyA 1 :keyB 2}`
- `(sorted-map 5 "five" 1 "one")` → `{1 "one" 5 "five"}` (le voci sono ordinate per tasto quando usate come sequenza)
- `#{}` → `#{}`
- `#{1 2 3 4 5}` → `#{4 3 2 5 1}` (non ordinato)
- `(hash-set 1 2 3 4 5)` → `#{2 5 4 1 3}` (non ordinato)
- `(sorted-set 2 5 4 3 1)` → `#{1 2 3 4 5}`

Examples

collezioni

Tutte le raccolte Clojure incorporate sono immutabili ed eterogenee, hanno una sintassi letterale e supportano le funzioni `conj` , `count` e `seq` .

- `conj` restituisce una nuova collezione che è equivalente a una collezione esistente con un elemento "aggiunto", in "tempo costante" o logaritmico. Cosa significa esattamente questo dipende dalla collezione.
- `count` restituisce il numero di elementi in una raccolta, in tempo costante.
- `seq` restituisce `nil` per una raccolta vuota o una sequenza di elementi per una raccolta non vuota, in tempo costante.

elenchi

Una lista è denotata da parentesi:

```
()
```

```
;;=> ()
```

Una lista di Clojure è una lista concatenata . `conj` "unisce" un nuovo elemento alla collezione nella posizione più efficiente. Per gli elenchi, questo è all'inizio:

```
(conj () :foo)
;;=> (:foo)

(conj (conj () :bar) :foo)
;;=> (:foo :bar)
```

A differenza di altre raccolte, gli elenchi non vuoti vengono valutati come chiamate a moduli speciali, macro o funzioni quando vengono valutati. Pertanto, mentre `(:foo)` è la rappresentazione letterale della lista che contiene `:foo` come unico elemento, la valutazione `(:foo)` in un REPL causerà il `IllegalArgumentException` di `IllegalArgumentException` perché una parola chiave non può essere invocata come una funzione nulla .

```
(:foo)
;; java.lang.IllegalArgumentException: Wrong number of args passed to keyword: :foo
```

Per evitare che Clojure valuti un elenco non vuoto, puoi `quote` :

```
'(:foo)
;;=> (:foo)

'(:foo :bar)
;;=> (:foo :bar)
```

Sfortunatamente, questo non causa la valutazione degli elementi:

```
(+ 1 1)
;;=> 2

'(1 (+ 1 1) 3)
;;=> (1 (+ 1 1) 3)
```

Per questo motivo, di solito vuoi usare la `list` , una funzione variadica che valuta tutti i suoi argomenti e usa quei risultati per costruire una lista:

```
(list)
;;=> ()

(list :foo)
;;=> (:foo)

(list :foo :bar)
;;=> (:foo :bar)

(list 1 (+ 1 1) 3)
;;=> (1 2 3)
```

`count` restituisce il numero di elementi, in tempo costante:

```
(count ())
;;=> 0

(count (conj () :foo))
;;=> 1

(count '(:foo :bar))
;;=> 2
```

Puoi verificare se qualcosa è una lista usando l' `list?` predicato:

```
(list? ())
;;=> true

(list? '(:foo :bar))
;;=> true

(list? nil)
;;=> false

(list? 42)
;;=> false

(list? :foo)
;;=> false
```

È possibile ottenere il primo elemento di una lista usando `peek` :

```
(peek ())
;;=> nil

(peek '(:foo))
;;=> :foo

(peek '(:foo :bar))
;;=> :foo
```

Puoi ottenere una nuova lista senza il primo elemento usando `pop` :

```
(pop '(:foo))
;;=> ()

(pop '(:foo :bar))
;;=> (:bar)
```

Si noti che se si tenta di `pop` un elenco vuoto, si otterrà un `IllegalStateException` :

```
(pop ())
;; java.lang.IllegalStateException: Can't pop empty list
```

Infine, tutte le liste sono sequenze, quindi puoi fare tutto con una lista che puoi fare con qualsiasi altra sequenza. Infatti, con l'eccezione della lista vuota, chiamare `seq` su una lista restituisce lo stesso identico oggetto:

```
(seq ())
;;=> nil

(seq '(:foo))
;;=> (:foo)

(seq '(:foo :bar))
;;=> (:foo :bar)

(let [x '(:foo :bar)]
  (identical? x (seq x)))
;;=> true
```

sequenze

Una sequenza è molto simile a una lista: è un oggetto immutabile che può darti il suo `first` elemento o il `rest` dei suoi elementi in un tempo costante. È possibile anche `cons` truct una nuova sequenza da una sequenza esistente e un elemento di attenersi al principio.

Puoi verificare se qualcosa è una sequenza usando il `seq?` predicato:

```
(seq? nil)
;;=> false

(seq? 42)
;;=> false

(seq? :foo)
;;=> false
```

Come già sai, le liste sono sequenze:

```
(seq? ())
;;=> true

(seq? '(:foo :bar))
;;=> true
```

Tutto ciò che ottieni chiamando `seq` o `rseq` o `keys` o `vals` su una raccolta non vuota è anche una sequenza:

```
(seq? (seq ()))
;;=> false

(seq? (seq '(:foo :bar)))
;;=> true

(seq? (seq []))
;;=> false

(seq? (seq [:foo :bar]))
;;=> true

(seq? (rseq []))
;;=> false
```

```
(seq? (rseq [:foo :bar]))
;;=> true

(seq? (seq {}))
;;=> false

(seq? (seq {:foo :bar :baz :qux}))
;;=> true

(seq? (keys {}))
;;=> false

(seq? (keys {:foo :bar :baz :qux}))
;;=> true

(seq? (vals {}))
;;=> false

(seq? (vals {:foo :bar :baz :qux}))
;;=> true

(seq? (seq #{}))
;;=> false

(seq? (seq #{:foo :bar}))
;;=> true
```

Ricorda che tutte le liste sono sequenze, ma non tutte le sequenze sono elenchi. Mentre le liste supportano lo `peek` e il `pop` e `count` in un tempo costante, in generale, una sequenza non ha bisogno di supportare nessuna di quelle funzioni. Se provi a chiamare `peek` o `pop` su una sequenza che non supporta anche l'interfaccia stack di Clojure, otterrai una `ClassCastException` :

```
(peek (seq [:foo :bar]))
;; java.lang.ClassCastException: clojure.lang.PersistentVector$ChunkedSeq cannot be cast to
clojure.lang.IPersistentStack

(pop (seq [:foo :bar]))
;; java.lang.ClassCastException: clojure.lang.PersistentVector$ChunkedSeq cannot be cast to
clojure.lang.IPersistentStack

(peek (seq #{:foo :bar}))
;; java.lang.ClassCastException: clojure.lang.APersistentMap$KeySeq cannot be cast to
clojure.lang.IPersistentStack

(pop (seq #{:foo :bar}))
;; java.lang.ClassCastException: clojure.lang.APersistentMap$KeySeq cannot be cast to
clojure.lang.IPersistentStack

(peek (seq {:foo :bar :baz :qux}))
;; java.lang.ClassCastException: clojure.lang.PersistentArrayMap$Seq cannot be cast to
clojure.lang.IPersistentStack

(pop (seq {:foo :bar :baz :qux}))
;; java.lang.ClassCastException: clojure.lang.PersistentArrayMap$Seq cannot be cast to
clojure.lang.IPersistentStack
```

Se si chiama `count` su una sequenza che non implementa il `count` in tempo costante, non si otterrà

un errore; invece, Clojure attraverserà l'intera sequenza fino a raggiungere la fine, quindi restituirà il numero di elementi che ha attraversato. Ciò significa che, per le sequenze generali, il `count` è un tempo lineare, non costante. È possibile verificare se qualcosa supporta il `count` tempo costante utilizzando il `counted?` predicato:

```
(counted? '(:foo :bar))
;;=> true

(counted? (seq '(:foo :bar)))
;;=> true

(counted? [:foo :bar])
;;=> true

(counted? (seq [:foo :bar]))
;;=> true

(counted? {:foo :bar :baz :qux})
;;=> true

(counted? (seq {:foo :bar :baz :qux}))
;;=> true

(counted? #{:foo :bar})
;;=> true

(counted? (seq #{:foo :bar}))
;;=> false
```

Come accennato in precedenza, è possibile utilizzare `first` per ottenere il primo elemento di una sequenza. Nota che `first` chiameremo `seq` sul loro argomento, quindi può essere usato su qualsiasi cosa "seqable", non solo sulle sequenze effettive:

```
(first nil)
;;=> nil

(first '(:foo))
;;=> :foo

(first '(:foo :bar))
;;=> :foo

(first [:foo])
;;=> :foo

(first [:foo :bar])
;;=> :foo

(first {:foo :bar})
;;=> [:foo :bar]

(first #{:foo})
;;=> :foo
```

Inoltre, come menzionato sopra, puoi usare il `rest` per ottenere una sequenza che contiene tutto tranne il primo elemento di una sequenza esistente. Come `first` , chiama `seq` sulla sua

argomentazione. Tuttavia, *non* chiama `seq` sul suo risultato! Ciò significa che, se chiami il `rest` su una sequenza che contiene meno di due elementi, tornerai indietro `()` invece di `nil` :

```
(rest nil)
;;=> ()

(rest '(:foo))
;;=> ()

(rest '(:foo :bar))
;;=> (:bar)

(rest [:foo])
;;=> ()

(rest [:foo :bar])
;;=> (:bar)

(rest {:foo :bar})
;;=> ()

(rest #{:foo})
;;=> ()
```

Se si vuole tornare `nil` , quando non ci sono più elementi in una sequenza, è possibile utilizzare `next` posto di `rest` :

```
(next nil)
;;=> nil

(next '(:foo))
;;=> nil

(next [:foo])
;;=> nil
```

È possibile utilizzare la funzione `cons` per creare una nuova sequenza che restituirà il primo argomento per il `first` e il secondo argomento per il `rest` :

```
(cons :foo nil)
;;=> (:foo)

(cons :foo (cons :bar nil))
;;=> (:foo :bar)
```

Clojure offre una grande libreria di sequenze con molte funzioni per gestire le sequenze. La cosa importante di questa libreria è che funziona con qualsiasi cosa "seqable", non solo con le liste. Ecco perché il concetto di una sequenza è così utile; significa che una singola funzione, come `reduce` , funziona perfettamente su qualsiasi raccolta:

```
(reduce + '(1 2 3))
;;=> 6

(reduce + [1 2 3])
;;=> 6
```

```
(reduce + #{1 2 3})
;;=> 6
```

L'altra ragione per cui le sequenze sono utili è che, poiché non impongono alcuna particolare implementazione del `first` e del `rest` , consentono sequenze pigre i cui elementi vengono realizzati solo quando necessario.

