eBook Gratuit

APPRENEZ clojure

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#clojure

Table des matières

À propos

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

Chapitre 1: Commencer avec clojure

Remarques

Clojure **est un langage de programmation généraliste à typage dynamique avec syntaxe Lisp.**

Ses fonctionnalités prennent en charge le style fonctionnel de programmation avec des fonctions de première classe et des valeurs immuables par défaut. L'utilisation de variables réaffectables n'est pas aussi facile dans Clojure que dans de nombreux langages traditionnels, car les variables doivent être créées et mises à jour comme des objets conteneurs. Cela encourage l'utilisation de valeurs pures qui resteront telles qu'elles étaient au moment où elles ont été vues pour la dernière fois. Cela rend généralement le code beaucoup plus prévisible, testable et compatible avec la concurrence. Cela fonctionne également pour les collections, car les structures de données intégrées de Clojure sont persistantes.

Pour des performances optimales, Clojure prend en charge les indices de type pour éliminer les réflexions inutiles lorsque cela est possible. En outre, des groupes de modifications de collections persistantes peuvent être apportées à *des* versions *transitoires* , ce qui réduit la quantité d'objets impliqués. Cela n'est pas nécessaire la plupart du temps, car les collections persistantes sont rapides à copier car elles partagent la plupart de leurs données. Leurs garanties de performance ne sont pas loin de leurs homologues mutables.

Clojure possède également:

- mémoire transactionnelle logicielle (STM)
- plusieurs primitives de concurrence n'impliquant pas de verrouillage manuel (atome, agent)
- transformateurs de séquences composables (transducteurs),
- installations de manipulation d'arbres fonctionnelles (fermetures éclair)

En raison de sa syntaxe simple et de son extensibilité élevée (via des macros, l'implémentation d'interfaces de base et la réflexion), certaines fonctionnalités de langage couramment rencontrées peuvent être ajoutées à Clojure avec des bibliothèques. Par exemple, `core.typed` apporte un vérificateur de type statique, `core.async` apporte des mécanismes de concurrence simples basés sur les `core.logic` , `core.logic` apporte la programmation logique.

Conçu comme un langage hébergé, il peut interagir avec la plate-forme sur laquelle il s'exécute. Bien que la cible principale soit JVM et l'écosystème entier derrière Java, d'autres implémentations peuvent également s'exécuter dans d'autres environnements, tels que ClojureCLR s'exécutant sur Common Language Runtime ou ClojureScript s'exécutant sur des

environnements d'exécution JavaScript (y compris les navigateurs Web). Bien que certaines implémentations puissent ne pas avoir certaines fonctionnalités de la version JVM, elles sont toujours considérées comme une famille de langues.

Versions

Version	Changer de journal	Date de sortie
1.8	Dernier journal des modifications	2016-01-19
1,7	Changer le journal 1.7	2015-06-30
1.6	Changer le journal 1.6	2014-03-25
1.5.1	Changer le journal 1.5.1	2013-03-10
1.4	Changer le journal 1.4	2012-04-15
1.3	Changer de journal 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

Installation et configuration

Option 1: Leiningen

Nécessite JDK 6 ou plus récent.

Le moyen le plus simple de commencer avec Clojure est de télécharger et d'installer Leiningen, l'outil standard de facto pour gérer les projets Clojure, puis d'exécuter `lein repl` pour ouvrir une REPL .

Linux

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

OS X

Suivez les étapes ci-dessus pour Linux ou installez-les avec les gestionnaires de paquets macOS.

Installer avec Homebrew

```
brew install leiningen
```

Installer avec MacPorts

Installez d'abord Clojure

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

Installez `Leiningen` , un outil de construction pour Clojure

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

```
lein self-install
```

les fenêtres

Voir la documentation officielle .

Option 2: Distribution officielle

Nécessite JRE 6 ou plus récent.

Les versions de Clojure sont publiées sous forme de simples fichiers JAR à exécuter sur la JVM. C'est ce qui se passe généralement dans les outils de construction Clojure.

1. Allez sur http://clojure.org et téléchargez la dernière archive de Clojure
2. Extrayez le fichier ZIP téléchargé dans un répertoire de votre choix
3. Exécutez `java -cp clojure-1.8.0.jar clojure.main` dans ce répertoire

   Vous devrez peut-être remplacer le `clojure-1.8.0.jar` dans cette commande par le nom du fichier JAR que vous avez réellement téléchargé.

   Pour une meilleure expérience REPL en ligne de commande (par exemple, passer en `rlwrap` vos commandes précédentes), vous pouvez installer `rlwrap` : `rlwrap java -cp clojure-1.8.0.jar clojure.main`

Option 3: démarrage

Nécessite JDK 7 ou plus récent.

Boot est un outil de construction Clojure polyvalent. Comprendre cela nécessite une certaine connaissance de Clojure, donc ce n'est peut-être pas la meilleure option pour les débutants. Consultez le site Web (cliquez sur *Démarrer*) pour obtenir des instructions d'installation.

Une fois installé et dans votre `PATH` , vous pouvez exécuter `boot repl` n'importe où pour démarrer une REPL Clojure.

"Bonjour le monde!" dans le REPL

La communauté Clojure accorde une grande importance au développement interactif, de sorte qu'une interaction avec Clojure se produit dans une REPL (read-eval-print-loop) . Lorsque vous entrez une expression, Clojure la **lit** , l' **évalue** et **imprime** le résultat de l'évaluation, le tout dans une **boucle** .

Vous devriez être en mesure de lancer une REPL Clojure maintenant. Si vous ne savez pas comment, suivez la section **Installation et configuration** de cette rubrique. Une fois que vous l'avez lancé, tapez ce qui suit:

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

Puis appuyez sur `Entrée` . Cela devrait imprimer `Hello, world!` , suivi de la valeur de retour de cette expression, `nil` .

Si vous souhaitez lancer des actions instantanément, essayez REPL en ligne. Par exemple, http://www.tryclj.com/ .

Créer une nouvelle application

Après avoir suivi les instructions ci-dessus et installé Leiningen, démarrez un nouveau projet en exécutant:

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

Cela va configurer un projet Clojure avec le modèle Leiningen par défaut dans le dossier `<project-name>` . Il existe plusieurs modèles pour Leiningen, qui affectent la structure du projet. Le plus commun est le modèle "app" utilisé, qui ajoute une fonction principale et prépare le projet à être compressé dans un fichier jar (dont la fonction principale est le point d'entrée de l'application). Cela peut être réalisé en exécutant:

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

En supposant que vous avez utilisé le modèle d'application pour créer une nouvelle application, vous pouvez tester que tout a été correctement configuré, en entrant le répertoire créé et en exécutant l'application en utilisant:

```
lein run
```

Si vous voyez `Hello, World!` sur votre console, vous êtes prêt et prêt à créer votre application.

Vous pouvez emballer cette application simple dans deux fichiers jar avec la commande suivante:

```
lein uberjar
```

"Bonjour le monde!" en utilisant Boot

Remarque: vous devez installer Boot avant d'essayer cet exemple. Voir la section **Installation et configuration** si vous ne l'avez pas encore installée.

Le démarrage permet de créer des fichiers Clojure exécutables en utilisant la ligne **shebang** (#!). Placez le texte suivant dans un fichier de votre choix (cet exemple suppose qu'il se trouve dans le "répertoire de travail actuel" et qu'il s'appelle `hello.clj`).

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

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

Puis marquez-le comme exécutable (le cas échéant, généralement en exécutant `chmod +x hello.clj`).
... et lancez-le (`./hello.clj`).

Le programme devrait sortir "Hello, world!" et fini.

Créer une nouvelle application (avec boot)

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

Cette commande indiquera à boot de récupérer la tâche `boot-new` depuis https://github.com/seancorfield/boot-new et d'exécuter la tâche avec le template de l' `app` (voir le lien pour les autres templates). La tâche créera un nouveau répertoire appelé `<appname>` avec une structure d'application Clojure typique. Voir le README généré pour plus d'informations.

Pour exécuter l'application: `boot run` . Les autres commandes sont spécifiées dans `build.boot` et décrites dans le fichier README.

Lire Commencer avec clojure en ligne: https://riptutorial.com/fr/clojure/topic/827/commencer-avec-clojure

Chapitre 2: Analyse des journaux avec clojure

Examples

Analyser une ligne de journal avec record et regex

```
(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)))
```

Définir une ligne d'échantillon:

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

Essaye-le :

```
(parser sample)
```

Résultat :

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

Lire Analyse des journaux avec clojure en ligne:
https://riptutorial.com/fr/clojure/topic/9822/analyse-des-journaux-avec-clojure

Chapitre 3: Atome

Introduction

Un atome dans Clojure est une variable qui peut être modifiée tout au long de votre programme (espace de noms). Comme la plupart des types de données dans Clojure sont immuables (ou non modifiables) - vous ne pouvez pas modifier la valeur d'un nombre sans le redéfinir - les atomes sont essentiels dans la programmation de Clojure.

