EBook Gratis

APRENDIZAJE clojure

Free unaffiliated eBook created from
Stack Overflow contributors.

#clojure

Tabla de contenido

Acerca de

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

Capítulo 1: Empezando con clojure

Observaciones

Clojure **es un lenguaje de programación de propósito general de tipo dinámico con la sintaxis de Lisp.**

Sus características admiten el estilo funcional de programación con funciones de primera clase y valores inmutables de forma predeterminada. El uso de variables reasignables no es tan fácil en Clojure como en muchos idiomas convencionales, ya que las variables deben crearse y actualizarse como objetos contenedores. Esto fomenta el uso de valores puros que permanecerán como estaban en el momento en que fueron vistos por última vez. Por lo general, esto hace que el código sea mucho más predecible, verificable y con capacidad de concurrencia. Esto también funciona para las colecciones, ya que las estructuras de datos integradas de Clojure son persistentes.

Para el rendimiento, Clojure admite sugerencias de tipo para eliminar la reflexión innecesaria cuando sea posible. Además, se pueden realizar grupos de cambios en las colecciones persistentes en versiones *transitorias* , lo que reduce la cantidad de objetos involucrados. Esto no es necesario la mayoría del tiempo, ya que las colecciones persistentes se copian rápidamente ya que comparten la mayoría de sus datos. Sus garantías de desempeño no están lejos de sus contrapartes mutables.

Entre otras características, Clojure también tiene:

- memoria transaccional de software (STM)
- varias primitivas de concurrencia que no implican bloqueo manual (átomo, agente)
- transformadores de secuencia composable (transductores),
- Instalaciones funcionales de manipulación de árboles (cremalleras).

Debido a su sintaxis simple y alta extensibilidad (a través de macros, implementación de interfaces centrales y reflexión), algunas características del lenguaje que se ven comúnmente se pueden agregar a Clojure con las bibliotecas. Por ejemplo, `core.typed` trae un verificador de tipo estático, `core.async` trae mecanismos de concurrencia simples basados en canales, `core.logic` trae programación lógica.

Diseñado como un lenguaje alojado, puede interoperar con la plataforma en la que se ejecuta. Si bien el objetivo principal es JVM y todo el ecosistema detrás de Java, también se pueden ejecutar implementaciones alternativas en otros entornos, como ClojureCLR ejecutándose en Common Language Runtime o ClojureScript ejecutándose en tiempos de ejecución de JavaScript (incluidos

los navegadores web). Si bien las implementaciones alternativas pueden carecer de algunas de las funcionalidades de la versión JVM, todavía se consideran una familia de idiomas.

Versiones

Versión	Registro de cambios	Fecha de lanzamiento
1.8	Último registro de cambios	2016-01-19
1.7	Cambio de registro 1.7	2015-06-30
1.6	Cambio de registro 1.6	2014-03-25
1.5.1	Cambio de registro 1.5.1	2013-03-10
1.4	Registro de cambios 1.4	15/04/2012
1.3	Registro de cambios 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

Instalación y configuración

Opción 1: Leiningen

Requiere JDK 6 o más nuevo.

La forma más fácil de comenzar con Clojure es descargar e instalar Leiningen, la herramienta estándar de facto para administrar proyectos de Clojure, y luego ejecutar el `lein repl` para abrir 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

Siga los pasos de Linux anteriores o instale con los administradores de paquetes macOS.

Instalar con Homebrew

```
brew install leiningen
```

Instalar con MacPorts

Primero instala Clojure

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

Instala `Leiningen` , una herramienta de compilación para Clojure

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

```
lein self-install
```

Windows

Consulte la documentación oficial .

Opción 2: Distribución oficial

Requiere JRE 6 o más nuevo.

Las versiones de Clojure se publican como archivos JAR simples que se ejecutan en la JVM. Esto es lo que sucede típicamente dentro de las herramientas de construcción de Clojure.

1. Vaya a http://clojure.org y descargue el último archivo de Clojure
2. Extraiga el archivo ZIP descargado en un directorio de su elección
3. Ejecute `java -cp clojure-1.8.0.jar clojure.main` en ese directorio

   Puede que tenga que sustituir el `clojure-1.8.0.jar` en ese comando por el nombre del archivo JAR que realmente descargó.

   Para una mejor experiencia de REPL en la línea de comandos (por ejemplo, `rlwrap` los comandos anteriores), es posible que desee instalar `rlwrap` : `rlwrap java -cp clojure-1.8.0.jar clojure.main`

Opción 3: Arranque

Requiere JDK 7 o más nuevo.

Boot es una herramienta de compilación Clojure de usos múltiples. Comprenderlo requiere cierto

conocimiento de Clojure, por lo que puede que no sea la mejor opción para los principiantes. Consulte el sitio web (haga clic en *Comenzar* allí) para obtener instrucciones de instalación.

Una vez que esté instalado y en su `PATH` , puede ejecutar `boot repl` cualquier lugar para iniciar un Clojure REPL.

"¡Hola Mundo!" en el REPL

La comunidad de Clojure pone un gran énfasis en el desarrollo interactivo, por lo que una gran cantidad de interacción con Clojure ocurre dentro de un REPL (read-eval-print-loop) . Cuando ingresas una expresión en ella, Clojure la **lee** , la **evalúa** e **imprime** el resultado de la evaluación, todo en un **bucle** .

Ya deberías poder lanzar un Clojure REPL. Si no sabe cómo hacerlo, siga las **instrucciones de instalación y configuración** de este tema. Una vez que lo tienes funcionando, escribe lo siguiente en él:

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

Luego presiona `Enter` . Esto debería imprimir `Hello, world!` , seguido del valor de retorno de esta expresión, `nil` .

Si desea ejecutar un poco de clojure al instante, intente REPL en línea. Por ejemplo http://www.tryclj.com/ .

Crear una nueva aplicación

Después de seguir las instrucciones anteriores e instalar Leiningen, inicie un nuevo proyecto ejecutando:

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

Esto configurará un proyecto de Clojure con la plantilla de Leiningen predeterminada dentro de la carpeta `<project-name>` . Hay varias plantillas para Leiningen, que afectan la estructura del proyecto. Más comúnmente es la plantilla "aplicación" utilizada, que agrega una función principal y prepara el proyecto para ser empaquetado en un archivo jar (cuya función principal es el punto de entrada de la aplicación). Esto se puede lograr con esto ejecutando:

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

Suponiendo que utilizó la plantilla de la aplicación para crear una nueva aplicación, puede probar que todo se configuró correctamente, ingresando al directorio creado y ejecutando la aplicación usando:

```
lein run
```

Si ves `Hello, World!` en su consola, está todo listo y listo para comenzar a construir su aplicación.

Puede empaquetar esta sencilla aplicación en dos archivos jar con el siguiente comando:

```
lein uberjar
```

"¡Hola Mundo!" usando Boot

Nota: necesitas instalar Boot antes de probar este ejemplo. Consulte la sección **Instalación y configuración** si aún no lo ha instalado.

Boot permite crear archivos ejecutables de Clojure utilizando la línea **shebang** (#!). Coloque el siguiente texto en un archivo de su elección (este ejemplo asume que está en el "directorio de trabajo actual" y se llama `hello.clj`).

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

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

Luego `chmod +x hello.clj` como ejecutable (si corresponde, normalmente ejecutando `chmod +x hello.clj`).
... y ejecutarlo (`./hello.clj`).

El programa debe mostrar "¡Hola, mundo!" y acaba.

Crear una nueva aplicación (con arranque)

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

Este comando le dirá a boot que tome la tarea `boot-new` desde https://github.com/seancorfield/boot-new y ejecute la tarea con la plantilla de la `app` (vea el enlace para otras plantillas). La tarea creará un nuevo directorio llamado `<appname>` con una estructura de aplicación típica de Clojure. Consulte el archivo README generado para obtener más información.

Para ejecutar la aplicación: `boot run` . Otros comandos se especifican en `build.boot` y se describen en README.

Lea Empezando con clojure en línea: https://riptutorial.com/es/clojure/topic/827/empezando-con-clojure

Capítulo 2: Analizando troncos con clojure

Examples

Analizar una línea de registro con registro y expresiones regulares

```
(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)))
```

Definir una línea de muestra:

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

Pruébalo :

```
(parser sample)
```

Resultado:

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

Lea Analizando troncos con clojure en línea:
https://riptutorial.com/es/clojure/topic/9822/analizando-troncos-con-clojure

Capítulo 3: Átomo

Introducción

Un átomo en Clojure es una variable que se puede cambiar a lo largo de su programa (espacio de nombres). Debido a que la mayoría de los tipos de datos en Clojure son inmutables (o inmutables), no se puede cambiar el valor de un número sin redefinirlo, los átomos son esenciales en la programación de Clojure.

Examples

Define un átomo

Para definir un átomo, use una `def` ordinaria, pero agregue una función de `atom` delante de él, así:

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

Esto crea un `atom` de valor `0` . Los átomos pueden ser de cualquier tipo:

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

Lee el valor de un átomo

Para leer el valor de un átomo, simplemente ponga el nombre del átomo, con una `@` antes de él:

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

Un ejemplo más grande:

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

Actualizar el valor de un átomo.