Data un'espressione che creerebbe una sequenza, puoi avvolgere quell'espressione nella macro `lazy-seq` per ottenere un oggetto che si comporta come una sequenza, ma in realtà valuterà quell'espressione solo quando viene richiesta dalla funzione `seq` , a che punto sarà cache il risultato dell'espressione e in avanti `first` e `rest` chiamate al risultato in cache.

Per le sequenze finite, una sequenza lenta di solito agisce come una sequenza desiderosa equivalente:

```
(seq [:foo :bar])
;;=> (:foo :bar)

(lazy-seq [:foo :bar])
;;=> (:foo :bar)
```

Tuttavia, la differenza diventa evidente per le sequenze infinite:

```
(defn eager-fibonacci [a b]
  (cons a (eager-fibonacci b (+' a b))))

(defn lazy-fibonacci [a b]
  (lazy-seq (cons a (lazy-fibonacci b (+' a b)))))

(take 10 (eager-fibonacci 0 1))
;; java.lang.StackOverflowError:

(take 10 (lazy-fibonacci 0 1))
;;=> (0 1 1 2 3 5 8 13 21 34)
```

Vettori

Un vettore è indicato da parentesi quadre:

```
[]
;;=> []

[:foo]
;;=> [:foo]

[:foo :bar]
;;=> [:foo :bar]

[1 (+ 1 1) 3]
;;=> [1 2 3]
```

Inoltre, utilizzando la sintassi letterale, puoi anche utilizzare la funzione `vector` per costruire un

vettore:

```
(vector)
;;=> []

(vector :foo)
;;=> [:foo]

(vector :foo :bar)
;;=> [:foo :bar]

(vector 1 (+ 1 1) 3)
;;=> [1 2 3]
```

Puoi verificare se qualcosa è un vettore che usa il `vector?` predicato:

```
(vector? [])
;;=> true

(vector? [:foo :bar])
;;=> true

(vector? nil)
;;=> false

(vector? 42)
;;=> false

(vector? :foo)
;;=> false
```

`conj` aggiunge elementi alla fine di un vettore:

```
(conj [] :foo)
;;=> [:foo]

(conj (conj [] :foo) :bar)
;;=> [:foo :bar]

(conj [] :foo :bar)
;;=> [:foo :bar]
```

`count` restituisce il numero di elementi, in tempo costante:

```
(count [])
;;=> 0

(count (conj [] :foo))
;;=> 1

(count [:foo :bar])
;;=> 2
```

Puoi ottenere l'ultimo elemento di un vettore usando la `peek` :

```
(peek [])
;;=> nil

(peek [:foo])
;;=> :foo

(peek [:foo :bar])
;;=> :bar
```

Puoi ottenere un nuovo vettore senza l'ultimo elemento usando `pop` :

```
(pop [:foo])
;;=> []

(pop [:foo :bar])
;;=> [:foo]
```

Nota che se provi a far `IllegalStateException` un vettore vuoto, otterrai una `IllegalStateException` :

```
(pop [])
;; java.lang.IllegalStateException: Can't pop empty vector
```

A differenza delle liste, i vettori sono indicizzati. Puoi ottenere un elemento di un vettore per indice in tempo "costante" usando `get` :

```
(get [:foo :bar] 0)
;;=> :foo

(get [:foo :bar] 1)
;;=> :bar

(get [:foo :bar] -1)
;;=> nil

(get [:foo :bar] 2)
;;=> nil
```

Inoltre, i vettori stessi sono funzioni che prendono un indice e restituiscono l'elemento in quell'indice:

```
([:foo :bar] 0)
;;=> :foo

([:foo :bar] 1)
;;=> :bar
```

Tuttavia, se chiami un vettore con un indice non valido, otterrai un valore `IndexOutOfBoundsException` anziché `nil` :

```
([:foo :bar] -1)
;; java.lang.IndexOutOfBoundsException:

([:foo :bar] 2)
;; java.lang.IndexOutOfBoundsException:
```

Puoi ottenere un nuovo vettore con un valore diverso in un indice particolare usando `assoc` :

```
(assoc [:foo :bar] 0 42)
;;=> [42 :bar]

(assoc [:foo :bar] 1 42)
;;=> [:foo 42]
```

Se passi un indice uguale al `count` del vettore, Clojure aggiungerà l'elemento come se avessi usato il `conj` . Tuttavia, se si passa un indice negativo o superiore al `count` , si otterrà un valore `IndexOutOfBoundsException` :

```
(assoc [:foo :bar] 2 42)
;;=> [:foo :bar 42]

(assoc [:foo :bar] -1 42)
;; java.lang.IndexOutOfBoundsException:

(assoc [:foo :bar] 3 42)
;; java.lang.IndexOutOfBoundsException:
```

Puoi ottenere una sequenza degli elementi in un vettore usando `seq` :

```
(seq [])
;;=> nil

(seq [:foo])
;;=> (:foo)

(seq [:foo :bar])
;;=> (:foo :bar)
```

Poiché i vettori sono indicizzati, puoi anche ottenere una sequenza invertita degli elementi di un vettore usando `rseq` :

```
(rseq [])
;;=> nil

(rseq [:foo])
;;=> (:foo)

(rseq [:foo :bar])
;;=> (:bar :foo)
```

Si noti che, sebbene tutte le liste siano sequenze e le sequenze siano visualizzate allo stesso modo delle liste, non tutte le sequenze sono elenchi!

```
'(:foo :bar)
;;=> (:foo :bar)

(seq [:foo :bar])
;;=> (:foo :bar)

(list? '(:foo :bar))
```

```
;;=> true

(list? (seq [:foo :bar]))
;;=> false

(list? (rseq [:foo :bar]))
;;=> false
```

Imposta

Come le mappe, gli insiemi sono associativi e non ordinati. A differenza delle mappe, che contengono mappature da chiavi a valori, imposta essenzialmente la mappa da chiavi a se stesse.

Un set è denotato da parentesi graffe precedute da un octothorpe:

```
#{}
;;=> #{}

#{:foo}
;;=> #{:foo}

#{:foo :bar}
;;=> #{:bar :foo}
```

Come con le mappe, l'ordine in cui gli elementi appaiono in un set letterale non ha importanza:

```
(= #{:foo :bar} #{:bar :foo})
;;=> true
```

Puoi verificare se qualcosa è un set usando il `set?` predicato:

```
(set? #{})
;;=> true

(set? #{:foo})
;;=> true

(set? #{:foo :bar})
;;=> true

(set? nil)
;;=> false

(set? 42)
;;=> false

(set? :foo)
;;=> false
```

È possibile verificare se una mappa contiene un dato elemento in "costante" utilizzando il `contains?` predicato:

```
(contains? #{} :foo)
;;=> false
```

```
(contains? #{:foo} :foo)
;;=> true

(contains? #{:foo} :bar)
;;=> false

(contains? #{} nil)
;;=> false

(contains? #{nil} nil)
;;=> true
```

Inoltre, gli insiemi stessi sono funzioni che accettano un elemento e restituiscono quell'elemento se è presente nell'insieme, o `nil` se non lo è:

```
(#{} :foo)
;;=> nil

(#{:foo} :foo)
;;=> :foo

(#{:foo} :bar)
;;=> nil

(#{} nil)
;;=> nil

(#{nil} nil)
;;=> nil
```

Puoi usare `conj` per ottenere un set che ha tutti gli elementi di un set esistente, oltre a un elemento aggiuntivo:

```
(conj #{} :foo)
;;=> #{:foo}

(conj (conj #{} :foo) :bar)
;;=> #{:bar :foo}

(conj #{:foo} :foo)
;;=> #{:foo}
```

Puoi usare `disj` per ottenere un set che ha tutti gli elementi di un set esistente, meno un elemento:

```
(disj #{} :foo)
;;=> #{}

(disj #{:foo} :foo)
;;=> #{}

(disj #{:foo} :bar)
;;=> #{:foo}

(disj #{:foo :bar} :foo)
;;=> #{:bar}
```

```
(disj #{:foo :bar} :bar)
;;=> #{:foo}
```

`count` restituisce il numero di elementi, in tempo costante:

```
(count #{})
;;=> 0

(count (conj #{} :foo))
;;=> 1

(count #{:foo :bar})
;;=> 2
```

Puoi ottenere una sequenza di tutti gli elementi in un set usando `seq` :

```
(seq #{})
;;=> nil

(seq #{:foo})
;;=> (:foo)

(seq #{:foo :bar})
;;=> (:bar :foo)
```

Mappe

A differenza della lista, che è una struttura dati sequenziale, e del vettore, che è sia sequenziale che associativo, la mappa è esclusivamente una struttura di dati associativa. Una mappa consiste in un insieme di mappature dalle chiavi ai valori. Tutte le chiavi sono univoche, quindi le mappe supportano la ricerca "costante" -time dalle chiavi ai valori.