Examples

Définir un atome

Pour définir un atome, utilisez un `def` ordinaire, mais ajoutez une fonction `atom` avant, comme ceci:

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

Cela crée un `atom` de valeur `0` . Les atomes peuvent être de tout type:

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

Lire la valeur d'un atome

Pour lire la valeur d'un atome, il suffit de mettre le nom de l'atome avec un `@` avant:

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

Un plus grand exemple:

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

Mettre à jour la valeur d'un atome

Il y a deux commandes pour changer un atome, `swap!` et `reset!` . `swap!` est donné des commandes et modifie l'atome en fonction de son état actuel. `reset!` change complètement la valeur de l'atome, quelle que soit la valeur de l'atome d'origine:

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

Cet exemple génère les 10 premières puissances de `2` utilisant des atomes:

```
(def count (atom 0))

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

Lire Atome en ligne: https://riptutorial.com/fr/clojure/topic/7519/atome

Chapitre 4: clj-time

Introduction

Ce document traite de la manière de manipuler la date et l'heure en clair.

Pour l'utiliser dans votre application, allez dans votre fichier project.clj et incluez [clj-time "<numéro_version>"] dans votre section: dependencies.

Examples

Créer un temps Joda

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

Vous donne une heure Joda du 20 janvier 2017 à 00:00:00.

Les heures, minutes et secondes peuvent également être spécifiées comme

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

Obtenir le Jour Mois Année Heure Minute Seconde de votre date-heure

```
(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
```

Comparer deux date-heure

```
(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
```

Vérifier si une heure est comprise dans un intervalle de temps

Cette fonction indique si une heure donnée se situe dans un intervalle de temps donné.

```
(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 fonction d'intervalle utilisée est **exclusive** , ce qui signifie qu'elle n'inclut pas le second argument de la fonction dans l'intervalle. Par exemple:

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

Ajouter joda date-heure à d'autres types d'heures

La bibliothèque clj-time.coerce peut aider à convertir d'autres formats de date et heure au format d'heure joda (clj-time.core / date-time). Les autres formats incluent le format **long Java** , la **chaîne** , la **date** , la **date SQL** .

Pour convertir l'heure à partir d'autres formats d'heure, incluez la bibliothèque et utilisez la fonction from-, par exemple

```
(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>
```

Ajout de date et heure à d'autres dates

cljs-time nous donne l'option d'ajouter / soustraire des dates-heures à d'autres dates-heures. La date à laquelle les temps ajoutés doivent être sous forme de jours, mois, années, heures, 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>
```

Lire clj-time en ligne: https://riptutorial.com/fr/clojure/topic/9127/clj-time

Chapitre 5: clojure.core

Introduction

Ce document donne diverses fonctionnalités de base offertes par clojure. Il n'y a pas de dépendance explicite nécessaire pour cela et cela fait partie de org.clojure.

Examples

Définir des fonctions 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 - Mise à jour des valeurs de carte / vecteur en clojure

Appliquée sur une carte, retourne une nouvelle carte avec des paires de valeurs de clés nouvelles ou mises à jour.

Il peut être utilisé pour ajouter de nouvelles informations dans une carte existante.

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

Il remplace l'ancienne valeur d'information si la clé existante est fournie.

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

Il peut également être utilisé sur un vecteur pour remplacer la valeur à l'index spécifié.

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

Opérateurs de comparaison à Clojure

Les comparaisons sont des fonctions dans la vie. Ce que cela signifie en `(2>1)` est `(> 2 1)` en clair. Voici tous les opérateurs de comparaison en cloche.

1. Plus grand que

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

2. Moins que

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

3. Plus grand ou égal à

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

4. Inférieur ou égal à

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

5. Égal à

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

6. Pas égal à

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

Dissoc - dissocier une clé d'une carte clojure

Cela retourne une carte sans les paires clé-valeur pour les clés mentionnées dans l'argument de la fonction. Il peut être utilisé pour supprimer des informations de la carte existante.

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

Il peut également être utilisé pour dissocier plusieurs clés comme:

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

Lire clojure.core en ligne: https://riptutorial.com/fr/clojure/topic/9585/clojure-core

Chapitre 6: clojure.spec

Syntaxe

- :: est un raccourci un mot-clé qualifié par un espace de noms. Par exemple, si nous sommes dans l'espace de noms user: :: foo est un raccourci pour: user / foo
- #: ou # - syntaxe littérale de la carte pour qualifier les clés d'une carte par un espace de noms

Remarques

Clojure spec est une nouvelle bibliothèque de spécifications / contrats pour clojure disponible à partir de la version 1.9.

Les spécifications sont exploitées de diverses manières, notamment en étant incluses dans la documentation, la validation des données, la génération de données pour les tests et bien plus encore.

Examples

Utiliser un prédicat comme spécification

Toute fonction de prédicat peut être utilisée en tant que spécification. Voici un exemple simple:

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

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

le `valid?` function prendra une spec et une valeur et retournera true si la valeur est conforme à la spécification et false sinon.

Un autre prédicat intéressant est défini pour l'appartenance:

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

fdef: écrire une spécification pour une fonction

Disons que nous avons la fonction suivante:

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

Nous pouvons écrire une spécification pour cette fonction en définissant une spécification de

fonction du même nom:

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

`:args` prend une spécification de regex qui décrit la séquence des arguments par une étiquette de mot-clé correspondant au nom de l'argument et à une spécification correspondante. La raison pour laquelle la spécification requise par `:args` est une spécification de regex est de prendre en charge plusieurs arités pour une fonction. `:ret` spécifie une spécification pour la valeur de retour de la fonction.

`:fn` est une spécification qui contraint la relation entre les `:args` et `:ret` . Il est utilisé comme propriété lorsqu'il est exécuté via test.check. Il est appelé avec un seul argument: une carte avec deux clés:: `:args` (les arguments conformes à la fonction) et `:ret` (la valeur de retour de la fonction).

Enregistrement d'une spec

En plus des prédicats fonctionnant en tant que spécifications, vous pouvez enregistrer une spécification globalement en utilisant `clojure.spec/def` . `def` exige qu'une spécification enregistrée soit nommée par un mot clé qualifié par un espace de nommage:

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

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

Une fois enregistrée, une spécification peut être référencée globalement n'importe où dans un programme Clojure.

La syntaxe `::odd-nums` est un raccourci pour `:user/odd-nums` , en supposant que nous sommes dans l'espace de noms de l' `user` . `::` qualifiera le symbole qu'il précède avec le nom actuel.

Plutôt que de passer le prédicat, nous pouvons passer le nom de la spécification à `valid?` , et cela fonctionnera de la même manière.

clojure.spec / et & clojure.spec / ou

`clojure.spec/and` & `clojure.spec/or` peuvent être utilisés pour créer des spécifications plus complexes, en utilisant plusieurs spécifications ou prédicats:

```
(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` fonctionne de manière similaire, avec une différence significative. Lors de la définition d'une `or` une spécification, vous devez marquer chaque branche possible avec un mot-clé. Ceci est utilisé pour fournir des branches spécifiques qui échouent dans les messages d'erreur:

```
(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
```

Lors de la conformité d'une spécification à l'aide de `or` , la spécification applicable sera renvoyée, ce qui a rendu la valeur conforme:

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

Fiche technique

Vous pouvez spécifier un enregistrement comme suit:

```
(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
```

À un moment donné, une syntaxe de lecteur ou une prise en charge intégrée pour la qualification des clés d'enregistrement par l'espace de nom des enregistrements peut être introduite. Ce support existe déjà pour les cartes.

Spécifications de la carte

Vous pouvez spécifier une carte en spécifiant les clés qui doivent être présentes sur la carte:

```
(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` est un vecteur de clés devant être présent dans la carte. Vous pouvez spécifier des options supplémentaires telles que `:opt` , un vecteur de clés facultatif.

Les exemples jusqu'à présent exigent que les clés du nom soient qualifiées par des espaces de noms. Mais il est courant que les clés de carte ne soient pas qualifiées. Pour ce cas, `clojure.spec` fournit: req et: opt équivalents pour les clés non qualifiées `:req-un` et `:opt-un` . Voici le même exemple, avec des clés non qualifiées:

```
(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
```

Notez que les spécifications fournies dans le vecteur `:req-un` sont toujours qualifiées. clojure.spec, confirmera automatiquement les versions non qualifiées dans la carte lors de la conformité des valeurs.

La syntaxe littérale de la carte d'espace de noms vous permet de qualifier toutes les clés d'une carte par un seul espace de noms succinct. Par exemple:

```
(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
```

Notez la syntaxe spéciale `#:` reader. Nous suivons cela avec l'espace de noms que nous souhaitons qualifier par toutes les clés de la carte. Celles-ci seront ensuite vérifiées par rapport aux spécifications correspondant à l'espace de noms fourni.

Collections

Vous pouvez spécifier des collections de différentes manières. coll-of vous permet de spécifier des collections et de fournir des contraintes supplémentaires. Voici un exemple 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
```

Les options de contrainte suivent les spécifications / prédicats principaux de la collection. Vous pouvez contraindre le type de collection avec `:kind` comme ceci:

```
(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
```

Ce qui précède est faux car la collection transmise n'est pas un vecteur.

```
(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
```

Ce qui précède est faux car tous les éléments de l'ensemble ne sont pas ints.

Vous pouvez également limiter la taille de la collection de plusieurs manières:

```
(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
```

Vous pouvez également imposer l'unicité des éléments de la collection avec `: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` s'assure que tous les éléments d'une séquence sont vérifiés. Pour les grandes collections, cela peut être très inefficace. `every` se comporte exactement comme `coll-of` , sauf qu'il échantillonne seulement un nombre relativement petit d'éléments de la séquence pour la conformité. Cela fonctionne bien pour les grandes collections. Voici un exemple:

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

`map-of` est similaire à `coll-of` , mais pour les cartes. Comme les cartes ont à la fois des clés et des valeurs, vous devez fournir une spécification pour la clé et une spécification pour la valeur:

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

Comme `coll-of` , la conformité de toutes les clés / valeurs `coll-of` `map-of` contrôles. Pour les grandes cartes, cela sera inefficace. A l'instar de `coll-of` , une `map-of` fournitures à `every-kv` pour échantillonner efficacement un nombre relativement petit de valeurs à partir d'une grande carte:

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

Séquences

spec peut décrire et être utilisé avec des séquences arbitraires. Il prend en charge cela via un certain nombre d'opérations de spécification regex.

