Transducers Introduction

Transducers in Practice Workshop - CUFP 2017

What are they?

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

What are they not?

- A library
- Replacing other basic sequence functions/macros
- Reducers

A comparison

```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))
```

Visible 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 visible differences

Plain

- 3 intermediate collections generated
- Transforming operations (e.g. map/filter) are applied on separated scans of the sequence
- Transforming functions always evaluate on a sequence (e.g. `(map inc xs)`)

Transducers

- Single iteration
- Transforming operations (e.g. map/filter) are applied as a composition during a single scan
- “Transduce” is using “reduce”
- Transforming functions are *not* evaluated at composition time

Why do we care?

- Transformations are isolated from input/output
- Transformations are composable/reusable
- Iteration happens once only
- Protocol-driven “create your own” experience

Why not using 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))`)
- Some scenario involving “extreme” laziness (e.g. `(take 3 (sequence (mapcat repeat) [1]))`)
- When large intermediate results are fully realized (e.g. `(first (sequence (comp (mapcat range) (mapcat range)) [3000 6000 9000]))`)
- Slower for small collections or not many transformations.

Using transducers

- `transduce`: eager, single pass. All input evaluated.
- `sequence`: delayed, cached. Input consumed on demand. Transformations applied once and cached.
- `eduction`: delayed, no caching. Input consumed on demand. Transformations repeating when re-used.
- `into`: eager. Transduce into another data type.

Transducers enabled functions

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
```

Resources

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

Lab 01
Transducers Introduction

Lab prerequisites

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

Example Application

- Receives regular updates of credit products (loans, mortgages, credit cards etc).
- Given a desired amount, period, type of credit etc. returns the best deal for the user.
- The feed contains thousands of products as a large list of Clojure maps.
- We want to filter, process and present the data to the user in a timely manner.

Goal of Lab1

- Task1: data preparation.
- Task 2: filter data by user search criteria.
- Task 3: store specific reusable searches.

Open `transducers-workshop.lab01` namespace for additional instructions.