Hay dos comandos para cambiar un átomo, `swap!` y `reset!` . `swap!` recibe comandos y cambia el átomo en función de su estado actual. `reset!` cambia el valor del átomo por completo, independientemente de cuál sea el valor del átomo original:

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

Este ejemplo produce los primeros 10 poderes de `2` usando átomos:

```
(def count (atom 0))

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

Lea Átomo en línea: https://riptutorial.com/es/clojure/topic/7519/atomo

Capítulo 4: clojure.core

Introducción

Este documento da varias funcionalidades básicas ofrecidas por clojure. No se necesita una dependencia explícita para esto y se incluye como parte de org.clojure.

Examples

Definiendo funciones en 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 - actualización de los valores del mapa / vector en clojure

Cuando se aplica en un mapa, devuelve un nuevo mapa con pares de valores clave nuevos o actualizados.

Se puede utilizar para agregar nueva información en el mapa existente.

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

Reemplaza el valor de información anterior si se suministra una clave existente.

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

También se puede utilizar en un vector para reemplazar el valor en el índice especificado.

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

Operadores de Comparación en Clojure

Las comparaciones son funciones en clojure. Lo que eso significa en `(2>1)` es `(> 2 1)` en clojure. Aquí están todos los operadores de comparación en clojure.

1. Mas grande que

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

2. Menos que

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

3. Mayor qué o igual a

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

4. Menos que o igual a

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

5. Igual a

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

6. No igual a

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

Dissoc: disociar una clave de un mapa de clojure

Esto devuelve un mapa sin los pares clave-valor para las claves mencionadas en el argumento de la función. Se puede utilizar para eliminar información del mapa existente.

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

También se puede utilizar para disociar varias claves como:

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

Lea clojure.core en línea: https://riptutorial.com/es/clojure/topic/9585/clojure-core

Capítulo 5: clojure.spec

Sintaxis

- :: es una abreviatura de palabra clave calificada para el espacio de nombres. Por ejemplo, si estamos en el espacio de nombres de usuario: :: foo es una abreviatura de: user / foo
- #: o # - sintaxis literal del mapa para calificar claves en un mapa por un espacio de nombres

Observaciones

Clojure spec es una nueva biblioteca de especificación / contratos para clojure disponible a partir de la versión 1.9.

Las especificaciones se aprovechan de varias maneras, incluida la inclusión en la documentación, la validación de datos, la generación de datos para pruebas y más.

Examples

Usando un predicado como una especificación

Cualquier función de predicado se puede utilizar como una especificación. Aquí hay un ejemplo simple:

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

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

el `valid?` la función tomará una especificación y un valor y devolverá verdadero si el valor se ajusta a la especificación y falso de lo contrario.

Otro predicado interesante es la membresía establecida:

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

fdef: escribiendo una especificación para una función

Digamos que tenemos la siguiente función:

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

Podemos escribir una especificación para esta función definiendo una especificación de función con el mismo nombre:

```
(clojure.spec/fdef nat-num-count
        :args (s/cat :nums (s/coll-of number?))
        :ret integer?
        :fn #(<= (:ret %) (-> % :args :nums count)))
```

`:args` toma una especificación de expresiones regulares que describe la secuencia de argumentos por una etiqueta de palabra clave correspondiente al nombre del argumento y una especificación correspondiente. La razón por la que la especificación requerida por `:args` es una especificación de expresiones regulares es para admitir múltiples aridades para una función. `:ret` especifica una especificación para el valor de retorno de la función.

`:fn` es una especificación que restringe la relación entre `:args` y `:ret` . Se utiliza como una propiedad cuando se ejecuta a través de test.check. Se llama con un solo argumento: un mapa con dos claves:: `:args` (los argumentos conformes a la función) y `:ret` (el valor de retorno conformado de la función).

Registro de una especificación

Además de los predicados que funcionan como especificaciones, puede registrar una especificación global usando `clojure.spec/def` . `def` requiere que una especificación que se está registrando sea nombrada por una palabra clave calificada para el espacio de nombres:

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

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

Una vez registrada, una especificación puede ser referenciada globalmente en cualquier lugar en un programa de Clojure.

La sintaxis de `::odd-nums` es una abreviatura de `:user/odd-nums` , asumiendo que estamos en el espacio de nombres del `user` . `::` calificará el símbolo que precede con el actual nombre.

En lugar de pasar el predicado, ¿podemos pasar el nombre de la especificación a `valid?` , y funcionará de la misma manera.

clojure.spec / y & clojure.spec / or

`clojure.spec/and` & `clojure.spec/or` puede usarse para crear especificaciones más complejas, utilizando múltiples especificaciones o predicados:

```
(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` funciona de manera similar, con una diferencia significativa. Al definir una `or` espec., Debe etiquetar cada rama posible con una palabra clave. Esto se usa para proporcionar ramas específicas que fallan en los mensajes de error:

```
(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
```

Al conformar una especificación utilizando `or` , se devolverá la especificación aplicable que hizo que el valor sea conforme:

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

Especificaciones de registro

Puede especificar un registro de la siguiente manera:

```
(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
```

En algún momento en el futuro, se puede introducir una sintaxis de lector o un soporte integrado para calificar las claves de registro por el espacio de nombres de los registros. Este soporte ya existe para los mapas.

Especificaciones del mapa

Puede especificar un mapa especificando qué claves deben estar presentes en el mapa:

```
(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` es un vector de claves que deben estar presentes en el mapa. Puede especificar opciones adicionales como `:opt` , un vector de claves que son opcionales.

Los ejemplos hasta ahora requieren que las claves en el nombre estén calificadas para el espacio de nombres. Pero es común que las claves del mapa no estén calificadas. Para este caso, `clojure.spec` proporciona: req y: opt equivalentes para claves no calificadas `:req-un` y `:opt-un` . Aquí está el mismo ejemplo, con claves no calificadas:

```
(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
```

Observe cómo las especificaciones proporcionadas en el vector `:req-un` aún calificadas. clojure.spec, confirmará automáticamente las versiones no calificadas en el mapa al conformar los valores.

La sintaxis literal del mapa de espacio de nombres le permite calificar de manera sucinta todas las claves de un mapa por un solo espacio de nombres. Por ejemplo:

```
(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
```

Fíjate en la `#:` especial `#:` sintaxis del lector. Seguimos esto con el espacio de nombres por el que deseamos calificar todas las claves de mapa. Estos se compararán con las especificaciones correspondientes al espacio de nombres proporcionado.

Colecciones

Puede especificar colecciones de varias maneras. coll-of le permite especificar colecciones y proporcionar algunas restricciones adicionales. Aquí hay un ejemplo simple:

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

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

Las opciones de restricción siguen la especificación / predicado principal de la colección. Puede restringir el tipo de colección con `:kind` como este:

```
(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
```

Lo anterior es falso porque la colección pasada no es un vector.

```
(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
```

Lo anterior es falso porque no todos los elementos en el conjunto son ints.

También puede restringir el tamaño de la colección de varias maneras:

```
(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
```

También puede imponer la unicidad de los elementos de la colección 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` asegura que todos los elementos en una secuencia son verificados. Para grandes colecciones, esto puede ser muy ineficiente. `every` comporta como una `coll-of` , excepto que solo muestrea un número relativamente pequeño de elementos de las secuencias para su conformidad. Esto funciona bien para grandes colecciones. Aquí hay un ejemplo:

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

`map-of` es similar a `coll-of` , pero para mapas. Como los mapas tienen tanto claves como valores,

debe proporcionar una especificación para la clave y una especificación para el valor:

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

Al igual que la `coll-of` , la `map-of` controles cumple con todas las claves / valores del mapa. Para mapas grandes esto será ineficiente. Al igual que la `coll-of` , `map-of` suministra `every-kv` para muestrear de manera eficiente un número relativamente pequeño de valores de un mapa grande:

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

Secuencias

La especificación puede describirse y usarse con secuencias arbitrarias. Es compatible con esto a través de una serie de operaciones de especificación de expresiones regulares.

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

`cat` requiere etiquetas para cada especificación utilizada para describir la secuencia. El gato describe una secuencia de elementos y una especificación para cada uno.

`alt` se utiliza para elegir entre una serie de posibles especificaciones para un elemento dado en una secuencia. Por ejemplo:

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

`alt` también requiere que cada especificación esté etiquetada por una palabra clave.

Las secuencias Regex se pueden componer de formas muy interesantes y potentes para crear especificaciones de descripciones de secuencias arbitrariamente complejas. Aquí hay un ejemplo un poco más complejo:

```
(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
```

Aquí `::complex-seq` validará una secuencia de uno o más pares de elementos, el primero es un int y el segundo es un mapa de palabra clave para int.

Lea clojure.spec en línea: https://riptutorial.com/es/clojure/topic/2325/clojure-spec

Capítulo 6: Coincidencia de patrones con core.match

Observaciones

La biblioteca `core.match` implementa un algoritmo de compilación de coincidencia de patrones que utiliza la noción de "necesidad" de la coincidencia de patrones diferida.

Examples

Literales a juego

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

;=> 4
```

Emparejando un vector

```
(let [v [1 2 3]]
  (match [v]
    [[1 1 1]] :a0
    [[1 _ 1]] :a1
    [[1 2 _]] :a2))  ;; _ is used for wildcard matching

;=> :a2
```

Coincidencia con un mapa

```
(let [x {:a 1 :b 1}]
   (match [x]
     [{:a _ :b 2}] :a0
     [{:a 1 :b _}] :a1
     [{:x 3 :y _ :z 4}] :a2))

;=> :a1
```

Coincidiendo con un símbolo literal

```
(match [['asymbol]]
  [['asymbol]] :success)
```

```
;=> :success
```

Lea Coincidencia de patrones con core.match en línea:
https://riptutorial.com/es/clojure/topic/2569/coincidencia-de-patrones-con-core-match

Capítulo 7: Colecciones y secuencias

Sintaxis

- `'()` → `()`
- `'(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"}` (las entradas se ordenan por clave cuando se usan como una secuencia)
- `#{}` → `#{}`
- `#{1 2 3 4 5}` → `#{4 3 2 5 1}` (sin ordenar)
- `(hash-set 1 2 3 4 5)` → `#{2 5 4 1 3}` (sin ordenar)
- `(sorted-set 2 5 4 3 1)` → `#{1 2 3 4 5}`

Examples

Colecciones

Todas las colecciones de Clojure integradas son inmutables y heterogéneas, tienen una sintaxis literal y admiten las funciones `conj` , `count` y `seq` .