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

`cat` nécessite des étiquettes pour chaque spécification utilisée pour décrire la séquence. cat décrit une séquence d'éléments et une spécification pour chacun.

`alt` est utilisé pour choisir parmi un certain nombre de spécifications possibles pour un élément donné dans une séquence. Par exemple:

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

`alt` exige également que chaque spécification soit étiquetée par un mot-clé.

Les séquences de regex peuvent être composées de manières très intéressantes et puissantes pour créer des spécifications de description de séquence arbitrairement complexes. Voici un exemple légèrement plus complexe:

```
(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
```

Ici `::complex-seq` validera une séquence d'une ou plusieurs paires d'éléments, la première étant un int et la seconde étant une carte de mot-clé à int.

Lire clojure.spec en ligne: https://riptutorial.com/fr/clojure/topic/2325/clojure-spec

Chapitre 7: clojure.test

Examples

est

Le `is` macro est au cœur de la `clojure.test` bibliothèque. Il renvoie la valeur de son expression de corps, imprimant un message d'erreur si l'expression renvoie une valeur 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
```

Regroupement des tests associés avec la macro de test

Vous pouvez regrouper des assertions associées dans des `deftest` unitaires de test dans un contexte à l'aide de la macro `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)))))
```

Cela aidera à clarifier la sortie du test lors de l'exécution. Notez que les `testing` doivent avoir lieu à l'intérieur d'un `deftest` .

Définir un test avec le deftest

`deftest` est une macro pour définir un test unitaire, similaire aux tests unitaires dans d'autres langues.

Vous pouvez créer un test comme suit:

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

Nous définissons ici un test appelé `add-nums` , qui teste la fonction `+` . Le test a deux assertions.

Vous pouvez ensuite exécuter le test comme ceci dans votre espace de noms actuel:

```
(run-tests)
```

Ou vous pouvez simplement exécuter les tests pour l'espace de noms dans lequel se trouve le test:

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

sont

Le `are` macro fait également partie de la `clojure.test` bibliothèque. Il vous permet de faire plusieurs assertions sur un modèle.

Par exemple:

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

Ici, `(= xy)` agit comme un modèle qui prend chaque argument et crée une `is` l' affirmation hors de celui - ci.

Cela élargit à de multiples `is` affirmations:

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

Enveloppez chaque test ou tous les tests avec des accessoires d'utilisation

`use-fixtures` permet d'emballer chaque `deftest` dans l'espace de nommage avec le code qui s'exécute avant et après le test. Il peut être utilisé pour des luminaires ou des bouts.

Les appareils ne sont que des fonctions qui prennent en charge la fonction de test et l’exécutent avec d’autres étapes nécessaires (avant / après, emballage).

```
(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)
```

Utilisé avec `:once` , il enveloppe une série complète de tests dans l'espace de noms actuel avec la fonction

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

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

Exécution de tests avec Leiningen

Si vous utilisez Leiningen et que vos tests se trouvent dans le répertoire de test de la racine de votre projet, vous pouvez exécuter vos tests en utilisant le `lein test`

Lire clojure.test en ligne: https://riptutorial.com/fr/clojure/topic/1901/clojure-test

Chapitre 8: Collections et séquences

Syntaxe

- `'() ⟶ ()`
- `'(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"}` (les entrées sont triées par clé lorsqu'elles sont utilisées comme une séquence)
- `#{} ⟶ #{}`
- `#{1 2 3 4 5} ⟶ #{4 3 2 5 1}` (non ordonné)
- `(hash-set 1 2 3 4 5) ⟶ #{2 5 4 1 3}` (non ordonné)
- `(sorted-set 2 5 4 3 1) ⟶ #{1 2 3 4 5}`

Examples

Collections

Toutes les collections Clojure intégrées sont immuables et hétérogènes, ont une syntaxe littérale et prennent en charge les fonctions `conj` , `count` et `seq` .

- `conj` renvoie une nouvelle collection équivalente à une collection existante avec un élément "ajouté", soit dans le temps "constant" ou logarithmique. Ce que cela signifie exactement dépend de la collection.
- `count` renvoie le nombre d'éléments d'une collection, en temps constant.
- `seq` renvoie `nil` pour une collection vide ou une séquence d'éléments pour une collection non vide, à temps constant.

Des listes

Une liste est indiquée par des parenthèses:

```
()
```

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

Une liste de Clojure est une liste à liens simples . `conj` "joint" un nouvel élément à la collection dans l'emplacement le plus efficace. Pour les listes, c'est au début:

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

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

Contrairement aux autres collections, les listes non vides sont évaluées comme des appels à des formulaires, macros ou fonctions spéciaux lorsqu'elles sont évaluées. Par conséquent, tant que `(:foo)` *est* la représentation littérale de la liste contenant `:foo` comme seul élément, l'évaluation de `(:foo)` dans une REPL provoquera une `IllegalArgumentException` car un mot clé ne peut pas être appelé comme fonction nullary .

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

Pour empêcher Clojure d'évaluer une liste non vide, vous pouvez la `quote` :

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

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

Malheureusement, les éléments ne sont pas évalués:

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

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

Pour cette raison, vous voudrez généralement utiliser `list` , une fonction variadic qui évalue tous ses arguments et utilise ces résultats pour construire une liste:

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

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

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

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

`count` renvoie le nombre d'éléments, en temps constant:

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

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

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

Vous pouvez tester si quelque chose est une liste en utilisant la `list?` prédicat:

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

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

(list? nil)
;;=> false

(list? 42)
;;=> false

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

Vous pouvez obtenir le premier élément d'une liste en utilisant `peek` :

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

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

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

Vous pouvez obtenir une nouvelle liste sans le premier élément en utilisant `pop` :

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

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

Notez que si vous essayez de faire `pop` une liste vide, vous obtiendrez une `IllegalStateException` :

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

Enfin, toutes les listes sont des séquences, vous pouvez donc tout faire avec une liste que vous pouvez faire avec toute autre séquence. En effet, à l'exception de la liste vide, l'appel de `seq` sur une liste renvoie exactement le même objet:

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

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

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

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

Séquences

Une séquence ressemble beaucoup à une liste: c'est un objet immuable qui peut vous donner son `first` élément ou le `rest` de ses éléments en temps constant. Vous pouvez également `cons` truct une nouvelle séquence d'une séquence existante et un élément de coller au début.

Vous pouvez tester si quelque chose est une séquence en utilisant le `seq?` prédicat:

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

(seq? 42)
;;=> false

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

Comme vous le savez déjà, les listes sont des séquences:

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

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

Tout ce que vous obtenez en appelant `seq` ou `rseq` ou des `keys` ou des `vals` sur une collection non vide est également une séquence:

```
(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
```

Rappelez-vous que toutes les listes sont des séquences, mais toutes les séquences ne sont pas des listes. Alors que les listes prennent en charge les fonctions de `peek` et de `pop` et `count` en temps constant, en général, une séquence n'a pas besoin de prendre en charge l'une de ces fonctions. Si vous essayez d'appeler `peek` ou `pop` sur une séquence qui ne prend pas également en charge l'interface de pile de Clojure, vous obtiendrez une `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 vous appelez `count` sur une séquence qui n'implémente pas le `count` en temps constant, vous n'obtiendrez pas d'erreur. Clojure traversera toute la séquence jusqu'à la fin, puis retournera le nombre d'éléments qu'il a parcourus. Cela signifie que, pour les séquences générales, le `count` est linéaire, pas constant, le temps. Vous pouvez tester si quelque chose prend en charge le `count` temps constant en utilisant le `counted?` prédicat:

```
(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
```

Comme mentionné ci-dessus, vous pouvez utiliser d' `first` pour obtenir le premier élément d'une séquence. Notez que le `first` appelera `seq` sur leur argument, donc il peut être utilisé sur tout "seqable", pas seulement sur les séquences réelles:

```
(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
```

Comme mentionné ci-dessus, vous pouvez utiliser `rest` pour obtenir une séquence contenant tous

les éléments sauf le premier élément d'une séquence existante. Comme `first` , il appelle `seq` sur son argument. Cependant, il n'appelle *pas* `seq` sur son résultat! Cela signifie que si vous appelez `rest` sur une séquence contenant moins de deux éléments, vous récupérerez `()` au lieu de `nil` :

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

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

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

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

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

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

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

Si vous voulez récupérer `nil` quand il n'y a plus d'éléments dans une séquence, vous pouvez utiliser `next` au lieu de `rest` :

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

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

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

Vous pouvez utiliser la fonction `cons` pour créer une nouvelle séquence qui renverra son premier argument pour le `first` et son deuxième argument pour le `rest` :

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

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

Clojure fournit une grande bibliothèque de séquences avec de nombreuses fonctions pour gérer les séquences. L'important à propos de cette bibliothèque est qu'elle fonctionne avec tout ce qui peut être "séqé", pas seulement les listes. C'est pourquoi le concept d'une séquence est si utile; cela signifie qu'une seule fonction, comme `reduce` , fonctionne parfaitement sur n'importe quelle collection:

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

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

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

L'autre raison pour laquelle les séquences sont utiles est que, puisqu'elles ne requièrent aucune implémentation particulière du `first` et du `rest` , elles permettent des séquences paresseuses dont les éléments ne sont réalisés que lorsque cela est nécessaire.