Una mappa è denotata da parentesi graffe:

```
{}
;;=> {}

{:foo :bar}
;;=> {:foo :bar}

{:foo :bar :baz :qux}
;;=> {:foo :bar, :baz :qux}
```

Ogni coppia di due elementi è una coppia chiave-valore. Quindi, ad esempio, la prima mappa in alto non ha mappature. Il secondo ha una mappatura, dalla chiave `:foo` al valore `:bar` . Il terzo ha due mapping, uno dalla chiave `:foo` al valore `:bar` , e uno dalla chiave `:baz` al valore `:qux` . Le mappe sono intrinsecamente non ordinate, quindi l'ordine in cui appaiono i mapping non ha importanza:

```
(= {:foo :bar :baz :qux}
   {:baz :qux :foo :bar})
;;=> true
```

Puoi verificare se qualcosa è una mappa usando la `map?` predicato:

```
(map? {})
;;=> true

(map? {:foo :bar})
;;=> true

(map? {:foo :bar :baz :qux})
;;=> true

(map? nil)
;;=> false

(map? 42)
;;=> false

(map? :foo)
;;=> false
```

È possibile verificare se una mappa contiene una determinata *chiave* in "costante" utilizzando il `contains?` predicato:

```
(contains? {:foo :bar :baz :qux} 42)
;;=> false

(contains? {:foo :bar :baz :qux} :foo)
;;=> true

(contains? {:foo :bar :baz :qux} :bar)
;;=> false

(contains? {:foo :bar :baz :qux} :baz)
;;=> true

(contains? {:foo :bar :baz :qux} :qux)
;;=> false

(contains? {:foo nil} :foo)
;;=> true

(contains? {:foo nil} :bar)
;;=> false
```

Puoi ottenere il valore associato a una chiave usando `get` :

```
(get {:foo :bar :baz :qux} 42)
;;=> nil

(get {:foo :bar :baz :qux} :foo)
;;=> :bar

(get {:foo :bar :baz :qux} :bar)
;;=> nil

(get {:foo :bar :baz :qux} :baz)
;;=> :qux
```

```
(get {:foo :bar :baz :qux} :qux)
;;=> nil

(get {:foo nil} :foo)
;;=> nil

(get {:foo nil} :bar)
;;=> nil
```

Inoltre, le mappe stesse sono funzioni che accettano una chiave e restituiscono il valore associato a quella chiave:

```
({:foo :bar :baz :qux} 42)
;;=> nil

({:foo :bar :baz :qux} :foo)
;;=> :bar

({:foo :bar :baz :qux} :bar)
;;=> nil

({:foo :bar :baz :qux} :baz)
;;=> :qux

({:foo :bar :baz :qux} :qux)
;;=> nil

({:foo nil} :foo)
;;=> nil

({:foo nil} :bar)
;;=> nil
```

Puoi trovare un'intera voce della mappa (chiave e valore insieme) come vettore a due elementi usando `find` :

```
(find {:foo :bar :baz :qux} 42)
;;=> nil

(find {:foo :bar :baz :qux} :foo)
;;=> [:foo :bar]

(find {:foo :bar :baz :qux} :bar)
;;=> nil

(find {:foo :bar :baz :qux} :baz)
;;=> [:baz :qux]

(find {:foo :bar :baz :qux} :qux)
;;=> nil

(find {:foo nil} :foo)
;;=> [:foo nil]

(find {:foo nil} :bar)
;;=> nil
```

È possibile estrarre la chiave o il valore da una voce della mappa utilizzando la `key` o `val` , rispettivamente:

```
(key (find {:foo :bar} :foo))
;;=> :foo

(val (find {:foo :bar} :foo))
;;=> :bar
```

Si noti che, sebbene tutte le voci della mappa Clojure siano vettori, non tutti i vettori sono voci della mappa. Se provi a chiamare la `key` o `val` su qualsiasi cosa che non è una voce della mappa, otterrai una `ClassCastException` :

```
(key [:foo :bar])
;; java.lang.ClassCastException:

(val [:foo :bar])
;; java.lang.ClassCastException:
```

Puoi verificare se qualcosa è una voce della `map-entry?` usando la voce della `map-entry?` predicato:

```
(map-entry? (find {:foo :bar} :foo))
;;=> true

(map-entry? [:foo :bar])
;;=> false
```

Puoi utilizzare `assoc` per ottenere una mappa che abbia tutte le stesse coppie chiave-valore di una mappa esistente, con una mappatura aggiunta o modificata:

```
(assoc {} :foo :bar)
;;=> {:foo :bar}

(assoc (assoc {} :foo :bar) :baz :qux)
;;=> {:foo :bar, :baz :qux}

(assoc {:baz :qux} :foo :bar)
;;=> {:baz :qux, :foo :bar}

(assoc {:foo :bar :baz :qux} :foo 42)
;;=> {:foo 42, :baz :qux}

(assoc {:foo :bar :baz :qux} :baz 42)
;;=> {:foo :bar, :baz 42}
```

Puoi usare `dissoc` per ottenere una mappa che abbia tutte le stesse coppie chiave-valore di una mappa esistente, con eventualmente una mappatura rimossa:

```
(dissoc {:foo :bar :baz :qux} 42)
;;=> {:foo :bar :baz :qux}

(dissoc {:foo :bar :baz :qux} :foo)
;;=> {:baz :qux}
```

```
(dissoc {:foo :bar :baz :qux} :bar)
;;=> {:foo :bar :baz :qux}

(dissoc {:foo :bar :baz :qux} :baz)
;;=> {:foo :bar}

(dissoc {:foo :bar :baz :qux} :qux)
;;=> {:foo :bar :baz :qux}

(dissoc {:foo nil} :foo)
;;=> {}
```

`count` restituisce il numero di mapping, in tempo costante:

```
(count {})
;;=> 0

(count (assoc {} :foo :bar))
;;=> 1

(count {:foo :bar :baz :qux})
;;=> 2
```

Puoi ottenere una sequenza di tutte le voci in una mappa usando `seq` :

```
(seq {})
;;=> nil

(seq {:foo :bar})
;;=> ([:foo :bar])

(seq {:foo :bar :baz :qux})
;;=> ([:foo :bar] [:baz :qux])
```

Di nuovo, le mappe non sono ordinate, quindi l'ordinamento degli elementi in una sequenza che ottieni chiamando `seq` su una mappa non è definito.

È possibile ottenere una sequenza di soli i tasti o solo i valori in una mappa utilizzando `keys` o `vals` , rispettivamente:

```
(keys {})
;;=> nil

(keys {:foo :bar})
;;=> (:foo)

(keys {:foo :bar :baz :qux})
;;=> (:foo :baz)

(vals {})
;;=> nil

(vals {:foo :bar})
;;=> (:bar)

(vals {:foo :bar :baz :qux})
```

```
;;=> (:bar :qux)
```

Clojure 1.9 aggiunge una sintassi letterale per rappresentare in modo più conciso una mappa in cui le chiavi condividono lo stesso spazio dei nomi. Si noti che la mappa in entrambi i casi è identica (la mappa non "conosce" lo spazio dei nomi predefinito), questa è solo una comodità sintattica.

```
;; typical map syntax
(def p {:person/first"Darth" :person/last "Vader" :person/email "darth@death.star"})

;; namespace map literal syntax
(def p #:person{:first "Darth" :last "Vader" :email "darth@death.star"})
```

Leggi Collezioni e sequenze online: https://riptutorial.com/it/clojure/topic/1389/collezioni-e-sequenze

Capitolo 9: core.async

Examples

operazioni di base del canale: creazione, inserimento, chiusura, chiusura e buffer.