- `conj` devuelve una nueva colección que es equivalente a una colección existente con un elemento "agregado", ya sea en "constante" o tiempo logarítmico. Lo que exactamente esto significa depende de la colección.
- `count` devuelve el número de elementos en una colección, en tiempo constante.
- `seq` devuelve `nil` para una colección vacía, o una secuencia de elementos para una colección no vacía, en tiempo constante.

Liza

Una lista se denota entre paréntesis:

```
()
```

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

Una lista de Clojure es una lista enlazada individualmente . `conj` "combina" un nuevo elemento de la colección en la ubicación más eficiente. Para listas, esto es al principio:

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

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

A diferencia de otras colecciones, las listas no vacías se evalúan como llamadas a formularios especiales, macros o funciones cuando se evalúan. Por lo tanto, mientras que `(:foo)` *es* la representación literal de la lista que contiene `:foo` como su único elemento, evaluar `(:foo)` en un REPL provocará que se `IllegalArgumentException` una `IllegalArgumentException` porque una palabra clave no puede invocarse como una función nula .

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

Para evitar que Clojure evalúe una lista no vacía, puede `quote` :

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

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

Desafortunadamente, esto hace que los elementos no sean evaluados:

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

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

Por esta razón, normalmente querrá usar `list` , una función variadic que evalúa todos sus argumentos y utiliza esos resultados para construir una lista:

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

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

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

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

`count` devuelve el número de elementos, en tiempo constante:

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

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

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

¿Puedes probar si algo es una lista usando la `list?` predicado:

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

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

(list? nil)
;;=> false

(list? 42)
;;=> false

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

Puedes obtener el primer elemento de una lista usando `peek` :

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

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

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

Puedes obtener una nueva lista sin el primer elemento usando `pop` :

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

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

Tenga en cuenta que si intenta `pop` una lista vacía, obtendrá un `IllegalStateException` :

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

Finalmente, todas las listas son secuencias, por lo que puede hacer todo con una lista que puede hacer con cualquier otra secuencia. De hecho, con la excepción de la lista vacía, llamar a `seq` en una lista devuelve el mismo objeto exacto:

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

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

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

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

Secuencias

Una secuencia es muy parecida a una lista: es un objeto inmutable que puede darle su `first` elemento o el `rest` de sus elementos en tiempo constante. También puede `cons` truct una nueva secuencia de una secuencia existente y un elemento para pegarse al principio.

Puedes probar si algo es una secuencia usando la secuencia `seq?` predicado:

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

(seq? 42)
;;=> false

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

Como ya sabes, las listas son secuencias:

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

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

Todo lo que obtenga al llamar a `seq` o `rseq` o las `keys` o `vals` en una colección no vacía también es una secuencia:

```
(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
```

Recuerde que todas las listas son secuencias, pero no todas las secuencias son listas. Mientras que las listas admiten `peek` y `pop` y `count` en tiempo constante, en general, una secuencia no necesita admitir ninguna de esas funciones. Si intentas llamar `peek` o `pop` en una secuencia que no es compatible con la interfaz de pila de Clojure, obtendrás 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
```

Si llama al `count` en una secuencia que no implementa el `count` en un tiempo constante, no

obtendrá un error; en su lugar, Clojure recorrerá toda la secuencia hasta que llegue al final, luego devolverá el número de elementos que atravesó. Esto significa que, para secuencias generales, el `count` es lineal, no constante, en el tiempo. ¿Puede probar si algo admite el `count` tiempo constante usando el `counted?` predicado:

```
(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
```

Como se mencionó anteriormente, puede usar `first` para obtener el primer elemento de una secuencia. Tenga en cuenta que `first` llamará a `seq` en su argumento, por lo que se puede usar en cualquier cosa "seqable", no solo en secuencias reales:

```
(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
```

También como se mencionó anteriormente, puede usar el `rest` para obtener una secuencia que contenga todo menos el primer elemento de una secuencia existente. Como `first` , llama a la `seq`

en su argumento. Sin embargo, *no* llama a la `seq` en su resultado! Esto significa que, si llama a `rest` en una secuencia que contiene menos de dos elementos, obtendrá back `()` lugar de `nil` :

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

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

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

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

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

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

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

Si desea volver a `nil` cuando no hay más elementos en una secuencia, puede usar `next` lugar de `rest` :

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

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

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

Puede usar la función `cons` para crear una nueva secuencia que devolverá su primer argumento para el `first` argumento y su segundo argumento para el `rest` :

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

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

Clojure proporciona una gran biblioteca de secuencias con muchas funciones para tratar con secuencias. Lo importante de esta biblioteca es que funciona con cualquier cosa "seqable", no solo con listas. Es por eso que el concepto de una secuencia es tan útil; significa que una sola función, como `reduce` , funciona perfectamente en cualquier colección:

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

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

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

La otra razón por la que las secuencias son útiles es que, dado que no requieren ninguna implementación particular de `first` y `rest` , permiten secuencias perezosas cuyos elementos solo se realizan cuando es necesario.

Dada una expresión que crearía una secuencia, puede envolver esa expresión en la macro `lazy-seq` para obtener un objeto que actúe como una secuencia, pero solo evaluará esa expresión cuando la función `seq` le pida que lo haga, en en qué punto almacenará en caché el resultado de la expresión y reenviará las `first` y `rest` llamadas al resultado en caché.

Para secuencias finitas, una secuencia perezosa generalmente actúa de la misma manera que una secuencia ansiosa equivalente:

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

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

Sin embargo, la diferencia se hace evidente para secuencias infinitas:

```
(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)
```

Vectores

Un vector se denota entre corchetes:

```
[]
;;=> []

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

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

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

Además de usar la sintaxis literal, también puede usar la función `vector` para construir un vector:

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

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

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

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

¿Puedes probar si algo es un vector usando el `vector?` predicado:

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

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

(vector? nil)
;;=> false

(vector? 42)
;;=> false

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

`conj` agrega elementos al final de un vector:

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

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

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

`count` devuelve el número de elementos, en tiempo constante:

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

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

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

Puedes obtener el último elemento de un vector usando `peek` :

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

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

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

Puedes obtener un nuevo vector sin el último elemento usando `pop` :

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

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

Tenga en cuenta que si intenta `IllegalStateException` un vector vacío, obtendrá una `IllegalStateException` :

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

A diferencia de las listas, los vectores están indexados. Puede obtener un elemento de un vector por índice en tiempo "constante" usando `get` :

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

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

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

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

Además, los vectores en sí mismos son funciones que toman un índice y devuelven el elemento en ese índice:

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

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

Sin embargo, si llama a un vector con un índice no válido, obtendrá una `IndexOutOfBoundsException` lugar de `nil` :

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

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

Puede obtener un nuevo vector con un valor diferente en un índice particular usando `assoc` :

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

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

Si pasa un índice igual al `count` del vector, Clojure agregará el elemento como si hubiera usado `conj` . Sin embargo, si pasa un índice que es negativo o mayor que el `count` , obtendrá una `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:
```

Puede obtener una secuencia de los elementos en un vector usando `seq` :

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

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

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

Dado que los vectores están indexados, también puede obtener una secuencia invertida de los elementos de un vector utilizando `rseq` :

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

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

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

Tenga en cuenta que, aunque todas las listas son secuencias, y las secuencias se muestran de la misma manera que las listas, ¡no todas las secuencias son listas!

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

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

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

```
;;=> true

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

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

Conjuntos

Al igual que los mapas, los conjuntos son asociativos y desordenados. A diferencia de los mapas, que contienen asignaciones de claves a valores, los conjuntos se asignan esencialmente de las claves a sí mismos.