Étant donné une expression qui créerait une séquence, vous pouvez envelopper cette expression dans la macro `lazy-seq` pour obtenir un objet qui agit comme une séquence, mais n'évaluera réellement cette expression que lorsque la fonction `seq` le demandera, à quel point il va mettre en cache le résultat de l'expression et en avant `first` et `rest` les appels vers le résultat mis en cache.

Pour les séquences finies, une séquence paresseuse agit généralement comme une séquence équivalente:

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

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

Cependant, la différence devient évidente pour des séquences infinies:

```
(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)
```

Vecteurs

Un vecteur est indiqué par des crochets:

```
[]
;;=> []

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

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

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

En plus de la syntaxe littérale, vous pouvez également utiliser la fonction `vector` pour construire un vecteur:

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

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

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

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

Vous pouvez tester si quelque chose est un vecteur en utilisant le `vector?` prédicat:

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

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

(vector? nil)
;;=> false

(vector? 42)
;;=> false

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

`conj` ajoute des éléments à la fin d'un vecteur:

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

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

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

`count` renvoie le nombre d'éléments, en temps constant:

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

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

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

Vous pouvez obtenir le dernier élément d'un vecteur en utilisant `peek` :

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

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

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

Vous pouvez obtenir un nouveau vecteur sans le dernier élément en utilisant `pop` :

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

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

Notez que si vous essayez de faire apparaître un vecteur vide, vous obtiendrez une `IllegalStateException` :

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

Contrairement aux listes, les vecteurs sont indexés. Vous pouvez obtenir un élément d'un vecteur par index dans le temps "constant" en utilisant `get` :

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

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

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

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

De plus, les vecteurs eux-mêmes sont des fonctions qui prennent un index et renvoient l'élément à cet index:

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

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

Cependant, si vous appelez un vecteur avec un index non valide, vous obtiendrez une `IndexOutOfBoundsException` au lieu de `nil` :

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

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

Vous pouvez obtenir un nouveau vecteur avec une valeur différente à un index particulier en utilisant `assoc` :

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

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

Si vous passez un index égal au `count` de vecteurs, Clojure ajoutera l'élément comme si vous aviez utilisé `conj` . Toutefois, si vous transmettez un index négatif ou supérieur au `count` , vous obtenez une `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:
```

Vous pouvez obtenir une séquence des éléments dans un vecteur en utilisant `seq` :

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

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

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

Comme les vecteurs sont indexés, vous pouvez également obtenir une séquence inversée des éléments d'un vecteur en utilisant `rseq` :

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

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

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

Notez que, bien que toutes les listes soient des séquences et que les séquences soient affichées de la même manière que les listes, toutes les séquences ne sont pas des listes!

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

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

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

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

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

Ensembles

Comme les cartes, les ensembles sont associatifs et non ordonnés. Contrairement aux cartes, qui contiennent des correspondances entre les clés et les valeurs, elles définissent essentiellement la correspondance entre les clés et elles-mêmes.

Un ensemble est désigné par des accolades précédé d'un octothorpe:

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

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

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

Comme pour les cartes, l'ordre dans lequel les éléments apparaissent dans un ensemble littéral n'a pas d'importance:

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

Vous pouvez tester si quelque chose est un ensemble en utilisant l' `set?` prédicat:

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

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

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

(set? nil)
;;=> false

(set? 42)
;;=> false

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

Vous pouvez tester si une carte contient un élément donné dans le temps "constant" en utilisant le `contains?` prédicat:

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

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

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

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

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

De plus, les ensembles eux-mêmes sont des fonctions qui prennent un élément et renvoient cet élément s'il est présent dans l'ensemble, ou `nil` s'il ne l'est pas:

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

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

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

(#{} nil)
;;=> nil

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

Vous pouvez utiliser `conj` pour obtenir un ensemble contenant tous les éléments d'un ensemble existant, plus un élément supplémentaire:

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

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

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

Vous pouvez utiliser `disj` pour obtenir un ensemble `disj` tous les éléments d'un ensemble existant, moins un élément:

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

(disj #{:foo} :foo)
```

```
;;=> #{}

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

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

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

`count` renvoie le nombre d'éléments, en temps constant:

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

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

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

Vous pouvez obtenir une séquence de tous les éléments d'un ensemble en utilisant `seq` :

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

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

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

Plans

Contrairement à la liste, qui est une structure de données séquentielle, et le vecteur, qui est à la fois séquentiel et associatif, la carte est exclusivement une structure de données associative. Une carte consiste en un ensemble de correspondances entre les clés et les valeurs. Toutes les clés sont uniques, donc les cartes prennent en charge la recherche "constante" entre les clés et les valeurs.

Une carte est indiquée par des accolades:

```
{}
;;=> {}

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

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

Chaque paire de deux éléments est une paire clé-valeur. Ainsi, par exemple, la première carte ci-dessus n'a pas de correspondance. Le second a un mapping, de la clé `:foo` à la valeur `:bar` . Le

troisième a deux mappages, un de la clé `:foo` à la valeur `:bar` , et un de la clé `:baz` à la valeur `:qux` . Les cartes sont intrinsèquement non ordonnées, donc l'ordre dans lequel les correspondances apparaissent n'a pas d'importance:

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

Vous pouvez tester si quelque chose est une carte en utilisant la `map?` prédicat:

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

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

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

(map? nil)
;;=> false

(map? 42)
;;=> false

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

Vous pouvez tester si une carte contient une *clé* donnée dans le temps "constant" en utilisant le `contains?` prédicat:

```
(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
```

Vous pouvez obtenir la valeur associée à une clé en utilisant `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
```

De plus, les cartes elles-mêmes sont des fonctions qui prennent une clé et renvoient la valeur associée à cette clé:

```
({: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
```

Vous pouvez obtenir une entrée de carte complète (clé et valeur ensemble) en tant que vecteur à deux éléments en utilisant `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
```

Vous pouvez extraire la clé ou la valeur d'une entrée de carte en utilisant la `key` ou `val` respectivement:

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

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

Notez que, bien que toutes les entrées de carte Clojure soient des vecteurs, tous les vecteurs ne sont pas des entrées de carte. Si vous essayez d'appeler `key` ou `val` sur quelque chose qui n'est pas une entrée de carte, vous obtiendrez une `ClassCastException` :

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

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

Vous pouvez tester si quelque chose est une entrée de carte en utilisant l' `map-entry?` prédicat:

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

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

Vous pouvez utiliser `assoc` pour obtenir une carte ayant toutes les mêmes paires clé-valeur qu'une carte existante, avec un mappage ajouté ou modifié:

```
(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}
```

Vous pouvez utiliser `dissoc` pour obtenir une carte ayant toutes les mêmes paires clé-valeur qu'une carte existante, avec éventuellement un mappage supprimé:

```
(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` renvoie le nombre de mappages, en temps constant:

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

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

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

Vous pouvez obtenir une séquence de toutes les entrées dans une carte en utilisant `seq` :

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

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

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

Encore une fois, les cartes ne sont pas ordonnées, de sorte que l'ordre des éléments dans une séquence obtenue en appelant `seq` sur une carte n'est pas défini.

Vous pouvez obtenir une séquence de seulement les clés ou simplement les valeurs dans une carte en utilisant respectivement des `keys` ou des `vals` :

```
(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 ajoute une syntaxe littérale pour représenter de manière plus concise une carte où les clés partagent le même espace de noms. Notez que la carte dans les deux cas est identique (la carte ne "connaît" pas l'espace de nommage par défaut), ceci est simplement une commodité syntaxique.