`core.async` riguarda i *processi* che prendono valori e inseriscono valori nei *canali* .

```
(require [clojure.core.async :as a])
```

Creazione di canali con `chan`

Crei un canale usando la funzione `chan` :

```
(def chan-0 (a/chan)) ;; unbuffered channel: acts as a rendez-vous point.
(def chan-1 (a/chan 3)) ;; channel with a buffer of size 3.
(def chan-2 (a/chan (a/dropping-buffer 3)) ;; channel with a *dropping* buffer of size 3
(def chan-3 (a/chan (a/sliding-buffer 3)) ;; channel with a *sliding* buffer of size 3
```

Mettere valori in canali con `>!!` e `>!`

Metti i valori in un canale con `>!!` :

```
(a/>!! my-channel :an-item)
```

Puoi inserire qualsiasi valore (stringhe, numeri, mappe, raccolte, oggetti, persino altri canali, ecc.) In un canale, tranne `nil` :

```
;; WON'T WORK
(a/>!! my-channel nil)
=> IllegalArgumentException Can't put nil on channel
```

A seconda del buffer del canale, `>!!` potrebbe bloccare il thread corrente.

```
(let [ch (a/chan)] ;; unbuffered channel
  (a/>!! ch :item)
  ;; the above call blocks, until another process
  ;; takes the item from the channel.
  )
(let [ch (a/chan 3)] ;; channel with 3-size buffer
  (a/>!! ch :item-1) ;; => true
  (a/>!! ch :item-2) ;; => true
  (a/>!! ch :item-3) ;; => true
  (a/>!! ch :item-4)
  ;; now the buffer is full; blocks until :item-1 is taken from ch.
  )
```

Dall'interno di un blocco `(go ...)` , puoi - e dovresti - usare `a/>!` invece di `a/>!!` :

```
 (a/go (a/>! ch :item))
```

Il comportamento logico sarà lo stesso di `a/>!!` , ma solo il processo logico della goroutine bloccherà invece del thread effettivo del sistema operativo.

Usando `a/>!!` all'interno di un blocco `(go ...)` c'è un anti-pattern:

```
;; NEVER DO THIS
(a/go
  (a/>!! ch :item))
```

Prendendo i valori dai canali con `<!!`

Prendi un valore da un canale usando `<!!` :

```
;; creating a channel
(def ch (a/chan 3))
;; putting some items in it
(do
  (a/>!! ch :item-1)
  (a/>!! ch :item-2)
  (a/>!! ch :item-3))
;; taking a value
(a/<!! ch) ;; => :item-1
(a/<!! ch) ;; => :item-2
```

Se nessun elemento è disponibile nel canale, `a/<!!` bloccherà il thread corrente fino a quando non viene inserito un valore nel canale (o il canale è chiuso, vedi dopo):

```
(def ch (a/chan))
(a/<!! ch) ;; blocks until another process puts something into ch or closes it
```

Dall'interno di un blocco `(go ...)` , puoi - e dovresti - usare `a/<!` invece di `a/<!!` :

```
 (a/go (let [x (a/<! ch)] ...))
```

Il comportamento logico sarà lo stesso di `a/<!!` , ma solo il processo logico della goroutine bloccherà invece del thread effettivo del sistema operativo.

Usando `a/<!!` all'interno di un blocco `(go ...)` c'è un anti-pattern:

```
;; NEVER DO THIS
(a/go
  (a/<!! ch))
```

Canali di chiusura

Chiudi un canale con `a/close!` :

```
(a/close! ch)
```

Una volta che un canale è stato chiuso e tutti i dati nel canale sono stati esauriti, i `nil` restituiranno sempre `nil` :

```
(def ch (a/chan 5))

;; putting 2 values in the channel, then closing it
(a/>!! ch :item-1)
(a/>!! ch :item-2)
(a/close! ch)

;; taking from ch will return the items that were put in it, then nil
(a/<!! ch) ;; => :item-1
(a/<!! ch) ;; => :item-2
(a/<!! ch) ;; => nil
(a/<!! ch) ;; => nil
(a/<!! ch) ;; => nil

;; once the channel is closed, >!! will have no effect on the channel:
(a/>!! ch :item-3)
=> false ;; false means the put did not succeed
(a/<!! ch) ;; => nil
```

Mette asincrono con `put!`

In alternativa a `a/>!!` (che può bloccare), puoi chiamare `a/put!` inserire un valore in un canale in un altro thread, con un callback opzionale.

```
(a/put! ch :item)
(a/put! ch :item (fn once-put [closed?] ...)) ;; callback function, will receive
```

In ClojureScript, poiché bloccare il thread corrente non è possibile, `a/>!!` non è supportato e `put!` è l'unico modo per mettere i dati in un canale al di fuori di un blocco `(go)` .

Asincrono prende con `take!`

In alternativa a `a/<!!` (che potrebbe bloccare il thread corrente), puoi usare `a/take!` per prendere un valore da un canale in modo asincrono, passandolo a un callback.

```
(a/take! ch (fn [x] (do something with x)))
```

Usando i buffers dropping e sliding

Con i buffer dropping e sliding, puts non blocca mai, tuttavia, quando il buffer è pieno, si perdono i dati. Il buffer in uscita perde gli ultimi dati aggiunti, mentre i buffer di scorrimento perdono i primi dati aggiunti.

Esempio di buffer di rilascio:

```
(def ch (a/chan (a/dropping-buffer 2)))

;; putting more items than buffer size
(a/>!! ch :item-1)
=> true ;; put succeeded
(a/>!! ch :item-2)
=> true
(a/>!! ch :item-3)
=> false ;; put failed

;; no we take from the channel
(a/<!! ch)
=> :item-1
(a/<!! ch)
=> :item-2
(a/<!! ch)
;; blocks! :item-3 is lost
```

Esempio di buffer scorrevole:

```
(def ch (a/chan (a/sliding-buffer 2)))

;; putting more items than buffer size
(a/>!! ch :item-1)
=> true
(a/>!! ch :item-2)
=> true
(a/>!! ch :item-3)
=> true

;; no when we take from the channel:
(a/<!! ch)
=> :item-2
(a/<!! ch)
=> :item-3
;; :item-1 was lost
```

Leggi core.async online: https://riptutorial.com/it/clojure/topic/5496/core-async

Capitolo 10: Emacs CIDER

introduzione

CIDER è l'acronimo di **C** lojure (sceneggiatura) **I** nteractive **D** evelopment **E** nvironment che **R** ocks. È un'estensione di emacs. CIDER mira a fornire un ambiente di sviluppo interattivo al programmatore. CIDER è basato su nREPL, un server REPL in rete e SLIME è stato l'ispirazione principale di CIDER.

Examples

Valutazione della funzione

La funzione CIDER `cider-eval-last-sexp` può essere utilizzata per eseguire il codice durante la modifica del codice all'interno del buffer. Di default questa funzione è associata a `Cx Ce` o `Cx Ce` .

Il manuale CIDER afferma che `Cx Ce` o `Cc Ce` :

> Valuta il modulo che precede il punto e visualizza il risultato nell'area di eco e / o in un overlay del buffer.

Per esempio:

```
(defn say-hello
  [username]
  (format "Hello, my name is %s" username))

(defn introducing-bob
  []
  (say-hello "Bob")) => "Hello, my name is Bob"
```

Eseguendo `Cx Ce` o `Cc Ce` mentre il cursore si trova appena prima del paren finale della funzione `say-hello` , la stringa restituirà la stringa `Hello, my name is Bob` .

Bella stampa

La funzione CIDER `cider-insert-last-sexp-in-repl` può essere utilizzata per eseguire il codice durante la modifica del codice all'interno del buffer e ottenere l'output stampato in un buffer differente. Questa funzione è normalmente associata a `Cc Cp` .

Il manuale CIDER dice che `Cc Cp` farà

> Valuta il modulo precedente e stampa il risultato in un buffer a comparsa.

Per esempio

```
(def databases {:database1  {:password "password"
                             :database "test"
                             :port "5432"
                             :host "localhost"
                             :user "username"}

                :database2 {:password "password"
                            :database "different_test_db"
                            :port "5432"
                            :host "localhost"
                            :user "vader"}})

(defn get-database-config
  []
  databases)

(get-database-config)
```

Eseguendo `Cc Cp` mentre il cursore si trova appena prima del paren finale della chiamata alla funzione `get-database-config` , verrà stampata la bella mappa stampata in un nuovo buffer popup.

```
{:database1
 {:password "password",
  :database "test",
  :port "5432",
  :host "localhost",
  :user "username"},
 :database2
 {:password "password",
  :database "different_test_db",
  :port "5432",
  :host "localhost",
  :user "vader"}}
```

Leggi Emacs CIDER online: https://riptutorial.com/it/clojure/topic/8847/emacs-cider

Capitolo 11: Esecuzione di semplici operazioni matematiche

introduzione

Ecco come aggiungerei alcuni numeri nella sintassi di Clojure. Poiché il metodo si verifica come primo argomento nell'elenco, stiamo valutando il metodo + (o addizione) sul resto degli argomenti nell'elenco.

Osservazioni

Eseguire operazioni matematiche è la base per manipolare i dati e lavorare con le liste. Pertanto, capire come funziona è la chiave per progredire nella comprensione di Clojure.

Examples

Esempi di matematica

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

;; returns 300
(+ 50 210 40)

;; returns 2
(/ 8 4)
```

Leggi Esecuzione di semplici operazioni matematiche online:
https://riptutorial.com/it/clojure/topic/8901/esecuzione-di-semplici-operazioni-matematiche

Capitolo 12: funzioni

Examples

Definizione di funzioni

Le funzioni sono definite con cinque componenti:

L'intestazione, che include la parola chiave `defn` , il nome della funzione.

```
(defn welcome ....)
```

Una docstring facoltativa che spiega e documenta cosa fa la funzione.

```
(defn welcome
    "Return a welcome message to the world"
     ...)
```

Parametri elencati tra parentesi.

```
(defn welcome
    "Return a welcome message"
    [name]
    ...)
```

Il corpo, che descrive le procedure eseguite dalla funzione.

```
(defn welcome
    "Return a welcome message"
    [name]
    (str "Hello, " name "!"))
```

Chiamandolo:

```
=> (welcome "World")

"Hello, World!"
```

Parametri e Arità

Le funzioni del clojure possono essere definite con zero o più parametri.