Un conjunto se denota con llaves, precedido por un octothorpe:

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

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

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

Al igual que con los mapas, el orden en que aparecen los elementos en un conjunto literal no importa:

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

¿Puedes probar si algo es un conjunto usando el `set?` predicado:

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

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

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

(set? nil)
;;=> false

(set? 42)
;;=> false

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

Puede probar si un mapa contiene un ítem dado en tiempo "constante" usando los `contains?` predicado:

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

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

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

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

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

Además, los propios conjuntos son funciones que toman un elemento y devuelven ese elemento si está presente en el conjunto, o `nil` si no lo está:

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

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

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

(#{} nil)
;;=> nil

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

Puede usar `conj` para obtener un conjunto que tenga todos los elementos de un conjunto existente, más un elemento adicional:

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

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

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

Puede usar `disj` para obtener un conjunto que tenga todos los elementos de un conjunto existente, menos un elemento:

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

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

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

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

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

`count` devuelve el número de elementos, en tiempo constante:

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

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

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

Puede obtener una secuencia de todos los elementos en un conjunto utilizando `seq` :

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

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

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

Mapas

A diferencia de la lista, que es una estructura de datos secuencial, y el vector, que es a la vez secuencial y asociativo, el mapa es exclusivamente una estructura de datos asociativa. Un mapa consiste en un conjunto de asignaciones de claves a valores. Todas las claves son únicas, por lo que los mapas admiten la búsqueda en tiempo "constante" de las claves a los valores.

Un mapa se denota con llaves:

```
{}
;;=> {}

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

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

Cada par de dos elementos es un par clave-valor. Entonces, por ejemplo, el primer mapa de arriba no tiene mapeos. El segundo tiene un mapeo, desde la clave `:foo` hasta el valor `:bar` . El tercero tiene dos asignaciones, una de la clave `:foo` al valor `:bar` , y una de la clave `:baz` al valor `:qux` . Los mapas están intrínsecamente desordenados, por lo que el orden en el que aparecen las asignaciones no importa:

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

¿Puedes probar si algo es un mapa usando el `map?` predicado:

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

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

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

(map? nil)
;;=> false

(map? 42)
;;=> false

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

¿Puede probar si un mapa contiene una *clave* dada en tiempo "constante" usando los `contains?` predicado:

```
(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
```

Puede obtener el valor asociado con una clave 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
```

Además, los mapas en sí mismos son funciones que toman una clave y devuelven el valor asociado con esa clave:

```
({: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
```

Puede obtener una entrada completa del mapa (clave y valor juntos) como un vector de dos elementos 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
```

Puede extraer la clave o el valor de una entrada de mapa usando `key` o `val` , respectivamente:

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

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

Tenga en cuenta que, aunque todas las entradas del mapa de Clojure son vectores, no todos los vectores son entradas del mapa. Si intenta llamar a `key` o `val` en algo que no sea una entrada de mapa, obtendrá una `ClassCastException` :

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

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

¿Puede probar si algo es una entrada de mapa usando la `map-entry?` predicado:

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

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

Puede usar `assoc` para obtener un mapa que tenga todos los mismos pares clave-valor que un mapa existente, con un mapa agregado o modificado:

```
(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}
```

Puede usar `dissoc` para obtener un mapa que tenga todos los mismos pares clave-valor que un mapa existente, con posiblemente un mapa eliminado:

```
(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` devuelve el número de mapeos, en tiempo constante:

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

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

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

Puede obtener una secuencia de todas las entradas en un mapa usando `seq` :

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

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

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

Nuevamente, los mapas no están ordenados, por lo que el orden de los elementos en una secuencia que se obtiene al llamar a `seq` en un mapa no está definido.

Usted puede obtener una secuencia de sólo las teclas o sólo los valores en un mapa utilizando `keys` o `vals` , respectivamente:

```
(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 agrega una sintaxis literal para representar de manera más concisa un mapa en el que las claves comparten el mismo espacio de nombres. Tenga en cuenta que el mapa en cualquier caso es idéntico (el mapa no "conoce" el espacio de nombres predeterminado), esto es simplemente una conveniencia sintáctica.

```
;; 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"})
```

Lea Colecciones y secuencias en línea: https://riptutorial.com/es/clojure/topic/1389/colecciones-y-secuencias

Capítulo 8: Comenzando con el desarrollo web

Examples

Crea una nueva aplicación de anillo con http-kit

Ring es una API estándar de facto para aplicaciones HTTP, similares a Ruby's Rack y al WSGI de Python.

Vamos a usarlo con el servidor web http-kit .

Crear nuevo proyecto Leiningen:

```
lein new app myapp
```

Agregue la dependencia de http-kit a `project.clj` :

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

Agregue `:require` para http-kit a `core.clj` :

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

Definir controlador de solicitud de anillo. Los manejadores de solicitudes son solo funciones de solicitud a respuesta y la respuesta es solo un mapa:

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

Aquí acabamos de devolver 200 OK con el mismo contenido para cualquier solicitud.

Inicie el servidor en la función `-main` :

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

Ejecute con `lein run` y abra `http://localhost:8080/` en el navegador.

Nueva aplicación web con Luminus.

Luminus es un micro-marco Clojure basado en un conjunto de bibliotecas ligeras. Su objetivo es proporcionar una plataforma robusta, escalable y fácil de usar. Con Luminus, puedes concentrarte en desarrollar tu aplicación de la manera que quieras sin distracciones. También tiene muy buena documentación que cubre algunos de los principales temas.

Es muy fácil empezar con luminus. Solo crea un nuevo proyecto con los siguientes comandos:

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

Su servidor se iniciará en el puerto 3000

Al ejecutar `lein new luminus myapp` se creará una aplicación utilizando la plantilla de perfil predeterminada. Sin embargo, si desea adjuntar más funciones a su plantilla, puede agregar sugerencias de perfil para la funcionalidad extendida.

Servidores web

- + aleph - agrega soporte del servidor Aleph al proyecto
- + embarcadero - agrega soporte de embarcadero al proyecto
- + http-kit - agrega el servidor web HTTP Kit al proyecto

bases de datos

- + h2: agrega las dependencias db.core namespace y H2 db
- + sqlite - agrega las dependencias db.core namespace y SQLite db
- + postgres: agrega el espacio de nombres db.core y agrega las dependencias de PostreSQL
- + mysql: agrega el espacio de nombres db.core y agrega las dependencias de MySQL
- + mongodb - agrega las dependencias de espacio de nombres db.core y MongoDB
- + datomic - agrega espacio de nombres db.core y dependencias de Datomic

diverso

- + auth - agrega middleware de autenticación y dependencia de Buddy
- + auth-jwe: agrega la dependencia de Buddy con el backend de JWE
- + sidra - agrega soporte para CIDER usando el complemento CIDER nREPL
- + cljs - agrega soporte de [ClojureScript] [cljs] con reactivo
- + re-marco - añade [ClojureScript] [cljs] soporte con re-marco
- + pepino - un perfil para pepino con clj-webdriver
- + swagger - agrega soporte para Swagger-UI usando la biblioteca compojure-api
- + sassc - agrega soporte para archivos SASS / SCSS usando el compilador de línea de comandos SassC
- + servicio: cree una aplicación de servicio sin la placa del final como las plantillas HTML
- + war: agregue el soporte para crear archivos WAR para su implementación en servidores

como Apache Tomcat (NO se debe usar para las aplicaciones Immutant que se ejecutan en WildFly)
- + sitio: crea una plantilla para el sitio utilizando la base de datos especificada (H2 de forma predeterminada) y ClojureScript

Para agregar un perfil, simplemente páselo como argumento después del nombre de su aplicación, por ejemplo:

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

También puede mezclar varios perfiles al crear la aplicación, por ejemplo:

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

Lea Comenzando con el desarrollo web en línea:
https://riptutorial.com/es/clojure/topic/2323/comenzando-con-el-desarrollo-web

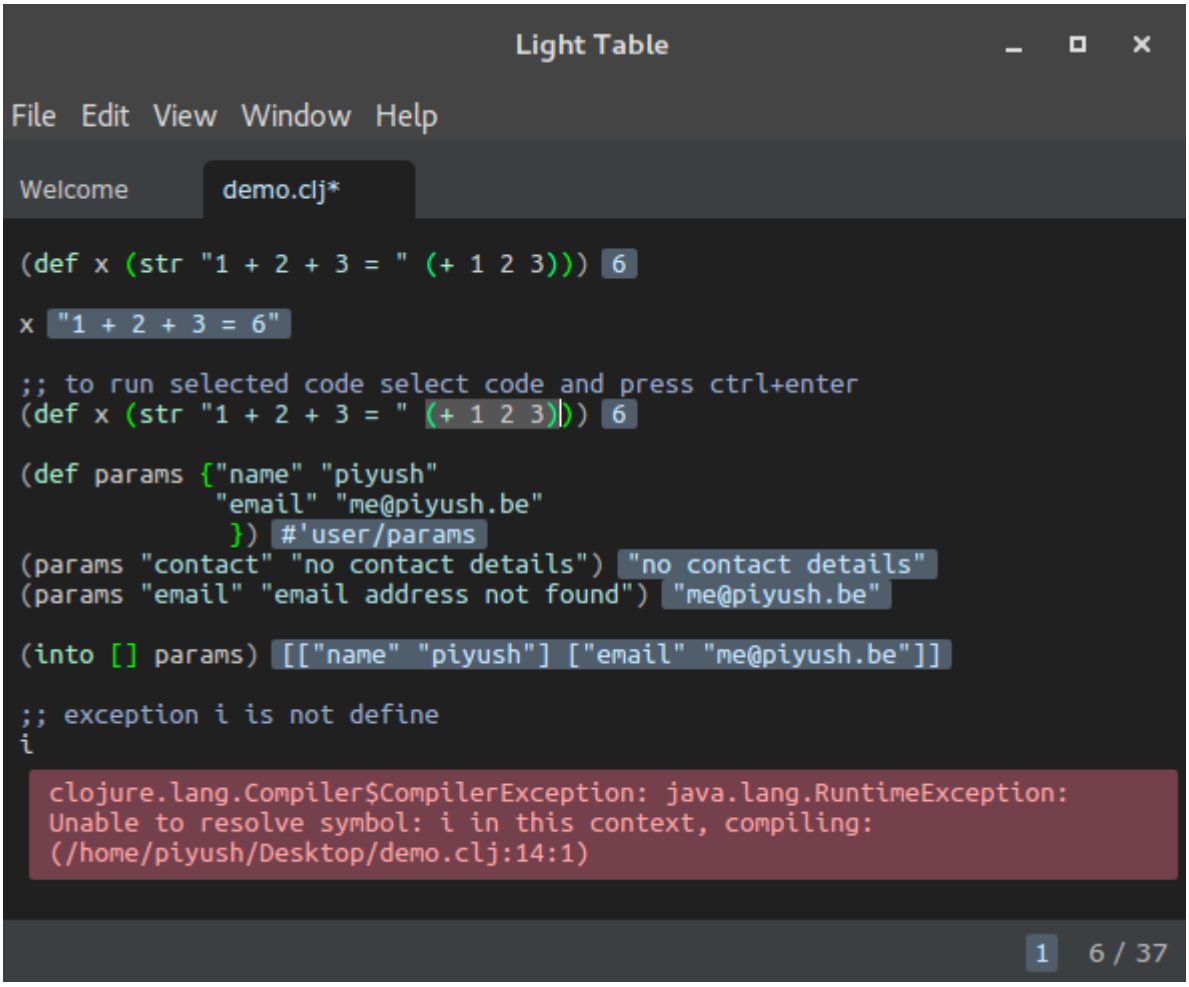
Capítulo 9: Configurando su entorno de desarrollo

Examples

Mesa ligera

Light Table es un buen editor para aprender, experimentar y ejecutar proyectos de Clojure. También puede ejecutar proyectos lein / boot abriendo el archivo `project.clj` . Se cargarán todas las dependencias del proyecto.

Admite la evaluación en línea, los complementos y mucho más, por lo que no es necesario agregar declaraciones de impresión y verificar el resultado en la consola. Puede ejecutar líneas individuales o código blocska presionando `ctrl + enter` . Para ejecutar código parcial, seleccione el código y presione `ctrl + enter` . consulte la siguiente captura de pantalla para saber cómo puede usar Light Table para aprender y experimentar con el código Clojure.



Binarios pre-construidos de Light Table se pueden encontrar aquí . No se requiere más configuración.

Light Table puede localizar automáticamente su proyecto Leiningen y evaluar su código. Si no

tiene instalado Leiningen, instálelo siguiendo las instrucciones que se encuentran aquí .

Documentación: docs.lighttable.com

Emacs

Para configurar Emacs para trabajar con Clojure, instale `clojure-mode` y el paquete de `cider` de melpa:

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

Ahora, cuando abra un archivo `.clj` , ejecute `Mx cider-jack-in` para conectarse a un REPL. Alternativamente, puede usar `Cu Mx` (cider-jack-in) para especificar el nombre de un proyecto de `boot` o de `lein` , sin tener que visitar ningún archivo en él. Ahora debería poder evaluar expresiones en su archivo usando `Cx Ce` .

La edición de código en lenguajes similares a lisp es mucho más cómoda con un complemento de edición con reconocimiento paren. Emacs tiene varias buenas opciones.

- `paredit` Un modo de edición de Lisp clásico que tiene una curva de aprendizaje más pronunciada, pero que proporciona mucha potencia una vez que se domina.

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

- `smartparens` Un proyecto más nuevo con objetivos y uso similares para `paredit` , pero también proporciona capacidades reducidas con lenguajes que no son Lisp.

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

- `parinfer` Un modo de edición de Lisp mucho más simple que funciona principalmente a través de inferir el anidamiento correcto de la sangría.

  La instalación es más complicada, vea la página de Github para `parinfer-mode` de `parinfer-mode` para las instrucciones de configuración .

Para habilitar la `paredit` en `clojure-mode` :

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

Para habilitar `smartparens` en `clojure-mode` :

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

Átomo

Instala Atom para tu distribución aquí .

Después de eso ejecuta los siguientes comandos desde una terminal:

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

IntelliJ IDEA + Cursivo

Descarga e instala la última versión de IDEA.

Descarga e instala la última versión del complemento Cursive.

Después de reiniciar IDEA, Cursive debería estar trabajando fuera de la caja. Siga la guía del usuario para ajustar la apariencia, las combinaciones de teclas, el estilo del código, etc.

Nota: al igual que `IntelliJ` , `Cursive` es un producto comercial, con un período de evaluación de 30 días. A diferencia de `IntelliJ` , no tiene una edición comunitaria. Las licencias gratuitas no comerciales están disponibles para individuos para uso no comercial, incluyendo piratería personal, código abierto y trabajo estudiantil. La licencia es válida por 6 meses y puede ser renovada.

Spacemacs + SIDRA

Spacemacs es una distribución de emacs que viene con muchos paquetes preconfigurados y fáciles de instalar. Además, es muy amigable para aquellos que están familiarizados con el estilo de edición vim. Spacemacs proporciona una capa de Clojure basada en CIDER .

Para instalarlo y configurarlo para usarlo con Clojure, primero instale emacs. Luego haga una copia de seguridad de sus configuraciones anteriores:

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

Luego clona el repositorio de spacemacs:

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

Ahora, abre emacs. Le hará algunas preguntas sobre sus preferencias. Luego descarga más paquetes y configura tus emacs. Después de que se haya instalado Spacemacs, estará listo para agregar el soporte de Clojure. Presione `SPC fed` para abrir su archivo `.spacemacs` para su configuración. Encuentra `dotspacemacs-configuration-layers` en el archivo, debajo de él hay un paren abierto. En cualquier lugar entre los parens en un nuevo tipo de línea `clojure` .

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

Presione `SPC fe R` para guardar e instalar la capa clojure. Ahora, en cualquier archivo `.clj` , si presiona `, si` spacemacs intentará generar una nueva conexión REPL para su proyecto, y si tiene éxito, se mostrará en la barra de estado, que luego puede presionar `, ss` para abrir un nuevo búfer REPL Para evaluar tus códigos.

Para más información sobre spacemacs y sidra contacte sus documentaciones. Spacemacs docs , documentos de sidra

Empuje

Instala los siguientes complementos usando tu administrador de complementos favorito:

1. fireplace.vim : Soporte REPL Clojure
2. vim-sexp : para dominar esos abrazos alrededor de tus llamadas a funciones
3. vim-sexp-mappings-for-regular-people : mapeos sexp modificados que son un poco más fáciles de soportar
4. vim-surround : elimine, cambie, agregue fácilmente "alrededores" en pareja
5. salve.vim : soporte de Static Vim para Leiningen y Boot.
6. rainbow_parentheses.vim : Better Rainbow Paréntesis

y también se relacionan con el resaltado de sintaxis, la finalización omni, el resaltado avanzado, etc.

1. vim-clojure-static (si tiene un vim anterior a 7.3.803, las versiones más nuevas se envían con este)
2. vim-clojure-highlight

Otras opciones en lugar de vim-sexp incluyen paredit.vim y vim-parinfer .

Lea Configurando su entorno de desarrollo en línea: https://riptutorial.com/es/clojure/topic/1387/configurando-su-entorno-de-desarrollo

Capítulo 10: core.async

Examples

Operaciones básicas del canal: crear, poner, tomar, cerrar y buffers.

`core.async` se trata de hacer *procesos* que tomen valores y pongan valores en *canales* .

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

Creando canales con `chan`

Se crea un canal utilizando la función `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
```

Poniendo valores en canales con `>!!` y `>!`

Pones valores en un canal con `>!!` :

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

Puede poner cualquier valor (cadenas, números, mapas, colecciones, objetos, incluso otros canales, etc.) en un canal, excepto `nil` :

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

Dependiendo del buffer del canal, `>!!` puede bloquear el hilo actual.

```
(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.
  )
```

Desde el interior de un bloque `(go ...)` , puede y debe usar `a/>!` en lugar de `a/>!!` :

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

El comportamiento lógico será el mismo que `a/>!!` , pero solo se bloqueará el proceso lógico de goroutine en lugar del hilo del sistema operativo real.

Usando `a/>!!` dentro de un bloque `(go ...)` hay un anti-patrón:

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

Tomando valores de canales con `<!!`

Toma un valor de un canal 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
```

Si no hay ningún elemento disponible en el canal, `a/<!!` bloqueará el hilo actual hasta que se coloque un valor en el canal (o se cierre el canal, ver más adelante):

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

Desde el interior de un bloque `(go ...)` , puede y debe usar `a/<!` en lugar de `a/<!!` :

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

El comportamiento lógico será el mismo que `a/<!!` , pero solo se bloqueará el proceso lógico de goroutine en lugar del hilo del sistema operativo real.

Usando `a/<!!` dentro de un bloque `(go ...)` hay un anti-patrón:

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

Canales de cierre

Cierra un canal con `a/close!` :

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

Una vez que se cierra un canal, y se han agotado todos los datos en el canal, las tomas siempre devolverán `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
```

Asíncrono pone con `put!`

Como alternativa a `a/>!!` (que puede bloquear), puedes llamar `a/put!` para poner un valor en un canal en otro hilo, con una devolución de llamada opcional.

