Transducers Introduction

Clojure Exchange 2017 - Workshops

What are they?

- A model for sequential processing
- Part of Clojure since 1.7 (end of 2014)
- A functional abstraction/pattern
- Allow for reusable computation recipes

What are they not?

- A library
- Replacing other basic sequence functions/macros
- Just a performance optimization
- Reducers (some overlapping though)

How do they look like?

```clojure
;; plain
(reduce +
  (filter odd?
    (map inc
      (range 10))))

;; Or plain with '->>' macro
(->> (range 10)
     (map inc)
     (filter odd?)
     (reduce +))
```

```clojure
;; transducers
(transduce
  (comp (map inc) (filter odd?))
  +
  (range 10))
```

Easily seen differences

Plain

- No “comp”
- Nested calls
- Using “reduce”

Transducers

- “Comp” (removing the nesting)
- “Transduce” instead of “reduce”
- Single call
- No “reduce” call, but “transduce”

Not so immediate differences

Plain

- 3 intermediate collections
- Transforming operations (e.g. map/filter) are applied on separated passes of the sequence
- Transforming functions evaluate at sequence application (e.g. (map inc xs))

Transducers

- No intermediate collections
- Transforming operations (e.g. map/filter) are applied as a composition during a single pass
- “Transduce” is using “reduce” underneath
- Transforming functions are *not* evaluated at composition time (e.g. (map inc))

Uhm, why do we care?

- Transformations become isolated from input/output
- Transformations become composable/reusable
- Performance boost single pass iteration
- Concentrate on your custom transducer, leave sequential iteration to somebody else.

So, should we just use them all the time?

- Some transformations are not straightforward to translate (e.g. `(->> [[0 1 2] [3 4 5] [6 7 8]] (apply map vector))`)
- Scenarios involving infinite laziness (e.g. `(take 3 (sequence (mapcat repeat) [1])) ;; boom!`)
- Realising intermediate results unnecessarily (e.g. `(first (sequence (comp (mapcat range) (mapcat range)) [3000 6000 9000])) ;; boom!`)
- Slower for small collections or just few transformations.

Main API

- **transduce**: eager, single pass. All input evaluated.
- **sequence**: delayed, cached. Chunked (32 items). Transudcers applied once then cached.
- **eduction**: delayed, no caching. Input consumed on demand. Transducers re-evalued on 2nd pass.
- **into**: eager. Transduce into another data type.

Current transducers line-up

Out of the box:

mapcat, remove, take, take-while, take-nth, drop, drop-while, replace, partition-by, partition-all, keep, keep-indexed, map-indexed, distinct, interpose, dedupe, random-sample, cat

Resources

- Transducers presentation by Rich
- Transducers official reference guide
- Article about the Transducers functional abstraction

Our driving example: financial products

- An app receives regular updates of fin products (loans, mortgages, credit cards etc).
- Users can search for the best product.
- Each update contains +10k products as clj maps.
- We want to process the data in a timely manner.

Lab 01
Transducers Introduction

Lab prerequisites

- JDK/Java 1.8 installation
- Install GIT
- Install leiningen
- git clone http://github.com/uswitch/transducers-workshop

Goal of Lab1

- **Task1**: data preparation.
- **Task2**: filter data by user search criteria.
- **Task3**: store specific reusable searches.

Open transducers-workshop.lab01 namespace for additional instructions.