```
(defn welcome
    "Without parameters"
```

```
    []
    "Hello!")

(defn square
    "Take one parameter"
    [x]
    (* x x))

(defn multiplier
    "Two parameters"
    [x y]
    (* x y))
```

arity

Il numero di argomenti di una funzione. Le funzioni supportano *l'overloading delle* funzioni, il che significa che le funzioni in Clojure consentono più di un "set" di argomenti.

```
(defn sum-args
  ;; 3 arguments
  ([x y z]
     (+ x y z))
  ;; 2 arguments
  ([x y]
     (+ x y))
  ;; 1 argument
  ([x]
     (+ x 1)))
```

Le unità non devono fare lo stesso lavoro, ciascuna delle quali può fare qualcosa di non correlato:

```
(defn do-something
  ;; 2 arguments
  ([first second]
     (str first " " second))
  ;; 1 argument
  ([x]
     (* x x x)))
```

Definizione delle funzioni variabili

Una funzione Clojure può essere definita per prendere un numero arbitrario di argomenti, usando il simbolo **e** nella sua lista di argomenti. Tutti gli argomenti rimanenti sono raccolti come una sequenza.

```
(defn sum [& args]
  (apply + args))

(defn sum-and-multiply [x & args]
  (* x (apply + args)))
```

Calling:

```
=> (sum 1 11 23 42)
77

=> (sum-and-multiply 2 1 2 3)  ;; 2*(1+2+3)
12
```

Definire funzioni anonime

Esistono due modi per definire una funzione anonima: la sintassi completa e una stenografia.

Sintassi della funzione anonima completa

```
(fn [x y] (+ x y))
```

Questa espressione valuta una funzione. Qualsiasi sintassi che è possibile utilizzare con una funzione definita con `defn` (`&` , argomento destructuring, ecc.), Si può fare anche con il modulo `fn` . `defn` è in realtà una macro che fa solo `(def (fn ...))` .

Sintassi della funzione anonima stenografia

```
#(+ %1 %2)
```

Questa è la notazione abbreviata. Usando la notazione abbreviata, non è necessario nominare esplicitamente gli argomenti; verranno assegnati i nomi `%1` , `%2` , `%3` e così via in base all'ordine in cui sono passati. Se la funzione ha solo un argomento, il suo argomento è chiamato solo `%` .

Quando utilizzare ciascuno

La notazione abbreviata ha alcune limitazioni. Non è possibile destrutturare un argomento e non è possibile nidificare funzioni anonime stenografiche. Il seguente codice genera un errore:

```
(def f #(map #(+ %1 2) %1))
```

Sintassi supportata

È *possibile* utilizzare vararg con funzioni anonime stenografiche. Questo è completamente legale:

```
#(every? even? %&)
```

Richiede un numero variabile di argomenti e restituisce true se ognuno di essi è pari:

```
(#(every? even? %&) 2 4 6 8)
;; true
(#(every? even? %&) 1 2 4 6)
;; false
```

Nonostante l'apparente contraddizione, è possibile scrivere una funzione anonima con un nome includendo un nome, come nell'esempio seguente. Ciò è particolarmente utile se la funzione ha bisogno di chiamare se stessa ma anche in tracce dello stack.

```
(fn addition [& addends] (apply + addends))
```

Leggi funzioni online: https://riptutorial.com/it/clojure/topic/3078/funzioni

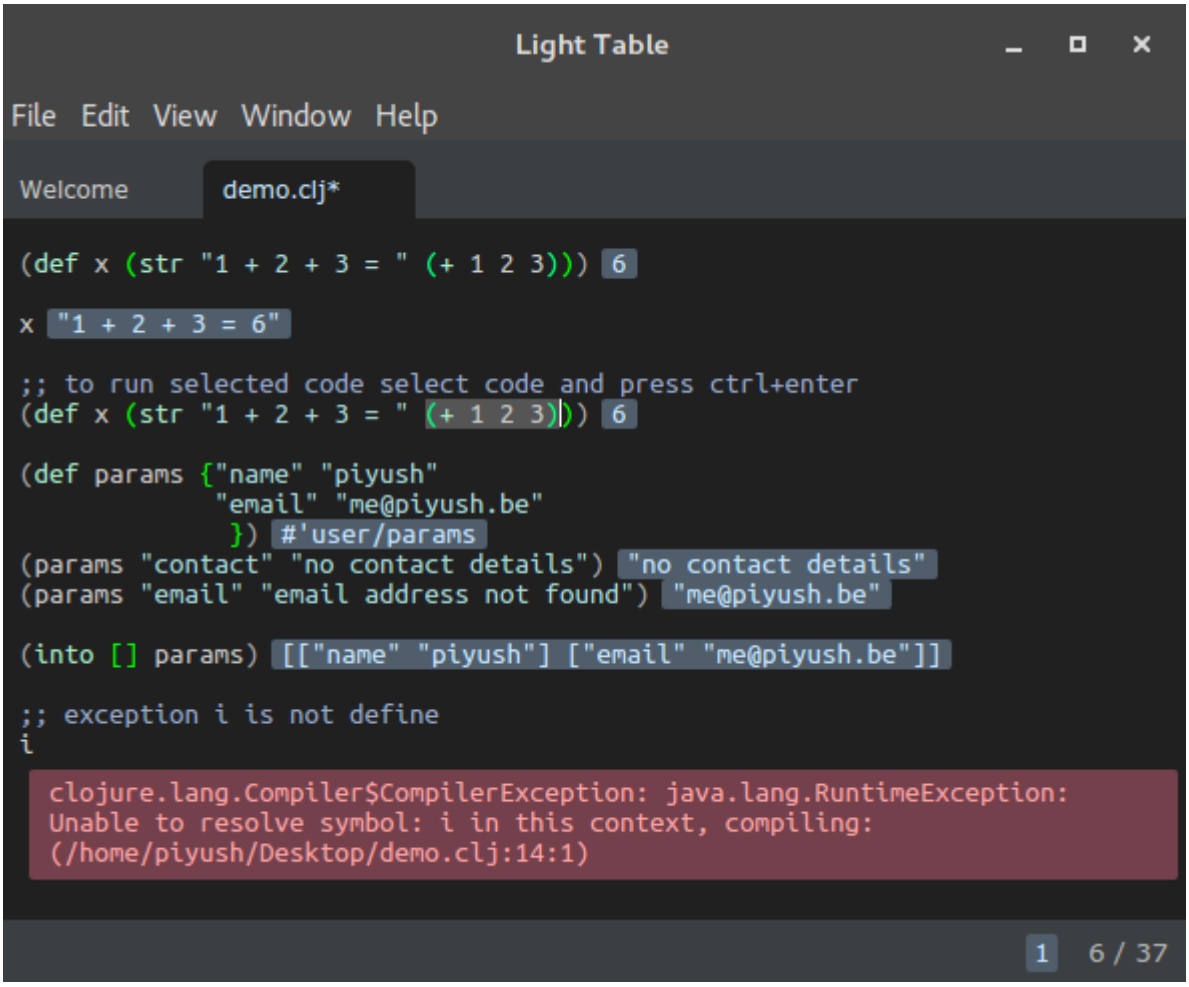
Capitolo 13: Impostazione del tuo ambiente di sviluppo

Examples

Tavolo luminoso

Light Table è un buon editor per imparare, sperimentare ed eseguire progetti Clojure.
È anche possibile eseguire progetti lein / boot aprendo il file `project.clj` . Caricherà tutte le dipendenze del progetto.

Supporta la valutazione inline, i plugin e molto altro, quindi non è necessario aggiungere istruzioni di stampa e controllare l'output nella console. Puoi eseguire singole linee o codice blocska premendo `ctrl + enter` invio. Per eseguire il codice parziale, selezionare il codice e premere `ctrl + enter` . controlla il seguente screenshot per sapere come utilizzare Light Table per imparare e sperimentare con il codice Clojure.



I binari pre-costruiti di Light Table possono essere trovati qui . Non è richiesta alcuna ulteriore configurazione.

Light Table è in grado di localizzare automaticamente il tuo progetto Leiningen e valutare il tuo

codice. Se non hai installato Leiningen, installalo seguendo le istruzioni qui .

Documentazione: docs.lighttable.com

Emacs

Per configurare Emacs per lavorare con Clojure, installa il `clojure-mode` e `cider` da melpa:

```
M-x package-install [RET] clojure-mode [RET]
M-x package-install [RET] cider [RET]
```

Ora quando apri un file `.clj` , esegui `Mx cider-jack-in` per connettersi a un REPL. In alternativa, puoi usare `Cu Mx` (sidro-jack-in) per specificare il nome di un `lein` o di un progetto di `boot` , senza dover visitare alcun file al suo interno. Ora dovresti essere in grado di valutare le espressioni nel tuo file usando `Cx Ce` .

La modifica del codice in lingue simili a Lisp è molto più comoda con un plug-in di modifica compatibile con Paren. Emacs ha diverse buone opzioni.

- `paredit` Una modalità di editing Lisp classica che ha una curva di apprendimento più ripida, ma fornisce un sacco di energia una volta acquisita padronanza.

  `Mx package-install [RET] paredit [RET]`

- `smartparens` Un nuovo progetto con obiettivi e utilizzo simili a `paredit` , ma offre anche funzionalità ridotte con linguaggi non-Lisp.

  `Mx package-install [RET] smartparens [RET]`

- `parinfer` Una modalità di editing Lisp molto più semplice che opera principalmente inferendo il corretto nidificazione paren da indentazione.

  L'installazione è più coinvolgente, vedere la pagina Github per la `parinfer-mode` per le istruzioni di installazione .

Per abilitare il `paredit` in `clojure-mode` :