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

En ClojureScript, ya que no es posible bloquear el hilo actual, `a/>!!` No es compatible, y `put!` es la única forma de colocar datos en un canal desde fuera de un bloque `(go)` .

Toma asíncrona con `take!`

Como alternativa a `a/<!!` (que puede bloquear el hilo actual), puede usar `a/take!` para tomar un valor de un canal de forma asíncrona, pasándolo a una devolución de llamada.

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

Utilizando buffers de caída y deslizamiento

Con los buffers que caen y se deslizan, los bloques nunca se bloquean, sin embargo, cuando el buffer está lleno, se pierden datos. Al eliminar el búfer se pierden los últimos datos agregados, mientras que los búferes deslizantes pierden los primeros datos agregados.

Ejemplo de búfer de caída:

```
(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
```

Ejemplo de búfer deslizante:

```
(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
```

Lea core.async en línea: https://riptutorial.com/es/clojure/topic/5496/core-async

Capítulo 11: Destrucción de clojure

Examples

Destructurando un vector

Así es como puedes destruir un vector:

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

Entonces, por ejemplo dentro de un `let` bloque, puede extraer los valores del vector de manera muy sucinta de la siguiente manera:

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

Destructurando un mapa

Así es como puedes destruir un mapa:

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

Luego, por ejemplo, dentro de un bloque Let puede extraer valores del mapa de manera muy sucinta de la siguiente manera:

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

Observe que los valores que se extraen en cada asignación están a la izquierda y las claves con las que están asociadas están a la derecha.

Si desea destruir valores en enlaces con los mismos nombres que las teclas, puede utilizar este acceso directo:

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

Si sus claves son cadenas, puede usar casi la misma estructura:

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

Y de manera similar para los símbolos:

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

Si desea destruir un mapa anidado, puede anidar los formularios de enlace explicados anteriormente:

```
(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]
```

Destrucción de los elementos restantes en una secuencia.

Digamos que tienes un vector así:

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

Y desea extraer los primeros 3 elementos y obtener los elementos restantes como una secuencia. Esto puede hacerse de la siguiente manera:

```
(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)
```

Destructurando vectores anidados

Puedes desestructurar vectores anidados:

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

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

Destrucción de un mapa con valores por defecto.

A veces desea destruir la clave debajo de un mapa que puede no estar presente en el mapa, pero desea un valor predeterminado para el valor desestructurado. Puedes hacerlo de esta manera:

```
(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
```

Destructurar params de un fn.

La destrucción se realiza en muchos lugares, así como en la lista de parámetros de una fn:

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

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

La destrucción también funciona para la construcción `& rest` en la lista de parámetros:

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

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

Convertir el resto de una secuencia en un mapa.

La destrucción también te da la capacidad de interpretar una secuencia como un mapa:

```
(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
```

Es útil para definir funciones con **parámetros nombrados** :

```
(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
```

Visión general

La destrucción le permite extraer datos de varios objetos en distintas variables. En cada ejemplo a continuación, cada variable se asigna a su propia cadena (`a` = `"a"` , b = `"b"` , & c.)

Tipo	Ejemplo	Valor de los `data` / *comentario*
`vec`	`(let [[abc] data ...)`	`["a" "b" "c"]`

Tipo	Ejemplo	Valor de los `data` / *comentario*
`vec` anidado	`(let [[[ab] [cd]] data ...)`	`[["a" "b"] ["c" "d"]]`
`map`	`(let [{a :ab :bc :c} data ...)`	`{:a "a" :b "b" :c "c"}`
- alternativa:	`(let [{:keys [abc]} data ...)`	*Cuando las variables llevan el nombre de las claves.*

Consejos:

- Los valores predeterminados se pueden proporcionar utilizando `:or` , de lo contrario, el valor predeterminado es `nil`
- Use `& rest` para almacenar una `seq` de valores extra en `rest` ; de lo contrario, los valores adicionales se ignorarán
- Un uso común y útil de la desestructuración es para parámetros de función.
- Puede asignar partes no deseadas a una variable desechable (convencionalmente: _)

Destrucción y enlace al nombre de las llaves.

A veces, al desestructurar mapas, le gustaría vincular los valores desestructurados a su respectivo nombre de clave. Dependiendo de la granularidad de la estructura de datos, el uso del esquema de desestructuración *estándar* puede ser un poco *detallado* .

Digamos, tenemos un registro basado en el mapa como tal:

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

Normalmente lo destruiríamos así:

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

aquí, la estructura de datos es bastante simple con solo 3 espacios (primer nombre , *apellido, país*), pero imagínese lo engorroso que sería si tuviéramos que repetir todos los nombres clave dos veces para obtener una estructura de datos más granular (con mucho más espacio que solo 3) .

En su lugar, una mejor manera de manejar esto es mediante `:keys` (ya que nuestras claves son *palabras clave* aquí) y seleccionando el nombre de la clave que nos gustaría vincular como así:

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

La misma *lógica intuitiva se* aplica a otros tipos de claves, como *símbolos* (con `:syms`) y *cadenas* simples y sin formato (con `: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"
```

Destrucción y nombre del argumento original.

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

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

Este ejemplo imprime la salida.

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

El argumento está desestructurado y los elementos `2` y `3` están asignados a los símbolos `a` y `b` . El argumento original, el vector completo `[2 3]` , también se asigna al símbolo `xs` .

Lea Destrucción de clojure en línea: https://riptutorial.com/es/clojure/topic/1786/destruccion-de-clojure

Capítulo 12: Emacs CIDER

Introducción

CIDER es el acrónimo de **C** lojure (script) **I** nteractive **D** evelopment **E** am Environment que **R** ocks. Es una extensión para emacs. CIDER tiene como objetivo proporcionar un entorno de desarrollo interactivo para el programador. CIDER está construido sobre nREPL, un servidor REPL en red y SLIME sirvió como la principal inspiración para CIDER.

Examples

Evaluación de la función

La función CIDER `cider-eval-last-sexp` puede usarse para ejecutar el código mientras se edita el código dentro del búfer. Esta función está vinculada por defecto a `Cx Ce` o `Cx Ce` .

El manual de la SIDRA dice que `Cx Ce` o `Cc Ce` :

> Evalúe el formulario que precede al punto y visualice el resultado en el área de eco y / o en una superposición de búfer.

Por ejemplo:

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

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

Si ejecuta `Cx Ce` o `Cc Ce` mientras su cursor está justo delante de la paren final de la llamada de la función `say-hello` , se emitirá la cadena `Hello, my name is Bob` .

Impresión bonita

La función CIDER `cider-insert-last-sexp-in-repl` puede usarse para ejecutar el código mientras se edita el código dentro del búfer y se imprime la salida en un búfer diferente. Esta función está vinculada por defecto a `Cc Cp` .

El manual de la SIDRA dice que `Cc Cp` hará.

> Evalúe el formulario que precede al punto e imprima el resultado en un búfer emergente.

Por ejemplo

```
(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)
```

Si ejecuta `Cc Cp` mientras el cursor está justo delante del paréntesis final de la llamada a la función `get-database-config` , se mostrará el mapa impreso en un nuevo búfer emergente.

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

Lea Emacs CIDER en línea: https://riptutorial.com/es/clojure/topic/8847/emacs-cider

Capítulo 13: Enhebrar macros

Introducción

También conocidas como macros de flecha, las macros de subprocesamiento convierten llamadas de función anidadas en un flujo lineal de llamadas de función.

Examples

Último hilo (- >>)

Esta macro da la salida de una línea dada como el último argumento de la siguiente llamada de función de línea. Por ejemplo

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

es igual que

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

Primero el hilo (->)

Esta macro da la salida de una línea dada como el primer argumento de la siguiente llamada de función de línea. Por ejemplo

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

No puedo entender nada, ¿verdad? Vamos a intentarlo de nuevo, con ->

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

Hilo como (como->)

Esta es una alternativa más flexible para subprocesar primero o subproceso último. Se puede insertar en cualquier parte de la lista de parámetros de la función.

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

Lea Enhebrar macros en línea: https://riptutorial.com/es/clojure/topic/9582/enhebrar-macros

Capítulo 14: Funciones

Examples

Definiendo funciones

Las funciones se definen con cinco componentes:

El encabezado, que incluye la palabra clave `defn` , el nombre de la función.

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

Una Docstring opcional que explica y documenta lo que hace la función.

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

Parámetros listados entre paréntesis.

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

El cuerpo, que describe los procedimientos que realiza la función.

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

Llamándolo:

```
=> (welcome "World")

"Hello, World!"
```

Parámetros y Arity

Las funciones de Clojure se pueden definir con cero o más parámetros.

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

```
    []
    "Hello!")

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

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

Aridad

El número de argumentos que toma una función. Las funciones admiten la *sobrecarga de arity* , lo que significa que las funciones en Clojure permiten más de un "conjunto" de argumentos.

