The Joy Of Clojure

인용

- 문법 인용 (`)
- 비인용 (~)
- 평가 이음 기호 (~@)

keyword

- :pre, :post

유용한 웹 사이트 모록

- https://github.com/joyofclojure/book-source
- https://github.com/ksseono/the-joy-of-clojure
- https://github.com/clojure-kr/translation
- https://eunmin.gitbooks.io/clojure-for-beginners/content
- https://github.com/clojure-kr/clojure-complete
- http://clojuredocs.org
- http://www.4clojure.com

Docs

- Clojure CLI tools
- Leiningen
- Boot

1. Clojure philosophy

2. Drinking from the Clojure firehose

2.4 Vars

2.5 Locals, loops, and blocks

2.5.3 Loops

RECUR

Clojure has a special form called recur that's specifically for tail recursion:

```
(defn print-down-from [x]
  (when (pos? x)
    (println x)
    (recur (dec x))))
```

```
(defn sum-down-from [sum x]
  (if (pos? x)
    (recur (+ sum x) (dec x))
    sum))
(sum-down-from 0 10)
;=> 55
```

LOOP

To help, there's a *loop* form that acts exactly like *let* but provides a target for *recur* to jump to. It's used like this:

snippet.clojure

```
(defn sum-down-from [initial-x]
  (loop [sum 0, x initial-x]
    (if (pos? x)
      (recur (+ sum x) (dec x))
      sum)))
```

2.9 Namespaces

Clojure also provide a Var *ns* that holds the value of the current namespace.

3. Dipping our toes in the pool

```
This chapter covers
  * Truthiness
  * Nil punning
  * Destructuring
  * Use the REPL to experiment
```

First we'll explore Clojure's straightforward notions of Truthiness[1)], or the distinctions between values considered logical true and those considered logical false.

Finally, we'll sit down and pair-program together to gain an appreciation for the power of Clojure's Read-Eval-Print-Loop (REPL).

3.1 Truthiness

Truthfulness may be an important virtue, but it doesn't come up much in programming. On the other hand, *Truthiness*, or the matter of logical truth values in Clojure, is critical.
Clojure has one Boolean context: the test expression of the *if* form. Other forms that expect Booleans –*and, or, when*, and so forth– are macros built on top of *if*. It's here that Truthiness matters.

```
(if true :truthy :falsey)  ;=> :truthy
(if [] :truthy :falsey)    ;=> :truthy
(if nil :truthy :falsey)   ;=> :falsey
(if false :truthy :falsey) ;=> :falsey
```

Every object is “true” all the time, unless it's *nil* or *false*.

3.1.2 Don't create Boolean objects

```
(if (Boolean/valueOf "false") :truthy :falsey)
;=> :falsey
```

3.1.3 nil versus false

3.2 Nil pun with care

Because empty collections act like true in Boolean contexts, we need an idiom for testing whether there's anything in a collection to process. Thankfully, Clojure provides just such a technique:

snippet.clojure

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

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

The seq function returns a sequence view of a collection, or nil if the collection is empty, In a language like Common Lisp, an empty list acts as a false value and can be used as a *pun* (a term with the same behavior) for such in determining a looping termination.

PREFER DOSEQ

An important point not mentioned is that it would be best to use *doseq* in this case, but that wouldn't allow us to illustrate our overarching points: the Clojure forms named with do at the start (doseq, dotimes, do, and so on) are intented for side-effects in their bodies and generally return nil as their result.

3.3 Destructuring

PATTERN MATCHING

Destructuring is loosely related to pattern matching found in Haskell, KRC, or Scala, but much more limited in scope. For more full-featured pattern matching in Clojure, consider using http://github.com/dcolthorp/matchure, which may in the future be included in contrib as clojure.core.match

3.3.2 Destructuring with a vector

So let's try that again, but use destructuring with let to create more convenient locals for the parts of Guy's name:

```
(let [[f-name m-name l-name] guys-whole-name]
  (str l-name ", " f-name " " m-name))
```

Positional destructuring
This positional destructuring doesn't work on maps and sets because they're not logically aligned sequentially. Thankfully, positional destructuring will work with Java's

java.util.regex.Mather and anything implementing the CharSequence and java.util.RandomAccess interfaces.

```
(let [range-vec (vec (range 10)) [a b c & more :as all] range-vec]
  (println "a b c are:" a b c)
  (println "more is:" more)
  (println "all is:" all)
)
; a b c are: 0 1 2
; more is: (3 4 5 6 7 8 9)
; all is: [0 1 2 3 4 5 6 7 8 9]
;=> nil
```

3.3.3 Destructuring with a map

ASSOCIATIVE DESTRUCTURING

One final technique worth mentioning is associative destructuring. Using a map to define a number of destructure bindings isn't limited to maps. We can also destructure a vector by providing a mpa declaring the local name as indices into them, as shown:

snippet.clojure

```
(let [{first-thing 0, last-thing 3} [1 2 3 4]]
  [first-thing last-thing])
;=> [1 4]
```

3.4 Using the REPL to experiment

Most software development projects include a stage where you're not sure what needs to happen next. Perhaps you need to use a libray or part of a library you've never touched before. Or perhaps you know what your input to a particular function will be, and what the output should be, but you aren't sure how to get form one to other. In more static languages, this can be time-consuming and frustrating; but by leveraging the power of the Clojure REPL, the interactive command prompt, it can actually by fun.