```
(add-hook 'clojure-mode-hook #'paredit-mode)
```

Per abilitare gli `smartparens` in `clojure-mode` :

```
(add-hook 'clojure-mode-hook #'smartparens-strict-mode)
```

Atomo

Installa Atom per la tua distribuzione qui .

Dopo di che esegui i seguenti comandi da un terminale:

```
apm install parinfer
apm install language-clojure
apm install proto-repl
```

IntelliJ IDEA + Cursive

Scarica e installa l'ultima versione IDEA.

Scarica e installa l'ultima versione del plugin Cursive.

Dopo aver riavviato IDEA, Cursive dovrebbe funzionare senza problemi. Segui la guida per l' utente per perfezionare l'aspetto, le combinazioni di tasti, lo stile del codice, ecc.

Nota: come `IntelliJ` , `Cursive` è un prodotto commerciale, con un periodo di valutazione di 30 giorni. A differenza di `IntelliJ` , non ha un'edizione community. Licenze gratuite non commerciali sono disponibili per gli individui per uso non commerciale, compreso l'hacking personale, l'open-source e il lavoro degli studenti. La licenza è valida per 6 mesi e può essere rinnovata.

Spacemac + CIDER

Spacemacs è una distribuzione di emacs che viene fornita con molti pacchetti preconfigurati e facilmente installabili. Inoltre, è molto amichevole per coloro che hanno familiarità con lo stile di editing vim. Spacemacs fornisce uno strato Clojure basato su CIDER .

Per installarlo e configurarlo per l'uso con Clojure, installare prima emacs. Quindi eseguire un backup delle configurazioni precedenti:

```
$ mv ~/.emacs.d ~/.emacs.d.backup
```

Quindi clona il repository di spacemacs:

```
$ git clone https://github.com/syl20bnr/spacemacs ~/.emacs.d
```

Ora, apri emacs. Ti farà alcune domande riguardanti le tue preferenze. Quindi scarica più pacchetti e configura i tuoi emacs. Dopo che è stato installato lo Spacemac, sei pronto per aggiungere il supporto Clojure. Premere `SPC fed` per aprire il file `.spacemacs` per la configurazione. Trova `dotspacemacs-configuration-layers` nel file, sotto di esso c'è un paren aperto. Ovunque tra i parenti in un nuovo tipo di linea `clojure` .

```
(defun dotspacemacs/layers ()
  (setq-default
   ;; ...
   dotspacemacs-configuration-layers
   '(clojure
     ;; ...
    )
   ;; ...
))
```

Premere `SPC fe R` per salvare e installare il livello clojure. Ora, in qualsiasi file `.clj` se si preme ,

`si` spacemacs proverà a generare una nuova connessione REPL clojure al progetto, e se ha successo verrà mostrato nella barra di stato, che in seguito si può premere `, ss` per aprire un nuovo buffer REPL per valutare i tuoi codici.

Per maggiori informazioni su spacemacs e sidro contattate le loro documentazioni. Documenti di Spacemacs , documenti di sidro

Vim

Installa i seguenti plugin usando il tuo gestore di plugin preferito:

1. fireplace.vim : supporto REPL Clojure
2. vim-sexp : per domare quegli abbracci intorno alle tue chiamate di funzione
3. vim-sexp-mappings-for-regular-people : mappature sexp modificate che sono un po 'più facili da sopportare
4. vim-surround : elimina, modifica, aggiungi "dintorni" in coppia
5. salve.vim : supporto statico Vim per Leiningen e Boot.
6. rainbow_parentheses.vim : Parentheses arcobaleno migliore

e riguardano anche l'evidenziazione della sintassi, il completamento dell'omni, l'evidenziazione avanzata e così via:

1. vim-clojure-static (se hai una vim più vecchia della 7.3.803, le versioni più recenti vengono spedite con questo)
2. vim-clojure-clou

Altre opzioni al posto di vim-sexp includono paredit.vim e vim-parinfer .

Leggi Impostazione del tuo ambiente di sviluppo online:
https://riptutorial.com/it/clojure/topic/1387/impostazione-del-tuo-ambiente-di-sviluppo

Capitolo 14: Iniziare con lo sviluppo web

Examples

Crea una nuova applicazione Ring con http-kit

Ring è de facto API standard per applicazioni HTTP clojure, simile a Ruby's Rack e Python WSGI.

Lo useremo con il webserver http-kit .

Crea un nuovo progetto Leiningen:

```
lein new app myapp
```

Aggiungi la dipendenza da http-kit a `project.clj` :

```
  :dependencies [[org.clojure/clojure "1.8.0"]
                 [http-kit "2.1.18"]]
```

Aggiungi `:require` http-kit per `core.clj` :

```
(ns test.core
  (:gen-class)
  (:require [org.httpkit.server :refer [run-server]]))
```

Definire il gestore della richiesta dell'anello. I gestori di richieste sono solo funzioni dalla richiesta alla risposta e la risposta è solo una mappa:

```
(defn app [req]
  {:status  200
   :headers {"Content-Type" "text/html"}
   :body    "hello HTTP!"})
```

Qui restituiamo solo 200 OK con lo stesso contenuto per qualsiasi richiesta.

Avvia il server in `-main` function:

```
(defn -main
  [& args]
  (run-server app {:port 8080}))
```

Esegui con `lein run` e apri `http://localhost:8080/` nel browser.

Nuova applicazione web con Luminus

Luminus è un micro-framework Clojure basato su una serie di librerie leggere. Mira a fornire una piattaforma robusta, scalabile e facile da usare. Con Luminus puoi concentrarti sullo sviluppo della

tua app come vuoi senza distrazioni. Ha anche un'ottima documentazione che copre alcuni degli argomenti più importanti

È molto facile iniziare con il luminus. Basta creare un nuovo progetto con i seguenti comandi:

```
lein new luminus my-app
cd my-app
lein run
```

Il tuo server inizierà sulla porta 3000

Eseguendo `lein new luminus myapp` verrà creata un'applicazione utilizzando il modello profilo predefinito. Tuttavia, se si desidera aggiungere ulteriori funzionalità al modello, è possibile aggiungere suggerimenti di profilo per la funzionalità estesa.

Server Web

- + aleph - aggiunge il supporto al server Aleph al progetto
- + jetty - aggiunge il supporto di Jetty al progetto
- + http-kit: aggiunge il server web del kit HTTP al progetto

banche dati

- + h2 - aggiunge lo spazio dei nomi db.core e le dipendenze di db di H2
- + sqlite - aggiunge lo spazio dei nomi db.core e le dipendenze db SQLite
- + postgres - aggiunge lo spazio dei nomi db.core e aggiunge le dipendenze di PostreSQL
- + mysql - aggiunge lo spazio dei nomi db.core e aggiunge le dipendenze MySQL
- + mongodb - aggiunge lo spazio dei nomi db.core e le dipendenze di MongoDB
- + datomic - aggiunge lo spazio dei nomi db.core e le dipendenze Datomic

miscellaneo

- + auth - aggiunge la dipendenza di Buddy e il middleware di autenticazione
- + auth-jwe - aggiunge la dipendenza di Buddy con il backend JWE
- + sidro: aggiunge il supporto per CIDER utilizzando il plug-in CIDER nREPL
- + cljs - aggiunge il supporto [ClojureScript] [cljs] con Reagent
- + re-frame - aggiunge il supporto [ClojureScript] [cljs] con re-frame
- + cetriolo - un profilo per cetriolo con clj-webdriver
- + swagger - aggiunge il supporto per Swagger-UI usando la libreria compojure-api
- + sassc - aggiunge il supporto per i file SASS / SCSS usando il compilatore della riga di comando SassC
- + servizio: crea un'applicazione di servizio senza lo schema di front-end come i modelli HTML
- + war - aggiungi il supporto per la costruzione di archivi WAR per la distribuzione su server come Apache Tomcat (NON dovrebbe essere usato per le applicazioni Immutant in esecuzione su WildFly)

- + sito: crea un modello per sito utilizzando il database specificato (H2 per impostazione predefinita) e ClojureScript

Per aggiungere un profilo basta passarlo come argomento dopo il nome dell'applicazione, ad esempio:

```
lein new luminus myapp +cljs
```

Puoi anche mescolare più profili durante la creazione dell'applicazione, ad esempio:

```
lein new luminus myapp +cljs +swagger +postgres
```

Leggi Iniziare con lo sviluppo web online: https://riptutorial.com/it/clojure/topic/2323/iniziare-con-lo-sviluppo-web

Capitolo 15: Interoperabilità Java

Sintassi

- . consentiamo l'accesso ai metodi di istanza
- .- consente di accedere ai campi di istanza
- .. macro espandibile a più invocazioni annidate di .

Osservazioni

Come linguaggio ospitato, Clojure offre un eccellente supporto di interoperabilità con Java. Il codice Clojure può anche essere chiamato direttamente da Java.

Examples

Chiamare un metodo di istanza su un oggetto Java

Puoi chiamare un metodo di istanza usando il . forma speciale:

```
(.trim " hello ")
;;=> "hello"
```

Puoi chiamare i metodi di istanza con argomenti come questo:

```
(.substring "hello" 0 2)
;;=> "he"
```

Fare riferimento a un campo di istanza su un oggetto Java

Puoi chiamare un campo istanza usando la sintassi .- :

```
(def p (java.awt.Point. 0 1))
(.-x p)
;;=> 0
(.-y p)
;;=> 1
```

Creare un nuovo oggetto Java

Puoi creare un'istanza di oggetti in due modi:

```
(java.awt.Point. 0 1)
;;=> => #object[java.awt.Point 0x3776d535 "java.awt.Point[x=0,y=1]"]
```

O

```
(new java.awt.Point 0 1)
;;=> => #object[java.awt.Point 0x3776d535 "java.awt.Point[x=0,y=1]"]
```

Chiamare un metodo statico

Puoi chiamare metodi statici come questo:

```
(System/currentTimeMillis)
;;=> 1469493415265
```

Oppure passa argomenti, come questo:

```
(System/setProperty "foo" "42")
;;=> nil
(System/getProperty "foo")
;;=> "42"
```

Chiamare una funzione Clojure da Java

Puoi chiamare una funzione Clojure dal codice Java cercando la funzione e invocandola:

```
IFn times = Clojure.var("clojure.core", "*");
times.invoke(2, 2);
```

Questo cerca la funzione `*` dal namespace `clojure.core` e la invoca con gli argomenti 2 e 2.

Leggi Interoperabilità Java online: https://riptutorial.com/it/clojure/topic/4036/interoperabilita-java

Capitolo 16: Macro

Sintassi

- Il `'` simbolo usato nell'esempio macroexpand è solo zucchero sintattico per l'operatore `quote` . Potresti aver scritto `(macroexpand (quote (infix 1 + 2)))` invece.

Osservazioni

Le macro sono solo funzioni che vengono eseguite in fase di compilazione, ovvero durante la fase di `eval` in un ciclo di lettura-eval-print .

Le macro di Reader sono un'altra forma di macro che viene espansa in fase di lettura piuttosto che in fase di compilazione.

Best practice durante la definizione della macro.

- alpha-renaming, poiché la macro è espandibile potrebbe sorgere conflitto di nomi di binding. Il conflitto vincolante non è molto intuitivo da risolvere quando si utilizza la macro. Questo è il motivo per cui, ogni volta che una macro aggiunge un'associazione all'ambito, è obbligatorio utilizzare il simbolo `#` alla fine di ogni simbolo.

Examples

Macro Infix semplice

Clojure usa la notazione del prefisso, ovvero: L'operatore viene prima dei suoi operandi.

Ad esempio, una semplice somma di due numeri sarebbe:

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

I macro ti consentono di manipolare la lingua Clojure in una certa misura. Ad esempio, è possibile implementare una macro che consente di scrivere codice in notazione infisso (ad es. `1 + 2`):

```
(defmacro infix [first-operand operator second-operand]
    "Converts an infix expression into a prefix expression"
    (list operator first-operand second-operand))
```

Analizziamo il codice sopra riportato:

- `defmacro` è un *modulo speciale* che usi per definire una macro.
- `infix` è il nome della macro che stiamo definendo.
- `[first-operand operator second-operand]` sono i parametri che questa macro si aspetta di ricevere quando viene chiamata.

- `(list operator first-operand second-operand)` è il corpo della nostra macro. Crea semplicemente una `list` con i valori dei parametri forniti alla macro `infix` e restituisce quello.

`defmacro` è una *forma speciale* perché si comporta in modo leggermente diverso rispetto ad altri costrutti Clojure: i suoi parametri non vengono immediatamente valutati (quando chiamiamo la macro). Questo è ciò che ci permette di scrivere qualcosa come:

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

La macro `infix` espanderà gli argomenti `1 + 2` in `(+ 1 2)` , che è un modulo Clojure valido che può essere valutato.

Se vuoi vedere che cosa genera la macro `infix` , puoi usare l'operatore `macroexpand` :

```
(macroexpand '(infix 1 + 2))
;; => (+ 1 2)
```

`macroexpand` , come implicito nel nome, espanderà la macro (in questo caso, utilizzerà la macro `infix` per trasformare `1 + 2` in `(+ 1 2)`) ma non consentirà di valutare il risultato dell'espansione macro Interprete di Clojure.

Sintassi quoting e unquotation

Esempio dalla libreria standard (core.clj: 807):

```
(defmacro and
  "Evaluates exprs one at a time, from left to right. If a form
  returns logical false (nil or false), and returns that value and
  doesn't evaluate any of the other expressions, otherwise it returns
  the value of the last expr. (and) returns true."
  {:added "1.0"}
  ([] true)
  ([x] x)
  ([x & next]
   `(let [and# ~x]
      (if and# (and ~@next) and#))))
```

- `` ` `` called syntax-quote è come `(quote)` , ma ricorsivo: causa `(let …)` , `(if …)` , ecc per non valutare durante l'espansione della macro ma per produrre come è
- `~` aka unquote annulla la sintassi-citazione per un modulo singolo all'interno di un modulo basato sulla sintassi. Quindi il valore di `x` viene emesso quando si espande la macro (invece di emettere il simbolo `x`)
- `~@` alias unquote-splicing è come unquote ma prende l'argomento list e lo espande, ogni elemento dell'elenco in forma separata
- `#` aggiunge un ID univoco ai simboli per prevenire conflitti di nome. Aggiunge lo stesso id per lo stesso simbolo all'interno dell'espressione con sintassi, così `and#` inside `let` e `and#` inside `if` otterrà lo stesso nome

Leggi Macro online: https://riptutorial.com/it/clojure/topic/2322/macro

Capitolo 17: Macro di threading

introduzione

Conosciuto anche come macro di freccia, le macro di thread convertono le chiamate di funzioni annidate in un flusso lineare di chiamate di funzione.

Examples

Thread Ultimo (- >>)

Questa macro fornisce l'output di una linea specifica come ultimo argomento della chiamata della funzione di linea successiva. Per es

```
(prn (str (+ 2 3)))
```

è uguale a

```
(->> 2
     (+ 3)
     (str)
     (prn))
```

Prima discussione (->)

Questa macro fornisce l'output di una determinata riga come primo argomento della chiamata della funzione di linea successiva. Per es

```
 (rename-keys (assoc {:a 1} :b 1) {:b :new-b}))
```

Non riesco a capire niente, giusto? Proviamo di nuovo, con ->

```
(-> {:a 1}
    (assoc :b 1)                ;;(assoc map key val)
    (rename-keys {:b :new-b}))  ;;(rename-keys map key-newkey-map)
```

Thread as (as->)

Questa è un'alternativa più flessibile al thread first o thread last. Può essere inserito ovunque nella lista dei parametri della funzione.

```
(as-> [1 2] x
      (map #(+ 1 %) x)
      (if (> (count x) 2) "Large" "Small"))
```

Leggi Macro di threading online: https://riptutorial.com/it/clojure/topic/9582/macro-di-threading

Capitolo 18: Operazioni sui file

Examples

Panoramica

Leggi un file tutto in una volta (non consigliato per file di grandi dimensioni):

```
(slurp "./small_file.txt")
```

Scrivi i dati su un file tutto in una volta:

```
(spit "./file.txt" "Ocelots are Awesome!")    ; overwrite existing content
(spit "./log.txt" "2016-07-26 New entry." :append true)
```

Leggi un file riga per riga:

```
(use 'clojure.java.io)
(with-open [rdr (reader "./file.txt")]
    (line-seq rdr)    ; returns lazy-seq
)    ; with-open macro calls (.close rdr)
```

Scrivi un file riga per riga:

```
(use 'clojure.java.io)
(with-open [wrtr (writer "./log.txt" :append true)]
    (.write wrtr "2016-07-26 New entry.")
)    ; with-open macro calls (.close wrtr)
```

Scrivi su un file, sostituendo il contenuto esistente:

```
(use 'clojure.java.io)
(with-open [wrtr (writer "./file.txt")]
    (.write wrtr "Everything in file.txt has been replaced with this text.")
)    ; with-open macro calls (.close wrtr)
```

Gli appunti:

- È possibile specificare URL e file
- Le opzioni a `(slurp)` e `(spit)` vengono passate rispettivamente a `clojure.java.io/reader` e `/writer` .

Leggi Operazioni sui file online: https://riptutorial.com/it/clojure/topic/3922/operazioni-sui-file

Capitolo 19: Parsing logs con clojure

Examples

Analizza una riga di registro con record ed espressioni regolari

```
(defrecord Logline [datetime action user id])
(def pattern #"(\d{8}-\d{2}:\d{2}:\d{2}.\d{3})\|.*\|(\w*),(\w*),(\d*)")
(defn parser [line]
  (if-let [[_ dt a u i] (re-find pattern line)]
          (->Logline dt a u i)))
```

Definire una linea di esempio:

```
(def sample "20170426-17:20:04.005|bip.com|1.0.0|alert|Update,john,12")
```

Provalo :

```
(parser sample)
```

Risultato:

```
#user.Logline{:datetime "20170426-17:20:04.005", :action "Update", :user "john", :id "12"}
```

Leggi Parsing logs con clojure online: https://riptutorial.com/it/clojure/topic/9822/parsing-logs-con-clojure

Capitolo 20: Pattern Matching con core.match

Osservazioni

La libreria `core.match` implementa un algoritmo di compilazione di corrispondenze di pattern che utilizza la nozione di "necessità" dalla corrispondenza di pattern lazy.

Examples

Letterali corrispondenti