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

Las aridades no tienen que hacer el mismo trabajo, cada aridad puede hacer algo sin relación:

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

Definiendo funciones variables

Se puede definir una función de Clojure para tomar un número arbitrario de argumentos, usando el símbolo **y** en su lista de argumentos. Todos los argumentos restantes se recogen como una secuencia.

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

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

Vocación:

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

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

Definiendo funciones anónimas.

Hay dos formas de definir una función anónima: la sintaxis completa y una taquigrafía.

Sintaxis completa de funciones anónimas

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

Esta expresión se evalúa como una función. Cualquier sintaxis que pueda usar con una función definida con `defn` (`&` , desestructuración de argumentos, etc.), también puede hacer con la forma `fn` . `defn` es en realidad una macro que solo hace `(def (fn ...))` .

Sintaxis de funciones anónimas

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

Esta es la notación taquigráfica. Usando la notación abreviada, no tiene que nombrar argumentos explícitamente; se les asignarán los nombres `%1` , `%2` , `%3` y así sucesivamente de acuerdo con el orden en que se pasaron. Si la función solo tiene un argumento, su argumento se llama solo `%` .

Cuándo usar cada

La notación abreviada tiene algunas limitaciones. No puede desestructurar un argumento y no puede anidar funciones anónimas abreviadas. El siguiente código arroja un error:

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

Sintaxis soportada

Puede utilizar varargs con funciones anónimas de la taquigrafía. Esto es completamente legal:

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

Toma un número variable de argumentos y devuelve verdadero si cada uno de ellos es par:

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

A pesar de la aparente contradicción, es posible escribir una función anónima con nombre incluyendo un nombre, como en el siguiente ejemplo. Esto es especialmente útil si la función necesita llamarse a sí misma pero también en los seguimientos de pila.

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

Lea Funciones en línea: https://riptutorial.com/es/clojure/topic/3078/funciones

Capítulo 15: Interoperabilidad de Java

Sintaxis

- . te permite acceder a métodos de instancia
- .- permite acceder a campos de instancia
- .. macro expandiéndose a múltiples invocaciones anidadas de .

Observaciones

Como lenguaje alojado, Clojure proporciona un excelente soporte de interoperabilidad con Java. El código Clojure también se puede llamar directamente desde Java.

Examples

Llamando a un método de instancia en un objeto Java

Puede llamar a un método de instancia utilizando el . forma especial:

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

Puedes llamar a métodos de instancia con argumentos como este:

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

Hacer referencia a un campo de instancia en un objeto Java

Puede llamar a un campo de instancia utilizando la .- sintaxis:

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

Creando un nuevo objeto Java

Puede crear una instancia de objetos de una de estas dos maneras:

```
(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]"]
```

Llamando a un método estático

Puedes llamar métodos estáticos como este:

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

O pasar en argumentos, como este:

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

Llamando a una función de Clojure desde Java

Puede llamar a una función Clojure desde el código Java buscando la función e invocándola:

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

Esto busca la función `*` desde el espacio de nombres `clojure.core` y la invoca con los argumentos 2 y 2.

Lea Interoperabilidad de Java en línea:
https://riptutorial.com/es/clojure/topic/4036/interoperabilidad-de-java

Capítulo 16: La verdad

Examples

La verdad

En Clojure todo lo que no es `nil` o `false` se considera lógico verdadero.

Ejemplos:

```
(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)
```

Booleanos

Cualquier valor en Clojure se considera veraz a menos que sea `false` o `nil` . Puedes encontrar la veracidad de un valor con `(boolean value)` . Puede encontrar la veracidad de una lista de valores usando `(or)` , que devuelve `true` si algún argumento es verdadero, o `(and)` que devuelve `true` si todos los argumentos son verdaderos.

```
=> (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
```

Lea La verdad en línea: https://riptutorial.com/es/clojure/topic/4116/la-verdad

Capítulo 17: Macros

Sintaxis

- El símbolo `'` usado en el ejemplo de macroexpand es solo azúcar sintáctica para el operador de `quote` . Podría haber escrito `(macroexpand (quote (infix 1 + 2)))` lugar.

Observaciones

Las macros son solo funciones que se ejecutan en tiempo de compilación, es decir, durante el paso de `eval` en un ciclo de lectura-eval-print .

Las macros del lector son otra forma de macro que se expande en el momento de la lectura, en lugar del tiempo de compilación.

Mejores prácticas a la hora de definir macro.

- alfa-cambio de nombre, ya que la macro es expandir enlace podría surgir un conflicto. El conflicto de enlace no es muy intuitivo de resolver cuando se utiliza la macro. Esta es la razón por la cual, cuando una macro agrega un enlace al alcance, es obligatorio usar el `#` al final de cada símbolo.

Examples

Macro Infix simple

Clojure usa la notación de prefijo, es decir: el operador aparece antes que sus operandos.

Por ejemplo, una suma simple de dos números sería:

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

Las macros te permiten manipular el lenguaje de Clojure hasta cierto punto. Por ejemplo, podría implementar una macro que le permita escribir código en notación de infijo (por ejemplo, `1 + 2`):

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

Vamos a desglosar lo que hace el código anterior:

- `defmacro` es una *forma especial* que se utiliza para definir una macro.
- `infix` es el nombre de la macro que estamos definiendo.
- `[first-operand operator second-operand]` son los parámetros que esta macro espera recibir cuando se llama.

- `(list operator first-operand second-operand)` es el cuerpo de nuestra macro. Simplemente crea una `list` con los valores de los parámetros proporcionados a la macro `infix` y los devuelve.

`defmacro` es una *forma especial* porque se comporta de manera un poco diferente en comparación con otras construcciones de Clojure: sus parámetros no se evalúan de inmediato (cuando llamamos a la macro). Esto es lo que nos permite escribir algo como:

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

La macro `infix` expandirá los argumentos `1 + 2` a `(+ 1 2)` , que es una forma de Clojure válida que puede ser evaluada.

Si desea ver lo que genera la macro `infix` , puede usar el operador de `macroexpand` :

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

`macroexpand` , como lo `macroexpand` su nombre, expandirá la macro (en este caso, usará la macro `infix` para transformar `1 + 2` en `(+ 1 2)`) pero no permitirá que el resultado de la expansión de macro sea evaluado por Intérprete de clojure

Sintaxis citando y sin citar

Ejemplo de la biblioteca estándar (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#))))
```

- `` ` `` llamado syntax-quote es como `(quote)` , pero recursivo: hace `(let …)` , `(if …)` , etc. no se evalúa durante la expansión de macros sino que se genera como está
- `~` aka unquote cancela syntax-quote para una sola forma dentro de la forma sintaxis entre comillas. Entonces, el valor de `x` se genera al expandir la macro (en lugar de mostrar el símbolo `x`)
- `~@` aka unquote-splicing es como unquote pero toma el argumento de la lista y lo expande, cada elemento de la lista se forma de forma separada
- `#` agrega una identificación única a los símbolos para evitar conflictos de nombre. Agrega la misma ID para el mismo símbolo dentro de la expresión entre comillas de sintaxis, por lo que `and#` dentro de `let` y `and#` inside `if` obtendremos el mismo nombre

Lea Macros en línea: https://riptutorial.com/es/clojure/topic/2322/macros

Capítulo 18: Operaciones de archivo

Examples

Visión general

Lea un archivo de una vez (no se recomienda para archivos grandes):

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

Escribir datos en un archivo a la vez:

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

Lee un archivo línea por línea:

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

Escribe un archivo línea por línea:

```
(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)
```

Escribir en un archivo, reemplazando el contenido existente:

```
(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)
```

Notas:

- Puedes especificar URLs así como archivos
- Las opciones para `(slurp)` y `(spit)` se pasan a `clojure.java.io/reader` y `/writer` , respectivamente.

Lea Operaciones de archivo en línea: https://riptutorial.com/es/clojure/topic/3922/operaciones-de-archivo

Capítulo 19: prueba de clojure

Examples

es

El `is` macro es el núcleo de la `clojure.test` biblioteca. Devuelve el valor de su expresión corporal, imprimiendo un mensaje de error si la expresión devuelve un valor falsey.

```
(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
```

Agrupando pruebas relacionadas con la macro de prueba

Puede agrupar aserciones relacionadas en las pruebas unitarias más `deftest` dentro de un contexto usando la macro de `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)))))
```

Esto ayudará a aclarar la salida de prueba cuando se ejecuta. Tenga en cuenta que las `testing` deben ocurrir dentro de un `deftest` .

Definiendo una prueba con más habilidad.

`deftest` es una macro para definir una prueba de unidad, similar a las pruebas de unidad en otros idiomas.

Puede crear una prueba de la siguiente manera:

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

Aquí estamos definiendo una prueba llamada `add-nums` , que prueba la función `+` . La prueba tiene dos afirmaciones.

A continuación, puede ejecutar la prueba como esta en su espacio de nombres actual:

```
(run-tests)
```

O simplemente puede ejecutar las pruebas para el espacio de nombres en el que se encuentra la prueba:

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

son

La macro `are` también es parte de la biblioteca `clojure.test` . Te permite hacer múltiples aserciones contra una plantilla.

Por ejemplo:

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

Aquí, `(= xy)` actúa como una plantilla que tiene cada argumento y crea una `is` la afirmación de ella.