3.4.1 Experimenting with seqs

Clojure provides find-doc, which searched not just function names but also their doc strings for the given term:

snippet.clojure

```
(find-doc "xor")
; -------------------------; clojure.core/bit-xor
; ([x y])
;   Bitwise exclusive or
;=> nil
```

So th function you need is called bit-xor:

snippet.clojure

```
(bit-xor 1 2)
;=> 3
```

snippet.clojure

```
(defn xors [max-x max-y] (for [x (range max-x) y (range max-y)] [x y
(bit-xor x y)]))
(xors 2 2)
;=> ([0 0 0] [0 1 1] [1 0 1] [1 1 0])
```

THE CONTRIB FUNCTION SHOW

The clojure-contrib library also has a function show in the clojure.contrib.repl-utils namespace that allows for more useful printouts of class members than we show using for.

4. On scalars

This chapter covers

- Understanding precision
- Trying to be rational
- When to use keywords
- Symbolic resolution
- Regular expressions - the second problem

4.1 Understanding precision

4.1.1 Truncation

snippet.clojure

```
(let [imadeuapi 3.14159265358979323846264338327950288419716939937M]
  (println (class imadeuapi))
  imadeuapi)
; java.math.BigDecimal
;=> 3.14159265358979323846264338327950288419716939937M

(let [butieatedit 3.14159265358979323846264338327950288419716939937]
  (println (class butieatedit))
  butieatedit)
; java.lang.Double
;=> 3.141592653589793
```

As we show, the local butieatedit is truncated because the default Java double type is insufficient. On the other hand, imadeuapi uses Clojure's literal notation, a suffix character M, to declare a value as requiring arbitrary decimal representation.

Overflow

snippet.clojure

```
(+ Integer/MAX_VALUE Integer/MAX_VALUE)
;=> java.lang.ArithmeticException: integer overflow

(unchecked-add (Integer/MAX_VALUE) (Integer/MAX_VALUE))
;=> -2
```

4.2 Trying to be rational

4.2.2 How to be rational

snippet.clojure

```
(def a (rationalize 1.0e50))
(def b (rationalize -1.0e50))
(def c (rationalize 17.0e00))

(+ (+ a b) c)
;=> 17

(+ a (+ b c))
;=> 17

(let [a (rationalize 0.1)
      b (rationalize 0.2)
      c (rationalize 0.3)]
  (=
    (* a (+ b c))
    (+ (* a b) (* a c))))
;=> true
```

There are a few rules of thumb to remember if you want to maintain perfect accuracy in your computations:

1. Never use Java math libraries unless they return results of BigDecimal, and even then be suspicious.
2. Don't rationalize values that are Java float or double primitives.
3. If you must write your own high-precision calculations, do so with rationals.
4. Only convert to a floating-point representation as a last resort.

4.3 When to use keywords

The purpose of Clojure *keywords*, or *symbolic identifiers*, can sometimes lead to confusion for first-time Clojure programmers, because their analogue isn't often found[2] in other languages. This section will attempt to alleviate the confusion and provide some tips for how keywords are typically used.

4.3.1 How are keywords different from symbols?

Keywords *always* refer to themselves. What this means is that the keyword :magma always has the value :magma, whereas the symbol *ruins* may refer to any legal Clojure value or reference.

AS KEYS

Because keywords are self-evaluating and provide fast equality checks, they're almost always used in the context of map keys. An equally important reason to use keywords as map keys is that they can be used as functions, taking a map as an argument, to perform value lookups:

snippet.clojure

```
(def population {:zombies 2700, :humans 9})

(:zombies population)
;=> 2700

(println (/ (:zombies population)
            (:humans population))
         "zombies per capita")
; 300 zombies per capita
```

This leads to much more concise code.

AS ENUMERATIONS

Often, Clojure code will use keywords as enumeration valus, such as :small, :medium, and :large. This provides a nice visual delineation within the source code.

AS MULTIMETHOD DISPATCH VALUES

Because keywords are used often as enumerations, they're ideal candidates for dispatch values for multimethods.

AS DIRECTIVES

Another common use for keywords is to provide a directive to a function, multimethod, or macro. A simple way to illustrate this is to imagine a simple function pour, shown in listing 4.5, that takes two numbers and returns a lazy sequence of the range of those numbers. But there;s also a mode for this function that takes a keyword :toujours, which will instead return an infinite lazy range starting with the first number and continuing "forever."