```
(let [x true
      y true
      z true]
  (match [x y z]
     [_ false true] 1
     [false true _ ] 2
     [_ _ false] 3
     [_ _ true] 4))

;=> 4
```

Abbinare un vettore

```
(let [v [1 2 3]]
  (match [v]
    [[1 1 1]] :a0
    [[1 _ 1]] :a1
    [[1 2 _]] :a2))  ;; _ is used for wildcard matching

;=> :a2
```

Abbinare una mappa

```
(let [x {:a 1 :b 1}]
   (match [x]
     [{:a _ :b 2}] :a0
     [{:a 1 :b _}] :a1
     [{:x 3 :y _ :z 4}] :a2))

;=> :a1
```

Abbinare un simbolo letterale

```
(match [['asymbol]]
  [['asymbol]] :success)

;=> :success
```

Leggi Pattern Matching con core.match online: https://riptutorial.com/it/clojure/topic/2569/pattern-matching-con-core-match

Capitolo 21: trasduttori

introduzione

I trasduttori sono componenti componibili per l'elaborazione di dati indipendentemente dal contesto. In questo modo possono essere utilizzati per elaborare raccolte, flussi, canali, ecc. Senza conoscere le loro sorgenti di input o i sink di output.

La libreria di core Clojure è stata estesa in 1.7 in modo che la sequenza funzioni come map, filter, take, etc. restituisca un trasduttore quando viene chiamato senza una sequenza. Poiché i trasduttori sono funzioni con contratti specifici, possono essere composti utilizzando la normale funzione `comp` .

Osservazioni

I trasduttori consentono di controllare la pigrizia man mano che vengono consumati. Ad esempio `into` è desideroso come ci si aspetterebbe, ma la `sequence` consumerà pigramente la sequenza attraverso il trasduttore. Tuttavia, la garanzia di pigrizia è diversa. Basta la fonte per consumare un elemento inizialmente:

```
(take 0 (sequence (map #(do (prn '-> %) %)) (range 5)))
;; -> 0
;; => ()
```

O decidere se l'elenco è vuoto:

```
(take 0 (sequence (comp (map #(do (prn '-> %) %)) (remove number?)) (range 5)))
;; -> 0
;; -> 1
;; -> 2
;; -> 3
;; -> 4
;; => ()
```

Che differisce dal solito comportamento a sequenza pigra:

```
(take 0 (map #(do (prn '-> %) %) (range 5)))
;; => ()
```

Examples

Piccolo trasduttore applicato a un vettore

```
(let [xf (comp
           (map inc)
           (filter even?))]
```

```
  (transduce xf + [1 2 3 4 5 6 7 8 9 10]))
;; => 30
```

Questo esempio crea un trasduttore assegnato al `xf` locale e usa `transduce` per applicarlo ad alcuni dati. Il trasduttore aggiunge uno a ciascuno dei suoi ingressi e restituisce solo i numeri pari.

`transduce` è come `reduce` e collassa la raccolta di input su un valore singolo usando la funzione `+` fornita.

Questo si legge come l'ultima macro del thread, ma separa i dati di input dai calcoli.

```
(->> [1 2 3 4 5 6 7 8 9 10]
     (map inc)
     (filter even?)
     (reduce +))
;; => 30
```

Applicazione dei trasduttori

```
(def xf (filter keyword?))
```

Applica a una raccolta, restituendo una sequenza:

```
(sequence xf [:a 1 2 :b :c]) ;; => (:a :b :c)
```

Applica a una raccolta, riducendo la raccolta risultante con un'altra funzione:

```
(transduce xf str [:a 1 2 :b :c]) ;; => ":a:b:c"
```

Applica a una raccolta e `conj` il risultato a un'altra raccolta:

```
(into [] xf [:a 1 2 :b :c]) ;; => [:a :b :c]
```

Crea un canale asincrono principale che utilizza un trasduttore per filtrare i messaggi:

```
(require '[clojure.core.async :refer [chan >!! <!! poll!]])
(doseq [e [:a 1 2 :b :c]] (>!! ch e))
(<!! ch) ;; => :a
(<!! ch) ;; => :b
(<!! ch) ;; => :c
(poll! ch) ;;=> nil
```

Creazione / uso di trasduttori

Quindi le funzioni più utilizzate sulla mappa e sul filtro Clojure sono state modificate per restituire trasduttori (trasformazioni algoritmiche componibili), se non chiamate con una raccolta. Questo significa:

`(map inc)` restituisce un trasduttore e lo fa `(filter odd?)`

Il vantaggio: le funzioni possono essere composte in una singola funzione da comp, il che significa attraversare la raccolta una sola volta. Risparmia tempo di esecuzione di oltre il 50% in alcuni scenari.

Definizione:

```
(def composed-fn (comp (map inc) (filter odd?)))
```

Uso:

```
;; So instead of doing this:
(->> [1 8 3 10 5]
     (map inc)
    (filter odd?))
;; Output [9 11]

;; We do this:
(into [] composed-fn [1 8 3 10 5])
;; Output: [9 11]
```

Leggi trasduttori online: https://riptutorial.com/it/clojure/topic/10814/trasduttori

Capitolo 22: truthiness

Examples

truthiness

In Clojure tutto ciò che non è `nil` o `false` è considerato logico vero.

Esempi:

```
(boolean nil)         ;=> false
(boolean false)       ;=> false
(boolean true)        ;=> true
(boolean :a)          ;=> true
(boolean "false")     ;=> true
(boolean 0)           ;=> true
(boolean "")          ;=> true
(boolean [])          ;=> true
(boolean '())         ;=> true

(filter identity [:a false :b true])   ;=> (:a :b true)
(remove identity [:a false :b true])   ;=> (false)
```

booleani

Qualsiasi valore in Clojure è considerato vero a meno che non sia `false` o `nil` . Puoi trovare la verità di un valore con `(boolean value)` . Puoi trovare la veridicità di un elenco di valori usando `(or)` , che restituisce `true` se qualsiasi argomento è true, o `(and)` che restituisce `true` se tutti gli argomenti sono veri.

```
=> (or false nil)
nil    ; none are truthy
=> (and '() [] {} #{} "" :x 0 1 true)
true    ; all are truthy
=> (boolean "false")
true    ; because naturally, all strings are truthy
```

Leggi truthiness online: https://riptutorial.com/it/clojure/topic/4116/truthiness

Capitolo 23: Vars

Sintassi

- (valore simbolo def)
- (valore def "docstring" valore)
- (dichiarare symbol_0 symbol_1 symbol_2 ...)

Osservazioni

Questo non dovrebbe essere confuso con `(defn)` , che è usato per definire le funzioni.

Examples

Tipi di variabili

Esistono diversi tipi di variabili in Clojure:

- numeri

  Tipi di numeri:

  - interi
  - lunghi (numeri maggiori di `2^31 - 1`)
  - galleggianti (decimali)

- stringhe

- collezioni

  Tipi di collezioni:

  - mappe
  - sequenze
  - vettori

- funzioni

Leggi Vars online: https://riptutorial.com/it/clojure/topic/4449/vars

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Clojure	adairdavid, Adeel Ansari, alejosocorro, Alex Miller, Arclite, avichalp, Blake Miller, Community, CP9, D-side, Geoff, Greg, KettuJKL, Kiran, Martin Janiczek, n2o, Nikita Prokopov, Sajjad, Sam Estep, Sean Allred, Zaz
2	Atomo	Qwerp-Derp, systemfreund
3	CLJ-time	Rishu Saniya, Akanksha, Mrinal Saurabh, Vishakha Silky
4	Clojure destrutturante	camdez, kaffein, kolen, leeor, Michał Marczyk, MuSaiXi, r00tt, RedBlueThing, ryo, tsleyson, Zaz
5	clojure.core	Akanksha, Mrinal Saurabh, Surbhi Garg
6	clojure.spec	Adam Lee, Alex Miller, kolen, leeor, nXqd
7	clojure.test	jisaw, kolen, leeor, porglezomp, Sam Estep
8	Collezioni e sequenze	Alex Miller, Kenogu Labz, nXqd, Sam Estep
9	core.async	Valentin Waeselynck
10	Emacs CIDER	avichalp
11	Esecuzione di semplici operazioni matematiche	Jim
12	funzioni	alejosocorro, fokz, Qwerp-Derp, tar, tsleyson
13	Impostazione del tuo ambiente di sviluppo	Adeel Ansari, agent_orange, amalloy, g1eny0ung, Geoff, Kiran, kolen, MuSaiXi, Piyush, Qwerp-Derp, spinningarrow, stardiviner, superkonduktr, swlkr
14	Iniziare con lo sviluppo web	Emin Tham, kolen, r00tt
15	Interoperabilità Java	leeor
16	Macro	Alex Miller, kolen, mathk, Sam Estep, snowcrshd
17	Macro di threading	Kusum Ijari, Mrinal Saurabh

18	Operazioni sui file	Zaz
19	Parsing logs con clojure	user2611740
20	Pattern Matching con core.match	Kiran
21	trasduttori	l0st3d, Mrinal Saurabh
22	truthiness	Alan Thompson, Michiel Borkent, Zaz
23	Vars	Aryaman Arora, Qwerp-Derp, Stephen Leppik, Zaz