Esto expande a múltiples `is` afirmaciones:

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

Envuelva cada prueba o todas las pruebas con accesorios de uso

`use-fixtures` permite envolver cada `deftest` en el espacio de nombres con el código que se ejecuta antes y después de la prueba. Puede ser utilizado para accesorios o stubbing.

Los accesorios son solo funciones que toman la función de prueba y la ejecutan con otros pasos necesarios (antes / después, ajuste).

```
(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)
```

Cuando se usa con `:once` , envuelve toda la ejecución de pruebas en el espacio de nombres actual con la función

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

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

Ejecutando pruebas con Leiningen.

Si está utilizando Leiningen y sus pruebas están ubicadas en el directorio de pruebas en la raíz de su proyecto, entonces puede ejecutar sus pruebas utilizando la `lein test`

Lea prueba de clojure en línea: https://riptutorial.com/es/clojure/topic/1901/prueba-de-clojure

Capítulo 20: Realización de operaciones matemáticas simples

Introducción

Así es como agregaría algunos números en la sintaxis de Clojure. Dado que el método aparece como el primer argumento en la lista, estamos evaluando el método + (o adición) en el resto de los argumentos en la lista.

Observaciones

Realizar operaciones matemáticas es la base para manipular datos y trabajar con listas. Por lo tanto, entender cómo funciona es clave para progresar en la comprensión de Clojure.

Examples

Ejemplos de matematicas

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

;; returns 300
(+ 50 210 40)

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

Lea Realización de operaciones matemáticas simples en línea:
https://riptutorial.com/es/clojure/topic/8901/realizacion-de-operaciones-matematicas-simples

Capítulo 21: tiempo de clj

Introducción

Este documento trata sobre cómo manipular la fecha y la hora en clojure.

Para usar esto en su aplicación, vaya a su archivo project.clj e incluya [clj-time "<version_number>"] en su: sección de dependencias.

Examples

Creando un tiempo de Joda

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

Te da un horario de Joda del 20 de enero de 2017 a las 00:00:00.

Horas, minutos y segundos también se pueden especificar como

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

Obtención Día Mes Año Hora Minuto Segundo desde su fecha y hora

```
(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
```

Comparando dos fecha y hora

```
(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
```

Comprobando si un tiempo está dentro de un intervalo de tiempo

Esta función indica si un tiempo dado se encuentra dentro de un intervalo de tiempo dado.

```
(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 función de intervalo que se usa es **exclusiva** , lo que significa que no incluye el segundo argumento de la función dentro del intervalo. Como ejemplo:

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

Agregando joda fecha-hora de otros tipos de tiempo

La biblioteca clj-time.coerce puede ayudar a convertir otros formatos de fecha y hora al formato de hora joda (clj-time.core / date-time). Los otros formatos incluyen formato **largo de Java** , **cadena** , **fecha** , **fecha de SQL** .

Para convertir el tiempo de otros formatos de tiempo, incluya la biblioteca y use la función from, por ejemplo

```
(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>
```

Añadiendo fecha-hora a otras fechas-hora

cljs-time nos da la opción de agregar / restar fechas y horas a otras fechas y fechas. La fecha en que se agregaron las restas debe ser en forma de días, meses, años, horas, etc.

```
(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>
```

Lea tiempo de clj en línea: https://riptutorial.com/es/clojure/topic/9127/tiempo-de-clj

Capítulo 22: Transductores

Introducción

Los transductores son componentes componibles para procesar datos independientemente del contexto. Por lo tanto, se pueden utilizar para procesar colecciones, flujos, canales, etc. sin conocimiento de sus fuentes de entrada o receptores de salida.

La biblioteca central de Clojure se amplió en 1.7 para que la secuencia funcione como mapa, filtro, toma, etc. devuelva un transductor cuando se le llame sin una secuencia. Debido a que los transductores son funciones con contratos específicos, se pueden componer utilizando la función de `comp` normal.

Observaciones

Los transductores permiten controlar la pereza a medida que se consumen. Por ejemplo `into` está ansioso como sería de esperar, pero `sequence` perezosamente consumirá la secuencia a través del transductor. Sin embargo, la garantía de pereza es diferente. Se consumirá suficiente de la fuente para producir un elemento inicialmente:

```
(take 0 (sequence (map #(do (prn '-> %) %)) (range 5)))
;; -> 0
;; => ()
```

O decide si la lista está vacía:

```
(take 0 (sequence (comp (map #(do (prn '-> %) %)) (remove number?)) (range 5)))
;; -> 0
;; -> 1
;; -> 2
;; -> 3
;; -> 4
;; => ()
```

Que difiere del comportamiento habitual de la secuencia perezosa:

```
(take 0 (map #(do (prn '-> %) %) (range 5)))
;; => ()
```

Examples

Pequeño transductor aplicado a un vector.

```
(let [xf (comp
           (map inc)
           (filter even?))]
```

```
  (transduce xf + [1 2 3 4 5 6 7 8 9 10]))
;; => 30
```

Este ejemplo crea un transductor asignado a la `xf` local y utiliza `transduce` para aplicarlo a algunos datos. El transductor agrega uno a cada una de sus entradas y solo devuelve los números pares.

`transduce` es como `reduce` , y colapsa la colección de entrada a un solo valor usando la función `+` provista.

Esto se lee como la macro del último hilo, pero separa los datos de entrada de los cálculos.

```
(->> [1 2 3 4 5 6 7 8 9 10]
     (map inc)
     (filter even?)
     (reduce +))
;; => 30
```

Aplicando transductores

```
(def xf (filter keyword?))
```

Aplicar a una colección, devolviendo una secuencia:

```
(sequence xf [:a 1 2 :b :c]) ;; => (:a :b :c)
```

Aplicar a una colección, reduciendo la colección resultante con otra función:

```
(transduce xf str [:a 1 2 :b :c]) ;; => ":a:b:c"
```

Aplicar a una colección y `conj` el resultado en otra colección:

```
(into [] xf [:a 1 2 :b :c]) ;; => [:a :b :c]
```

Cree un canal asíncrono central que use un transductor para filtrar mensajes:

```
(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
```

Creando / Usando Transductores

Por lo tanto, las funciones más utilizadas en el mapa y el filtro de Clojure se han modificado para devolver los transductores (transformaciones algorítmicas compuestas), si no se llaman con una colección. Eso significa:

`(map inc)` devuelve un transductor y también lo hace `(filter odd?)`

La ventaja: las funciones se pueden componer en una sola función por comp, lo que significa atravesar la colección solo una vez. Ahorra tiempo de ejecución en más del 50% en algunos escenarios.

Definición:

```
(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]
```

Lea Transductores en línea: https://riptutorial.com/es/clojure/topic/10814/transductores

Capítulo 23: Vars

Sintaxis

- (valor del símbolo def)
- (def símbolo "docstring" valor)
- (declarar símbolo_0 símbolo_1 símbolo_2 ...)

Observaciones

Esto no debe confundirse con `(defn)` , que se utiliza para definir funciones.

Examples

Tipos de variables

Hay diferentes tipos de variables en Clojure:

- números

  Tipos de números:

  - enteros
  - largos (números mayores que `2^31 - 1`)
  - flotadores (decimales)

- instrumentos de cuerda
- colecciones

  Tipos de colecciones:

  - mapas
  - secuencias
  - vectores

- funciones

Lea Vars en línea: https://riptutorial.com/es/clojure/topic/4449/vars

Creditos

S. No	Capítulos	Contributors
1	Empezando 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	Analizando troncos con clojure	user2611740
3	Átomo	Qwerp-Derp, systemfreund
4	clojure.core	Akanksha, Mrinal Saurabh, Surbhi Garg
5	clojure.spec	Adam Lee, Alex Miller, kolen, leeor, nXqd
6	Coincidencia de patrones con core.match	Kiran
7	Colecciones y secuencias	Alex Miller, Kenogu Labz, nXqd, Sam Estep
8	Comenzando con el desarrollo web	Emin Tham, kolen, r00tt
9	Configurando su entorno de desarrollo	Adeel Ansari, agent_orange, amalloy, g1eny0ung, Geoff, Kiran, kolen, MuSaiXi, Piyush, Qwerp-Derp, spinningarrow, stardiviner, superkonduktr, swlkr
10	core.async	Valentin Waeselynck
11	Destrucción de clojure	camdez, kaffein, kolen, leeor, Michał Marczyk, MuSaiXi, r00tt, RedBlueThing, ryo, tsleyson, Zaz
12	Emacs CIDER	avichalp
13	Enhebrar macros	Kusum Ijari, Mrinal Saurabh
14	Funciones	alejosocorro, fokz, Qwerp-Derp, tar, tsleyson
15	Interoperabilidad de Java	leeor
16	La verdad	Alan Thompson, Michiel Borkent, Zaz

17	Macros	Alex Miller, kolen, mathk, Sam Estep, snowcrshd
18	Operaciones de archivo	Zaz
19	prueba de clojure	jisaw, kolen, leeor, porglezomp, Sam Estep
20	Realización de operaciones matemáticas simples	Jim
21	tiempo de clj	Rishu Saniya, Akanksha, Mrinal Saurabh, Vishakha Silky
22	Transductores	l0st3d, Mrinal Saurabh
23	Vars	Aryaman Arora, Qwerp-Derp, Stephen Leppik, Zaz