Listing 4.5 Using a keyword as a function directive

snippet.clojure

```clojure
(defn pour [lb ub]
  (cond
    (= ub :toujours) (iterate inc lb)
    :else (range lb ub)))

(pour 1 10)
;=> (1 2 3 4 5 6 7 8 9)

(pour 1 :toujours)
; ... runs forever
```

An illustrative bonus with pour is that the macro **cond** itself used a directive **:else** to mark the default conditional case. In this case, **cond** uses the fact that the keyword **:else** is truthy; any keyword (or truthy vlaue) would've worked just as well.

4.3.2 Qualifying your keywords

Keywords don't belong to any specific namespace, although they may appear to if namespace qualification is used:

snippet.clojure

```
::not-in-ns
;=>:user/not-in-ns
```

Separating the plumbing from the domain
Within a namespace named **crypto**, the keywords **::rsa** and **::blowfish** make sense as being namespace-qualified. Likewise, should we create a namespace aquarium, then using ::blowfish within is contextually meaningful. Likewise, when adding metadata to structures, you should consider using qualified keywords as keys and directives if their intention is domain-oriented. Observe the following code:

snippet.clojure

```
(defn do-blowfish [directive]
  (case directive
    :aquarium/blowfish (println "feed the fish")
    :crypto/blowfish   (println "encode the message")
    :blowfish          (println "not sure what to do")))

(ns crypto)
(user/do-blowfish:blowfish)
; not sure what to do

(user/do-blowfish::blowfish)
; encode the message

(ns aquarium)
(user/do-blowfish:blowfish)
; not sure what to do

(user/do-blowfish::blowfish)
; feed the fish
```

When switching to different namespaces using ns, you can use the namespace-qualified keyword syntax to ensure that the correct domain-specific code path is executed.

4.4 Symbolic resolution

Symbols in Clojure are roughly analogous to identifiers in many other languages–words that refer to other things. In a nutshell, symbols are primarily used to provide a name for a given value. But in Clojure, symbols can also be referred to directly, by using the symbol or quote function or the ' special operator.

snippet.clojure

```clojure
(identical? 'goat 'goat)
;=>false

(= 'goat 'goat)
;=>true

(name 'goat)
"goat"

(let [x 'goat y x] (identical? x y))
;=> true
```

4.4.1 Metadata

Clojure allows the attachment of metadata to various objects, but for now we'll focus on attaching metadata to symbols. The *with-meta* function takes an object and a map and returns another object of the same type with the metadata attached. The reason why equally named symbols are often not the same instance is because each can have its own unique metadata:

snippet.clojure

```clojure
(let [x (with-meta 'goat {:ornery true})
      y (with-meta 'goat {:ornery false})]
  [(= x y)
    (identical? x y)
    (meta x)
    (meta y)])
;=> [true false {:ornery true} {:ornery false}]
```

4.4.2 Symbols and namespaces

Like keywords, symbols don't belong to any specific namespace. Take, for example, the following code:

snippet.clojure

```clojure
(ns where-is)
(def a-symbol 'where-am-i)

a-symbol
;=> where-am-i

(resolve 'a-symbol)
;=> #'where-is/a-symbol
```

```
`a-symbol
;=> where-is/a-symbol
```

5. Composite data types

This chapter covers

- Persistence, sequences, and complexity
- Vectors: creating and using them in all their varieties
- Lists: Clojure's code from data structure
- How to use persistent queues
- Persistent sets
- Thinking in maps
- Putting it all together: finding the position of items in a sequence

5.1 Persistence, sequences, and complexity

Clojure's composite data types have some unique properties compared to composites in many mainstream languages. Terms such as *persistent* and *sequence* come up, and not always in a way that makes their meaning clear.

5.2 Vectors: creating and using them in all their varieties

5.2.1 Building vectors

snippet.clojure

```
(let [my-vector [:a :b :c]]
(into my-vector (range 10)))
;=> [:a :b :c 0 1 2 3 4 5 6 7 8 9]
```

5.3 Lists: Clojure's code form data structure

5.4 How to use persistent queues

6. Being lazy and set in your ways

This chapter covers

- Immutability
- Designing a persistent toy
- Laziness
- Putting it all together: a lazy quicksort

6.1 On immutability

7. Functional programming

This chapter covers

- Functions in all their forms
- Closures
- Thinking recursively
- Putting it all together: A* pathfinding

Prefer higher-order functions when processing sequences
We mentioned in section 6.3 that one way to ensure that lazy sequences are never fully realized in memory is to prefer (Hutton 1999) higher-order functions for processing. Most collection processing can be performed with some combination of the following functions: map, reduce, filter, for, some, repeatedly, sort-by, keep take-while, and drop-while But higher-order functions aren't a panacea for every solution. Therefore, we'll cover the topic of recursive solutions deeper in section 7.3 for those cases when higherorder functions fail or are less than clear.

관련 문서

- Clojure

[1]
As a deviation from the definitionn coined by Stephen Colbert in his television show *The Colbert Report*. Ours isn't about matters of gut feeling but rather about matters of Clojure's logical truth ideal
[2]
Ruby has a symbole type that acts, looks, and is used similarly to Clojure keywords