```
;; 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"})
```

Lire Collections et séquences en ligne: https://riptutorial.com/fr/clojure/topic/1389/collections-et-sequences

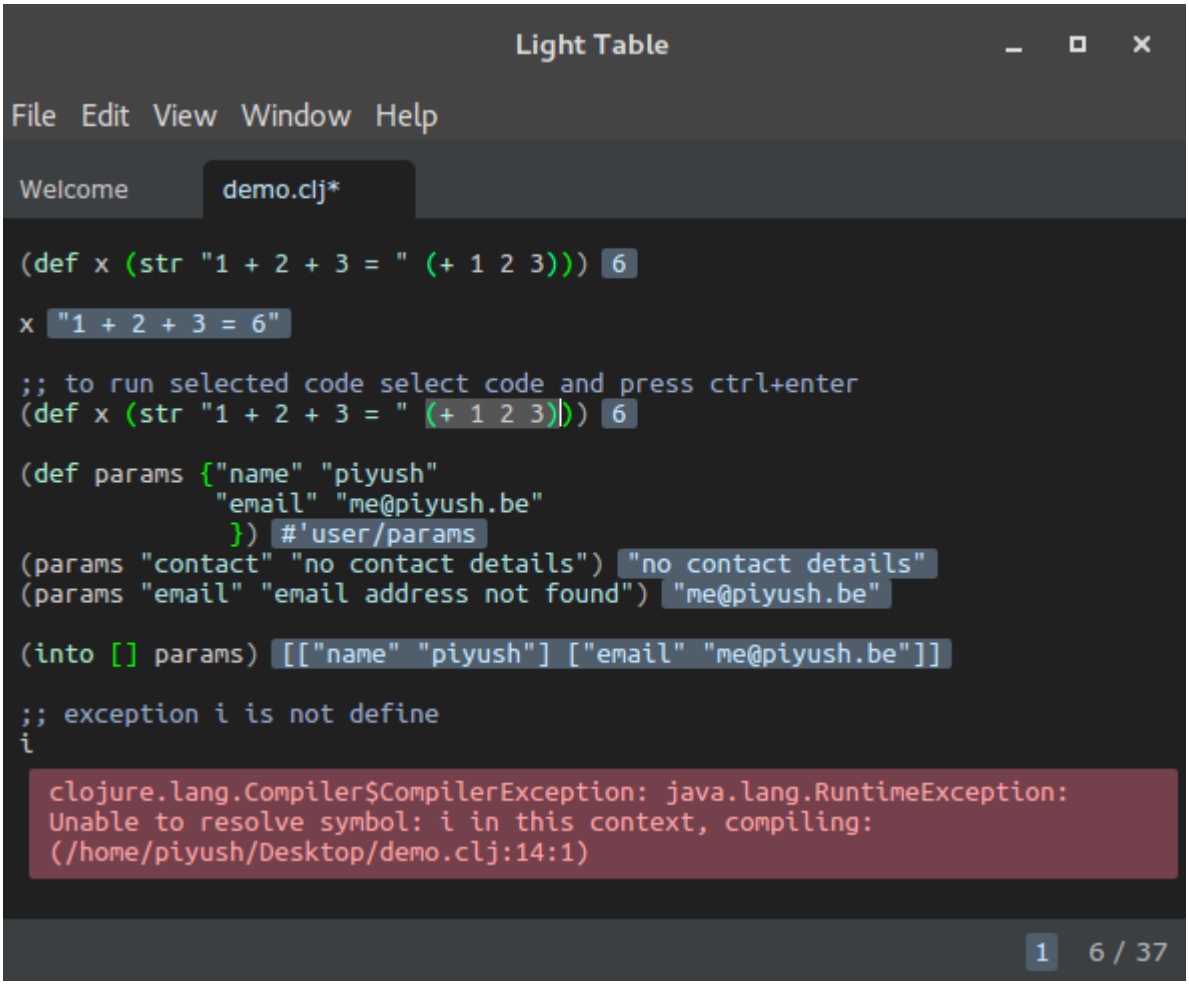
Chapitre 9: Configuration de votre environnement de développement

Examples

Table lumineuse

Light Table est un bon éditeur pour apprendre, expérimenter et exécuter des projets Clojure. Vous pouvez également exécuter des projets lein / boot en ouvrant le fichier `project.clj` . Il va charger toutes les dépendances du projet.

Il prend en charge l'évaluation en ligne, les plug-ins et bien plus encore. Il n'est donc pas nécessaire d'ajouter des instructions d'impression et de vérifier la sortie dans la console. Vous pouvez exécuter des lignes individuelles ou des blocs de code en appuyant sur Ctrl `ctrl + enter` . Pour exécuter un code partiel, sélectionnez le code et appuyez sur `ctrl + enter` . Vérifiez la capture d'écran suivante pour savoir comment utiliser Light Table pour apprendre et tester le code Clojure.



Des binaires pré-construits de Light Table peuvent être trouvés ici . Aucune autre configuration n'est requise.

Light Table est capable de localiser automatiquement votre projet Leiningen et d'évaluer votre code. Si vous n'avez pas installé Leiningen, installez-le en suivant les instructions ci- dessous .

Documentation: docs.lighttable.com

Emacs

Pour configurer Emacs afin qu'il fonctionne avec Clojure, installez `clojure-mode` et `cider` package à partir de melpa:

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

Maintenant, lorsque vous ouvrez un fichier `.clj` , exécutez `Mx cider-jack-in` pour vous connecter à une REPL. Alternativement, vous pouvez utiliser `Cu Mx` (cider-jack-in) pour spécifier le nom d'un projet `lein` ou `boot` sans avoir à visiter aucun fichier. Vous devriez maintenant pouvoir évaluer les expressions dans votre fichier en utilisant `Cx Ce` .

Éditer du code dans des langages de type lisp est beaucoup plus confortable avec un plugin d'édition compatible avec Paren. Emacs a plusieurs bonnes options.

- `paredit` Un mode d'édition classique de Lisp qui a une courbe d'apprentissage plus raide, mais qui fournit beaucoup de puissance une fois maîtrisé.

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

- `smartparens` Un nouveau projet avec des objectifs et une utilisation similaires à `paredit` , mais offre également des capacités réduites avec les langages non-Lisp.

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

- `parinfer` Un mode d'édition Lisp beaucoup plus simple qui opère principalement par inférence d'imbrication paren appropriée à partir de l'indentation.

  L'installation est plus `parinfer-mode` , consultez la page Github pour le `parinfer-mode` pour les instructions de configuration .

Pour activer `paredit` en `clojure-mode` :

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

Pour activer `smartparens` en `clojure-mode` :

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

Atome

Installez Atom pour votre distribution ici .

Après cela, lancez les commandes suivantes depuis un terminal:

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

IntelliJ IDEA + Cursive

Téléchargez et installez la dernière version d'IDEA.

Téléchargez et installez la dernière version du plug-in Cursive.

Après avoir redémarré IDEA, Cursive devrait être prêt à l'emploi. Suivez le guide de l' utilisateur pour affiner l'apparence, les raccourcis clavier, le style de code, etc.

Note: Comme `IntelliJ` , `Cursive` est un produit commercial, avec une période d'évaluation de 30 jours. Contrairement à `IntelliJ` , il n'a pas d'édition communautaire. Des licences non commerciales gratuites sont disponibles pour les particuliers à des fins non commerciales, y compris le piratage personnel, le logiciel libre et le travail d'étudiant. La licence est valable 6 mois et peut être renouvelée.

Spacemacs + CIDER

Spacemacs est une distribution d'emacs livrée avec de nombreux paquets préconfigurés et faciles à installer. En outre, il est très convivial pour ceux qui sont familiers avec le style d'édition vim. Spacemacs fournit une couche de Clojure basée sur CIDER .

Pour l'installer et le configurer pour l'utiliser avec Clojure, installez d'abord emacs. Faites ensuite une sauvegarde de vos configurations précédentes:

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

Ensuite, clonez le référentiel des spacemacs:

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

Maintenant, ouvrez emacs. Il vous posera quelques questions concernant vos préférences. Ensuite, il télécharge plus de paquets et configure vos emacs. Après que les spacemacs soient installés, vous êtes prêt à ajouter le support de Clojure. Appuyez sur `SPC fed` pour ouvrir votre fichier `.spacemacs` pour la configuration. Trouvez `dotspacemacs-configuration-layers` dans le fichier, sous `dotspacemacs-configuration-layers` se trouve un paren ouvert. Partout entre les parens dans un nouveau type de ligne `clojure` .

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

Appuyez sur `SPC fe R` pour sauvegarder et installer le calque. Maintenant, dans n'importe `.clj` fichier `.clj` si vous appuyez sur `, si` spacemacs essaiera de générer une nouvelle connexion REPL cloure à votre projet, et si elle réussit, elle apparaîtra dans la barre d'état, puis vous pourrez appuyer sur `, ss` pour ouvrir un nouveau tampon REPL pour évaluer vos codes.

Pour plus d'informations sur spacemacs et cidre contacter leurs documentations. Documents spacemacs , documents sur le cidre

Vim

Installez les plug-ins suivants en utilisant votre gestionnaire de plug-ins préféré:

1. fireplace.vim : le support de Clojure REPL
2. vim-sexp : pour apprivoiser ces calins autour de vos appels de fonction
3. vim-sexp-mappings-pour-regular-people : cartographie sexp modifiée, plus facile à supporter
4. vim-surround : effacez facilement, modifiez, ajoutez "
5. salve.vim : Prise en charge de Vim statique pour Leiningen et Boot.
6. rainbow_parentheses.vim : De meilleures parenthèses arc-en-ciel

et se rapportent également à la coloration syntaxique, à l'achèvement omni, à la mise en évidence avancée, etc.

1. vim-clojure-static (si vous avez un vim plus ancien que 7.3.803, les nouvelles versions sont livrées avec)
2. vim-clojure-highlight

Paredit.vim et vim-parinfer sont d' autres options à la place de vim-sexp .

Lire Configuration de votre environnement de développement en ligne: https://riptutorial.com/fr/clojure/topic/1387/configuration-de-votre-environnement-de-developpement

Chapitre 10: core.async

Examples

opérations de base du canal: création, mise en place, prise, fermeture et tampons.

`core.async` consiste à créer des *processus* qui prennent des valeurs et placent des valeurs dans des *canaux* .

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

Créer des canaux avec `chan`

Vous créez un canal en utilisant la fonction `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
```

Mettre des valeurs dans les canaux avec `>!!` et `>!`

Vous mettez des valeurs dans un canal avec `>!!` :

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

Vous pouvez mettre n'importe quelle valeur (chaînes, nombres, cartes, collections, objets, même d'autres canaux, etc.) dans un canal, sauf le `nil` :

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

Selon le tampon de la chaîne, `>!!` peut bloquer le thread en cours.

```
(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.
```

```
  )
```

A l'intérieur d'un bloc `(go ...)` , vous pouvez et devez utiliser `a/>!` au lieu d' `a/>!!` :

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

Le comportement logique sera le même `a/>!!` , mais seul le processus logique de la goroutine sera bloqué à la place du thread OS réel.

Utiliser `a/>!!` l'intérieur d'un bloc `(go ...)` est un anti-pattern:

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

Prendre des valeurs de canaux avec `<!!`

Vous prenez une valeur d'un canal en utilisant `<!!` :

```
;; 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 aucun élément n'est disponible dans le canal, `a/<!!` va bloquer le thread actuel jusqu'à ce qu'une valeur soit placée dans le canal (ou que le canal soit fermé, voir plus loin):

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

A l'intérieur d'un bloc `(go ...)` , vous pouvez et devez utiliser `a/<!` au lieu d' `a/<!!` :

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

Le comportement logique sera le même `a/<!!` , mais seul le processus logique de la goroutine sera bloqué à la place du thread OS réel.

Utiliser `a/<!!` l'intérieur d'un bloc `(go ...)` est un anti-pattern:

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

Canaux de fermeture

Vous *fermez* une chaîne avec `a/close!` :

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

Une fois qu'un canal est fermé et que toutes les données du canal ont été épuisées, les prises renvoient toujours `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
```

Asynchrone met avec `put!`

En alternative à `a/>!!` (qui peut bloquer), vous pouvez appeler `a/put!` mettre une valeur dans un canal dans un autre thread, avec un rappel facultatif.

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

En ClojureScript, depuis le blocage du thread en cours n'est pas possible, `a/>!!` n'est pas supporté et `put!` est le seul moyen de placer des données dans un canal en dehors d'un bloc `(go)` .

Asynchrone prend à `take!`

En alternative à `a/<!!` (qui peut bloquer le thread en cours), vous pouvez utiliser `a/take!` pour prendre une valeur d'un canal de manière asynchrone, en le passant à un rappel.

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

Utilisation de tampons de largage et de glissement

Avec les tampons de dépôt et de glissement, ne bloque jamais, mais lorsque le tampon est plein, vous perdez des données. Le tampon de suppression perd les dernières données ajoutées, tandis que les tampons glissants perdent les premières données ajoutées.

Exemple de tampon de suppression:

```
(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
```

Exemple de tampon coulissant:

```
(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
```

Lire core.async en ligne: https://riptutorial.com/fr/clojure/topic/5496/core-async

Chapitre 11: Correspondance de motif avec core.match

Remarques

La bibliothèque `core.match` implémente un algorithme de compilation de correspondance de modèle qui utilise la notion de "nécessité" à partir de la correspondance de modèle paresseux.

Examples

Littéraux correspondants

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

;=> 4
```

Faire correspondre un vecteur

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

;=> :a2
```

Faire correspondre une carte

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

;=> :a1
```

Correspondant à un symbole littéral

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

```
;=> :success
```

Lire Correspondance de motif avec core.match en ligne:
https://riptutorial.com/fr/clojure/topic/2569/correspondance-de-motif-avec-core-match

Chapitre 12: Démarrer avec le développement web

Examples

Créer une nouvelle application Ring avec http-kit

Ring est une API standard de facto pour les applications HTTP clojure, similaire à Ruby's Rack et WSGI de Python.

Nous allons l'utiliser avec le serveur Web http-kit .

Créer un nouveau projet Leiningen:

```
lein new app myapp
```

Ajoutez la dépendance http-kit à `project.clj` :

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

Ajouter `:require core.clj` pour http-kit à `core.clj` :

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

Définir le gestionnaire de demande de sonnerie. Les gestionnaires de requêtes ne sont que des fonctions allant de la requête à la réponse et la réponse est juste une carte:

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

Ici, il suffit de retourner 200 OK avec le même contenu pour toute demande.

Démarrer le serveur dans la fonction `-main` :

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

Exécutez avec `lein run` et ouvrez `http://localhost:8080/` dans le navigateur.

Nouvelle application web avec Luminus

Luminus est un micro-framework Clojure basé sur un ensemble de bibliothèques légères. Il vise à fournir une plate-forme robuste, évolutive et facile à utiliser. Avec Luminus, vous pouvez vous concentrer sur le développement de votre application comme vous le souhaitez sans aucune distraction. Il a également une très bonne documentation qui couvre certains des sujets de majour

Il est très facile de commencer avec le luminus. Créez simplement un nouveau projet avec les commandes suivantes:

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

Votre serveur démarrera sur le port 3000

`lein new luminus myapp` créera une application en utilisant le modèle de profil par défaut. Toutefois, si vous souhaitez ajouter des fonctionnalités supplémentaires à votre modèle, vous pouvez ajouter des conseils de profil pour les fonctionnalités étendues.

Serveurs Web

- + aleph - ajoute le support du serveur Aleph au projet
- + jetée - ajoute le soutien Jetty au projet
- + http-kit - ajoute le serveur Web HTTP Kit au projet

bases de données

- + h2 - ajoute un espace de noms db.core et des dépendances H2 db
- + sqlite - ajoute un espace de noms db.core et des dépendances SQLite db
- + postgres - ajoute un espace de nommage db.core et ajoute des dépendances PostreSQL
- + mysql - ajoute un espace de noms db.core et ajoute des dépendances MySQL
- + mongodb - ajoute les dépendances de l'espace de noms db.core et de MongoDB
- + datomic - ajoute un espace de noms db.core et des dépendances Datomic

divers

- + auth - ajoute un middleware de dépendance et d'authentification Buddy
- + auth-jwe - ajoute la dépendance Buddy avec le backend JWE
- + cider - ajoute le support de CIDER en utilisant le plugin CIDER nREPL
- + cljs - ajoute le support [ClojureScript] [cljs] avec Reagent
- + re-frame - ajoute le support [ClojureScript] [cljs] avec re-frame
- + concombre - un profil pour concombre avec clj-webdriver
- + swagger - ajoute la prise en charge de Swagger-UI à l'aide de la bibliothèque compojure-api
- + sassc - ajoute le support pour les fichiers SASS / SCSS en utilisant le compilateur en ligne de commande SassC
- + service - créer une application de service sans la configuration frontale telle que les

modèles HTML
- + war - ajoute la prise en charge de la création d'archives WAR pour un déploiement sur des serveurs tels qu'Apache Tomcat (ne doit PAS être utilisé pour les applications Immutant exécutées sur WildFly)
- + site - crée un modèle pour le site en utilisant la base de données spécifiée (H2 par défaut) et ClojureScript

Pour ajouter un profil, transmettez-le simplement comme argument après le nom de votre application, par exemple:

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

Vous pouvez également mélanger plusieurs profils lors de la création de l'application, par exemple:

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

Lire Démarrer avec le développement web en ligne:
https://riptutorial.com/fr/clojure/topic/2323/demarrer-avec-le-developpement-web

Chapitre 13: Déstructuration Clojure

Examples

Destruction d'un vecteur

Voici comment vous pouvez déstructurer un vecteur:

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

Ensuite, par exemple dans un bloc `let` , vous pouvez extraire les valeurs du vecteur très succinctement comme suit:

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

Détruire une carte

Voici comment vous pouvez déstructurer une carte:

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

Ensuite, par exemple, dans un bloc let, vous pouvez extraire les valeurs de la carte très succinctement comme suit:

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

Notez que les valeurs extraites dans chaque mappage sont à gauche et les clés auxquelles elles sont associées sont à droite.

Si vous souhaitez déstructurer des valeurs vers des liaisons ayant les mêmes noms que les clés, vous pouvez utiliser ce raccourci:

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

Si vos clés sont des chaînes, vous pouvez utiliser presque la même structure:

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

Et de même pour les symboles:

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

Si vous souhaitez déstructurer une carte imbriquée, vous pouvez imbriquer les formes de liaison expliquées ci-dessus:

```
(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]
```

Destruction des éléments restants en une séquence

Disons que vous avez un vecteur comme ça:

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

Et vous voulez extraire les 3 premiers éléments et obtenir les éléments restants sous forme de séquence. Cela peut être fait comme suit:

```
(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)
```

Destructuration de vecteurs imbriqués

Vous pouvez déstructurer les vecteurs imbriqués:

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

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

Destruction d'une carte avec des valeurs par défaut

Parfois, vous voulez détruire la clé sous une carte qui pourrait ne pas être présente dans la carte, mais vous voulez une valeur par défaut pour la valeur déstructurée. Vous pouvez le faire de cette façon:

```
(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
```

Destruction des paramètres d'une fn

La destruction fonctionne dans de nombreux endroits, ainsi que dans la liste des paramètres d'une fn:

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

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

La destruction fonctionne également pour la construction `& rest` dans la liste des paramètres:

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

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

Conversion du reste d'une séquence en carte

La destruction vous permet également d'interpréter une séquence comme une carte:

```
(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
```

C'est utile pour définir des fonctions avec des **paramètres nommés** :

```
(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
```

Vue d'ensemble

La destruction vous permet d'extraire des données de divers objets dans des variables distinctes. Dans chaque exemple ci-dessous, chaque variable est affectée à sa propre chaîne (`a` = `"a"` , b = `"b"` , & c.)

Type	Exemple	Valeur des `data` / *commentaire*
`vec`	`(let [[abc] data ...)`	`["a" "b" "c"]`
`vec` imbriqué	`(let [[[ab] [cd]] data ...)`	`[["a" "b"] ["c" "d"]]`
`map`	`(let [{a :ab :bc :c} data ...)`	`{:a "a" :b "b" :c "c"}`
- alternative:	`(let [{:keys [abc]} data ...)`	*Lorsque les variables sont nommées d'après les clés.*

Conseils:

- Les valeurs par défaut peuvent être fournies en utilisant `:or` , sinon la valeur par défaut est `nil`
- Utilisez `& rest` pour stocker un `seq` de toutes les valeurs supplémentaires au `rest` , sinon les valeurs supplémentaires sont ignorées
- Une utilisation commune et utile de la déstructuration concerne les paramètres de fonction
- Vous pouvez affecter des parties non désirées à une variable jetable (classiquement: _)

Destructuration et liaison au nom des clés

Parfois, lors de la déstructuration des cartes, vous souhaitez lier les valeurs déstructurées à leur nom de clé respectif. Selon la granularité de la structure de données, l'utilisation du schéma de déstructuration *standard* peut être un peu *verbeux* .

Disons que nous avons un enregistrement basé sur la carte comme ceci:

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

nous le déstructurions normalement comme ça:

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

ici, la structure des données est assez simple avec seulement 3 slots (*prénom, nom, pays*) mais imaginez la lourdeur si nous devions répéter tous les noms de clés deux fois pour une structure de données plus granulaire (avec bien plus de slots que 3) .

Au lieu de cela, une meilleure façon de gérer cela consiste à utiliser `:keys` (puisque nos clés sont des *mots - clés* ici) et à sélectionner le nom de la clé que nous aimerions lier ainsi:

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

La même *logique intuitive* s'applique à d'autres types de clés comme les *symboles* (en utilisant `:syms`) et les anciennes *chaînes* simples (en utilisant `: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"
```

Destructuration et attribution d'un nom à la valeur d'argument d'origine

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

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

Cet exemple imprime la sortie

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

L'argument est déstructuré et les éléments `2` et `3` sont affectés aux symboles `a` et `b` . L'argument d'origine, le vecteur entier `[2 3]` , est également affecté au symbole `xs` .

Lire Déstructuration Clojure en ligne: https://riptutorial.com/fr/clojure/topic/1786/destructuration-clojure

Chapitre 14: Effectuer des opérations mathématiques simples

Introduction

Voici comment vous ajouteriez des nombres dans la syntaxe Clojure. Comme la méthode est le premier argument de la liste, nous évaluons la méthode + (ou addition) sur le reste des arguments de la liste.

Remarques

L'exécution d'opérations mathématiques est la base de la manipulation des données et de l'utilisation des listes. Par conséquent, comprendre comment cela fonctionne est essentiel pour progresser dans la compréhension de Clojure.

Examples

Exemples mathématiques

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

;; returns 300
(+ 50 210 40)

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

Lire Effectuer des opérations mathématiques simples en ligne:
https://riptutorial.com/fr/clojure/topic/8901/effectuer-des-operations-mathematiques-simples

Chapitre 15: Emacs CIDER

Introduction

CIDER est l'acronyme de **C** lojure (script) **I** nteractive **D** veloppement **E** nvironment que **R** troupeaux. C'est une extension à emacs. CIDER vise à fournir un environnement de développement interactif au programmeur. CIDER est construit sur nREPL, un serveur REPL en réseau et SLIME a servi d'inspiration principale pour CIDER.

Examples

Évaluation de la fonction

La fonction CIDER `cider-eval-last-sexp` peut être utilisée pour exécuter le code lors de l'édition du code à l'intérieur du tampon. Cette fonction est par défaut liée à `Cx Ce` ou `Cx Ce` .

Le manuel du CIDER indique que `Cx Ce` ou `Cc Ce` va:

> Évaluez le point précédent et affichez le résultat dans la zone d'écho et / ou dans un cache.

Par exemple:

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

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

Exécuter `Cx Ce` ou `Cc Ce` alors que votre curseur est juste avant la fin paren de l'appel de la fonction `say-hello` affichera la chaîne `Hello, my name is Bob` .

Jolie impression

La fonction CIDER `cider-insert-last-sexp-in-repl` peut être utilisée pour exécuter le code lors de l'édition du code à l'intérieur du tampon et obtenir une sortie assez imprimée dans un tampon différent. Cette fonction est par défaut liée à `Cc Cp` .

Le manuel CIDER indique que `Cc Cp` va

> Evaluez le formulaire précédent et imprimez le résultat dans un tampon popup.

Par exemple

```
(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)
```

Effectuer `Cc Cp` alors que votre curseur est juste avant la fin du paren de l'appel de la fonction `get-database-config` affichera la jolie carte imprimée dans un nouveau tampon popup.

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

Lire Emacs CIDER en ligne: https://riptutorial.com/fr/clojure/topic/8847/emacs-cider

Chapitre 16: Interop Java

Syntaxe

- . vous permet d'accéder aux méthodes d'instance
- .- vous permet d'accéder aux champs d'instance
- .. macro en expansion à plusieurs invocations imbriquées de .

Remarques

En tant que langage hébergé, Clojure fournit un excellent support d'interopérabilité avec Java. Le code Clojure peut également être appelé directement depuis Java.

Examples

Appeler une méthode d'instance sur un objet Java

Vous pouvez appeler une méthode d'instance avec le . forme spéciale:

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

Vous pouvez appeler des méthodes d'instance avec des arguments comme ceci:

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

Référencement d'un champ d'instance sur un objet Java

Vous pouvez appeler un champ d'instance à l'aide de la syntaxe .- :

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

Créer un nouvel objet Java

Vous pouvez créer une instance d'objets de deux manières:

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

Ou

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

Appeler une méthode statique

Vous pouvez appeler des méthodes statiques comme ceci:

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

Ou passer des arguments, comme ceci:

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

Appeler une fonction Clojure à partir de Java

Vous pouvez appeler une fonction Clojure à partir du code Java en recherchant la fonction et en l'appelant:

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

Cela recherche la fonction `*` depuis l' `clojure.core` noms `clojure.core` et l'invoque avec les arguments 2 & 2.

Lire Interop Java en ligne: https://riptutorial.com/fr/clojure/topic/4036/interop-java

Chapitre 17: Les fonctions

Examples

Définition des fonctions

Les fonctions sont définies avec cinq composants:

L'en-tête, qui inclut le mot-clé `defn` , le nom de la fonction.

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

Un Docstring facultatif qui explique et documente ce que fait la fonction.

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

Paramètres listés entre parenthèses.

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

Le corps, qui décrit les procédures exécutées par la fonction.

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

En l'appelant:

```
=> (welcome "World")

"Hello, World!"
```

Paramètres et Arity

Les fonctions de clojure peuvent être définies avec zéro ou plusieurs paramètres.

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

```
    []
    "Hello!")

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

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

Arity

Le nombre d'arguments pris par une fonction Les fonctions prennent en charge la *surcharge d'arity* , ce qui signifie que les fonctions de Clojure permettent plusieurs "ensembles" d'arguments.

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

Les arités n'ont pas à faire le même travail, chaque arité peut faire quelque chose sans rapport:

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

Définition des fonctions variadiques

Une fonction Clojure peut être définie pour prendre un nombre arbitraire d'arguments, en utilisant le symbole **&** dans sa liste d'arguments. Tous les arguments restants sont collectés sous forme de séquence.

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

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

Appel:

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

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

Définir des fonctions anonymes

Il existe deux manières de définir une fonction anonyme: la syntaxe complète et un raccourci.

Syntaxe complète de la fonction anonyme

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

Cette expression évalue une fonction. Toute syntaxe que vous pouvez utiliser avec une fonction définie avec `defn` (`&` , argument de déstructuration, etc.), vous pouvez également faire avec le formulaire `fn` . `defn` est en fait une macro qui ne fait que `(def (fn ...))` .

Abréviation de la fonction anonyme

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

C'est la notation abrégée. En utilisant la notation abrégée, vous n'avez pas à nommer explicitement les arguments; ils se verront attribuer les noms `%1` , `%2` , `%3` et ainsi de suite en fonction de l'ordre dans lequel ils sont passés. Si la fonction n'a qu'un seul argument, son argument s'appelle simplement `%` .

Quand utiliser chaque

La notation abrégée comporte certaines limites. Vous ne pouvez pas déstructurer un argument et vous ne pouvez pas imbriquer des fonctions anonymes abrégées. Le code suivant déclenche une erreur:

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

Syntaxe prise en charge

Vous *pouvez* utiliser varargs avec des fonctions anonymes abrégées. Ceci est tout à fait légal:

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

Il prend un nombre variable d'arguments et renvoie true si chacun d'entre eux est pair:

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

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

Malgré l'apparente contradiction, il est possible d'écrire une fonction anonyme nommée en incluant un nom, comme dans l'exemple suivant. Ceci est particulièrement utile si la fonction doit s'appeler elle-même mais aussi dans les traces de pile.

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

Lire Les fonctions en ligne: https://riptutorial.com/fr/clojure/topic/3078/les-fonctions

Chapitre 18: Macros

Syntaxe

- Le `'` symbole utilisé dans l'exemple macroexpand est juste du sucre syntaxique pour la `quote` opérateur. Vous pourriez avoir écrit `(macroexpand (quote (infix 1 + 2)))` place.

Remarques

Les macros ne sont que des fonctions qui s'exécutent au moment de la compilation, c'est-à-dire pendant l'étape d' `eval` dans une boucle read-eval-print-loop .

Les macros de lecteur sont une autre forme de macro qui est développée au moment de la lecture, plutôt que lors de la compilation.

Meilleure pratique lors de la définition de la macro.

- Alpha-renaming, Puisque la macro est développée, le conflit de nom de liaison peut survenir. Le conflit de liaison n'est pas très intuitif à résoudre lors de l'utilisation de la macro. C'est pourquoi, chaque fois qu'une macro ajoute une liaison à la portée, il est obligatoire d'utiliser le `#` à la fin de chaque symbole.

Examples

Simple Infix Macro

Clojure utilise la notation préfixée, à savoir: l'opérateur vient avant ses opérandes.

Par exemple, une simple somme de deux nombres serait:

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

Les macros vous permettent de manipuler le langage Clojure dans une certaine mesure. Par exemple, vous pouvez implémenter une macro qui vous permet d'écrire du code en notation infixe (par exemple, `1 + 2`):

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

Décomposons ce que le code ci-dessus fait:

- `defmacro` est un *formulaire spécial que* vous utilisez pour définir une macro.
- `infix` est le nom de la macro que nous définissons.
- `[first-operand operator second-operand]` sont les paramètres que cette macro s'attend à

recevoir lorsqu'elle est appelée.
- `(list operator first-operand second-operand)` est le corps de notre macro. Il crée simplement une `list` avec les valeurs des paramètres fournis à la macro `infix` et la retourne.

`defmacro` est une *forme particulière* car elle se comporte un peu différemment des autres constructions Clojure: ses paramètres ne sont pas immédiatement évalués (quand on appelle la macro). C'est ce qui nous permet d'écrire quelque chose comme:

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

La macro `infix` étendra les arguments `1 + 2` dans `(+ 1 2)` , qui est un formulaire Clojure valide pouvant être évalué.

Si vous voulez voir ce que la macro `infix` génère, vous pouvez utiliser l'opérateur `macroexpand` :

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

`macroexpand` , implicite par son nom, développera la macro (dans ce cas, elle utilisera la macro `infix` pour transformer `1 + 2` en `(+ 1 2)`) mais ne permettra pas d'évaluer le résultat de l'extension de la macro par L'interprète de Clojure.

Syntaxe entre guillemets

Exemple de la bibliothèque standard (core.clj: 807):

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

- `` ` `` appelé syntaxe-citation est comme `(quote)` , mais récursif: il provoque `(let …)` , `(if …)` , etc. à ne pas évaluer lors de l'expansion de la macro mais à produire tel quel
- `~` aka unquote annule les guillemets syntaxiques pour un formulaire unique dans une forme citée avec la syntaxe. Donc, la valeur de `x` est sortie lors du développement de la macro (au lieu de sortir le symbole `x`)
- `~@` aka unquote-splicing est comme unquote mais prend un argument de liste et le développe, chaque élément de la liste se sépare
- `#` ajoute un identifiant unique aux symboles pour éviter les conflits de noms. Il ajoute le même identifiant pour le même symbole à l'intérieur de l'expression citée entre syntaxes, `and#` inside `let` et `and#` inside `if` obtiendra le même nom

Lire Macros en ligne: https://riptutorial.com/fr/clojure/topic/2322/macros

Chapitre 19: Macros de filetage

Introduction

Également appelées macros à flèche, les macros de threads convertissent les appels de fonctions imbriqués en un flux linéaire d'appels de fonctions.

Examples

Fil Dernier (- >>)

Cette macro donne la sortie d'une ligne donnée en tant que dernier argument de l'appel de fonction de ligne suivant. Par exemple

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

est la même que

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

Fil Premier (->)

Cette macro donne la sortie d'une ligne donnée en tant que premier argument de l'appel de fonction de ligne suivant. Par exemple

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

Je ne peux rien comprendre, non? Essayons encore, avec ->

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

Thread as (as->)

C'est une alternative plus flexible au premier thread ou au dernier thread. Il peut être inséré n'importe où dans la liste des paramètres de la fonction.

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

Lire Macros de filetage en ligne: https://riptutorial.com/fr/clojure/topic/9582/macros-de-filetage

Chapitre 20: Opérations sur les fichiers

Examples

Vue d'ensemble

Lire un fichier en une seule fois (déconseillé pour les gros fichiers):

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

Ecrire des données dans un fichier en une seule fois:

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

Lire un fichier ligne par ligne:

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

Ecrire un fichier ligne par ligne:

```
(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)
```

Écrivez dans un fichier en remplaçant le contenu existant:

```
(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)
```

Remarques:

- Vous pouvez spécifier des URL ainsi que des fichiers
- Les options vers `(slurp)` et `(spit)` sont respectivement transmises à `clojure.java.io/reader` et `/writer` .

Lire Opérations sur les fichiers en ligne: https://riptutorial.com/fr/clojure/topic/3922/operations-sur-les-fichiers

Chapitre 21: Transducteurs

Introduction

Les transducteurs sont des composants composables permettant de traiter des données indépendamment du contexte. Ils peuvent donc être utilisés pour traiter des collections, des flux, des canaux, etc. sans connaître leurs sources d'entrée ou leurs puits de sortie.

La bibliothèque principale de Clojure a été étendue en 1.7 pour que les fonctions de séquence telles que carte, filtre, prise, etc. renvoient un transducteur lorsqu'il est appelé sans séquence. Parce que les transducteurs sont des fonctions avec des contrats spécifiques, ils peuvent être composés en utilisant la normale `comp` fonction.

Remarques

Les transducteurs permettent de contrôler la paresse au fur et à mesure de leur consommation. Par exemple, `into` est comme on peut s'y attendre, mais la `sequence` consommera paresseusement la séquence à travers le transducteur. Cependant, la garantie de paresse est différente. Assez de la source sera consommée pour produire un élément initialement:

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

Ou décidez si la liste est vide:

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

Ce qui diffère du comportement habituel de la séquence paresseuse:

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

Examples

Petit transducteur appliqué à un vecteur

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

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

Cet exemple crée un transducteur affecté au `xf` local et utilise la `transduce` pour l'appliquer à certaines données. Le transducteur en ajoute un à chacune de ses entrées et ne renvoie que les nombres pairs.

`transduce` est comme `reduce` et réduit la collection d'entrée à une valeur unique en utilisant la fonction `+` fournie.

Cela se lit comme la dernière macro thread, mais sépare les données d'entrée des calculs.

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

Application de transducteurs

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

Appliquer à une collection en renvoyant une séquence:

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

Appliquer à une collection en réduisant la collection résultante avec une autre fonction:

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

Appliquer à une collection et `conj` le résultat à une autre collection:

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

Créez un canal asynchrone principal qui utilise un transducteur pour filtrer les messages:

```
(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
```

Création / utilisation de transducteurs

Ainsi, les fonctions les plus utilisées sur les cartes et les filtres de Clojure ont été modifiées pour renvoyer des transducteurs (transformations algorithmiques composables), sinon appelés avec une collection. Cela signifie:

`(map inc)` renvoie un transducteur et le fait aussi `(filter odd?)`

L'avantage: les fonctions peuvent être composées en une seule fonction par comp, ce qui signifie traverser la collection une seule fois. Enregistre le temps d'exécution de plus de 50% dans certains scénarios.

Définition:

```
(def composed-fn (comp (map inc) (filter odd?)))
```

Usage:

```
;; 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]
```

Lire Transducteurs en ligne: https://riptutorial.com/fr/clojure/topic/10814/transducteurs

Chapitre 22: Vars

Syntaxe

- (valeur du symbole def)
- (valeur de symbole déf "docstring")
- (déclarer symbole_0 symbole_1 symbole_2 ...)

Remarques

Cela ne doit pas être confondu avec `(defn)` , qui est utilisé pour définir des fonctions.

Examples

Types de variables

Il existe différents types de variables dans Clojure:

- Nombres

  Types de nombres:

  - entiers
  - longues (nombre supérieur à `2^31 - 1`)
  - flotteurs (décimaux)

- cordes

- collections

  Types de collections:

  - Plans
  - séquences
  - vecteurs

- les fonctions

Lire Vars en ligne: https://riptutorial.com/fr/clojure/topic/4449/vars

Chapitre 23: Vérité

Examples

Vérité

Dans Clojure, tout ce qui n'est pas `nil` ou `false` est considéré comme vrai.

Exemples:

```
(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)
```

Booléens

Toute valeur dans Clojure est considérée comme véridique sauf si elle est `false` ou `nil` . Vous pouvez trouver la véracité d'une valeur avec `(boolean value)` . Vous pouvez trouver la véracité d'une liste de valeurs en utilisant `(or)` , qui renvoie `true` si des arguments sont vrais, ou `(and)` qui renvoie `true` si tous les arguments sont vrais.

```
=> (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
```

Lire Vérité en ligne: https://riptutorial.com/fr/clojure/topic/4116/verite

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec 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	Analyse des journaux avec clojure	user2611740
3	Atome	Qwerp-Derp, systemfreund
4	clj-time	Rishu Saniya, Akanksha, Mrinal Saurabh, Vishakha Silky
5	clojure.core	Akanksha, Mrinal Saurabh, Surbhi Garg
6	clojure.spec	Adam Lee, Alex Miller, kolen, leeor, nXqd
7	clojure.test	jisaw, kolen, leeor, porglezomp, Sam Estep
8	Collections et séquences	Alex Miller, Kenogu Labz, nXqd, Sam Estep
9	Configuration de votre environnement de développement	Adeel Ansari, agent_orange, amalloy, g1eny0ung, Geoff, Kiran, kolen, MuSaiXi, Piyush, Qwerp-Derp, spinningarrow, stardiviner, superkonduktr, swlkr
10	core.async	Valentin Waeselynck
11	Correspondance de motif avec core.match	Kiran
12	Démarrer avec le développement web	Emin Tham, kolen, r00tt
13	Déstructuration Clojure	camdez, kaffein, kolen, leeor, Michał Marczyk, MuSaiXi, r00tt, RedBlueThing, ryo, tsleyson, Zaz
14	Effectuer des opérations mathématiques simples	Jim
15	Emacs CIDER	avichalp

16	Interop Java	leeor
17	Les fonctions	alejosocorro, fokz, Qwerp-Derp, tar, tsleyson
18	Macros	Alex Miller, kolen, mathk, Sam Estep, snowcrshd
19	Macros de filetage	Kusum Ijari, Mrinal Saurabh
20	Opérations sur les fichiers	Zaz
21	Transducteurs	l0st3d, Mrinal Saurabh
22	Vars	Aryaman Arora, Qwerp-Derp, Stephen Leppik, Zaz
23	Vérité	Alan Thompson, Michiel Borkent, Zaz